THEORY AND APPLICATIONS IN NUMERICAL ALGEBRAIC GEOMETRY

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Daniel James Bates, B.A., M.S.

Andrew J. Sommese, Director

Graduate Program in Mathematics
Notre Dame, Indiana
April 2006

THEORY AND APPLICATIONS IN NUMERICAL ALGEBRAIC GEOMETRY

Abstract

by

Daniel James Bates

Homotopy continuation techniques may be used to approximate all isolated solutions of a polynomial system. More recent methods which form the crux of the young field known as numerical algebraic geometry may be used to produce a description of the complete solution set of a polynomial system, including the positive-dimensional solution components. There are four main topics in the present thesis: three novel numerical methods and one new software package. The first algorithm is a way to increase precision as needed during homotopy continuation path tracking in order to decrease the computational cost of using high precision. The second technique is a new way to compute the scheme structure (including the multiplicity and a bound on the Castelnuovo-Mumford regularity) of an ideal supported at a single point. The third method is a new way to approximate all solutions of a certain class of two-point boundary value problems based on homotopy continuation. Finally, the software package, Bertini, may be used for many calculations in numerical algebraic geometry, including the three new algorithms described above.

For Sarah, my precious bride.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I would like to begin by thanking the members of my thesis committee for the time and effort that they have put into reviewing this thesis and for their comments and guidance while I was writing it. The contents of this thesis represent several years of work, so the organization of the material proved to be something of a nightmare. The ideas and suggestions of my advisor and readers helped me to convey this material far more effectively than I would have managed if left to my own devices.

Taking a step back in time, I am eternally grateful to my parents and my family (Bates, York, Payne, Carson, etc.) for all of their love and support over the years. They have done more for me than I could hope to record presently. All I can offer is that I would not have made it to this point were it not for many years of hard work, support, and sacrifice on their part.

I learned the background necessary for graduate school in mathematics as an undergrad at The College of Wooster. John Ramsay and the other math and computer science professors at Wooster are responsible for sparking my interest in mathematics in the first place. Many of my fellow Notre Dame graduate students have helped to keep that spark going over the years. I am especially grateful for all of the conversations (both mathematical and social) that I have had over the past few years with Cabral Balreira, Ryan Hutchinson, Ben Jones, Heather Hannah, Stuart Ambler, Ye Lu, Yumi Watanabe, and Jon Hauenstein. I am also

very grateful for the fellowships provided by the Arthur J. Schmitt Foundation and the Notre Dame Center for Applied Mathematics that enabled me to attend Notre Dame in the first place.

It is difficult to develop into a research mathematician without friendly colleagues and strong role models. I have been fortunate to have both. The guidance, training, and intuition provided to me over the past few years by Charles Wampler, Chris Peterson, and Gene Allgower have been key in my development, and I am excited to continue my relationships with these fine researchers far into the future. I am grateful as well for the many other researchers that have been supportive over the years, particularly T. Y. Li, Jan Verschelde, Zhonggang Zeng, Dave Nicholls, and Wenqin Zhou.

Now we come to the two individuals to whom I owe the most: my advisor and my wife. Although their plans were sometimes at odds, both were clearly focused on my success both in graduate school and life in general. If only I had had more hours in the day, I could have spent as much time as I would have liked with each of them over the past few years.

I will begin with my advisor, Andrew Sommese. Andrew falls into many of the categories of mathematicians listed above, but he has grown to be much more than a mathematician or an advisor; he is a friend and mentor. Andrew has been incredibly selfless with his time and resources since I began working with him, sending me to many conferences and spending hours upon hours helping me grow into the mathematician that I am today. Through all of his work, Andrew has guided me along a path that appears to offer many different options for the future, any one of which is far greater than what I thought possible when I started graduate school. I am glad that I can count Andrew as both a collaborator and a

good friend.

Finally, I must thank my wife, Sarah. I must also apologize, for I chose the path of a scientist and not that of a poet, so I cannot possibly put into words the many great things that I want to say to and about Sarah. Through my many tribulations during graduate school, including my candidacy exams, my (first) job search, and the writing of this thesis, the related papers, and Bertini, Sarah has stood by me and supported me in every way that a man could ever hope that his wife would (and more). Sarah has been a constant source of strength, encouragement, and happiness for me since we first met. I am truly blessed to have her as my wife. I am dedicating this thesis to Sarah, for it would likely never have come to fruition were it not for her constant stream of love and support for me.

CHAPTER 1

INTRODUCTION

Algebraic geometry is one of the oldest branches of mathematics, with its earliest roots tracing back four thousand years to Babylon (see [30]). Early modern concepts of interest included the closed form solutions of low degree univariate polynomials (as discovered by various Italian mathematicians in the 16th century) and the eventual proofs by Ruffini and Abel of the insolubility of the quintic in the 19th century (see [8], [31]). Even before the celebrated proof of what has come to be known as Abel's Impossibility Theorem, mathematicians became interested in the connection between algebraic objects and their corresponding geometric structures. As suggested in [29], algebraic geometry is the study of solutions of systems of polynomial equations.

Over the past two centuries, there have been several major schools of thought in the field of algebraic geometry, each building upon and generalizing the ideas of the previous schools. As discussed in [29], one of the earliest formal attempts at understanding the dictionary between algebra and geometry was the function analytic approach of Riemann in the late 19th century. Following Riemann and his contemporaries came the Italian school, including such luminaries as Castelnuovo and Severi. This school took a more geometric approach, and their work culminated in the classification of algebraic surfaces.

Commutative algebra was developed in part to study algebraic geometry from a more algebraic viewpoint. The work of E. Noether and Macaulay was key in this direction, and [7] is a great introduction to these concepts. In the middle of the 20th century, a new French school came into prominence, beginning with the concepts of sheaves and sheaf cohomology by Serre (see [51]) and leading to the general idea of a scheme, as developed by Grothendieck (see [42]). From a (symbolic) computational standpoint, a number of interesting and important objects related to an ideal may be computed using Gröbner basis techniques, as introduced by Buchberger in his Ph.D. thesis in 1966. All told, algebraic geometry is a huge field that has grown in many directions, so it is unfortunately impossible to go into any more details presently. Please refer to [29], [28], [49], [15], or [16] for more discussion and details.

Predating the development of modern algebraic geometry, Euler and Newton developed their well-known methods for solving ordinary differential equations and iteratively improving approximations to solutions of nonlinear equations (respectively). In 1953, Davidenko realized that these numerical methods could be applied together to find the solutions of systems of nonlinear equations (see [17]). As elaborated in [43], this method of Davidenko, known as homotopy continuation, gives a way to find the isolated solutions of polynomial systems. This opened the door for a new numerical approach to attacking problems from algebraic geometry, as developed in [45], [46], [47], [52], [53], [54], and [55]. This new approach has come to be called numerical algebraic geometry. Excellent references for numerical algebraic geometry include [38] for the zero-dimensional case and [56] in general.

There has been significant growth in the theory of and interest in numerical

and numeric-symbolic methods in algebraic geometry since the 1980s. The techniques presented in this thesis further broaden the understanding of algebraic sets from a numerical standpoint. In particular, the three algorithms presented in the following chapters broaden the range of what can be computed numerically and make certain numerical methods more trustworthy.

As is always the case, some knowledge of mathematics is necessary to understand this thesis. Chapter 2 contains some of the necessary foundational concepts, though many details are omitted. References are of course provided for those readers interested in more details.

Not surprisingly, the presence of singularities causes numerical difficulties in homotopy continuation. Although finite precision techniques can never completely account for a true singularity, increasing precision will temporarily alleviate the numerical problems caused by ill-conditioning, perhaps allowing for the convergence of the methods being employed. The standard algorithm for increasing precision in homotopy continuation (i.e., re-running paths that fail due to ill-conditioning) is more computationally expensive than necessary. A new technique for changing the precision as it is needed is given in Chapter 3.

One situation in which ill-conditioning may occur is near a multiple root. Therefore, understanding the multiplicity structure of an isolated solution of a polynomial system is important, and this scheme structure is also of interest in a more pure setting. Chapter 4 contains a new numerical method for calculating the multiplicity, as well as several other interesting invariants, of a zero-scheme. Also included in that chapter is a discussion of how to tell which singular values of a numerical matrix are truly zero, a major problem for many algorithms of symbolic-numeric algebra.

Homotopy continuation is a useful tool, and it may be applied in a variety of settings. Even within the polynomial systems setting, this method may be applied to solve a number of seemingly unrelated problems. One such application is the subject of Chapter 5, in which it is shown how to compute approximate solutions to two-point boundary value problems of a certain type using a carefully-constructed homotopy.

Bertini is a new software package that may be used for a number of different computations in numerical algebraic geometry, including homotopy continuation, numerical irreducible decomposition, and those methods described in Chapters 3, 4, and 5. Chapter 6 contains many details about Bertini, including the basic structure of a run and many implementation-specific details, while Appendix A contains not only useful snippets of source code describing some of the data types of Bertini but also a complete set of input and output files for one problem.

CHAPTER 2

BACKGROUND

2.1 Systems of multivariate polynomials

2.1.1 Basic theory

There are several different, but related, ways to regard polynomial systems. Of course there is the standard list of polynomials, given in any of a variety of formats (e.g., standard form, various factored forms, straight-line programs, etc.). One may also consider a polynomial system as a particular basis for an ideal in a polynomial ring, $k[x_1, ..., x_n]$. Given a basis, the *ideal generated by the basis* is defined to be the basis itself along with all sums of elements in the ideal and products of ideal elements with ring elements. See [31] for more details about ideals in general and ideals of polynomial rings in particular.

One may also wish to approach these algebraic objects from a geometric viewpoint. To each ideal, one may associate a geometric object known as an algebraic set or variety[1]. An *algebraic set* is just the common zero locus of an ideal, i.e., the set of points at which all of the polynomials in the ideal vanish. Although there is a unique algebraic set assigned to each ideal, the relation is not bijective. For example, in one variable, x and x^2 form different ideals but each has the origin as the associated algebraic set. There are ways of accounting for this difference

[1]The term "variety" has unfortunately been used by different authors to mean different things, so the term has become somewhat ambiguous.

between algebra and geometry (e.g., by using schemes), but such concepts are beyond the scope of this thesis. More about the dictionary between ideals and algebraic sets is described below in Section 2.3.1.

This interplay between ideals and algebraic sets is the central theme of algebraic geometry. The focus of homotopy continuation and basic numerical algebraic geometry is to produce a numerical description of the algebraic set associated to an ideal given a particular basis, i.e., a polynomial system. In particular, homotopy continuation seeks to find all isolated points in the algebraic set, while techniques from numerical algebraic geometry may be used to classify all positive-dimensional components. These methods, along with a more sophisticated description of the relationship between ideals and algebraic sets, constitute the remainder of this chapter.

Before delving into the various methods for finding the isolated solutions of a polynomial system, though, it is important to describe projective space and related topics that will play a role throughout the remainder of this chapter. Consider the equivalence relation $\sim$ on $\mathbb{C}^{n+1}$ given by $x \sim y$ if and only if $x = \lambda y$ for some $\lambda \neq 0$. Then *n-dimensional projective space*, $\mathbb{P}^n$ is defined to be the set of equivalence classes of $\mathbb{C}^{n+1}$ under $\sim$, i.e., the set of lines through the origin in $\mathbb{C}^{n+1}$.

There is no way to define a function on $\mathbb{P}^n$ since the function value at $x \in \mathbb{P}^n$ depends on the chosen representative of the equivalence class of x. *Homogeneous polynomials* are those polynomials having all terms of the same degree (where the degree of a monomial is the sum of all exponents of all variables appearing in the monomial). By restricting attention to homogeneous polynomials, it is possible to define zero sets of homogeneous polynomials. This is because of the fact that

$p(\lambda x) = \lambda^d p(x)$ if p is homogeneous where d is the degree of $p(x)$.

It may be seen below that it is at times beneficial to use homogeneous polynomials in $\mathbb{P}^n$ instead of inhomogeneous polynomials in $\mathbb{C}^n$. To homogenize an inhomogeneous polynomial, one may simply add a new variable, x_{HOM}, multiplying enough copies of x_{HOM} into each term of the polynomial to bring the degree of each term up to the maximum degree of any term of the original inhomogeneous polynomial. Similarly, one could choose to separate the variables into k *variable groups* and homogenize the polynomial with each group separately, using a multi-degree of size k and a different new variable for each variable group. This corresponds to moving from $\mathbb{C}^n$ to $\mathbb{P}^{n_1} \times \mathbb{P}^{n_2} \times ...\mathbb{P}^{n_k}$ rather than $\mathbb{P}^n$, where $\sum_{i=1}^{k}(n_i - 1) = n$. For more details about projective space and related ideas, please refer to [16].

2.1.2 Finding isolated solutions

There are a number of methods for finding all of the isolated solutions of a polynomial system. One may wonder then whether there is any need to consider methods such as homotopy continuation when existing symbolic methods will theoretically produce the same output. These methods and their drawbacks are the focus of this and the following section.

Before describing the first method for solving polynomial systems, it should be noted that such a method at least has the potential for terminating. From the Fundamental Theorem of Algebra, it is known that the number of real solutions of a single univariate polynomial is bounded from above by the degree of the polynomial (with the bound realized if working over $\mathbb{C}$ and counting multiplicity). A similar bound holds in the case of a multivariate polynomial system. Indeed, a

version of Bézout's Theorem states that the number of complex isolated solutions of a polynomial system is bounded above by the product of the total degrees of the polynomials in the system.

A number of methods for computing the isolated solutions of a polynomial system rely on the computation of a Gröbner basis for the ideal. To compute a Gröbner basis, one must first choose a term order (i.e., an ordering of the variables and all monomials). This provides a way to identify the leading term of any given polynomial. This choice is not trivial, for different orders can result in different bases. After choosing a term order, there are many methods for actually computing the Gröbner basis, although most of these are variants of the standard technique known as Buchberger's algorithm. The details of that algorithm are not important to this thesis but are described well in [15].

Although the process of producing a Gröbner basis may seem complicated, the structure itself is not. As pointed out in [15], a *Gröbner basis* of an ideal is just a special basis such that the leading term of any element of the ideal is guaranteed to be divisible by the leading term of one of the basis elements. There are many variations on these Gröbner basis methods, and there are other special types of bases of interest (e.g., border bases). More information regarding Gröbner bases may be found in [1] and [15] and the references therein.

One commonly used cluster of methods for finding the isolated solutions of a polynomial system goes by the collective name of elimination theory. The main thrust of elimination theory is to reduce the given polynomial system to a new system having the same number of solutions plus the added benefit of being easily solved. In particular, the goal is to "eliminate" variables from the system so that the resulting system is simply a univariate polynomial which is easily solved by

numerical means. After solving the eliminated system, one must reconstruct the solutions of the original system from those of the eliminated system. There is a bit of theory related to this last step which falls outside the scope of this thesis.

As discussed in [16], one may choose to employ Gröbner basis techniques using a certain term order to eliminate one or more variables. Another option is the use of resultants. A *resultant* is a polynomial constructed by taking the determinant of a specially constructed matrix formed from the coefficients of the polynomials in the system. Gröbner bases and resultants also form the core of methods for solving systems that are not a part of elimination theory. See [1], [15], [16], and [59] and the references therein for descriptions of some of these other methods. It should also be noted that a number of software packages now include these symbolic techniques. In particular, Maple, CoCoA (see [34], [35]), Macaulay (see [22]), and Singular (see [25]) all include commands for at least some of the methods described in this section.

Another class of techniques for discovering the isolated solutions of a polynomial system goes by the name of exclusion or bisection methods. The basic idea of these methods comes in the form of a two-stage loop. Each step of the loop begins by breaking all existing cells into a number of smaller cells, perhaps using bisection in each dimension. This set of cells is initially just the region where solutions could possibly exist (or any other region of interest to the user). Once the cells have been refined in this way, one may detect for each cell whether a solution could possibly occur in that cell, excluding those cells which could not contain a solution. A common way to detect whether a given cell could contain a solution is to use interval arithmetic, i.e., to find the interval of function values that could possibly occur throughout the cell and exclude the cell if zero is not in

that interval. Details about exclusion methods may be found in [2], [32], and [24].

2.1.3 Drawbacks of existing methods

Each of the methods of the previous section works well, at least in certain situations. However, as is always the case, each method also has a number of drawbacks. Thus, one must be careful about which method to apply to a given polynomial system. The goal of this section is to briefly describe the disadvantages of the methods of the previous section not to discount their usefulness, but to illustrate the need for and usefulness of other methods, such as homotopy continuation. Homotopy continuation of course suffers from its own disadvantages, although they are largely different than those encountered by the methods of the previous section.

Gröbner basis methods are probably the most commonly used methods for symbolic polynomial system solving. One difficulty that frequently arises when using such techniques over $\mathbb{Q}$ is *coefficient blowup*. If the coefficients of the polynomial system involve many different prime numbers, then the rational operations involved in the computation of a Gröbner basis leads to fractions whose numerator and denominator are arbitrarily large. For instance, as discussed in [9], a system consisting of five randomly chosen degree four polynomials in four variables with random integer coefficients between 0 and 100 has a root at the origin in affine space (since homogeneous polynomials have no constant terms). This ideal has a Gröbner basis whose elements have rational coefficients that are ratios of integers with over 200 digits.

This coefficient blowup is one type of *expression swell*. Another type also occurs when working with Gröbner bases. Namely, the number of polynomials in the

basis can grow very large by changing an ideal only slightly or by increasing the dimension. Although Gröbner basis methods generally suffer from these difficulties, some progress is being made in resolving these problems, and these techniques are often superior for small problems. Gröbner basis techniques generally work for systems of size up to 5×5 (i.e., five equations in five variables) or perhaps 10×10, although the computations are likely to overwhelm the available computational resources for systems any larger than that, except in very special cases. Techniques employing resultants have disadvantages similar to those relying on Gröbner basis methods.

Exclusion methods also have a few disadvantages. These algorithms, like the others, suffer from the "curse of dimensionality": for each additional dimension, the number of cells to consider at each stage doubles (or worse, depending upon the type of cell division being used). Also, if interval arithmetic is used, the errors induced could easily compound to the point at which it appears that most cells contain isolated solutions, whether they actually do or not. Finally, if there is a positive-dimensional component in the solution set, any cell containing a portion of that component obviously contains a solution, so the number of active cells will increase rapidly as the algorithm progresses.

2.2 Homotopy continuation

Homotopy continuation is a way of finding all isolated solutions of a system of nonlinear equations. The subroutines of this method are classical, dating back to Euler and Newton, although Davidenko appears to have been the first to employ predictor/corrector methods together (see [17]). Good modern references include [43], [6], and [56]. The next two sections describe basic homotopy continuation

and various difficulties that may be encountered when using this method.

2.2.1 The basic method

Suppose one wants to apply homotopy continuation to find the isolated roots of a polynomial system $f : \mathbb{C}^n \to \mathbb{C}^n$. One must first construct a homotopy, $H : \mathbb{C}^n \times \mathbb{C} \to \mathbb{C}^n$, depending upon a parameter $t \in \mathbb{C}$ such that the isolated solutions to $H(z, 1)$ are known and the solutions of $H(z, 0) = f(z)$ are sought, where $z \in \mathbb{C}^n$. The new parameter t is sometimes referred to as the *path tracking variable*. From theory (see Appendix A of [56] regarding Grauert's proper mapping theorem and Stein's lemma), it may be assumed that there exists at least one differentiable path $z(t) : (0, 1] \to \mathbb{C}^n \times \mathbb{C}$ such that $H(z(t); t) = 0$ for each solution of $H(z, 0) = 0$. There will be multiple such paths for isolated solutions of multiplicity greater than one. The crux of the theory is that the number of isolated solutions of each member of a parametrized family of polynomial systems is the same (counting multiplicity and points at infinity) for a Zariski open set of the parameter space. One may numerically follow these paths to estimate the desired limit $z(0) := \lim_{t \to 0} H(z(t); t)$.

Letting

$$\frac{dz(t)}{dt} := \begin{bmatrix} \frac{dz_1(t)}{dt} \\ \vdots \\ \frac{dz_N(t)}{dt} \end{bmatrix}$$

and letting $J(z, t)$ denote the Jacobian

$$\begin{bmatrix} \frac{\partial H_1(z,t)}{\partial z_1} & \cdots & \frac{\partial H_1(z,t)}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial H_n(z,t)}{\partial z_1} & \cdots & \frac{\partial H_n(z,t)}{\partial z_n} \end{bmatrix}$$

we have the differential equation that the solution path $z(t)$ satisfies (often called the Davidenko differential equation):

$$J(z(t),t)\cdot\frac{dz(t)}{dt}+\frac{\partial H(z(t),t)}{\partial t}=0. \tag{2.1}$$

Each step of homotopy continuation begins with a prediction from a sample point at $t=t_0$ to $t=t_1:=t_0-\Delta t$, typically using Euler's method, with t_0 set to 1.0 for the first step. The quantity Δt is known as the steplengh and is generally set to 0.1 at the beginning of the procedure, although, as is discussed below, it is changed according to the behavior of the predictor and corrector steps. Euler's method consists of solving the Davidenko equation (2.1) for $\frac{dz(t)}{dt}$ and adding the result to the point at $t=t_0$.

Following the predictor step comes a sequence of corrector steps which refine the initial approximation. Given an estimate to the solution curve, z_0, from the predictor step, one may iteratively apply Newton's method while holding t constant, i.e., one may iteratively compute

$$z_{i+1}:=z_i+dz \text{ where } dz:=-(J(z_i,t))^{-1}H(z_i,t). \tag{2.2}$$

When two solution paths, $z_1(t)$ and $z_2(t)$ such that $z_1(1)\neq z_2(1)$, are near one another for some value of t, a prediction for the path $z_1(t)$ may lie in the convergence basin of $z_2(t)$ rather than the basin of $z_1(t)$. The corrector, if allowed to iterate too many times, then corrects towards $z_2(t)$, and the path $z_1(t)$ is thereafter neglected. This phenomenon is known as *pathcrossing*. Pathcrossing may often be avoided by placing a limit on the number of corrector steps, declaring a failure to correct if the limit is exceeded. Naturally, setting this maximum number of steps

allowed too low will result in a premature declaration of step failure.

A given corrector step is declared to be a success if two successive estimates are within a specified tolerance, hereafter referred to as the Newton tolerance or target tolerance. Note that this and all other tolerances are absolute, rather than relative, measures. In general, the use of absolute measures often yields trouble in the presence of poor scaling, but that concern is beyond the scope of this thesis.

Homotopy continuation employs the notion of adaptive steplength. In the case of a failed step, the steplength, Δt, is decreased by some factor and the step is attempted again. If several consecutive steps have been successful, the steplength is increased by some factor. Thus the tracking procedure accelerates while the path is well-behaved but slows down as the behavior of the path worsens.

There are several ways in which a path may be declared complete. Ideally, each path will end as a success, with the final two estimates at $t = 0$ within the user-specified tolerance. The path may be declared a failure if it seems to be going to infinity, i.e., the approximations to the path grow large. Such paths are not uncommon, for they occur whenever the isolated solutions of the start system outnumber those of the target system. Any excess paths necessarily diverge. Other paths that are declared failures fail because either too many steps have been taken or so many corrector steps have failed that the steplength has been driven below some minimal threshold. This threshold is a necessary stopping criterion to keep slow-moving paths that are likely to fail from wasting CPU time.

One may not want to rely on homotopy continuation techniques to converge to an endpoint at $t = 0$ due to the ill-conditioning that occurs near singularities. Special techniques known as *endgames* have been developed to speed convergence in these situations. The endgames that are discussed in the following chapters are

introduced at the end of the next section.

2.2.2 Obstacles of homotopy continuation and their resolutions

Singularities encountered during path tracking constitute a major numerical problem. The Jacobian of the homotopy function is (by definition) singular at singularities. However, numerical woes occur not just at the singularities themselves but in neighborhoods of the singularities. Although matrix inversion is possible with exact arithmetic for all nonsingular matrices, that operation is numerically unstable for near-singular matrices. One is therefore led to define the condition number of a nonsingular matrix A with respect to a matrix norm $||\cdot||$ as $\text{cond}(A) = ||A||\cdot||A^{-1}||$. The condition number is comparable to the inverse of the distance from the matrix A to the nearest singular matrix. As a path approaches a singularity, the condition number increases, with singular matrices having an infinite condition number.

A result of Wilkinson in [64] shows exactly where the problem is. Letting $\mathbf{CN} := \log_{10}(\text{cond}(A))$ for a nonsingular matrix A and **PREC** designate the number of decimal digits being used to represent floating point numbers, the solution x of $Ax = f$ may be computed with approximately $\mathbf{PREC}-\mathbf{CN}$ accurate digits. Thus, increasing the precision of the data and the arithmetic used in the computation will result in more accuracy. The use of higher than normal precision to counteract the effects of singularities on path tracking techniques is the subject of Chapter 3.

It can be very expensive to track paths that are diverging since they have infinite length! One way to avoid this computational difficulty is to homogenize the polynomial system, i.e., work on $\mathbb{P}^m$ rather than $\mathbb{C}^n$, with $m \geq n$. However,

in homogenizing the system, extra variables are added so that the system is no longer square. To account for this, one linear polynomial should be added to the system for each variable group. For example, if one group consists of the variables $z_1, ..., z_k$, a polynomial of the form $c_1 z_1 + ... + c_k z_k + c_{k+1}$ should be added to the system, where the coefficients are all randomly chosen. This forces the computation to occur on a certain patch of $\mathbb{P}^m$, with the infinity of the original $\mathbb{C}^n$ replaced by a finite number, making the tracking of paths "to infinity" feasible. This trick is known as *projective transformation* and was introduced in [43].

Since singularities tend to occur near $t = 0$, a number of sophisticated methods known as endgames have been developed to speed convergence at the ends of paths. There is a well-developed theory regarding endgames. Please refer to [56] for the appropriate theory and descriptions of the major endgames. The most commonly used endgames were introduced in [45], [46], and [47]. Only two endgames play a prominent role in the following chapters, so only those two are described in any detail in this section.

The simplest endgame, known hereafter as the 0th order endgame, is simply to track with loose path tracking tolerances (e.g., a large Newton tolerance) until some pre-specified value of t, at which point the tolerances are tightened to their final desired settings. After reaching the endgame zone, the path should be sampled at values of t chosen in a geometric progression. The endgame then terminates if once two consecutive samples agree to the desired number of digits. This endgame is not very sophisticated and will generally do little to overcome singularities, although it does save computational time for nonsingular paths for obvious reasons.

Let

$$\Delta(r) := \left\{ s \in \mathbb{C} \;\middle|\; |s| < r \right\}$$

and let

$$\Delta^*(r) := \Delta(r) \setminus \{0\}.$$

The key analytic fact (see [45]) used in many endgames more sophisticated than the 0th order endgame is that there is a real number $R > 0$ and a positive integer c called the cycle number, such that there is a homomorphic map

$$\phi : \Delta\left(R^{\frac{1}{c}}\right) \to \mathbb{C}^N \times \mathbb{C}$$

satisfying

1. the map ϕ is an embedding on $\Delta^*\left(R^{\frac{1}{c}}\right)$;
2. $H(\phi(s)) = 0$;
3. $t = \phi_{N+1}(s) = s^c$; and
4. $z(t) = \left(\phi_1\left(t^{\frac{1}{c}}\right), \ldots, \phi_N\left(t^{\frac{1}{c}}\right)\right)$ for $0 < t < R^{\frac{1}{c}}$.

The idea of the fractional power series method is to use interpolation to discover the cycle number and then use the cycle number together with another round of interpolation to approximate $z(0)$. In particular, one may use regular path tracking techniques to collect samples $z(t_1)$, $z(t_2)$, and $z(t_3)$ at $t_1 > t_2 > t_3$. The samples and the derivatives at t_1 and t_2 may then be fit using Hermite interpolation in s, assuming $t = s^c$ for different values of the cycle number c. Naturally, not all possible values of c may be tried, so a maximum possible cycle number must be defined. The value of c that yields the best approximation to $z(t_3)$ is assumed to

be the cycle number, and two estimates of the value of $z(0)$ are made, one using the data from t_1 and t_2 and one using data from t_2 and t_3. If these estimates are within the prescribed tolerance, the path is a success. Otherwise, it is necessary to set $t_1 := t_2$, $t_2 := t_3$, choose a new value of t_3, and try again. More data points may be used for higher order fits. The fractional power series endgame often converges much more rapidly than basic path tracking alone and often provides solutions of the target system when basic path tracking (with no endgame) or less sophisticated endgames fail.

2.3 Numerical algebraic geometry

So far, all methods discussed in this chapter deal with isolated (zero-dimensional) solutions of polynomial systems. However, in practice, positive-dimensional solutions are also of interest. For example, for various engineering applications, it is useful to know how many irreducible components there are in the solution set in each dimension. It is also often useful to know at least one point on each of these components. The goal of the remainder of this chapter is to introduce some of the theory surrounding positive-dimensional components as well as some of the algorithms of numerical algebraic geometry that may be used to produce a "numerical irreducible decomposition." However, since Chapters 3 and 5 do not rely on this material in any way and Chapter 4 requires only a little of this theory, the reader would be wise to consult [23], [29], [16], or [56] for anything more than the superficial survey given below.

2.3.1 More basic algebraic geometry

To understand the "numerical irreducible decomposition" of the solution set of a polynomial system, it is first necessary to understand the irreducible decomposition, which in turn relies on an understanding of the primary decomposition of an ideal. Much of the following may be found in [9]. Consider the ring of polynomials $R = \mathbb{C}[z_1, z_2, \ldots, z_n]$. As a set, R consists of all polynomials in the variables $z_1, z_2, \ldots, z_n$ with complex coefficients.

Some ideals have very special properties. Thus, it is useful to try to write arbitrary ideals in terms of these special ideals. Here are some of the types of ideals that will play a role in the following chapters:

Definition 2.1. Let f and g be arbitrary elements in R. Let I be an ideal in R.

- I is *prime* if $fg \in I \implies f \in I$ or $g \in I$.
- I is *primary* if $fg \in I \implies f \in I$ or $g^m \in I$ for some m.
- I is *radical* if $f^m \in I \implies f \in I$.
- The *radical of* I is the set $\sqrt{I} = \{f \in R | f^m \in I \text{ for some } m\}$.
- I is a *radical ideal* if $I = \sqrt{I}$.

It should be noted that the radical of a primary ideal is a prime ideal. If I is a primary ideal and if $\mathfrak{p} = \sqrt{I}$ then I is said to be $\mathfrak{p}$-primary. As discussed above, there is a dictionary between ideals and algebraic sets. The following definition helps to solidify that idea.

Definition 2.2. Let U be an ideal in $\mathbb{C}$, let U' be a homogeneous ideal in $\mathbb{C}[z_1, z_2, \ldots, z_{n+1}]$, let $T \subset \mathbb{C}^n$, and let $T' \subset \mathbb{P}^n$.

- Define $V(U) := \{z \in \mathbb{C}^n | f(z) = 0 \text{ for every } f \in U\}$.

- Define $V_h(U') := \{z \in \mathbb{P}^n | f(z) = 0 \text{ for every } f \in U'\}$.
- Define $I(T) := \{f \in \mathbb{C}[z_1, z_2, \ldots, z_n] | f(z) = 0 \text{ for every } z \in T\}$.
- Define $I_h(T') := \{f \in \mathbb{C}[z_1, z_2, \ldots, z_{n+1}] | f \text{ is homogeneous and } f(z) = 0 \text{ for every } z \in T'\}$

These operations associate geometric objects to algebraic objects and algebraic objects to geometric objects. In particular, the operation $I(-)$ (resp. $I_h(-)$) takes as input a subset of $\mathbb{C}^{n+1}$ (resp. $\mathbb{P}^n$) and produces a subset of a polynomial ring. The operation $V(-)$ (resp. $V_h(-)$) takes as input a subset of a polynomial ring and produces a subset of $\mathbb{C}^{n+1}$ (resp. $\mathbb{P}^n$). Although the term "algebraic set" is defined above, a technical definition follows.

Definition 2.3. A subset $T \subset \mathbb{C}^n$ is called an *affine algebraic set* if $T = V(U)$ for some subset $U \subset \mathbb{C}[z_1, z_2, \ldots, z_n]$. A subset $T \subset \mathbb{P}^n$ is called a *projective algebraic set* if $T = V_h(U)$ for some subset of homogeneous polynomials $U \subset \mathbb{C}[z_1, z_2, \ldots, z_{n+1}]$.

Here are some of the basic properties of the tools introduced thus far:

Proposition 2.1. *For any subset $T \subset \mathbb{C}^{n+1}$ (resp. $T' \subset \mathbb{P}^n$) and for any subset $U \subset \mathbb{C}[z_1, z_2, \ldots, z_{n+1}]$,*

(i) $I(T)$ is a radical ideal. $I_h(T')$ is a homogeneous radical ideal.

(ii) $T \subseteq V(I(T))$ with equality if and only if T is an affine algebraic set. $T' \subseteq V_h(I_h(T'))$ with equality if and only if T' is a projective algebraic set.

(iii) $U \subseteq I(V(U))$ with equality if and only if U is a radical ideal. $U \subseteq I_h(V_h(U))$ with equality if and only if U is a homogeneous radical ideal.

(iv) If U is an ideal then $I(V(U)) = \sqrt{U}$. If U is a homogeneous ideal then $I_h(V_h(U)) = \sqrt{U}$.

The word *algebraic set* will be used to refer to both affine algebraic sets and projective algebraic sets. An algebraic set, V, is said to be *reducible* if it is possible to write $V = V_1 \cup V_2$ with V_1, V_2 algebraic sets and with $V \neq V_1$ and with $V \neq V_2$. Algebraic sets which are not reducible are called *irreducible*. Irreducible algebraic sets are sometimes called *varieties*, although some authors choose to refer to any algebraic set as a variety. Algebraic sets, ideals and radical ideals have nice decomposition properties and relationships that are summarized as follows:

Proposition 2.2 (Decomposition properties)**.** *The following are several important properties related to the decomposition of ideals and algebraic sets.*

- *Every algebraic set can be written uniquely as the union of a finite number of irreducible algebraic sets, none of which are a subset of another.*
- *Every (homogeneous) radical ideal can be written uniquely as the intersection of a finite number of (homogeneous) prime ideals none of which are contained in another.*
- *Every (homogeneous) ideal can be written as the intersection of a finite number of (homogeneous) primary ideals.*
- *If V is an affine algebraic set then $I(V)$ is a prime ideal. If V is a projective algebreic set then $I_h(V)$ is a homogeneous prime ideal.*
- *If I is a primary ideal then $V(I)$ is an affine algebraic set. If I is a homogeneous primary ideal then $V_h(I)$ is a projective algebraic set.*

- *If* $I = I_1 \cap I_2$ *then* $V(I) = V(I_1) \cup V(I_2)$. *If* $I = I_1 \cap I_2$ *with* I, I_1, I_2 *homogeneous then* $V_h(I) = V_h(I_1) \cup V_h(I_2)$.

In the proposition above, it is seen that every ideal can be written as the intersection of a finite number of primary ideals. Something much stronger can be said.

Definition 2.4. Let I be an ideal and let $I = I_1 \cap I_2 \cap \cdots \cap I_t$ be a primary decomposition of I. Suppose I_i is $\mathfrak{p}_i$-primary for each i. The primary decomposition is called *reduced* if

(i) $\mathfrak{p}_i \neq \mathfrak{p}_j$ whenever $i \neq j$.

(ii) For each i, $\cap_{j \neq i} I_j \nsubseteq I_i$.

The prime ideals $\mathfrak{p}_1, \mathfrak{p}_2, \ldots, \mathfrak{p}_t$ are called *associated primes*. As defined, they depend on the choice of primary decomposition. However, the following proposition simplifies the situation and demonstrates that the set of associated primes play a more central role than it first appears.

Proposition 2.3 (Primary decompositions)**.** *The following facts are related to the primary decomposition of an ideal.*

- *Every ideal has a reduced primary decomposition.*
- *Any reduced primary decomposition of a given ideal has the same set of associated primes.*
- *A radical ideal has a unique reduced primary decomposition.*
- *The associated primes of the radical of an ideal are a subset of the associated primes of the ideal.*

As a consequence of the proposition above, it makes sense to talk about the associated primes of an ideal (rather than the associated primes of a primary decomposition of an ideal). The associated primes of an ideal that are not associated primes of the radical of the ideal are called *embedded primes*. It should be emphasized that the proposition does not claim there is a *unique* reduced primary decomposition for a general ideal, it only claims that the associated primes are uniquely determined. To each ideal, I, one may associate a degree and a dimension, denoted $deg(I)$ and $dim(I)$ respectively. The degree and dimension of an ideal can be defined in terms of the *Hilbert Polynomial* of R/I. Precise defintions of these terms may be found in [23], [29], and [16]. The degree function allows one to define the multiplicity of a primary ideal.

Given the primary decomposition of an ideal, there is an analogous decomposition of the algebraic set associated to that ideal, known as the *irreducible decomposition*. Indeed, since the algebraic sets associated to prime ideals are irreducible, a primary decomposition of an ideal gives rise to a unique irreducible decomposition of the algebraic set associated to the ideal. This is the case since the radical of a primary is prime and, given an ideal I, the *associated primes* of I (i.e., those primes that appear as the radicals of primary ideals in a primary decomposition of I) are independent of the choice of primary decomposition.

Definition 2.5. A *witness point set* of an irreducible algebraic set Y is a set of $deg(Y)$ points on Y, found by slicing Y with a linear space of complementary dimension. These points are known as *witness points*. A *numerical irreducible decomposition* for an algebraic set Z is then a set of points W that decomposes into subsets by dimension and again by component, just as Z does, so that W consists of one witness point set for every irreducible component of Z. In particular, if

$Z = \bigcup Z_{ij}$ with i running over all dimensions and j as the component number within each dimension, then $W = \bigcup W_{ij}$ with $W_{ij} \subset Z_{ij}$ a witness point set for all i and j.

2.3.2 Methods

The key algorithms of numerical algebraic geometry are those that are used to find a numerical irreducible decomposition of the solution set of a given polynomial system. The goal of this section is to describe those methods very briefly. For more details, please refer to [56].

It is known that almost every general linear space of dimension k in $\mathbb{C}^n$ will intersect a given irreducible algebraic set of dimension d if $d+k \geq n$. In particular, this intersection will be zero-dimensional (i.e., just points) if $d+k = n$. By "almost every", it is meant that there is a Zariski open set U of some vector space V of all linear spaces of dimension k such that every point of U corresponds to a linear space having the desired property. Equivalently, the set of all points of V at which the property does not hold is a proper algebraic set of V. This is what is meant by a *probability one* occurrence. Almost all (no pun intended) algorithms of numerical algebraic geometry succeed with probability one.

To produce a witness set (i.e., numerical irreducible decomposition) for a polynomial system, one must first produce a *witness superset*. A witness superset is simply a witness set that could have extra points in each W_{ij} that actually lie on components of dimension greater than i. Once a witness superset W has been produced, there are methods to remove the "junk" and thereby reduce W to a witness set.

The algorithm for producing a witness superset begins at codimension one.

One may add linear polynomials with randomly chosen coefficients to the system in order to "slice" the algebraic set with a one-dimensional general linear space. With probability one, this results in a system having as isolated solutions points lying on the codimension one components of the original system. This system is easily solved with homotopy continuation methods. One may then proceed to codimension two and so on until all dimensions (down to zero) have been addressed. The result is a witness point superset (not a witness point set) since this slicing mechanism will produce points on higher-dimensional components for all but the highest-dimensional components. Also, the points are only broken up by the dimension of the component upon which they are presumed to reside. Pure-dimensional decomposition needs to be carried out as well.

Once a witness point superset has been produced, it is easy to remove the "junk," i.e., the points lying on higher-dimensional components. Indeed, to check whether a point x presumed to lie on a component of dimension i actually does so, one may produce for each known higher-dimensional witness point set W a homotopy beginning with the slice used to find W and ending with a general slice through x. After tracking all points of W to the general slice through x, if any endpoint equals x, then x lies on the same higher-dimensional component as W and may therefore be deemed "junk." If no such match is found after considering all such W, x is known to lie on a component of dimension i.

As for pure-dimensional decomposition, one may use monodromy to move all unclassified points in a given dimension around a loop and back to themselves. Any points that swap places are known to lie on the same irreducible component since the set of regular points of an irreducible component is connected. Finally, there is a special test for confirming whether a suspected component is actually

complete. This test relies on linear traces, which, with monodromy and many more sophisticated mechanisms, are described in full detail in [56]. Since these techniques require even more theory than has been described above and since they do not play a role in the subsequent chapters, no further details will be provided presently.

CHAPTER 3

ADAPTIVE PRECISION IN HOMOTOPY CONTINUATION

As described in the previous chapter, certain numerical difficulties may arise while attempting to track paths using predictor/corrector methods. In particular, when near a singularity, the condition number of the Jacobian of the system becomes large enough to overwhelm most or all of the available precision. In this chapter, a novel numerical method to address such numerical problems is presented. It should be noted that, while the technique of this chapter can theoretically overcome near-singularities, it will still result in failure in the presence of true singularities (which only occur at $t = 0$, with probability one). A more sophisticated discussion of the underlying theory and a more detailed version of the algorithm may be found in [10]. The goal of this chapter is to convey the basic idea of the new adaptive precision technique of [10]. The final two sections of this chapter provide a translation from the notation of this chapter to that of [10], a statement of the criteria of that paper, and a few examples showing the usefulness of this algorithm (from the same paper).

3.1 Background

Suppose one tries to approximate the solutions of a polynomial system $f : \mathbb{C}^n \to \mathbb{C}^n$ via homotopy continuation techniques, using the homotopy $H : \mathbb{C}^n \times$

$\mathbb{C} \to \mathbb{C}^n$ with $H(z,0) \equiv f(z)$. It is known (see [43], [6], [38], or [56]) that the solutions of f lie on paths that pass through $t = 1$ so that it is possible to track these paths from $t = 1$ to $t = 0$. The predictor/corrector methods of homotopy continuation generally perform well away from singularities. However, in the presence of singularities, the linear algebra behind these methods breaks down, leading to path failure and, ultimately, an incomplete description of the solution set of f.

By definition, when a singularity is encountered along a path, the Jacobian of the homotopy function, $J := \left(\frac{\partial H_i}{\partial x_j}\right)$, is singular, so that the condition number of J is infinite. As a path moves closer to a singularity, the condition number of the Jacobian swells. As discussed in the previous chapter, this ill-conditioning overwhelms the amount of available precision, making it impossible to attain the desired level of accuracy when inverting the Jacobian.

There are two obvious ways of overcoming ill-conditioning in this situation. One way is to somehow rewrite the polynomials or add extra polynomials so that the singularity of the original system is no longer a singularity. Deflation is one such method (see [37]). The other basic solution to the problem of ill-conditioning is to increase the level of precision being used. The standard way of increasing precision in the homotopy continuation setting is to allow paths suffering from ill-conditioning to fail at low precision and then follow them again at a higher level of precision. One typically sets a maximum number of reruns, the violation of which triggers a declaration of total path failure. Figure 3.1 gives a flowchart depiction of this method.

The point of the new step-adaptive precision path tracking technique of this chapter is that much time and effort may be saved by changing to higher precision

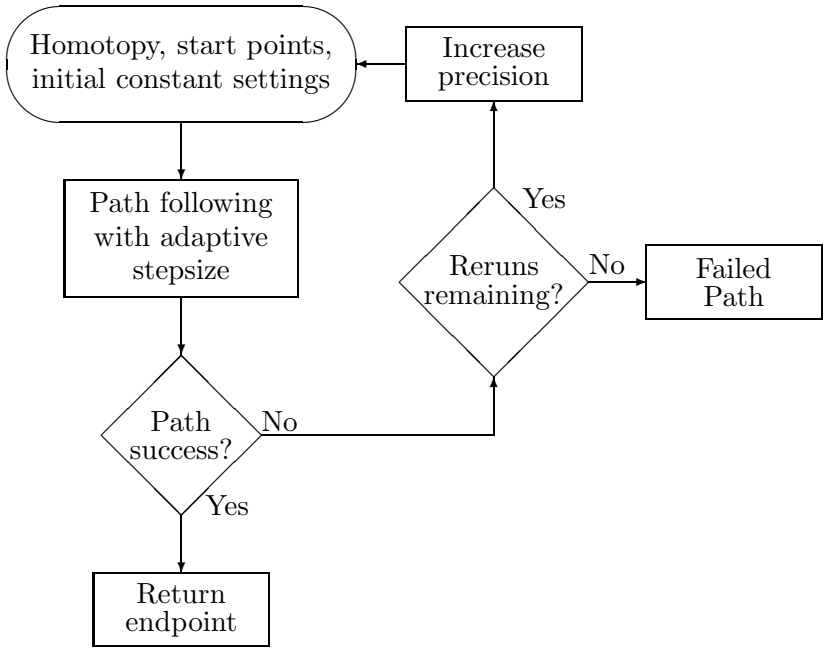

Figure 3.1. Path-adaptive precision path tracking

on the fly and only when necessary. Suppose that, for some path, the level of precision would need to be increased by 10 digits 10 times in order to have success when tracking it. Using path-adaptive precision path tracking, the same path would be tracked 11 times in its entirety before success. Using step-adaptive precision path tracking, the path would only be run once, and higher precision would be used only when necessary.

The key difficulty with this method is deciding when it is necessary to increase precision. Naturally, one must observe the behavior of the predictor/corrector methods and, when certain conditions are observed, the precision should be increased. The development of such criteria is the subject of the following section. Recall from the previous chapter that when solving the matrix equation $A \cdot x = b$

for x, one may expect that the number of accurate digits of x is approximately the number of digits being used to represent floating point numbers minus the exponent of the condition number of A. This is the key fact for the following section.

The following is a list of the notation used in the next two sections. The notation of [10] is introduced later in this chapter both to state more sophisticated versions of the criteria and to serve as a bridge to that paper.

- $\text{cond}(A)$ is the condition number of the nonsingular matrix A, and **CN** := $\log_{10}(\text{cond}(A))$ is the exponent of the condition number;

- **PREC** designates the number of decimal digits used for the representation of floating point numbers;

- **PredictorStepSize** is the predictor residual, i.e., the size of a predictor step, and **PSS** := $-\log_{10}$ **PredictorStepSize**;

- **CorrectorResidual** is the corrector residual, i.e., the size of a corrector step, and **CR** := $-\log_{10}$ **CorrectorResidual**;

- **MaxIts** is the maximum number of corrector steps allowed following a predictor step;

- **TrackTol** represents the desired accuracy while tracking, **FinalTol** is the desired final accuracy, and **TOL** := $-\log_{10}$ **TrackTol** is the desired number of correct decimal digits while tracking; and

- **StepLength** is the current size of a step while **MinStepLength** is the level of **StepLength** below which a path is declared a failure.

3.2 Criteria indicating the need for higher precision

The technique of adaptive precision path tracking is similar in many ways to fixed precision path tracking. The main differences are that there is more bookkeeping and computation and, using the extra information, the precision of the pertinent data is changed. One may use the same starting values for **StepLength**, **TrackTol**, and the other heuristic constants as with the fixed precision method.

To develop the adaptive precision technique, consider the linear algebra needed in a typical predictor-corrector step. Suppose an approximate solution, x_0, is known at $t = t_0$ and that the goal of the step is to produce an approximate solution x_1 at $t = t_1 := t_0 - \mathbf{StepLength}$, with the requirement that the corrector residual at the new point is less than **TrackTol** in absolute value. From the convergence criterion of the corrector, the first **TOL** digits of x_0 are correct. Enough digits of precision are needed to represent the first **TOL** digits of x, giving the first adaptive precision rule, or relation:

$$\mathbf{PREC} - \mathbf{TOL} > \mathbf{SD}_1. \tag{3.1}$$

For each of the relations derived in this section, the exact inequality is necessary. However, for safety in practice, one ought to include one or more *safety digits*. Using a large number of safety digits unnecessarily triggers a precision increase, thereby increasing run time and memory usage. However, safety digits are necessary due to the approximation or sporadic observation of the related quantities. Typical settings of $\mathbf{SD}_1$ are 1 or 2. If (3.1) is violated, one may choose to increase the precision or relax the tracking tolerance. However, relaxing the tracking tolerance can lead to pathcrossing, so the alternative is a better choice.

Call the approximate solution after the predictor $\tilde{x}_1$, so that $\tilde{x}_1 := x_0 +$

PredictorStepSize . As discussed in Chapter 2, only the first $\mathbf{PREC} - \mathbf{CN}$ nonzero digits of **PredictorStepSize** are accurate, where **CN** represents the log of the condition number of the Jacobian. Since **PredictorStepSize** begins with a number of zeroes after the decimal point (which it often will, assuming reasonable scaling and a small enough **StepLength**), the first $\mathbf{PSS} + (\mathbf{PREC} - \mathbf{CN})$ digits of **PredictorStepSize** may be trusted. Naturally, at least the first nonzero digit of **PredictorStepSize** must be accurate, since that much is necessary to guarantee that the prediction is in a reasonable direction. If no digits of **PredictorStepSize** are trustworthy, there is no point in taking the step. The only thing saving the tracking procedure in such an event is the corrector, so for a poorly behaved path, **StepLength** is certain to drop below **MinStepLength** if the precision is not increased. This leads to the rule

$$\mathbf{PREC} - \mathbf{CN} > \mathbf{SD}_2. \tag{3.2}$$

In practice, the computation of the exact condition number is too expensive, so estimates are used instead. There exist several techniques for estimating the condition number of a matrix, but the method found in [63] is typically correct up to a factor of ten. Setting $\mathbf{SD}_2$ to a number between 2 and 4 is therefore reasonable.

Following the predictor step comes a sequence of corrector steps. **CorrectorResidual** is a measurement of the accuracy of the previous estimate, so the value of **CR** from the first corrector step indicates approximately how many digits are correct after the predictor step. Given the value of **CN** at that estimate, each corrector step may add at most $\mathbf{PREC} - \mathbf{CN}$ additional correct digits to the estimate. One may be concerned by the fact that the condition number of the

Jacobian will be so unstable near a singularity that the approximation of **CN** obtained before the first corrector step is inaccurate after a few corrector steps. To account for this and add security to the algorithm, **CN** should be estimated prior to each corrector step. This will, of course, significantly increase the amount of CPU time needed for each path. Letting **ItsLeft** denote the number of corrector steps remaining after some number of corrector steps, the resulting rule is

$$\mathbf{CR} + \mathbf{ItsLeft} * (\mathbf{PREC} - \mathbf{CN}) - \mathbf{TOL} > \mathbf{SD}_3. \tag{3.3}$$

As with (3.2), an estimate of the condition number is used, so $\mathbf{SD}_3$ is generally set to a number between 2 and 4.

So, when path tracking with adaptive precision, the inequalities (3.1), (3.2), and (3.3) should be monitored. The violation of any of these rules results in a precision increase. It is sensible to check from time to time, via the formulas discussed above, whether the precision may be lowered when using multiprecision away from $t = 0$, so as to avoid wasting CPU time using many digits when fewer will suffice. Of course, theoretically, singularities will not be encountered except at $t = 0$ with probability one. However, in the case of a near singularity in $(0, 1]$, it would be unfortunate to increase the precision, thereby slowing down the tracker but never allowing the tracker to accelerate after moving beyond the singularity.

It should be noted that for adaptive precision path tracking to be possible, the input data must be available at any given level of precision. At first, this seems to remove from consideration many problems from the catalog of polynomial systems that necessitate numerical techniques such as those being discussed. However, in most cases, coefficients and parameters are known exactly, and as long as the implementation of the algorithm can handle holomorphic coefficients and account

for any algebraic relations between the coefficients, the polynomial system may be approximated to any level of precision needed.

It should be noted that this algorithm is guaranteed to terminate, at worst because more than the allowed number of predictor-corrector steps have been taken. One might be concerned about the behavior of the algorithm if the estimate at some $t > 0$ is very good, resulting in a corrector residual very near 0. However, this is no problem since the $\mathbf{CR} > \mathbf{TOL}$ step termination check (which indicates that the number of correct digits exceeds the desired number of correct digits) is absolute rather than relative in nature, so the corrector will simply not attempt a correction in such a scenario.

Endgames, such as the fractional power series endgame discussed in Chapter 2, often converge to regular and even some singular solutions without needing precision increases. However, endgames cannot account for near-singularities away from $t = 0$. Also, given a singular solution with high enough multiplicity, higher than regular precision will be needed even when using endgames. More sophisticated endgames will generally converge from larger values of t, thereby allowing for lower final precision.

Conversely, one might suspect that arbitrary precision may replace the need for endgames. Although this might be true in theory, it is definitely not the case in practice. When path tracking near a singular solution at $t = 0$ using fixed precision, the tracking algorithm will fail for the reasons described above. By using higher precision, the tracking procedure may advance towards $t = 0$, but it will again fail since the condition number of the Jacobian will continue to grow. In fact, the condition number at $t = 0$ will be infinite for a singular solution, so it is impossible for the basic tracking procedure to converge to a tight final tolerance

without unrealistically high levels of precision. Therefore, multiprecision cannot alleviate the need for endgames.

The best course of action is to use a sophisticated endgame such as the fractional power series endgame along with adaptive precision path tracking. The cost associated to such endgames is minimal for regular endpoints, and the added cost in the case of singular endpoints is necessary since simpler endgames may not converge at all.

Exact versions of the criteria discussed loosely above may be found in the next section. At that point, a flowchart representation of the algorithm is given. The final section of the chapter consists of three examples of this algorithm in practice, as discussed in [10].

3.3 Translation to new notation and exact criteria

As mentioned in previous sections, the criteria of the preceding section were carefully developed and analyzed in [10]. In order to discuss the examples of that paper, it is necessary to have the exact criteria as they appear in that paper. To shorten the length of the criteria and to serve as a bridge between the discussion above and [10], the notation of that paper is introduced in this section and a table relating the two forms of notation is provided.

3.3.1 New notation

Let J deonte the Jacobian of the homotopy. Precision is denoted by $u = 10^{-P}$. There is of course error associated with both the evaluation and the inversion of the Jacobian. The two types of errors may be accounted for together, with the error bounded from above by $\mathcal{E}u\|J(z)\| + \phi(z, u)$. As described in more detail

in [10], $\mathcal{E} > 1$ captures the roundoff error and its amplification resulting from the linear solve. The function ϕ covers the evaluation errors that occur if $J(z)$ is nonzero. Also, $\phi = \Phi u$ for some function Φ which can be bounded in the polynomial systems settings, as described below.

The safety digit settings are now known as σ_1 and σ_2. There are only two because $\mathbf{SD}_2$ and $\mathbf{SD}_3$ actually appear for the same reason and can taken to be equal (and renamed σ_1). The logarithm of the final tolerance, **TOL** is important, so it is given the new name τ. The symbol d stands for the Newton residual while N denotes the maximum number of Newton iterations allowed in one step. Also, v is the notation for the current approximate solution. Finally, $\psi = \Psi u$ denotes the error in evaluating the homotopy function. Table 3.1 indicates the translation from the notation above to the set of notation used below.

3.3.2 Exact criteria

By carefully analyzing Newton's method (as in [60]) and keeping track of all error bounds, one may convert the adaptive precision criteria (3.1), (3.2), and (3.3) above to the criteria of this section. Although this careful analysis is omitted from this thesis, it may be found in [10]. (3.2) becomes

$$P > \sigma_1 + \log_{10}[\|J^{-1}\|(\mathcal{E}\|J\| + \Phi)]. \qquad (\mathbf{A})$$

Letting i denote the number of Newton iterations that have already been carried out, (3.3) becomes the following:

TABLE 3.1

CONVERSION BETWEEN SETS OF NOTATION

New symbol	Old symbol
J	N/A
u	$10^{-\mathbf{PREC}}$
P	**PREC**
$\mathcal{E}$	N/A
ϕ, Φ	N/A
ψ, Ψ	N/A
σ_1	$\mathbf{SD}_2$, $\mathbf{SD}_3$
σ_2	$\mathbf{SD}_1$
τ	**TOL**
d	$10^{-\mathbf{CR}}$
N	**MaxIts**

$$P > \sigma_1 + \log_{10}\left(\|J^{-1}\|[(2+\mathcal{E})\|J\| + \Phi] + 1\right) + (\tau + \log_{10}\|d\|)/(N-i). \qquad (\mathbf{B})$$

Finally, (3.1) may be restated as follows:

$$P > \sigma_2 + \tau + \log_{10}(\|J^{-1}\|\Psi + \|v\|). \qquad (\mathbf{C})$$

As discussed in [10], it is very easy to place bounds on Φ and Ψ in the case

of homogeneous polynomials. Letting D denote the degree of a homogeneous polynomial f, and letting c_i for i from 1 to M be the set of nonzero coefficients of f (for some M), Ψ is certainly bounded by $D\sum_{i=1}^{M}|c_i|$. This is because there are no more than D multiplications per term, and the sum bounds the error coming from the addition of the terms. Similarly, Φ may be bounded by $D(D-1)\sum_{i=1}^{M}|c_i|$. Also, according to statements made in [20], n^2 is an ample upper bound for most $n \times n$ polynomial systems.

For more discussion regarding the theory behind and development of the criteria and bounds of this section, please see [10]. Given the exact criteria of this section, it is possible to give a flowchart description of the step-adaptive precision algorithm of this chapter. That flowchart is given in Figure 3.2.

3.4 Examples

As is evident from Figure 3.2, the implementation of adaptive precision homotopy continuation given existing path-tracking software amounts to adding several conditional statements based on conditions A, B, and C and a mechanism for increasing and decreasing precision when necessary. Section 3.4.1 consists of a few details regarding the implementation of this method for polynomial systems. In particular, details of the implementation of adaptive precision homotopy continuation in the Bertini software package will be described briefly. The remainder of the chapter consists of a discussion of the effects of applying adaptive precision to a few representative polynomial systems. Please note that the rest of this chapter closely follows the sections of [10] regarding computational results.

3.4.1 Implementation details

To implemenent adaptive precision path tracking, one must obviously be able to compute the various quantities appearing in conditions A, B, and C. Most quantities appearing in conditions A, B, and C, such as norms of matrices, are easily computed with Bertini and will not be discussed further here. Possible exceptions include $\mathcal{E}$, ϕ, and ψ. As noted above, $\mathcal{E}$ is generally bounded by n^2 for an $n \times n$ system. Since pathological examples exceed this bound, a decision regarding the bound placed on $\mathcal{E}$ must be made at implementation time. The n^2 bound is hard-coded into Bertini.

Bounds on ϕ and ψ that depend on the quantities $\sum_{i=1}^{M} |c_i|$ and D are also described briefly above. Although a good mechanism for bounding ϕ and ψ for polynomials given in straight-line program format is given in [10], Bertini currently takes a simpler approach. It is left to the user to provide a bound for $\sum_{i \in \mathcal{I}} |c_i|$ as well as a bound on D. The obvious method of computing exactly $\sum_{i \in \mathcal{I}} |c_i|$ is not difficult for polynomials in standard form, but Bertini accepts polynomials in non-standard forms, e.g., factored polynomials, complicating the detection of the coefficients. There is no pressure on the user to specify particularly tight bounds, since ϕ and ψ typically play a minor role in conditions A, B, and C and any overestimate in the bounds is equivalent to adding extra safety digits.

The only other difficulty in implementing adaptive precision path tracking is in changing the precision of variables on the fly. Details of how Bertini handles this may be found in Chapter 6. Since the use of adaptive precision variables is highly implementation-specific, no other details are described here.

3.4.2 Behavior of adaptive precision near a singularity

Consider the polynomial system of Griewank and Osborne [27],

$$f = \begin{bmatrix} \frac{29}{16}z_1^3 - 2z_1z_2 \\ z_2 - z_1^2 \end{bmatrix}.$$

This system is interesting because Newton's method diverges for any initial guess near the triple root at the origin. A homotopy of the form

$$h(z,t) = tg(z) + (1-t)f(z),$$

where $g(z)$ is the following start system, constructed as described in [56]:

$$\begin{aligned} g1 = \; & ((-0.74924187 + 0.13780686i)z_2 + (+0.18480353 - 0.41277609i)H_2)* \\ & ((-0.75689854 - 0.14979830i)z_1 + (-0.85948442 + 0.60841378i)H_1)* \\ & ((+0.63572306 - 0.62817501i)z_1 + (-0.23366512 - 0.46870314i)H_1)* \\ & ((+0.86102153 + 0.27872286i)z_1 + (-0.29470257 + 0.33646578i)H_1), \end{aligned}$$

$$\begin{aligned} g2 = \; & ((+0.35642681 + 0.94511728i)z_2 + (0.61051543 + 0.76031375i)H_2)* \\ & ((-0.84353895 + 0.93981958i)z_1 + (0.57266034 + 0.80575085i)H_1)* \\ & ((-0.13349728 - 0.51170231i)z_1 + (0.42999170 + 0.98290700i)H_1), \end{aligned}$$

with H_1 and H_2 the extra variables added to the variable groups $\{z_1\}$ and $\{z_2\}$, respectively. In the process of homogenizing, two linear polynomials (one per variable group) are added to the system and do not depend on t. In this case,

those polynomials are:

$$(-0.42423834 + 0.84693089i)z_1 + H_1 - 0.71988539 + 0.59651665i,$$
$$(+0.30408917 + 0.78336869i)z_2 + H_2 + 0.35005211 - 0.52159537i.$$

This homotopy has three paths converging to the origin as $t \to 0$. These paths remain nonsingular for $t \in (0, 1]$, so it is interesting to see how the adaptive precision algorithm behaves as t approaches zero.

Using the prescription of two equations from [10] to bound errors, it is reasonable to take $D = 3$ and $\sum_{i \in \mathcal{I}} |c_i| \approx 4$, so $\Psi = 12$ and $\Phi = 24$. (This is actually quite conservative, because a more realistic bound would be $\Psi = 12||z||$ and $||z||$ is approaching zero.) The safety digits are set to $\sigma_1 = \sigma_2 = 0$, as these serve only to force precision to increase at a slightly larger value of t.

The desired accuracy was set to $\tau = 8$ for $t \in [0.1, 1]$ and increased it to $\tau = 12$ digits thereafter. To watch the behavior of the algorithm for very small values of t, the usual stopping criterion was turned off and the path was allowed to run to $t = 10^{-30}$. That is, in the flowchart of Figure 3.3.2, after a successful correction step, the algorithm was modified to always loop back to step ahead in t until $t = 10^{-30}$. Since for small t and for $||g|| \approx 1$, the homotopy path has $||z|| \approx |t|^{1/3}$, all three roots heading to the origin were within 10^{-10} of the origin at the end of tracking. With an accurate predictor, such as a fractional power series endgame [47], the solution at $t = 0$ can be computed to an accuracy of 10^{-12} using only double precision, but the purpose of this example is to show that the adaptive precision tracking algorithm succeeds beyond the point where double precision would fail.

The result is shown in Figure 3.4.2. The right-hand side of rule C is shown, as

this rule determines the increases in precision for this problem. The jump in this value at $t = 0.1$ is due to our prescribed increase in τ at that point. Since the path is heading to the origin, if the less conservative error estimate of $\psi(z, u) = 12||z||u$ were used, increases in precision would be delayed somewhat, as rule C would be shifted down by approximately $\log_{10} ||z|| \approx (1/3) \log_{10} t$.

In cases where multiple roots are clustered close together or even approaching the same endpoint, as in this example, the path tracking tolerance must be kept substantially smaller than the distance between roots to avoid jumping between paths. Rather than prescribing τ versus t, as in this example, it would be preferable to automatically adjust τ as needed. This is postponed this for future research, but note that the rules given here for adjusting precision should still be effective when τ is adjusted automatically.

3.4.3 Behavior of adaptive precision under tighter tolerances

To illustrate the effect of tightening the tracking tolerance (i.e., increasing τ) on adaptive precision path tracking, we will consider a polynomial system coming from chemistry. To determine chemical equilibria, one may pass from a set of reaction and conservation equations to polynomials, as described in [56], [43]. One such polynomial system, discussed in [47], is the following:

$$f = \begin{bmatrix} 14z_1^2 + 6z_1z_2 + 5z_1 - 72z_2^2 - 18z_2 - 850z_3 + 0.000000002 \\ 0.5z_1z_2^2 + 0.01z_1z_2 + 0.13z_2^2 + 0.04z_2 - 40000 \\ 0.03z_1z_3 + 0.04z_3 - 850 \end{bmatrix}.$$

As in the previous example, this system was homogenized, although this time with only one variable group, so there is just a single homogenizing coordinate.

TABLE 3.2

SOLUTIONS OF THE CHEMICAL SYSTEM

H_1	2.15811678208e−03 − 2.32076062821e−03*i	7.75265879929e−03 + 5.61530748382e−03*i
z_1	1.21933862567e−01 + 4.02115643024e−01*i	1.26177608967e−01 − 1.26295173168e+00*i
z_2	-2.29688938707e−02 − 7.44021609426e−02*i	-2.45549175888e−02 + 2.33918398619e−01*i
z_3	-6.62622511387e−01 − 1.55538216233e−00*i	-1.87267506123e+00 + 8.48441541195e−01*i
H_1	-2.54171295092e−03 + 1.43777404446e−03*i	6.32152240723e−03 + 1.64022773970e−03*i
z_1	-3.34864497185e−01 + 1.89423218369e−01*i	-2.20546409488e−01 − 7.88700342178e−01*i
z_2	6.25969991088e−02 − 3.54093238711e−02*i	-4.39498685300e−02 − 1.59117743373e−01*i
z_3	-5.41138529778e−01 + 3.06106507778e−01*i	-1.03394901752e+00 + 1.06381058693e+00*i
H_1	-2.64917471213e−05 + 5.83090377404e−06*i	2.72428081371e−03 − 2.22348561510e−03*i
z_1	1.23749450722e−05 − 2.72846568805e−06*i	6.93769305944e−02 + 4.35467392206e−01*i
z_2	-3.62126436085e−03 − 1.64631735533e−02*i	1.42630599439e−02 + 8.77317115664e−02*i
z_3	-8.66534113884e−01 + 1.90904229879e−01*i	-7.45011925697e−01 − 2.88085192442e−01*i
H_1	-2.37392215058e−03 + 9.07039153390e−04*i	-2.73688783636e−05 + 4.95136100653e−06*i
z_1	-2.96180844307e−01 + 1.13166145980e−01*i	1.27865341710e−05 − 2.30844830185e−06*i
z_2	-6.00257143378e−02 + 2.29349024594e−02*i	3.07919899933e−03 + 1.70073434711e−02*i
z_3	-5.33404946327e−01 + 2.03805834055e−01*i	-8.95294964314e−01 + 1.61788761616e−01*i
H_1	-8.36782294691e−26 − 1.06193180980e−26*i	1.73680568470e−24 + 2.20412066255e−25*i
z_1	1.11570976734e−25 + 1.41590902837e−26*i	-2.31574091294e−24 − 2.93882755007e−25*i
z_2	3.64909247567e−14 + 1.01571832542e−12*i	-4.31048495249e−12 + 1.54859448543e−13*i
z_3	-8.80995273590e−01 + 1.76801830530e−01*i	-8.80995302370e−01 + 1.76801836789e−01*i
H_1	-1.73680568470e−24 − 2.20412066243e−25*i	1.73680568471e−24 + 2.20412066239e−25*i
z_1	2.31574091293e−24 + 2.93882754991e−25*i	-2.31574091295e−24 − 2.93882754986e−25*i
z_2	-1.54859448573e−13 − 4.31048495249e−12*i	4.31048495252e−12 − 1.54859448574e−13*i
z_3	-8.80995302371e−01 + 1.76801836796e−01*i	-8.80995302378e−01 + 1.76801836795e−01*i

Using a compatible total-degree linear product start system, that is, one whose polynomials have degrees 2, 3, and 2, respectively, results in a homotopy with 12 paths. The solution set of this system consists of eight regular solutions, two of which are real, and an order four solution at infinity. These solutions are given in Table 3.2, where H_1 is the extra variable used for homogenization. Each solution is listed to 12 digits, and the infinite solutions may be found in the last two rows. The number and location of the infinite solutions may of course vary according to the choice of homotopy; indeed, a linear product homotopy exists that has no divergent paths.

Due to the poor scaling of the problem, the error bounds were set to $\Psi = \Phi = 20,000$. As opposed to the previous example, the usual stopping criterion

described in Chapter 2 was employed. No endgame was used, though, since the use of endgames speeds convergence (which is generally desired!), resulting in the lack of any interesting data.

Tracking to a final tolerance of 10^{-8} using fixed regular (IEEE double) precision, all eight finite solutions were discovered, two of which had some coordinates of size 10^5. In tightening the tolerance to 10^{-12}, however, the linear algebra involved with path tracking broke down for the paths leading to the two large solutions. This resulted in step failures and, ultimately, path failure for those two paths. The paths leading to infinity also failed. By increasing precision to 96 bits, all eight regular solutions were again found. The solutions at infinity were not found to the final tolerance of 10^{-12} until the precision was increased to 192 bits.

Using adaptive precision with the number of safety digits set to 0, 1, or 2, the six finite solutions of moderate size required only one precision increase (to 64 bits from 53 bits). This increase occurred at the very end of the path. The two large finite solutions each needed 96 bits of precision (as expected), and the paths leading to infinite solutions all required precision increases all the way up to 192 bits for convergence. The use of the fractional power series endgame resulted in the same precision increases for all paths except those leading to infinity, which needed only 96 bits of precision rather than 192.

This use of more precision than necessary is one of the risks of using adaptive precision methods: higher than necessary levels of precision will sometimes be used, resulting in an unnecessary increase in computational cost. Compared to the cost of multiple reruns of paths or the consistent use of unnecessarily high precision for all paths, though, adaptive precision path tracking yields a savings. This is illustrated in the next example.

3.4.4 Overhead associated with adaptive precision

Given the discussion of the previous example, one might be concerned about the extra cost of adaptive precision path tracking. One may analyze how the adaptive precision method compares to fixed precision methods for a polynomial system that does not require higher than regular precision. In particular, from an inverse pole-placement problem, one may construct the following polynomial system [44], truncated to four digits for this run:

$$
\begin{aligned}
f_1 &= x_1^2 + x_2^2 - 1 \\
f_2 &= x_3^2 + x_4^2 - 1 \\
f_3 &= x_5^2 + x_6^2 - 1 \\
f_4 &= x_7^2 + x_8^2 - 1 \\
f_5 &= -0.2492x_1x_3 + 1.609x_1x_4 + 0.2794x_2x_3 + 1.435x_2x_4 + 0.4003x_5x_8 \\
&\quad -0.8005x_6x_7 + 0.07405x_1 - 0.08305x_2 - 0.3862x_3 - 0.7553x_4 + \\
&\quad 0.5042x_5 - 1.092x_6 + 0.4003x_8 + 0.04921 \\
f_6 &= 0.1250x_1x_3 - 0.6866x_1x_4 - 0.1192x_2x_3 - 0.7199x_2x_4 - 0.4324x_5x_7 \\
&\quad -0.8648x_6x_8 - 0.03716x_1 + 0.03544x_2 + 0.08538x_3 - 0.03925x_5 - \\
&\quad 0.4324x_7 + 0.01387 \\
f_7 &= -0.6356x_1x_3 - 0.1157x_1x_4 - 0.6664x_2x_3 + 0.1104x_2x_4 + 0.2907x_5x_7 \\
&\quad +1.259x_5x_8 - 0.6294x_6x_7 + 0.5814x_6x_8 + 0.1959x_1 - 1.228x_2 - \\
&\quad 0.07903x_4 + 0.02639x_5 - 0.05713x_6 - 1.163x_7 + 1.259x_8 + 2.163 \\
f_8 &= 1.489x_1x_3 + 0.2306x_1x_4 + 1.328x_2x_3 - 0.2586x_2x_4 + 1.165x_5x_7 - \\
&\quad 0.2691x_5x_8 + 0.5382x_6x_7 + 0.5826x_6x_8 - 0.2082x_1 + 2.687x_2 - \\
&\quad 0.6991x_3 + 0.3574x_4 + 1.250x_5 + 1.468x_6 + 1.165x_7 + 1.108x_8 \\
&\quad -0.6969
\end{aligned}
$$

The homotopy used to solve this system again makes use of a 2-homogenized form of this system, as with the first two examples. In this case, the variable sets are $\{x_1, x_2, x_5, x_6\}$ and $\{x_3, x_4, x_7, x_8\}$. The 2-homogenized system has 96 paths, one half of which end at regular finite solutions, the other half of which go to infinity. This example was run using error bounds $\Psi = \Phi = 200$. The paths were tracked to a tolerance of 10^{-4} until $t = 0.1$ and to a final tolerance of 10^{-6}. As opposed to the previous example, condition B usually caused increases in precision, likely due to the size of the system (10×10 after 2-homogenization, leading to a bound of 100 on $\mathcal{E}$).

Table 3.3 shows how adaptive precision with various safety digit settings compares to fixed precision path tracking. The final column is the number of paths requiring higher than regular precision in each case. Of course, exact timings are platform- and implementation-dependent, but the fourth column of Table 3 shows that adaptive precision tracking behaves as expected in Bertini; it is slower than regular precision tracking but faster than multiple precision tracking. As mentioned in Section 3.4.1, using MPFR greatly slows down the computations, even more than is expected given that more digits are being used.

In comparing the first two rows of Table 3.3, one might conclude that the extra computational cost associated to checking conditions A, B, and C is very great. However, of the 19.98 total seconds needed to run all 96 paths with the number of safety digits set to 0, 15.18 seconds were spent on the 32 paths that went to higher precision. In other words, 76% of the time was spent on 33% of the paths, due to the extra cost of using MPFR described above. Ignoring those 32 paths, the paths not making use of higher precision ran at a rate of 13.33 paths per second,

TABLE 3.3

EFFECT OF SAFETY DIGIT SETTINGS ON COMPUTATION TIME

Precision type	σ_1	σ_2	Paths/second	# high precision paths
fixed, regular	N/A	N/A	13.71	0
adaptive	0	0	4.80	32
adaptive	1	1	4.66	44
adaptive	2	2	4.22	56
adaptive	3	3	3.53	57
fixed, 96 bits	N/A	N/A	1.31	96

which is only slightly less than the rate for the regular precision run of the first row.

The 32 paths using higher precision ran at a rate of 2.11 paths per second, which is faster than the rate when using a fixed precision of 96 bits as shown in the final row of Table 3.3. This is due to the fact that many of the paths using higher precision only went to higher precision near the end of the path. One might also wonder why the adaptive precision runs with safety digit settings of 2 and 3 had different rates, since the almost the same number of paths used higher precision in each case. The difference is that with the safety digit settings set to 3, the paths using higher precision began using higher precision for slightly larger values of t than in the other run. In no case did the precision exceed 96 bits for any path.

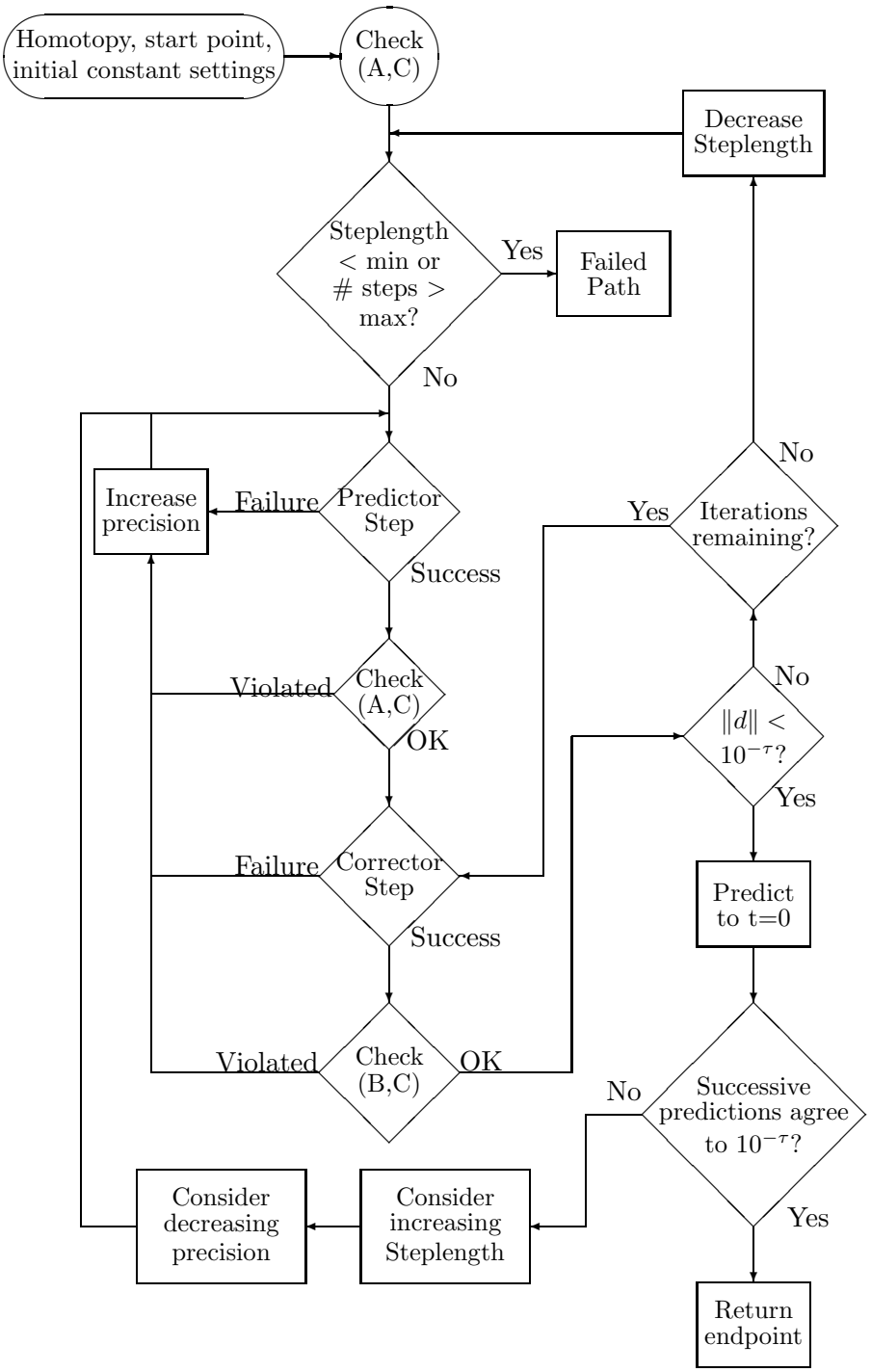

Figure 3.2. Step-adaptive precision path tracking

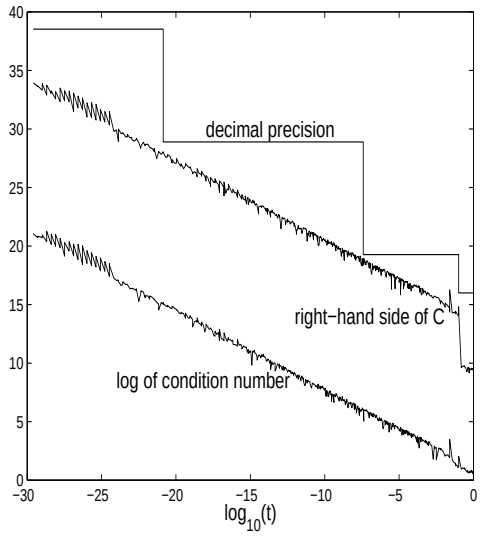

Figure 3.3. Right-hand side of C, condition number, and decimal precision versus $\log(t)$

CHAPTER 4

SYMBOLIC-NUMERIC COMPUTATION OF MULTIPLICITY

The goal of this chapter is to introduce a novel algorithm for the numerical computation of the multiplicity of a zero-scheme. This algorithm also produces a bound on the regularity of the ideal and other quantities related to the scheme structure. This algorithm is presented in greater detail in [9] and much of this chapter follows that paper very closely. For some of the necessary background, please refer to Chapter 2, [9], or the references therein. It should be noted that an algorithm with similar output is given in [19], although that method relies on observations of Macaulay in [41] while the present method relies on the key result of Bayer and Stillman in [11].

4.1 Multiplicity of an ideal and related ideas

As discussed in 2.3.1, to every ideal I one can associate a degree and a dimension, denoted $deg(I)$ and $dim(I)$ respectively. The degree and dimension of an ideal can be defined in terms of the Hilbert Polynomial of R/I. Definitions of these terms are not given presently as they are beyond the scope of this chapter. These definitions can be found in [16], [23], [29]. Please note that the dimension of an ideal means the dimension of the algebraic set related to the ideal in this chapter (not the Krull dimension of the ideal). The degree function allows one to define the multiplicity of a primary ideal.

Definition 4.1. Let I be a $\mathfrak{p}$-primary ideal. The *multiplicity* of I at $\mathfrak{p}$ is defined to be $\mu(I) = deg(I)/deg(\mathfrak{p})$.

Since multiplicity is defined as a fraction, it may at first seem a bit surprising that the following proposition is true. However, the multiplicity of any $\mathfrak{p}_i$-primary ideal can be shown to be a positive integer. The next proposition makes the definition that follows seem very natural.

Proposition 4.1. *If $\mathfrak{p}_i$ is not an embedded prime of I then the $\mathfrak{p}_i$-primary component that appears in any reduced primary decomposition of I is the same.*

Definition 4.2. Let I be an ideal. If $\mathfrak{p}_i$ is an associated prime of I but is not an embedded prime of I then the multiplicity of I at $\mathfrak{p}_i$ is defined to be the multiplicity of the $\mathfrak{p}_i$-primary component of I at $\mathfrak{p}_i$.

Multiplicity arises is many situations, but it is very natural in the case of polynomial factorization. By the fundamental theorem of algebra, every univariate polynomial factors into a product of linear polynomials over $\mathbb{C}$. That is to say, if $f(z)$ is a polynomial in the single variable, z, then one can write $f(z) = a(z - c_1)^{d_1}(z - c_2)^{d_2} \dots (z - c_t)^{d_t}$ where a is a nonzero complex number, $c_1, c_2, \dots, c_t$ are distinct complex numbers and $d_1, d_2, \dots, d_t$ are positive integers. Let $I = (f)$, the ideal generated by f, also sometimes written $I(f)$. Then $V(I)$ will be the set of points in $\mathbb{C}$ corresponding to the roots of f (i.e. $V(I) = \{c_1, c_2, \dots, c_t\}$). The radical of I will be the ideal $\sqrt{I} = (g)$ where $g = (z - c_1)(z - c_2) \dots (z - c_t)$. The reduced primary decomposition of I is $I = ((z-c_1)^{d_1}) \cap ((z-c_2)^{d_2}) \cap \dots \cap ((z-c_t)^{d_t})$. The associated primes of I are $(z-c_1), (z-c_2), \dots, (z-c_t)$. The associated primes of $\sqrt{I}$ yield the same list, thus (f) has no embedded primes. By the previous definitions, (f) has multiplicity d_i at the prime ideal $(z - c_i)$ for each i.

In general, if f is a multivariate polynomial then f can be written as $f = af_1^{d_1} f_2^{d_2} \dots f_t^{d_t}$ with a a nonzero constant, with each f_i an irreducible polynomial and with each d_i a positive integer. Furthermore, the factorization can be made so that f_i is not a multiple of f_j whenever $i \neq j$. The (unique) reduced primary decomposition of (f) is $(f) = (f_1^{d_1}) \cap (f_2^{d_2}) \cap \dots \cap (f_t^{d_t})$. To each irreducible factor, f_i, there is associated a variety, $V(f_i)$. $V(f)$ is the union of these varieties. $(f_i^{d_i})$ is a primary ideal with multiplicity d_i at (f_i). (f) has multiplicity d_i at (f_i) for each i. If f itself is irreducible then the ideal $I = (f)$ is a prime ideal. In general, for ideals with more than one generator, the multiplicity of the ideal at a given prime ideal is rather subtle and requires a more thorough understanding of the degree function, which is beyond the scope of this chapter. Nevertheless, it arises naturally in a number of engineering problems, poses significant challenges to numerical computation and is often associated with slow convergence rates. The next few sections present a few standard theorems from commutative algebra that will aid in the development of a numerical-symbolic algorithm to compute multiplicity.

4.1.1 Results about multiplicity

In the previous section, it was seen that it makes sense to talk about the multiplicity of an ideal I at a prime ideal $\mathfrak{p}$ provided $\mathfrak{p}$ is not an embedded prime of I. Let I_i be a $\mathfrak{p}_i$-primary ideal appearing in a reduced primary decomposition of I where $\mathfrak{p}_i$ is not an embedded prime of I. Let V_i be the variety associated to I_i. Let q be a generic point on V_i. The goal of this chapter is to give an overview of an algorithm that takes as input the point q and the dimension d_i of the variety V_i and produces as output the multiplicity of I at $\mathfrak{p}_i$. Let I_q be the prime ideal

corresponding to the point q. This problem may be reduced to the computation of the multiplicity of an I_q-primary ideal. In order to do this, first form the ideal, $J = (I, L_1, L_2, \ldots, L_{d_i})$ where $L_1, L_2, \ldots, L_{d_i}$ are generic linear forms in I_q. Then J has an I_q-primary component, I_q is not an embedded prime, and the multiplicity of the I_q-primary component of J is the same as the multiplicity of the $\mathfrak{p}_i$- primary component of I. This multiplicity may then be computed by a numerical-symbolic method. Throughout this section, assume that all ideals are homogeneous. In order to make use of this simplifying assumption, a homogenization procedure for non-homogeneous polynomials is needed. All of the theorems and propositions found in this and the next few sections are well known to experts. The proofs are beyond the scope of this thesis but may be found in the literature. No proofs will therefore be given, especially since the point of this chapter is the algorithm which makes use of this theory, not the theory itself.

Definition 4.3. Let $f \in \mathbb{C}[z_1, z_2, \ldots, z_n]$. The *homogenization of f with respect to* z_{n+1} is defined to be the element

$$f^h = z_{n+1}^{\deg(f)} f\left(\frac{z_1}{z_{n+1}}, \frac{z_2}{z_{n+1}}, \ldots, \frac{z_n}{z_{n+1}}\right) \in \mathbb{C}[z_1, z_2, \ldots, z_{n+1}].$$

Example 4.1.1. Let $f = x^2 + y^3 + 1 \in \mathbb{C}[x, y]$. The homogenization of f with respect to z is

$$f^h = z^3 f\left(\frac{x}{z}, \frac{y}{z}\right) = (z^3)\left(\left(\frac{x}{z}\right)^2 + \left(\frac{y}{z}\right)^3 + 1\right) = x^2 z + y^3 + z^3.$$

Since all polynomials will be made homogeneous, it is important to know that this homogenization procedure will not affect the multiplicity of an ideal.

Proposition 4.2 (Multiplicity unaffected by homogenization)**.** *Let* $f_1, f_2, \ldots, f_t \in$

$\mathbb{C}[z_1, z_2, \ldots, z_n]$. *Let* f^h *denote the homogenization of* f *with respect to* z_{n+1}. *Let* $q = (q_1, q_2, \ldots, q_n) \in \mathbb{C}^n$ *and let* $q' = [q_1 : q_2 : \cdots : q_n : 1] \in \mathbb{P}^n$. *The multiplicity of* $(f_1, f_2, \ldots, f_t)$ *at* I_q *is equal to the multiplicity of* $(f_1^h, f_2^h, \ldots, f_t^h)$ *at* $I_{q'}$.

The next theorem allows for the reduction of a general multiplicity computation to the multiplicity of a zero-dimensional object via slicing.

Theorem 4.3 (Multiplicity preserved by generic hyperplane sections)**.** *Let* I *be a homogeneous ideal. Let* $\mathfrak{p}$ *be a non-embedded, associated prime of* I. *Let* $V_h(\mathfrak{p})$ *be the projective variety associated to* $\mathfrak{p}$. *Let* q *be a generic point on* $V_h(\mathfrak{p})$ *and let* $D = Dim(V_h(\mathfrak{p}))$. *Let* I_q *be the homogeneous prime ideal associated to the point* q. *Let* $L_1, L_2, \ldots, L_D$ *be generic linear forms in* I_q. *Let* $J = (I, L_1, L_2, \ldots, L_D)$. *Then*

(i) I_q *is an associated prime of* J.

(ii) I_q *is not an embedded prime of* J.

(iii) The multiplicity of J *at* I_q *is equal to the multiplicity of* I *at* $\mathfrak{p}$.

This fact is of particular interest in the applied polynomial systems setting. When using standard numerical algebraic geometry techniques to find witness sets, one may compute a bound on the multiplicity of a component by counting the number of paths leading to a witness point of that component. If only one path leads to the witness point, the multiplicity is surely one. However, if multiple paths lead to the point, the multiplicity of the component represented by the witness point is bounded by the number of paths, and this bound can be artificially inflated if using randomization to square the polynomial system prior to tracking. The theorem above and the algorithm below are therefore useful in this setting since the true multiplicity, not just a bound, may be computed.

Definition 4.4. Let $R = \mathbb{C}[z_1, z_2, \ldots, z_n]$. The *maximal ideal of* R is the ideal $\mathfrak{m} = (z_1, z_2, \ldots, z_n)$.

It should be noted that if I is an $\mathfrak{m}$-primary ideal then $V(I)$ is the origin while $V_h(I)$ is the empty set. If I is a homogeneous ideal whose associated projective variety is a single point, q, then I_q is a non-embedded, associated prime of I. It does not necessarily follow that I is I_q primary. A reduced primary decomposition of I may still contain an $\mathfrak{m}$-primary component. In other words, $\mathfrak{m}$ may be an embedded prime. In any case, the computation of the multiplicity of I at I_q is aided by the following theorem. First, though, some notation is needed.

Definition 4.5. Let I be a homogeneous ideal in $R = \mathbb{C}[z_1, z_2, \ldots, z_n]$. The k^{th} *homogeneous part of* I is defined to be the set of all elements of I which are homogeneous of degree k. It is denoted $(I)_k$ and is a finite dimensional vector space over $\mathbb{C}$.

Recall that the product of ideals I and J is the ideal generated by products of elements in I and J. In particular, I^k is generated by the products of k elements of I. Then, using the previous definition, $(R)_k = (\mathfrak{m}^k)_k$.

Theorem 4.4 (Multiplicity of a homogeneous ideal supported at a point)**.** *Let q be a point in $\mathbb{P}^n$. Let I be a homogeneous ideal with $V_h(I) = q$. Let $R = \mathbb{C}[z_1, z_2, \ldots, z_{n+1}]$. The multiplicity of I at I_q is the dimension of $(R/I)_d$ as a $\mathbb{C}$-vector space for a large enough value of d. Furthermore, if I is I_q-primary then the multiplicity of I at I_q is the dimension of $R/(I, L)$ as a $\mathbb{C}$-vector space where L is a generic, homogeneous linear form in R.*

4.1.2 Regularity

The next proposition relates the multiplicity of a homogeneous ideal supported at a point to the regularity of the ideal. For the definition and some of the basic theorems concerning regularity, see [48], [21], or [12]. As always in algebraic geometry, one must be careful about whether quantities such as the regularity are being attached to the ideal or the quotient of the polynomial ring by the ideal (since the same computation taken each of those two ways will produce quantities that differ by one). For the purposes of this chapter, it suffices to make the following connection between regularity and multiplicity.

Proposition 4.5 (Regularity and multiplicity)**.** *Let q be a point in $\mathbb{P}^n$. Let I be a homogeneous ideal with $V(I) = q$. Let $R = \mathbb{C}[z_1, z_2, \ldots, z_{n+1}]$. The dimension of $(R/I)_d$ as a $\mathbb{C}$-vector space is equal to the dimension of $(R/I)_{d+1}$ as a $\mathbb{C}$-vector space for all d greater than or equal to the regularity of I.*

The algorithm developed below utilizes the previous theorem and proposition. In order to use these tools, one needs a method for computing the regularity of an ideal. This will provide a stopping criterion in the computation of the dimension of $(R/I)_d$.

Definition 4.6. Let I and J be ideals in $R = k[x_1, x_2, \ldots, x_n]$. The *ideal quotient* of I by J is defined to be $I : J = (\{f \in R | fJ \subseteq I\})$. Then $I : J^\infty$ is simply $(\{f \in R | f \in I : J^k \text{ for some } k\})$. The *saturation* of I, denoted I^{sat}, is then defined to be $I^{sat} = I : m^\infty$ where m is maximal.

Note that since R is Noetherian, there exists an N such that $I : m^n = I : m^{n+1}$ for all $n \geq N$. The ideal quotient is a form of "algebraic set subtraction" in that the algebraic set of $I : J$ is that of I minus that of J. The point of the saturation

of an ideal I is that it removes all m-primary components of I, i.e., all those components of the primary decomposition of I that have no geometric content.

Definition 4.7. Let $R = k[x_1, x_2, \ldots, x_n]$. An element $h \in R$ is *generic* for I if h is not a zero-divisor on R/I^{sat}. If $\dim(R/I) = 0$ then the general $h \in S$ is generic for I. For $j > 0$, define $U_j(I)$ to be the subset $\{(h_1, h_2, \ldots, h_j) \in R_1^j | h_i \text{ is generic for } (I, h_1, h_2, \ldots, h_{i-1}), 1 \leq i \leq j\}$.

With these definitions in place, the following crucial theorem of Bayer and Stillman (see [12]) may now be stated.

Theorem 4.6 (Criterion for m-regularity)**.** *Let k be an infinite field and let $R = k[x_1, x_2, \ldots, x_n]$. Let $I \subset R$ be a homogeneous ideal generated in degree at most m. The following conditions are equivalent:*

1. *I is m-regular.*
2. *There exists $h_1, h_2, \ldots, h_j \in R_1$ for some $j \geq 0$ such that*

$$((I, h_1, h_2, \ldots, h_{i-1}) : h_i)_m = (I, h_1, h_2, \ldots, h_{i-1})_m \quad \text{for} \quad i = 1, 2, \ldots, j$$

   and $(I, h_1, h_2, \ldots, h_j)_m = R_m$.

3. *Let $r = \dim(R/I)$. For all $(h_1, h_2, \ldots, h_r) \in U_r(I)$, and all $p \geq m$,*

$$((I, h_1, h_2, \ldots, h_{i-1}) : h_i)_p = (I, h_1, h_2, \ldots, h_{i-1})_p \quad \text{for} \quad i = 1, 2, \ldots, r$$

   and $(I, h_1, h_2, \ldots, h_r)_p = R_p$.

Furthermore, if $h_1, h_2, \ldots, h_j$ satisfy condition 2 then $(h_1, h_2, \ldots, h_j) \in U_j(I)$.

Corollary 4.7 (Stopping Criterion for regularity computation)**.** *Let I be a homogeneous ideal which is generated in degree at most k. Suppose $V_h(I)$ is a single point $q \in \mathbb{P}^n$. Let L be a linear form which is not contained in I_q. The regularity of I is less than or equal to k if and only if $(I : L)_k = (I)_k$ and if $(I, L)_k = (R)_k$.*

Proof. The corollary follows immediately from the theorem of Bayer and Stillman under the assumption that I defines a zero dimensional scheme. □

Recall that the dimension of a scheme in this chapter is the dimension of the support of the scheme rather than the Krull dimension of the ideal.

4.1.3 Multiplicity at a point

Theorem 4.8 (Convergence of multiplicity at an isolated point)**.** *Let $\mathbf{p}$ be a point in $\mathbb{P}^n$. Suppose $I_{\mathbf{p}}$ is an associated, non-embedded prime of an ideal I. Let $J_k = (I, I_{\mathbf{p}}^k)$. Then the multiplicity of I at $I_{\mathbf{p}}$ is equal to the multiplicity of J_k at $I_{\mathbf{p}}$ for $k \gg 0$.*

Proposition 4.9 (Persistence of multiplicity in neighborhood of a point)**.** *If the multiplicity of J_k at $I_{\mathbf{p}}$ is equal to the multiplicity of J_{k+1} at $I_{\mathbf{p}}$ then the multiplicity of I at $I_{\mathbf{p}}$ is equal to the multiplicity of J_k at $I_{\mathbf{p}}$,*

With the theorems in this section in place, it is now easy to describe the algorithm. This is done in the following section.

4.2 Numerical method for the determination of the multiplicity of a point

Recall that if I is a homogeneous ideal, if p is a point in $\mathbb{P}^n$ and if $V_h(I) = p$ then the multiplicity of I at I_p is the same as the dimension of $(R/I)_k$ as a $\mathbb{C}$-vector space for sufficiently large k. Note that the dimension of $(R/I)_k$ is equal

to $\dim(R)_k - \dim(I)_k$. When this is combined with the stopping criterion given above, the result is:

Algorithm 1. *find_mult*$(\{f_1, f_2, \ldots, f_r\}, p; \mu)$

Input: A set of homogeneous polynomials $\{f_1, f_2, \ldots f_r\} \subset \mathbb{C}[z_0, z_1, \ldots, z_n]$ and an isolated point $p = [p_0 : p_1 : \cdots : p_n] \in \mathbb{P}^n$ of $V_h(f_1, f_2, \ldots, f_r)$ with $p \notin V_h(z_n)$.

Output: μ = the multiplicity of the ideal, $(f_1, f_2, \ldots, f_r)$, at I_p.

Algorithm:

Form $I_p := (\{p_i z_j - p_j z_i \mid 0 \leq i, j \leq n\})$.

Form $\mathfrak{m} := (z_0, z_1, \ldots, z_n)$.

Let $k := 1$, $\mu(0) := 0$, and $\mu(1) := 1$.

while $\mu(k) \neq \mu(k-1)$ do

Form I_p^k.

Form $J_k := (I, I_p^k)$.

Form $\mathfrak{m}^k$ and $z_n \cdot \mathfrak{m}^k$.

Let $A := 0$ and $B := 1$.

while $A \neq B$ do

Form $(J_k)_{k+1}$.

Compute $P := (J_k)_{k+1} \cap z_n \cdot \mathfrak{m}^k$. (*)

Compute the preimage $\overline{P} \subseteq (\mathfrak{m})_k$ of P. (**)

Compute $A := \mathrm{r}ank((J_k)_k)$.

Compute $B := \mathrm{r}ank(\overline{P})$.

if $A = B$ then let $\mu(k) := rank((\mathfrak{m})_k) - A$, else let $J_k := \overline{P}$.

if $\mu(k) = \mu(k-1)$, then $\mu := \mu(k)$, else let $k = k + 1$.

Two steps of this algorithm are of particular interest, so they have been marked with the symbols (*) and (**). The algorithm, *find_mult*, has been implemented as a module of the Bertini software package, as discussed in Chapter 6. Although varying levels of precision may be used to pinpoint exactly which singular values are truly zero (see the discussion below), this approach was found to be unnecessary for the following examples. Rather, it was found that setting a threshold of 10^{-8}, below which all singular values are considered to be zero, is sufficient.

Step (*) of *find_mult* involves the intersection of the degree $k+1$ component of two graded ideals, a procedure that deserves some explanation. Since the degree $k+1$ component of each of the two ideals is represented as a vector space (with the generators representing bases), the intersection of the degree $k+1$ component of the two graded ideals is equivalent to the intersection of two vector spaces, say V and W. Briefly, the left singular vectors of V and W corresponding to zero singular values form bases for $V^\perp$ and $W^\perp$, respectively. Then, by concatenating these bases into a single matrix and computing the SVD of that matrix, the left singular vectors corresponding to the zero singular values form a basis for $(V^\perp \cup W^\perp)^\perp = V \cap W$. Step (**) is simply a matter of mechanically removing a factor of z_n from each polynomial of P.

4.3 Examples

For each example in this section, the ideal was run through the Bertini implementation of *find_mult* discussed in the previous section, on a single processor 3 GHz Pentium 4 machine running Linux. To simplify parsing, all exact numbers

were first converted to 16 digit floating point approximations. In each example, the invariants predicted by theory or computed symbolically were confirmed numerically.

4.3.1 Monomial ideals

Let I be a monomial ideal of the form $I = (M_1, M_2, \ldots, M_N)$, with $M_i = z_1^{k_{i,1}} z_2^{k_{i,2}} \ldots \ z_n^{k_{i,n}}$ (where the $k_{i,j}$ are nonnegative integers). Suppose $V(I) = (0, 0, \ldots, 0)$ (or equivalently, that $V_h(I) = \emptyset$). Then the multiplicity of I at the prime ideal $(z_1, z_2, \ldots, z_n)$ is exactly the number of monomials that are not in I. This leads to the easily understood staircase counting method, as described in [16]. For example, the ideal $I = (x^5, y^5, x^2y^4, x^3y)$ in $\mathbb{C}[x, y]$ has multiplicity 16 at (x, y) since $\{1, x, y, x^2, xy, y^2, x^3, \ x^2y, xy^2, y^3, x^4, x^2y^2, xy^3, y^4, x^2y^3, xy^4\}$ is a full list of the monomials not in I. Using *find_mult*, this multiplicity was confirmed (by considering I as an ideal in $\mathbb{C}[x, y, z]$). The multiplicities of a large number of similar problems was also confirmed by this method.

An upper bound on the multiplicity of a zero-scheme defined by a square ideal, i.e., an ideal with a generating set which possesses the same number of polynomials and variables, may be obtained via homotopy continuation by counting the number of solution paths converging to the point in question (see [56]). However, when considering a polynomial system with more equations than unknowns, this bound is generally much too high. It is therefore interesting to note that *find_mult* works for any zero-scheme, whether the ideal is square or not.

4.3.2 A nontrivial exact problem

Consider the ideal

$$I = (x^4 + 2x^2y^2 + y^4 + 3x^2yz - y^3z, x^6 + 3x^4y^2 + 3x^2y^4 + y^6 - 4x^2y^2z^2) \quad (1)$$

in $\mathbb{C}[x, y, z]$. This ideal is a homogenized form of an example discussed in [23]. As described in the book, the multiplicity of I at $I_{[0:0:1]}$ can be computed as the intersection number of the two generators of the ideal at $[0 : 0 : 1]$. Using the techniques described in that text, a hand calculation determines that the multiplicity of I at $I_{[0:0:1]}$ is 14. The multiplicity was confirmed numerically to be 14 via the Bertini implementation of *find_mult*.

4.3.3 A related inexact problem

Although symbolic algorithms for computing the previously mentioned multiplicities could easily handle either of the preceding examples, such techniques cannot be applied if the input data is inexact. This is due to the fact that small perturbations in the generators of an ideal can have drastic effects on the (exact) zero locus of the generators.

For example, consider the following ideal in $\mathbb{C}[x, y, z]$,

$$I = (x^4+2x^2y^2+y^4+3x^2yz-y^3z+.001z^4, x^6+3x^4y^2+3x^2y^4+y^6-4x^2y^2z^2), \quad (2)$$

created by perturbing a single coefficient of (1) by 10^{-3}. A symbolic algorithm will compute the multiplicity of I at $I_{[0:0:1]}$ to be 0 (since $[0 : 0 : 1]$ is not a point on $V_h(I)$). However, *find_mult* reports that the multiplicity of the associated zero-scheme and Hilbert function are the same as those for (1).

This may seem very strange, and the numerical analysis behind this algorithm is not yet fully understood. However, it is known that *find_mult* will fail to find the correct solution using tight settings of the zero threshold placed on the singular values. The idea is that as the coefficients are perturbed away from the original ideal, the numerical rank and vector spaces computed in the algorithm will move continuously so that the more severe the perturbation, the looser the tolerance must be to achieve success.

The point of this example is not to illustrate any careful analytical result regarding *find_mult*. Rather, the goal is to indicate that there is at least some hope that this algorithm will work with inexact data. A careful analysis of these ideas is certainly warranted.

4.3.4 Another inexact problem

Frequently, a symbolic version of roundoff is done to prevent coefficient blowup (which was discussed in Section 2.1.3). The computations are carried out with exact arithmetic but over a field with finite characteristic. There are settings in which this technique can lead to some uncertainty as to the meaning of the results. In the following example, symbolic techniques over $\mathbb{Q}$ were used to determine the multiplicity at a certain prime. In carrying out this computation, some of the coefficients involved in the Gröbner basis computation reached 42 digits!

Consider the ideal in $\mathbb{Q}[w,x,y,z]$ generated by the following polynomials:

$$3x^2 - y^2 - z^2 + 2yz - 12xw - \frac{1212}{161}yw + \frac{729}{161}zw + \frac{383678}{77763}w^2,$$
$$x^3 - 8x^2w - 6xyw - 4y^2w + z^2w + \frac{1047}{63}xw^2 + \frac{117}{7}yw^2 - \frac{183}{23}zw^2 + \frac{2600452}{699867}w^3,$$
$$z^3 + 4x^2w + 2xyw - \frac{102}{23}z^2w - \frac{415}{21}xw^2 - \frac{19}{3}yw^2 + \frac{5055}{529}zw^2 + \frac{12496664}{766521}w^3,$$
$$x^2 - \frac{16}{3}xw + \frac{64}{9}w^2.$$

This ideal was confirmed symbolically to have multiplicity 2 at the prime ideal corresponding to the point $[1 : \frac{8}{3} : -\frac{2}{7} : \frac{34}{23}]$ in projective space. The example is a modified version of an example found in [57]. The modification involves slicing by x^2 and then applying a change of variables. Using *find_mult*, the multiplicity of the zero-scheme has been confirmed to be 2. This multiplicity information was obtained using a fairly meager amount of computational resources, and coefficient blowup was completely avoided.

4.4 A method for determining which singular values are zero

Extensions of the above algorithm to the numerical computation of free resolutions and other objects of interest are now on the horizon. However, a fundamental problem is the decision of precisely which singular values should be considered zero. There are a growing number of symbolic-numeric methods that require such a decision (in fact, it seems that many such methods, if cast in such a way, rely on a numerical rank computation), but the standard way of making this decision is to leave it to the user of the algorithm.

However, if exact input data is available, there is a way of determining which singular values are zero and which are nonzero, given a confidence threshold θ.

Using one level of precision, compute the singular values of the matrix at hand. Then increase precision and recompute the singular values. Those singular values that are smaller than the first number of digits used will move towards zero in the second computation. This procedure may be carried out iteratively until all singular values either stabilize (meaning that they are nonzero) or drop below θ, at which point they may be considered to be zero. Although this procedure does not guarantee whether a singular value is exactly zero given exact data, it may be used to find whether the singular value is "small enough" to be ignored in the computation at hand.

One might think that this procedure is no different than setting a threshold *a priori* for considering singular values to be zero. The difference is that no guarantee about whether the computed singular values are actually insignficant may be given without using multiple levels of precision. One could also argue that most data provided in the real world is empirical so that exact data is not generally available. This is, however, not usually the case with polynomial systems arising in engineering and science. Those polynomial systems appearing in the literature that have inexact coefficients are usually just truncations of known, exact polynomial systems made approximate in order to study the effects of rounding. It should be noted, though, that some systems with empirically-discovered coefficients have been studied in the past, although it might be possible to increase the number of digits in the coefficients by further empirical investigation, if desired.

CHAPTER 5

SOLVING TWO-POINT BOUNDARY VALUE PROBLEMS WITH HOMOTOPY CONTINUATION

The technique of homotopy continuation may be applied in many situations. For example, a method employing homotopy continuation to solve two-point boundary value problems was developed in [5]. That method is the subject of this chapter, and this chapter follows that paper rather closely. In particular, section 5.1 introduces the types of problems to be considered and conveys the basic idea of the algorithm. As discussed in section 5.2, much may be gained by restricting the algorithm to a special class of problems - those with polynomial nonlinearity. Finally, section 5.3 describes the success of an implementation of the algorithm on several problems of interest.

5.1 The basic algorithm

Consider a two-point boundary value problem on the interval $[a, b] \subset \mathbb{R}$,

$$y'' = f(x, y, y'), \tag{1}$$

with boundary conditions $y(a) = \alpha$ and $y(b) = \beta$. One may wish to approximate solutions to (1) via numerical techniques rather than attempting to compute exact, closed form solutions. With such numerical methods, one typically constructs

a mesh on $[a, b]$ and chooses a difference operator to use in place of the derivative. One common choice is the standard central difference approximation with a uniform mesh. In particular, let N be a positive integer, $h := \frac{b-a}{N+1}$, and $x_i := a + ih$ for $i = 0, \ldots, N+1$. Setting $y_0 = \alpha$ and $y_{N+1} = \beta$, the discretization of (1) takes the form of the system $\mathcal{D}_N$:

$$\begin{array}{ccccccc} y_0 & - & 2y_1 & + & y_2 & = h^2 f(x_1, y_1, \frac{y_2 - y_0}{2h}) \\ \vdots & & \vdots & & \vdots & = \quad \vdots \\ y_{N-1} & - & 2y_N & + & y_{N+1} & = h^2 f(x_N, y_N, \frac{y_{N+1} - y_{N-1}}{2h}) \end{array}$$

A solution $y(x)$ of (1) may then be approximated by an N-tuple of real numbers $(y_1, \ldots, y_N)$ such that $y_i \approx y(x_i)$ for $i = 1, \ldots, N$.

(1) may have any number of solutions, ranging from none to infinitely many, depending upon f. There are some existence theorems for solutions of such equations, but there are very few theorems about the number of solutions when existence is known. Thus, a procedure that indicates the number of solutions could be useful. Unfortunately, a discretization such as $\mathcal{D}_N$ may have solutions that do not converge to a solution of (1) as $N \to \infty$. Such solutions are called spurious. Any solution y will be approximated with $\mathcal{O}(h^2)$ accuracy on the mesh by some solution $\bar{y} \in \mathbb{R}^N$ of $\mathcal{D}_N$, for N large enough.

The algorithm discussed in this chapter is a relatively secure numerical method for finding the solutions of the discretization of certain types of two-point boundary value problems without requiring highly refined meshes. The key idea of the algorithm is to perform homotopy deformations between discretizations with increasingly many mesh points, as suggested in [4]. One first computes approximate solutions using only a few meshpoints, after which carefully-chosen homotopies

make it possible to move iteratively from a mesh with N mesh points to one having $N+1$ mesh points. By restricting the method to problems having polynomial nonlinearity, one can easily compute the solutions at the initial stage and can usually assure that all solutions are found at each stage of the algorithm.

Even though it is not possible to guarantee that all solutions will be found, the method generates multiple solutions that, in all test cases, include approximations to all known solutions. While symbolic techniques such as Gröbner basis methods (see [16]) or cellular exclusion methods (see [24]) could be applied to solve the discretizations, homotopy continuation is useful due to its ability to handle large systems. Other numerical methods have been developed to treat two-point boundary value problems (see [33] and [50]), but such methods require adequate initial solution estimates. The method of this chapter allows one to begin with no such estimates.

Here is a rough draft of the procedure:

1. Compute all solutions of $\mathcal{D}_N$ for some small N. It is reasonable to choose the smallest N for which the system may be solved.

2. (optional) Remove all solutions that do not display properties known to be exhibited by solutions of the differential equation. This step is known as "filtering". It could be that nothing is known *a priori* about the solutions of (1), so it is necessary to skip this step. In any case, denote by $\mathcal{V}_N$ the set of solutions which remain after filtering.

3. If the estimates are not yet adequate, add a mesh point to obtain the discretization $\mathcal{D}_{N+1}$. Use the solutions in $\mathcal{V}_N$ to generate solutions $\mathcal{V}_{N+1}$ of $\mathcal{D}_{N+1}$ via polynomial solving and homotopy continuation. Then go to Step 2. If the estimates are adequate, the process is complete.

Step 1 is difficult in general, so the restriction to certain cases is necessary, as described in the next section. Step 2 is optional and will also be discussed in section 5.2. Step 3 is of particular interest and is the key to the entire algorithm. There are two parts to step 3. First, solutions in $\mathcal{V}_N$ are extended to account for the new mesh point. That will be discussed shortly. Second, the extended solutions of $\mathcal{V}_N$ must be tracked using homtopy continuation to produce solutions in $\mathcal{V}_{N+1}$. The construction of the appropriate homotopy will be addressed presently.

Consider the homotopy function

$$H_{N+1}(y_1,\ldots,y_{N+1},t) :=$$
$$\begin{bmatrix} y_0 - 2y_1 + y_2 & - & h(t)^2 f\left(x_1(t), y_1, \frac{y_2-y_0}{2h(t)}\right) \\ & \vdots & \\ y_{N-2} - 2y_{N-1} + y_N & - & h(t)^2 f\left(x_{N-1}(t), y_{N-1}, \frac{y_N-y_{N-2}}{2h(t)}\right) \\ y_{N-1} - 2y_N + Y_{N+1}(t) & - & h(t)^2 f\left(x_N(t), y_N, \frac{Y_{N+1}(t)-y_{N-1}}{2h(t)}\right) \\ y_N - 2y_{N+1} + Y_{N+2}(t) & - & h(t)^2 f\left(x_{N+1}(t), y_{N+1}, \frac{Y_{N+2}(t)-y_N}{2h(t)}\right) \end{bmatrix}$$

with

$$\begin{aligned} y_0 &:= \alpha \\ h(t) &:= t\left(\tfrac{b-a}{N+1}\right) + (1-t)\left(\tfrac{b-a}{N+2}\right) \\ Y_{N+1}(t) &:= (1-t)y_{N+1} + \beta t \\ Y_{N+2}(t) &:= \beta(1-t) \\ x_i(t) &:= a + ih(t), \qquad i = 1,\ldots,N+1. \end{aligned}$$

To see that this is a good choice for the homotopy function, note that at $t = 0$, the homotopy is clearly equal to $\mathcal{D}_{N+1}$, while at $t = 1$, it is $\mathcal{D}_N$ plus an extra equation. The extra equation comes from having extended the solution in $\mathcal{V}_N$ to have the value $Y_{N+2}(1)$ at the new right-hand boundary, $x = b + h(1)$. As t goes

from 1 to 0, the mesh points are squeezed back inside $[a, b]$, and the right-hand boundary condition $y(b) = \beta$ is transferred from Y_{N+1} to Y_{N+2} as Y_{N+1} is forced to equal y_{N+1}. It should be noted that the choice of the homotopy function is not unique. All that is claimed presently is that the above homotopy function is adequate.

As mentioned prior to the presentation of the homotopy function, it is necessary to construct the homotopy continuation start points out of $\mathcal{V}_N$. For each solution of $\mathcal{V}_N$, one must find the value of y_{N+1} which satisfies the homotopy at $t = 1$. The final equation is the only equation in which y_{N+1} occurs. So, for each solution of $(y_1, \ldots, y_N)$ in $\mathcal{V}_N$, one must use this equation to solve for y_{N+1}.

One may avoid (with probability one) path singularities away from $t = 0$ by inserting a random $\gamma \in \mathbb{C}$ into any homotopy function. For completeness, the resulting homotopy function is recorded here:

$$
\begin{array}{l}
H_{N+1}(y_1, \ldots, y_{N+1}, t) := \\
\left[\begin{array}{rcl}
\Gamma(t)\,(y_0 - 2y_1 + y_2) & - & h(t)^2 f\left(x_1(t), y_1, \frac{y_2 - y_0}{2h(t)}\right) \\
 & \vdots & \\
\Gamma(t)\,(y_{N-2} - 2y_{N-1} + y_N) & - & h(t)^2 f\left(x_{N-1}(t), y_{N-1}, \frac{y_N - y_{N-2}}{2h(t)}\right) \\
\Gamma(t)\,(y_{N-1} - 2y_N) + Y_{N+1}(t) & - & h(t)^2 f\left(x_N(t), y_N, \frac{Y_{N+1}(t) - y_{N-1}}{2h(t)}\right) \\
\Gamma(t)\,(y_N - 2y_{N+1} + \beta) & - & h(t)^2 f\left(x_{N+1}(t), y_{N+1}, \frac{\beta - y_N}{2h(t)}\right)
\end{array}\right]
\end{array}
\tag{2}
$$

with

$$
\begin{array}{rcl}
\Gamma(t) & := & \gamma^2 t + (1 - t) \\
h(t) & := & \gamma t \left(\frac{b-a}{N+1}\right) + (1 - t)\left(\frac{b-a}{N+2}\right) \\
Y_{N+1}(t) & := & (1 - t) y_{N+1} + \gamma^2 \beta t \\
x_i(t) & := & a + i h(t), \qquad i = 1, \ldots, N + 1.
\end{array}
$$

5.2 Details of the algorithm

As mentioned in the previous section, it is beneficial to restrict to cases for which $f(x, y, y')$ is easily solved. Thus, it shall be assumed in the following that f is actually a real polynomial $p(y)$. This causes the right-hand side of the equations of $H_{N+1}(y, t)$ to have the simple form $h^2(t)p(y_i)$. This restriction makes it simple to obtain start points for the current level of the algorithm from the end points of the previous level.

When solving the last polynomial of the homotopy function to obtain the start points, one will obtain d complex values of y_{N+1} for each solution in $\mathcal{V}_N$. Suppose that at each level of the algorithm all paths end in finite, nonsingular solutions. Then, the solution list $\mathcal{V}_N$ will have d^N entries. This gives a complete list of the solutions of the discretized problem, but the exponential growth in the size of the list is obviously prohibitive as N increases.

Fortunately, if some property of the solutions of (1) is known *a priori*, those solutions in $\mathcal{V}_N$ not exhibiting those properties may be discarded. This is filtering. Naturally, there are some differential equations for which no special filters are known while there are other problems for which several filters may be chosen. Regardless of the problem, for small N, it is probably necessary and reasonable to retain all solutions. For larger N (e.g., $N \geq 5$), a filter becomes necessary in order to proceed.

One filter that is not specific to a certain class of boundary value problems is based upon the idea that a solution of (1) will not grow "too fast" while spurious solutions are likely to grow "fast". In particular, one may take as a filter the discretization of $y''' = p'(y)y'$. To get the discretization, use the central difference

approximations

$$y'(x_i) = \frac{y_{i+1} - y_{i-1}}{2h},$$

and

$$y'''(x_i) = \frac{y_{i+2} - 2y_{i+1} + 2y_{i-1} - y_{i-2}}{2h^3}$$

applied only at the mesh points $y_2, \ldots, y_{N-1}$. So, one might choose to throw away the point $z = (y_1, \ldots, y_N) \in \mathcal{V}_N$ if

$$\sum_{i=2}^{N-1} \left| \frac{y_{i+2} - 2y_{i+1} + 2y_{i-1} - y_{i-2}}{2h^3} - p'(y_i) \frac{y_{i+1} - y_{i-1}}{2h} \right| > \epsilon_2$$

for some $\epsilon_2 > 0$. As is usual in such circumstances, one must weigh computational resources against completeness in needing to specify ϵ_2.

Other filters may be derived from known properties of the solutions of the problem at hand. For example, it may be known that solutions are symmetric about $x = \frac{a+b}{2}$, are always positive, oscillate with a specific period, or exhibit some other easily-detected behavior. Although it is tempting to filter out solutions having a nonzero complex part, this is not a valid filtering rule. Examples in which nonreal approximate solutions converge to real approximate solutions have been observed. Similarly, it is possible that oscillating solutions may arise from a sequence of non-oscillating solutions and that similar difficulties may occur with other filters. Thus, the use of filters may be computationally beneficial, but with it comes the risk of not finding all real solutions to the problem.

Now the final version of the algorithm may be stated:

1. For a small value of N (usually $N = 1$), solve $H_N(y_1, \ldots, y_N, 1)$, by any appropriate method in order to produce $\mathcal{V}_1$.

2. Repeat the following (incrementing N before each loop) until some desired stopping criterion occurs:

   (a) Form the homotopy $H_N(y_1, \ldots, y_N, t)$.

   (b) Solve the last polynomial of $H_N(y_1, \ldots, y_N, t)$ for y_N using each solution in $\mathcal{V}_{N-1}$, thereby forming the set S of the start solutions for $H_N(y_1, \ldots, y_N, t)$.

   (c) Track all paths beginning at points in S at $t = 1$. The set of endpoints of these paths is $\mathcal{V}_N$.

   (d) If desired, apply a filter to $\mathcal{V}_N$ to reduce the number of paths to be tracked in stage $N + 1$.

5.3 Examples

As discussed in Chapter 6, the algorithm of this chapter has been implemented in Maple, making use of an unreleased version of Bertini for the homotopy continuation. Although Bertini can change precision adaptively, each of the examples of this section ran successfully using only regular precision. This is not surprising as the continuation paths were all nonsingular throughout $[0, 1]$.

In this section, N is used to denote the number of mesh points, SOLS(N) denotes the total number of solutions (real or complex), and REAL(N) denotes the number of real solutions. As a convention, say that a solution is real if the imaginary part at each mesh point is zero to at least eight decimal digits.

TABLE 5.1

EVIDENCE OF $\mathcal{O}(h^2)$ CONVERGENCE FOR PROBLEM (3)

N	Maximal error at any mesh point	h^2	Maximal error/h^2
3	1.570846e-04	4.000000e-02	3.927115e-03
4	1.042635e-04	2.777778e-02	3.753486e-03
5	7.069710e-05	2.040816e-02	3.464158e-03
6	5.348790e-05	1.562500e-02	3.423226e-03
7	4.078910e-05	1.234568e-02	3.303917e-03
8	3.230130e-05	1.000000e-02	3.230130e-03
9	2.624560e-05	8.264463e-03	3.175718e-03

5.3.1 A basic example

As a first, simple example, consider the two-point boundary value problem

$$y'' = 2y^3 \tag{3}$$

with boundary conditions $y(0) = \frac{1}{2}$ and $y(1) = \frac{1}{3}$.

It is known that there is a unique solution, $y = \frac{1}{x+2}$, to (3). The algorithm of this chapter produces one real solution among a total of 3^N complex solutions found for $N = 1, \ldots, 9$. Furthermore, the error between the computed solution and the unique exact solution behaves as $\mathcal{O}(h^2)$, as shown in Table 5.1.

5.3.2 An example with a filter

Consider the problem

$$y'' = -\lambda\left(1 + y^2\right) \tag{4}$$

with zero boundary conditions, $y(0) = 0$ and $y(1) = 0$, and $\lambda > 0$.

It is known that there are different numbers of solutions depending on the value of λ. In particular, there are two solutions if $\lambda < 4$, a unique solution if $\lambda = 4$, and no solutions if $\lambda > 4$. This example was run without any filter, resulting in the confirmation of the expected number of real solutions in the cases $\lambda = 2 < 4$ and $\lambda = 6 > 4$. As expected by theory (see [36]), the computed solutions were symmetric about the midpoint of the interval. The case of $\lambda = 4$ is numerically difficult since, in that case, the Jacobian of the associated polynomial system is not of full rank.

Tracking all 2^{17} paths for $N = 17$ took just under an hour of CPU time on a single processor Pentium 4, 3 GHz machine running Linux. At this rate, ignoring the time-consuming data management part of the algorithm, it would take well over one year to track all 2^{30} paths for $N = 30$ mesh points. As discussed in the previous section, filtering rules may be used to discard most paths at each stage, making the use of many mesh points feasible. Taking advantage of the symmetry condition for this problem, a filter forcing $||y_1| - |y_N|| < 10^{-8}$ was applied to the case $\lambda = 2$. This reduced the path-tracking time to less than half a second for $N = 17$ mesh points. This drastic reduction in time for path-tracking as well as data management allowed for the confirmation of the existence of two real solutions for up to 100 mesh points. Despite the size of the polynomial system when $N = 100$, each path took less than 4 seconds to track from $t = 1$ to $t = 0$. A graph of the two real solutions for $N = 20$ mesh points is given in Figure 5.1.

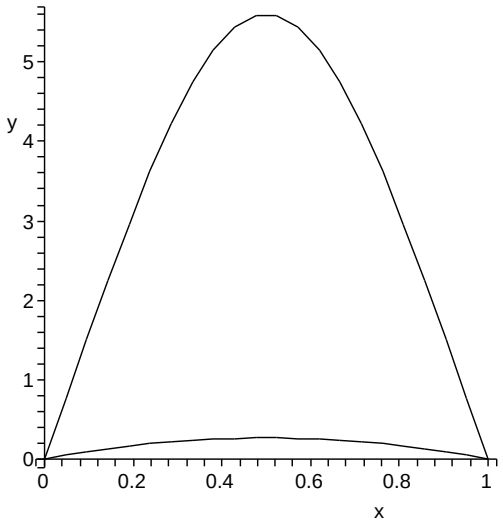

Figure 5.1. The real solutions of (4) with $N = 20$.

5.3.3 A problem with infinitely many solutions

It was shown in [14] that the two-point boundary value problem

$$y'' = -\lambda y^3, \qquad y(0) = y(1) = 0, \tag{5}$$

with $\lambda > 0$ has infinitely many oscillating real solutions on the interval $[0, 1]$. These solutions occur in pairs in the sense that $-y$ is a solution whenever y is a solution. Due to the trivial solution, one would therefore always expect to have an odd number of real solutions. This fact was confirmed computationally, as shown in Table 5.2. Only the case of $\lambda = 1$ is displayed as all other cases are identical up to scaling. The number of real solutions found by Bertini grows without bound for this problem as the number of mesh points increases. In fact, beyond some

TABLE 5.2

SOLUTIONS OF PROBLEM (5)

N	SOLS(N)	REAL(N)
1	3	3
2	3	3
3	9	3
4	27	7
5	81	11
6	243	23
7	729	47
8	2187	91

small value of N, the number of real solutions approximately doubles for each subsequent value of N. More discussion of this problem may be found in [3].

5.3.4 The Duffing problem

One representation (see [18]) of the Duffing problem is the two-point boundary value problem

$$y'' = -\lambda \sin(y) \tag{6}$$

on the interval $[0, 1]$ with $y = 0$ at the endpoints and $\lambda > 0$. Since only the case of polynomial nonlinearity is being considered presently, $\sin(y)$ is approximated

by truncating its Taylor series expansion, yielding the problem

$$y'' = -\lambda \left(y - \frac{y^3}{6} \right) \tag{7}$$

using two terms or

$$y'' = -\lambda \left(y - \frac{y^3}{6} + \frac{y^5}{120} \right) \tag{8}$$

using three terms.

As with (4), the number of real solutions to the exact Duffing problem (6) depends on the value of λ. In this case, the number of real solutions increases as λ grows. In particular, it is known that there are $2k+1$ real solutions when $k\pi < \lambda < (k+1)\,\pi$. For a given value of λ, the $2k+1$ real solutions include the trivial solution $y \equiv 0$ and k pairs of solutions $(y_1(x), y_2(x))$ such that $y_1(x) = -y_2(x)$. Each solution pair oscillates with a different period. Unfortunately, as Taylor series truncations for $\sin(y)$ do not approximate $\sin(y)$ well outside of a small neighborhood, the behavior of solutions to (7) and (8) may be quite different than the behavior of solutions of (6).

Table 5.3 indicates the number of real solutions computed for problems (7) and (8) for $\lambda = 0.5\pi$, 1.5π, and 2.5π. All solutions have either odd or even symmetry about $x = \frac{1}{2}$, so the filter $||y_1| - |y_N|| < 10^{-8}$ was used as in (4). The filter was first applied when $N = 4$, so the number of real solutions reported in each case of Table 5.3 is the number of real solutions found for $N \geq 5$. For $\lambda = 0.5\pi$ and $\lambda = 1.5\pi$, there were more real solutions found for (8) than predicted for the exact problem (6). However, the computed solutions in each case included a pair of wildly-oscillating solutions. These poorly-behaved solutions are readily identified by the y''' filter discussed in the previous section: using $N = 25$ mesh

TABLE 5.3

NUMBER OF REAL SOLUTIONS FOR APPROXIMATIONS OF THE DUFFING PROBLEM

λ	$f(y) = y - \frac{y^3}{6}$	$f(y) = y - \frac{y^3}{6} + \frac{y^5}{120}$	$f(y) = \sin(y)$
0.5π	1	3	1
1.5π	1	5	3
2.5π	1	5	5

points, the poorly-behaved solutions had residuals four orders of magnitude larger than those of the well-behaved solutions.

5.3.5 The Bratu problem

The Bratu problem on the interval $[0, 1]$ has the form

$$y'' = -\lambda e^y, \qquad y(0) = y(1) = 0, \tag{9}$$

with $\lambda > 0$. As in the case of the Duffing problem, the right-hand side may be made into a polynomial by truncating the power series expansion of e^y, yielding

$$y'' = -\lambda \left(1 + y + \frac{y^2}{2}\right) \tag{10}$$

As discussed in [18], there are two real solutions if λ is near zero and no real solutions if λ is large. The real solutions for small λ are symmetric and

nonnegative. The expected number and properties of the real solutions in the cases of $\lambda = 0.5$ and $\lambda = 10$ were confirmed, and, as anticipated, 2^N total solutions were found in each case for $N = 1, \ldots, 15$.

CHAPTER 6

BERTINI: IMPLEMENTATION DETAILS

Several software packages focused specifically on homotopy continuation, numerical algebraic geometry, and related techniques have been developed over the past two decades, such as PHC (see [62]). Although these packages have been quite successful, none made use of adaptive precision as described in Chapter 3, of the numerical algorithm for computing multiplicity described in Chapter 4, or of several other recent algorithms in numerical algebraic geometry. Thus, there was need for a new software package including these, and other, advances.

Bertini is a software package under ongoing development by D. Bates, A. Sommese, and C. Wampler, with some early work by C. Monico. Bertini is written in the C programming language and makes use of several advanced computational tools, such as MPFR for multiprecision floating point support and the tandem of flex and bison for parsing. The following sections contain detailed descriptions of various aspects of Bertini, incluing the basic structures and linear algebra subroutines, the use of straight-line programs and automatic differentiation, and the implementation of adaptive precision, as well as several others. Appendix A also contains some source code related to the fundamental types of Bertini as well as a complete set of input and output files. The details of this chapter and Appendix A are correct for Bertini 1.0, but since Bertini is still under development, they are subject to change.

6.1 Basic structure of a Bertini run

As mentioned above, Bertini is written in the C programming language. To run Bertini for path-tracking purposes (i.e., for the computation of zero- and/or positive-dimensional witness sets), one must provide two to three simple text files. The number and syntax of these files depends on various aspects of the run, such as whether automatic m-homogenization is desired. The required files differ significantly for specialized runs (e.g., the computation of the multiplicity of a zero-dimensional ideal at a specified point), so those files will be described in the appropriate sections below.

6.1.1 The *input* file

Naturally, one must provide a file containing the polynomial system or homotopy along with other data such as the names of the polynomials and variables. This file is typically named *input*. For zero-dimensional solving, the user may specify either a target system only or an entire homotopy. If the user chooses to provide only a target system, he or she must also specify the m variable groupings for m-homogenization as the only option at present for the automatic creation of a start system is the m-homogeneous start system for $m \geq 1$. Other options, such as total degree and linear product start systems, will be implemented soon. Also, the user must provide a name for the path variable, which is typically called "t". Please note that it is imperative (for technical reasons) that the m of "mhom" in the example below is in the first space of the first line of the *input* file. This is an unfortunate requirement that will be removed in later releases of Bertini. As an example, the *input* file for the Griewank-Osborne problem (as in [27]) should have the following form:

```
mhom 2;
variable_group z1;
variable_group z2;
function f1, f2;
pathvariable t;
f1 = (29/16)*z1^3 - 2*z1*z2;
f2 = z2 - z1^2;
END
```

As mentioned above, the user may also choose to specify the entire homotopy, rather than only the target system. In that case, he or she must also provide the names of any parameters used in the homotopy. For technical reasons, there must be at least one parameter, and the role of the path variable must be carried out by a parameter. Please see the last example of this section to see why this this awkward requirement is necessary. In this case, the user provides a list of variables rather than variable groups. The *input* file for the Griewank-Osborne problem in this case would be:

```
variable z1, z2;
function f1, f2;
pathvariable t;
parameter s;
constant g;
s=t;
g=0.123247542+I*0.76253746298;
X = z1*z2;
Y = z1^2;
f1 = ((29/16)*z1^3 - 2*X)*(1.0-s) + g*s*(z1^3-1.0);
f2 = (z2 - Y)*(1.0-s) + g*s*(z2^2-1.0);
END
```

It may be useful to define constants outside of the actual definition of the polynomials. Bertini allows for such variables, although they must be declared before use. Also, due to the symmetry that is often present in polynomial systems, especially those coming from mechanisms, the use of subfunctions may result in increased efficiency for polynomial evaluation. For example, if some expression appears repeatedly throughout a polynomial system, it is more efficient to compute the value of that expression once and thereafter use the result than to compute the value of the expression every time that it is encountered. Bertini allows for subfunctions, the names of which need not be declared. The previous example illustrates the use of both constants and subfunctions.

For positive-dimensional Bertini runs, the *input* file resembles a hybrid of the two types of *input* files for zero-dimensional runs. Bertini will create an appropriate total degree start system and attach it to the provided polynomial system, so one should not specify a homotopy in this case. Again, one must specify both a path variable name and a parameter name, although neither of these names should appear in the definition of the target system. The *input* file for a positive-dimensional Bertini run for the "illustrative example" of [53] should have the following form:

```
variable x, y, z;
function f1, f2, f3;
pathvariable t;
parameter s;
s=t;
f1 = (y-x^2)*(x^2+y^2+z^2-1)*(x-0.5);
f2 = (z-x^3)*(x^2+y^2+z^2-1)*(y-0.5);
f3 = (y-x^2)*(z-x^3)*(x^2+y^2+z^2-1)*(z-0.5);
END
```

Bertini is also capable of handling parameter homotopies. The idea of the parameter homotopy is to move between problems of the same class by adjusting the coefficients as necessary. A linear homotopy is just a special type of parameter homotopy, which is why this strange "s=t" line is necessary if specifying an entire homotopy. To remove that requirement, there would need to be another version of the basic tracking functions built specifically to use data from the path variable rather than a parameter, and as may be seen in this chapter and Appendix A, there are already a number of versions of some of these functions!

The following example appears again in Appendix A, but with more explanation. This input file may be used to move from a solved instance of a certain problem class (finding the intersection of two circles) to an unsolved instance. The main points of interest for this example are the use of constants to define parameters which depend on t and the appearance of multiple parameters in the function. Please refer to Appendix A for more details.

```
variable x, y;
function f1, f2;
pathvariable t;
constant xc1Start, yc1Start, xc2Start, yc2Start, r1Start,
r2Start, xc1End, yc1End, xc2End, yc2End, r1End, r2End;
parameter xc1, yc1, xc2, yc2, r1, r2;
xc1Start = -1.0;
yc1Start = 0.0;
r1Start = 3.0;
xc2Start = 1.0;
yc2Start = 0.0;
r2Start = 3.0;
xc1End = 0.0;
yc1End = 0.5;
```

```
r1End = 0.5;
xc2End = 0.0;
yc2End = 1.5;
r2End = 0.5;
xc1 = xc1Start*t+xc1End*(1-t);
xc2 = xc2Start*t+xc2End*(1-t);
yc1 = yc1Start*t+yc1End*(1-t);
yc2 = yc2Start*t+yc2End*(1-t);
r1 = r1Start*t+r1End*(1-t);
r2 = r2Start*t+r2End*(1-t);
f1 = (x-xc1)^2 + (y-yc1)^2 - r1^2;
f2 = (x-xc2)^2 + (y-yc2)^2 - r2^2;
END
```

6.1.2 The *config* file

The second sort of file always needed by Bertini for a path-tracking run is a configuration file. In the configuration file, typically named *config*, the user sets a number of thresholds and constants that effect the quality of the run. These settings may be manipulated to weigh security against speed. Some of the settings relate to only one sort of run (e.g., runs using the power series endgame) while others are always needed. Please note that the length of the configuration file is expanded frequently, so one should always check the examples released with the latest version of Bertini for the appropriate format. It is unfortunate that *config* currently has such a fixed, user-unfriendly format; this is an artifact of the way in which Bertini was developed. Eventually, Bertini will be changed so that the *config* file will look like a list of equalities of the form "A = a" where "A" is some string representing a specific setting and "a" is the value of that setting.

The next several lines constitue the contents of a typical configuration file. The only data required are those before the "<-" symbols. All other text is present simply for explanation.

```
1 <-Precision type (0 = machine, 1 = multiple, 2 = adaptive).
96 <-# bits to be used (neglected if previous line = 0).
0 <-Toggle for output to screen (0 = no, 1 = yes).
0 <-Level of output (from -1 (minimal) to 3 (maximal)).
3 <-# of successful steps after which step size is increased.
3 <-Maximum number of Newton iterations.
0.1 <-Maximum (and starting) step size.
1e-10 <-Minimum step size before endgame.
1e-13 <-Minimum step size for endgame.
1e-100 <-Minimum allowed distance from target time.
1e-6 <-Newton tolerance until endgame.
1e-9 <-Newton tolerance for endgame.
1e2 <-Threshold for declaring a path infinite.
0.1 <-Path variable value at which we switch to end game.
0 <-Final path variable value desired.
5000 <-Max number of steps allowed per path.
1 <-Endgame number (1 = simple, 3 = power series, 4 = Cauchy).
2 <-Power series sample factor (for power series endgame).
4 <-Power series max cycle number (for power series endgame).
1.0e-6 <-Final PS tolerance (for power series endgame).
0 <-Toggle for real variety project of [40] and [39].
1.0e-4 <-Threshold for considering imaginary parts to be zero.
```

6.1.3 The *start* file

In addition to the data within the *input* and *config* files, Bertini must also know starting points for each path to be tracked. These are specified in a file typically

called *start*. The first line of this file must contain the number of paths to be followed, after which comes a blank line. After that, each point is listed using one variable per line with real and imaginary parts separated by a single space, with a semicolon at the end of the line, and with different points separated by blank lines. For example, if the starting points for a homotopy are $(1.0 - 2.0 * I, 3.0 + 4.0 * I)$ and $(5.0 + 6.0 * I, -7.0 - 8.0 * I)$, the *start* file would be as follows:

```
2

1.0 -2.0;
3.0 4.0;

5.0 6.0;
-7.0 -8.0;
```

If one wishes to provide only a target system, Bertini will form a homotopy by adding a multihomogeneous start system in the case of zero-dimensional runs or a total degree start system in the case of positive-dimensional runs. It will then solve the start systems and create the *start* file automatically. Thus, in these situations, the user need only specify two files rather than three.

6.1.4 Running Bertini

Bertini is typically run from the command line using the Linux operating system. One may build Bertini on a computer by navigating to the source directory within the Bertini folder and making use of the Makefile there, by typing "make". This will call gcc to compile the source code and link it to the various necessary libraries. Then, to run zero-dimensional Bertini, one may simply create the appropriate *input* and *config* files (and, possibly, the *start* file) and type "./bertini" to

call the executable program. Bertini detects whether m-homogenization is needed by checking the first character of the first line of the *input* file. To run Bertini for positive-dimensional tracking, simply type "./bertini -c" instead. This extra command line argument for running Bertini is somewhat annoying. There are other ways of providing such settings, e.g., in *config*, one of which will be employed to replace "-c" in a future release of Bertini. Finally, if help is needed, type "./bertini -help" for instructions.

6.1.5 Output files

Depending upon the type of run, Bertini produces several different types of output files. Any time tracking takes place, a file named *output* is created. The data reported in this file depends upon the setting of the output level in the *config* file, and this file is created even if the toggle for output to screen is turned on in the *config* file. If the level of output is set to 0, the starting point and ending point of each path will be reported, as will other pertinent information such as the final path variable value, the number of steps taken along the path, and the value of the polynomials evaluated at the ending point.

For zero-dimensional tracking, two other output files of interest are created. One of these, named *raw_solutions*, contains a variety of information about the path, including not only the ending point but also data such as the approximate condition number at the endpoint, the cycle number (if using the power series endgame), and the last computed corrector residual. This file is created as Bertini runs, and the format is opaque for the nonexpert.

After the run, a postprocessor reads in the data of *raw_solutions* and creates the file *refined_solutions*. The latter file contains the same data as the former,

except that it is displayed in a human-readable format. The postprocessor also collects endpoints that agree up to the user-specified final tolerance (from the *config* file) times a factor of ten. In this way, the postprocessor computes the multiplicity of multiple points and reports these data in *refined_solutions*.

The output is somewhat different in the case of a positive-dimensional run. In this case, all data of interest in stored in a file called *cascade_output*. In particular, that file contains information regarding each path tracked (as with *output* in the case of a zero-dimensinal run), monodromy, linear trace tests, and so on. Also, a list of all witness points and their vital statistics (e.g., condition numbers), a table summarizing the results of all paths tracked and endpoints classified, and a list of all components in all dimensions (including witness point sets) is displayed on the standard output following a positive-dimensional run.

Bertini also produces a number of other files which store data needed for the run as well as several other forms of output. For example, the straight-line program of the input file is stored in one such intermediate file. However, these files are not generally human-readable, so they will not be discussed presently.

6.2 Data types and linear algebra

Of the available computational mathematics software packages (e.g., Maple and Matlab) and libraries (e.g., Linpack and LAPack), only Maple supports multiprecision floating point arithmetic. However, to maximize control over various other aspects of the software, it was decided to write Bertini without relying on such existing software platforms. As a result, it was necessary to build basic data types and linear algebra subroutines, which are the topic of this section.

6.2.1 Basic data types

The GMP-based library called MPFR provides data types for arbitrary precision floating point numbers as well as basic declaration, assignment, and arithmetic functions. The basic floating point type for in MPFR is the mpf_t data type, which contains a field for the precision of the instance and a several fields for the data itself. Using MPFR, bits may only be assigned in packets of 32. When increasing the level of precision of a variable of type mpf_t, the added bits are assigned randomly (i.e., they are not all 0 as might be expected), so care must be taken in case that is not appropriate.

The use of MPFR data types and functions is very computationally expensive, so it was necessary to have available in Bertini duplicate data types and functions. One set of types and functions (those ending with "d" in the source code) uses entirely basic C double data types while the other set of types and functions (those ending with "mp" in the source code) uses the MPFR types and functions. Although this duality makes code-writing and maintenance somewhat tedious, it is a transparent solution to the need for both types.

The most basic nontrivial data type in Bertini is the complex number, which is a structure consisting of two of the basic floating point type (double or mpf_t), named "r" and "i" for obvious reasons. Vectors (and the identical data type, points) and arrays are then built as structures consisting of an array of complex numbers and one or two integer fields containing the dimension(s) of the instance. Many more advanced structures are included in Bertini. Those types will be introduced in the appropriate sections below and in Appendix A.

Adaptive precision creates a special problem obviously not present with fixed precision runs: Bertini must somehow convert from regular precision types and

functions to multiple precision versions. The solution to this problem used in Bertini is to have both versions of any given function available and using a control function sitting over these two functions. When called, the control function will begin by calling the regular precision version of the function. The regular precision version must be able to recognize when higher precision is needed using, for example, the criteria developed in Chapter 3.

Once the regular precision version of the function at hand runs out of precision, it returns a special error code to the control function. The control function then uses special MPFR functions for copying regular precision data into MPFR data types to convert all necessary data from regular precision to the minimal level (64 bits) of multiple precision. After the conversion of all data to multiple precision types, the control function calls the multiple precision version of the function. That version of the function may then use the criteria to judge whether higher precision is needed, and if so, it can handle the precision increase itself, without reporting back to the control function.

There is at least one major drawback to this way of implementing the functions of numerical algebraic geometry. Every function comes in several versions. For example, for the power series endgame, there is currently a fixed regular precision version, a fixed multiple precision version, a version for regular precision that knows how to watch for precision problems, an analogous version for multiple precision, and a control function sitting over the adaptive precision versions. Whenever a bug is found or an extension is made, the developer must be very careful to correct or extend *every* version of the function!

It should also be noted that when one extends precision in MPFR, the added digits are chosen randomly. Although this may cause concern in some contexts,

Bertini makes use of a refining function based on Newton's method to make certain the extended precision versions of all data types have adequate accuracy immediately following precision increases. Note that this is not a concern with the numerical data in the *input* file! Indeed, integers in *input* are stored exactly, and all floating point numbers are stored as rational numbers with denominators the appropriate factor of ten. Then, when straight-line program evaluation begins, these numbers are computed to the current working precision. The use of rational numbers to represent numerical data forces any digits added beyond the original precision of the input data to be set to zero, working under the assumption that the user is providing exact (or at least as exact as possible) information.

6.2.2 Linear algebra

Although more sophisticated techniques could be employed (e.g., Strassen-like matrix multiplication formulas), Bertini includes naive implementations of arithmetic and norms for matrices and vectors. Simple Gaussian elimination with pivoting is used for matrix inversion, and failure is declared in that subroutine if any pivot is smaller than some threshold that has been hardcoded in the source code as a formula based on precision. As with the basic data types and most other sorts of subroutines, they are two copies of each linear algebraic operation, one in regular precision and one in arbitrary precision using MPFR.

One of the endgames, the power series endgame, makes use of interpolation. Basic Hermite interpolation has been implemented since derivatives are readily available all along the path. These implementations of interpolation allow for an arbitrary number of data points, although the fractional power series endgame generally uses only three data points.

The most sophisticated linear algebra routine in Bertini is the computation of the condition number of a matrix. The singular value decomposition of a matrix yields as a byproduct the condition number of the matrix, which is just the ratio of the largest and the smallest nonzero (interpreted numerically, of course) singular values. An implementation of the numerical computation of the singular value decomposition of a matrix, based on the algorithm found in [58], is included in Bertini, The idea of that algorithm is to reduce a general complex matrix first to a bidiagonal complex matrix via Householder transformations, then to a real bidiagonal matrix using a naive transformation, and finally to a matrix consisting of numerically negligible nondiagonal entries via an iterative process related to QR factorization. This final step is still the source of much research, and various algorithms (going by names such as dqd and dqds) have been developed. [61] is another excellent resource regarding the singular value decomposition.

Although the condition number may be gleaned from the singular value decomposition of a matrix, one might prefer to use only an approximation of the condition number to save computing resources. There are several algorithms that could be used to approximate the condition number of a matrix. Bertini includes an implementation based on an algorithm found in [20] in which one estimates the condition number of a matrix A as the product $||A^{-1}|| \cdot ||A||$, where $||A^{-1}||$ is approximated as the norm of the result of solving $Ax = y$ for x with y a random unit vector. This algorithm generally produces an underestimate of the condition number, although the estimate is frequently within a factor of ten of the true condition number.

6.3 Preprocessing of input files in Bertini

Before any actual tracking is carried out, Bertini passes through a preprocessing stage. It is during this part of the run that Bertini read in all of the data contained within the various input files and converts the polynomials of the *input* file to straight-line format. All such information is currently written to severl files at the end of the preprocessing stage. The main part of Bertini, the tracker, then reads in these files at the beginning of the tracking phase. There is also a short post-processing stage during which the output data is analyzed and summarized in a file called *refined_solutions*. This last phase is very simple and will not be described in any more detail.

The fundamental storage structure for polynomials in Bertini is the straight-line program. A straight-line program of a polynomial (or a function in general) is a list of unary and binary operations that may be used to evaluate the polynomial (or function). For example, the expression $(x + y)^2 * z$ may be broken into the following straight-line program:

$$t_1 = x + y\,,$$
$$t_2 = t_1^2\,,$$
$$t_3 = t_2 * z\,.$$

Then, to evaluate the expression, one may simply substitute in the values of x, y, and z as they are needed. The next section describes several advantages to using straght-line programs. The section after that describes specifically how Bertini converts polynomials in *input* into straight-line programs.

6.3.1 Advantages of using straight-line programs

There are several advantages to using straight-line programs for the representation of polynomial systems. For one, it allows for the use of subfunctions, as described above. Also, it sidesteps the difficulty of expression swell by allowing for non-standard expression representation, e.g., many more operations are needed to evaluate the expansion of $(x+y)^4$ (at least 17 operations) than are needed when evaluating the factored form of the expression (three operations). Finally, the use of straight-line programs makes automatic differentiation and homogenization very easy.

To homogenize a polynomial system given by a straight-line program, one must first record the breakup of the variables into m variable groups and add one new variable to each of the m groups. Then, when parsing the system into a straight-line program (as described in the next section), for each operation, one must simply compare the multidegrees of the operands and multiply in copies of the appropriate new homogenizing variables as needed. For example, if the operands of an addition have multidegrees $[0,1,0]$ and $[1,2,0]$, before recording the straight-line instruction for this operation, one must record two extra instructions corresponding to multiplying the first operand by the first and second homogenizing variables. Note that after a system is homogenized, it is again just a matter of bookkeeping to produce the corresponding start system and to solve this start system in order to produce the set of start points.

Automatic differentiation is also very simple in the straight-line program setting. Please note that automatic differentiation is not the same as numerical or approximate differentiation - the result of automatic differentiation is exact. The idea of automatic differenation is to produce a set of instructions for evaluating

the derivative of a polynomial (or other function) given a straight-line program for the evaluation of the original polynomial (or function). There are many variations on the basic method, but the most basic method (called forward automatic differentiation) is used within a pair of loops to produce instructions for evaluating the Jacobian of a polynomial system in Bertini. To learn more about the state of the art in automatic differentiation, please refer to [26].

After producing the straight-line program for a polynomial system, one may read through the set of instructions, writing down appropriate derivative evaluation instructions for each evaluation instruction. For example, given an instruction $x * y$ where x and y are memory locations, one could produce a set of three instructions for evaluating the derivative of this instruction. In particular, the first instruction would be to multiply the derivative of x (which should already have a memory location set aside) by y, the next would multiply x by the derivative of y, and the third would add these two, storing the result in the appropriate memory location. Naturally, there is a great deal of bookkeeping involved, but that is just a matter of careful implementation. The details of how this is carried out in Bertini may be found in the comments of the source code.

6.3.2 Implementation details for straight-line programs

Before Bertini performs path-tracking techniques or any other methods, it calls a parser to convert the Maple-style, human-readable input file into straight-line program. This preprocessing stage also includes automatic m-homogenization and automatic differentiation, as described in the previous section. Details of how to use lex and yacc (or the variants flex and bison) to parse polynomial systems into straight-line programs and details about the actual straight-line program data

structure in Bertini are given in this section. To learn about the use of lex and yacc in general, please refer to [13].

To use lex and yacc (or their variants), one must produce two files that, together, describe how to convert the input structure into an output structure. For example, these files convert from Maple-style input files to straight-line programs (in the form of a list of integers) in Bertini. The main part of the lex file contains descriptions of the tokens to be detected within the input file. In the case of Bertini, these include tokens such as integers, floating point numbers, string names, and other implementation-specific words such as "VARIABLE" and "END."

The main part of the yacc file consists of an ordered set of rules to which the tokens may be set. There are a variety of nuances in creating such a yacc file, but the main idea is to create a list of unambiguous rules that together completely specify the structure of the input file. For example, the first two rules of the Bertini yacc file specify that the input file should have the structure of a "command" followed by a "command_list" or followed by the word "END." Each command is then a declaration or an assignment, and so on. The main part of this set of rules for Bertini is the definition of an "expression", the right hand side of a polynomial definition. Each expression is a sum of terms, each term is a product of monomials, each monomial is some kind of primary, and each primary is a number, a name, or another expression contained in a pair of parentheses.

Details of the Bertini lex and yacc files may be found in the source code. It should be noted that there are actually multiple parsers within Bertini - one for path tracking without homogenization, one for path tracking with homogenization, and one for the multiplicity algorithm of Chapter 4. In every case, the parsing

stage of the run takes no more than a few hundredths of a second at most, so the use of lex and yacc to parse polynomials into straight-line programs is quite efficient.

Bertini does not only convert from Maple-style input to straight-line programs. Consider the example of $f = (x + y)^2 * z$ given above. By moving through the yacc rules, Bertini would first recognize $x + y$. Since x and y are variables that were presumably already declared before this statement, they have already been assigned addresses in an "evaluation array" of the complex numbers type. A few constants like 0, 1, and $i = \sqrt{-1}$ are so important that they are given the first few addresses in this array, after which all other names and numbers that appear in the input file are given addresses. Assuming f, x, y, and z were already declared before this assignment statement, the evaluation array would look like:

index	0	1	2	3	4	5	6
name	i	0	1	f	x	y	z

The first instruction is then $t_1 = x + y$, written "+ t_1 x y" for convenience. This is then converted into address notation as "+ 7 4 5", i.e., the sum of the values in memory locations 4 and 5 is stored in memory location 7. When the 2 is encountered, it is also assigned an address in the evaluation array. In the end, the array is as follows, where IR means "intermediate result":

index	0	1	2	3	4	5	6	7	8	9	10
name	i	0	1	f	x	y	z	IR	2	IR	IR

To save work, all operations are assigned integer codes, so, for example, "+ 7 4 5" becomes "0 7 4 5." Table 6.1 lists all currently supported operations and their corresponding codes. Other operations, such as trigonometric functions of constants, are very simple to implement and will be included very soon.

TABLE 6.1

LEGAL OPERATIONS IN BERTINI

Operation	Code
+	0
- (binary)	1
*	2
/ (second operand must be numeric)	3
^ (second operand only must be integer)	4
=	5
- (unary)	6

Thus, the Bertini-style straight-line program for the current example would be "0 7 4 5 4 9 7 8 2 10 9 6 5 3 10." If m-homogenization was being carried out during parsing, Bertini would simply compare the degrees of each operand while parsing and add extra instructions corresponding to multiplication by a homogenizing variable, as necessary. Once the straight-line program for a polynomial system has been created, a function called diff reads through the straight-line program one instruction at a time, creating instructions for the Jacobian of the system, as described above. Finally, Bertini concatenates the two straight-line programs to make one complete straight-line program for computing the function and derivative values of the polynomial system.

6.4 Structures and methods for zero-dimensional solutions

Following the preprocessing stage described in the previous section, Bertini may follow one of several execution paths. One of these is zero-dimensional solving, the subject of this section. After parsing is complete, Bertini next reads in each start point, one at a time. The paths are tracked serially, so only one start point is needed at a time. The evaluation of the system and its Jacobian are of course fundamental, so that is described in the next section. The functions for basic path tracking and a few details regarding endgames are given in the two sections after that.

6.4.1 Evaluation of straight-line programs

The evaluation of straight-line programs in Bertini is of particular interest. Depending upon the setting (e.g., zero-dimensional tracking, positive-dimensional tracking, monodromy, etc.), the user may wish to use the function and derivative values of one or more straight-line programs in several different ways. For example, one might be interested simply in the function values at a point. On the other hand, one might wish to evaluate two different straight-line programs (one for a target system and one for a start system) and combine their values using some homotopy.

Rather than making a large number of separate but nearly identical functions for function evaluation or a large switch statement inside some main evaluation function, Bertini has a general straight-line program evaluation function that may be called from any number of special evaluation functions. This straight-line program evaluator takes in a straight-line program data type and pointers to several Bertini variables (one for function values, one for Jacobian values, etc.).

Bertini then evaluates the straight-line program at the point that was passed in and stores the result in the appropriate variables. Other evaluation functions (such as the one mentioned above for gluing two systems together) then make use of this basic evaluator. The virtues of having a basic straight-line program evaluator and a number of small, specialized evaluation functions are that future developers need not write complete (long) evaluation functions any time a new form of evaluation is needed and, better, basic straight-line program evaluation may be maintained and upgraded without needing to change dozens of different functions. This technique also allows the possibility of using one call to the straight-line program evaluation code for multiple types of evaluation, which is beneficial since function evaluation is generally regarded as one of the most expensive parts of homotopy continuation.

The actual evaluation of the straight-line program is not particularly difficult. Given an instance of the straight-line program data type (which contains a few other fields, such as an integer indicating the total number of instructions and another indicating the size of the evaluation array), Bertini first creates an appropriately-sized array to be used as the evaluation array, of the complex number data type. This array is then populated as much as possible with data passed in from the calling function, e.g., the point, the value of the path variable, all numbers that appear in the input file, etc. Bertini then reads through the integer array of instructions and carries out all of the operations of the straight-line program. Finally, the output data is copied into the appropriate Bertini variables. As described in Section 6.2.1, Bertini will always extend the precision of all numbers in the input file to the current working precision and carry out all operations to that precision.

6.4.2 Basic path following

After preprocessing (and other various actions, such as the reading of the *config* file), control is passed to the user-chosen endgame. The endgame then calls the basic path tracking functions as necessary. A general tracking function accepts the input for path tracking and calls the functions for Euler's method and Newton's method as needed. This general tracking function keep tracks of and changes various settings related to path tracking such as the current steplength. All of these functions come in two versions, one for regular (double) precision and one for higher precision. The case of adaptive precision is more sophisticated and will be described below.

Bertini makes use of several data structures designed for path-tracking. The main three structures hold the straight-line program, various tracking data, and the current path value, parameter values, and path variable value. The straight-line program structure holds not only the list of instructions for system and Jacobian evaluation but also various useful integers, such as the number of instructions needed for function evaluation only, the number of variables and functions, and the current working precision (among other data). The data structure holding tracking data, called "tracker_config_t," contains data from the *config* file as well as some data specific to the run at hand, such as the target value of the path variable. Between the preprocessor and the actual path tracking, the *config* file is read and stored in an instance of this data type. Various settings, such as the target Newton tolerance, are changed by the endgame functions during the path tracking. All of these data types, and most others used in Bertini, are C structs.

6.4.3 Endgames

Endgames are specialized numerical methods used to speed convergence to $t = 0$ in path tracking, especially in the presence of singularities. Bertini offers several different endgames, although each has a different level of functionality. The main two endgames are the 0th order endgame and the fractional power series endgame. The former involves tracking to a user-specified value of t (typically $t = 0.1$) using loose tracking tolerances and then switching to tighter tolerances for the remainder of the path. This switch is just a matter of changing various settings at the endgame function level. This endgame is implemented in both types of fixed precision as well as adaptive precision.

The fractional power series endgame is the only other endgame available in every type of precision. This endgame relies on a special kind of Hermite interpolation and careful bookkeeping in addition to basic path tracking. For details about the endgame, please refer to Section 2.2.2 or [47]. At present, only two sets of path values and derivatives are used to interpolate, although it would be beneficial to use more points or to use path data from both $t > 0$ and $t < 0$. The adaptive precision version of this endgame is somewhat difficult, so it is described in more detail below.

The Cauchy endgame has also been implemented in both kinds of fixed precision but not in adaptive precision. The idea of this endgame is to track in a circle about $t = 0$ using complex path variable values and then apply Cauchy's Integral Formula to approximate the value at $t = 0$. The size of the circle about $t = 0$ is decreased iteratively until the approximation converges as desired. Since the target value of t is specified by the endgame, this endgame is just a matter of choosing equally-spaced complex t values around $t = 0$, tracking from one to

the next, and applying the formula. Bertini is currently set to use eight values of t about $t = 0$, although that number is easily changed. As the Cauchy and power series endgames have similar convergence properties, it was decided not to implement the Cauchy endgame in adaptive precision.

6.5 Adaptive precision in Bertini

Due to the dichotomous nature of Bertini (forced by the regular precision and more than regular precision versions of all data types and functions), adaptive precision is more difficult to implement than might be anticipated. If MPFR data types were used throughout (i.e., if Bertini made use of MPFR data types for regular precision), adaptive precision would just be a matter of changing precision levels, but it was decided that it is worthwhile to retain C double types and functions since MPFR is so computationally expensive. Thus, to implement adaptive precision, it is necessary to have some sort of control structure combining the two kinds of data types and functions. Bertini begins with regular precision types and functions and then switches to MPFR types and functions when necessary, after which further precision changes are simple. The actual changing from regular precision data types and functions to multiple precision types and functions is described in Section 6.2.1.

Since endgames control the flow of data for path tracking, the key to implementing adaptive precision path tracking in Bertini is the implementation of adaptive precision endgames. For the 0th order endgame, there is a main control function and two tracking functions, one for regular precision and one for higher precision. The idea is that the control function may call the fixed regular precision tracker which tracks until either the precision needs to be increased or the path

ends successfully. Once higher precision is needed, the regular precision tracker returns a special value to the control function which then calls the multiple precision tracker to finish the path. To know when higher precision is needed, the basic Euler and Newton functions of Bertini were augmented with extra code to check the inequalities discussed in Chapter 3. These inequalities are checked only if adaptive precision is being used, so there is no loss in the fixed precision setting.

Three new functions were used to implement the adaptive precision version as well, although the control structure is somewhat different. Again, there is a control function and two endgame functions (one for regular precision and one for higher precision.), but this time, the control function does not simply hand the control to the multiple precision version of the endgame function once higher than regular precision is needed. The reason for this difference between the implementations of the two endgames is technical in nature. Basically, the adaptive precision endgame is so sophisticated and involves so many more data structures than the 0th order endgame that many variables need to be changed when increasing precision after the first increase. This is very difficult without relying on the control function, so every time a higher level of precision is needed, the multiple precision version of the endgame returns a special code to the control function, after which the control function increases the precision of the appropriate variables and once again calls the multiple precision version of the endgame. For more details, please refer to the source code.

6.6 Structures and methods for positive-dimensional solutions

The cascade algorithm for producing the numerical irreducible decomposition of a solution set of a polynomial system necessitates a number of special structures.

The most fundamental of these is the "cascadeData" data type which holds an array of "levelData" data types (one entry for each codimension) as well as a pointer to an instance of a "cascade_eval_data" data type. The last of these types is small relative to "levelData"; it holds the straight-line program of the original polynomial system given in the *input* file as well as a number of constants, vectors, and matrices that are necessary for the entire cascade (not just one stage). Please refer to Appendix A for details regarding these data types.

For example, letting f denote the target system, Bertini produces the following homotopy to decompose f:

$$\begin{aligned}
h_1(x_1,...,x_N) &= f_1(x_1,...,x_N) + \sum_{i=1}^{N-1} a_{1i}t_iL_i(x_1,...,x_N) \\
h_2(x_1,...,x_N) &= f_2(x_1,...,x_N) + \sum_{i=1}^{N-1} a_{2i}t_iL_i(x_1,...,x_N) \\
&... \\
h_1(x_1,...,x_N) &= f_1(x_1,...,x_N) + \sum_{i=1}^{N-1} a_{Ni}t_iL_i(x_1,...,x_N)
\end{aligned}$$

where

$$L_j(x_1,...,x_N) = \sum_{j=1}^{N} b_jx_j$$

and the a_j and b_j are random complex numbers. The cascade algorithm is then a matter of beginning with all t_i set to 1 and successively making the last nonzero t_i the path variable and tracking it to $t_i = 0$ (via special evaluation functions, as describe above). This homotopy may be stored as a straight-line program for f and a set of three matrices, A, B, and T, where A and B hold the obvious random numbers and T is diagonal and holds the t_i. The homotopy is then easily evaluated at a point $\bar{x}$ as $h(\bar{x}) = f(\bar{x}) + ATB\bar{x}$. The three matrices, as well as

information such as the current codimension, are stored in "cascade_eval_data."

Bertini collects trace information by moving all nonsingular, finite endpoints of each witness point set through certain homotopies. In particular, for a pure-dimensional witness set, Bertini constructs a homotopy that moves the slice used to obtain the witness set towards two randomly chosen slices, both of which are parallel to the original slice. The path data at these three slices (the original slice plus the two randomly chosen slices) is then evaluated in a randomly chosen linear, and a simple formula may be used to compute a trace for each path. If the traces of a subset of endpoints sum to zero, those endpoints constitute one or more complete witness sets. If one such subset is as small as possible (i.e., no subset of the subset has a trace sum of zero), it is a witness set for a single irreducible component. Thus, one way to perform the pure-dimensional breakpoint is to compute this trace information and then combinatorially test every possible combination of witness points (beginning with single points, then pairs, and so on) until all points have been classified. Although this is an efficient algorithm if there are few points to be classified, monodromy, described next, is far superior if there are many points to consider.

Monodromy is performed by moving the slice to a randomly chosen slice, and then moving the slice back via various paths through the parameter space. Two points are known to lie on the same component if they can be connected with a monodromy loop. The details of how these pieces of information are organized may be found in Appendix A.

After all paths for a given codimension have been run, Bertini tries to classify the endpoints by first sorting out the "junk" points (those lying on higher dimensions). This is done as described in Chapter 2. Once the junk has been removed

for the given codimension, Bertini tries to sort the points into components using a user-specified number of monodromy loops. While executing monodromy, if Bertini finds that some number of points lie on the same component, the trace data for those points is combined to see if all points on that component have been found. A sum of zero indicates that the classification of that component is complete.

If monodromy ends without having accounted for all endpoints in the given codimension, Bertini cycles through all possible sets of points, checking to see if any yield a trace sum of zero, as described above. If there are still unclassified witness points remaining after this combinatorial application of the trace test, Bertini reports an error. Otherwise, it moves on to the next codimension.

6.7 Implementation specifics for the ODE method

The two-point boundary value problem method of Chapter 5 has been implemented using Bertini in conjunction with the computer algebra system Maple. The nonlinearity and boundary data are entered into a Maple worksheet which then produces homotopies and calls Bertini for path tracking when necessary. Since the output of Bertini must be read back into Maple for processing, Bertini produces a special (simple) form of output for this algorithm. The output of the algorithm itself is displayed in the Maple worksheet.

The only difficulty with using Maple in Bertini in this way is in calling Bertini from Maple. However, Maple provides a "system()" function that executes from the command line any argument provided to the function. For example, "system(./bertini)" runs Bertini from within Maple. This special interface with Maple makes Bertini much more powerful than if it were an entirely separate program.

6.8 Implementation specifics for the multiplicity algorithm

The multiplicity algorithm of Chapter 4 has also been implemented in Bertini. However, the nature of the computations for that algorithm are so different than those for path tracking that a nearly separate module was built specifically for that algorithm. The most significant difference in this special *find_mult* module is the representation of polynomials. Since every polynomial is required to be homogeneous in the input file, the polynomials are stored in vectors with one entry for each monomial of the degree of the polynomial. For example, using three variables, there are 15 possible monomials in degree four, so the coefficients of a polynomial of that degree would be stored in a vector of size 15. Naturally, a monomial ordering must be fixed. This is accomplished by having a fixed algorithm for producing the monomial basis for a degree based on the basis from one degree less.

Once the input polynomials and point are stored in vector form, the computations needed by this algorithm amount to a variety of symbolic and linear algebra operations. For example, to expand a polynomial to a higher degree, one must multiply the polynomial by each monomial of the appropriate degree and store the results in an array of vectors. Polynomial multiplication is simple: given the vectors corresponding to two polynomials, one may multiply them by finding each nonzero entry in the first polynomial (one at a time) and multiplying that entry by each nonzero entry of the second polynomial, storing the product in the entry of the resulting polynomial's vector corresponding to the appropriate monomial. Since monomials may be represented by integer arrays containing the degrees of the monomial, finding the monomial that is the product of two others is just a matter of adding the degree arrays. Bertini has functions for identifying the mono-

mial represented by a degree array and vice versa, so such symbolic polynomial manipulations are really quite simple.

As Bertini descends through the main loop of the *find_mult* algorithm, it produces the coefficients of the Hilbert function of the ideal, which it prints to the screen. Other pertinent information is also displayed during the run. At the end of the algorithm, Bertini displays the computed multiplicity and the computed bound on the regularity. As opposed to the path tracking methods described above, there is no output file for the *find_mult* algorithm.

APPENDIX A

MORE DETAILS REGARDING BERTINI

A.1 Complete sets of Bertini input and output files

A.1.1 The example

Suppose one is interested in computing the intersection of two circles in the plane, not just once, but over and over for different circles, i.e., different centers and radii. This is of course a very simple problem, but that is exactly what makes it a good example of how to use homotopy continuation in general and Bertini in particular. This problem is also discussed in [56] in relation to the software package HomLab.

In the homotopy continuation setting, the idea is to first solve one problem of the type (hopefully one that is easy to solve). After solving this initial system, one simply builds a homotopy from the initial system to any other target system. This is known as a parameter homotopy. Bertini input and output files for the first step, solving the initial system, are provided in the next section, while the input and output files for a particular parameter homotopy are given in the subsequent section. It should be noted that since Bertini is still a work in progress, the exact structure of these files will likely change over time, especially the output files.

A.1.2 The initial system, using homogenization

Here is an input file for solving the initial system:

```
mhom 1;
variable_group x, y;
function f1, f2;
pathvariable t;
constant xc1Start, yc1Start, xc2Start, yc2Start, r1Start,
r2Start;
xc1Start = -1.0;
yc1Start = 0.0;
r1Start = 3.0;
xc2Start = 1.0;
yc2Start = 0.0;
r2Start = 3.0;
f1 = (x-xc1Start)^2 + (y-yc1Start)^2 - r1Start^2;
f2 = (x-xc2Start)^2 + (y-yc2Start)^2 - r2Start^2;
END
```

The constants, listed on line 5, are the two coordinates for the first circle, the two coordinates for the second circle, and the radii of the two circles. The constants are set to describe circles of radius 3 with centers $(-1, 0))$ and $(1, 0)$ in lines 6 through 11. Since the system (given in lines 12 and 13) has two degree two polynomials, there will be four paths to follow.

This is an example of an m-homogeneous *input* file. Bertini will automatically homogenize the target system (found in the two lines prior to the word "END") and create an appropriate homotopy. This problem could just as easily be solved using its original, nonhomogeneous form with a different homotopy, such as a total degree start system attached with the usual linear homotopy, although that would involve tracking roots going to infinity, which can be costly.

Many settings of the *config* file never come into play. For instance, for a run in which all paths end at nonsingular, finite endpoints, neither the minimum step sizes nor the threshold for declaring a path infinite should matter. The settings of those quantities in the following *config* file should not therefore be taken to be set that way for any particular reason. One could of course find optimal values for such tolerances by trial and error, but that is beside the point since the goal of this part of Appendix A is to provide the files for a common run. Here is the *config* file that was used to run the above *input* file:

```
0 <-Precision type (0 = machine, 1 = fixed multi, 2 = adaptive).
64 <-# of bits to be used (neglected if previous line = 0).
0 <-Toggle for output to screen (0 = no, 1 = yes).
0 <-Level of output (from -1 (minimal) to 3 (maximal)).
3 <-# of successful steps after which step size is increased.
3 <-Maximum number of Newton iterations.
0.1 <-Maximum (and starting) step size.
1e-15 <-Minimum step size before endgame.
1e-40 <-Minimum step size for endgame.
1e-50 <-Minimum allowed distance from target time.
1e-5 <-Newton tolerance until endgame.
1e-8 <-Newton tolerance for endgame.
1e4 <-Threshold for declaring a path infinite.
0.1 <-Path variable value at which we switch to end game.
0 <-Final path variable value desired.
10000 <-Max number of steps allowed per path.
1 <-Endgame number (1 = simple, 3 = power series, 4 = Cauchy).
2 <-Power series sample factor (for power series endgame).
4 <-Power series max cycle number (for power series endgame).
1.0e-12 <-Final PS tolerance (for power series endgame).
0 <-Toggle for real variety project of [40] and [39].
1.0e-4 <-Threshold for considering imaginary parts to be zero.
```

As mentioned above, no *start* file needs to be created by the user for this run as Bertini will automatically produce the start points. The output files of Bertini are not terribly attractive, although there are several from which the user can choose. The two most human-readable output files are named *output* and *refined_solutions*, although *output* contains far too much data to be recorded presently, even for this small problem and even with the level of data set to the minimum in the *config* file. Here is the file named *refined_solutions* that was produced during this run:

```
-------------------------
Solution 0 (path number 0)
Estimated condition number:  2.223731422424316e+01
Function residual:  1.730069135810763e-15
Latest Newton residual:  1.092259441047363e-16
T value at final sample point:  0.000000000000000e+00
Cycle number:  0
HOM_VAR_0; 2.504541277885437e-01 -1.614952385425568e-01
x; 1.130340735410548e-17 -2.123611610072467e-19
y; 7.083912491798401e-01 -4.567775130271912e-01
Paths with the same endpoint, to the prescribed tolerance:
Multiplicity:  1
-------------------------
Solution 1 (path number 1)
Estimated condition number:  3.605774307250977e+01
Function residual:  1.052013332303065e-15
Latest Newton residual:  1.170691383240661e-16
T value at final sample point:  0.000000000000000e+00
Cycle number:  0
HOM_VAR_0; -1.803561942861838e-17 -3.582913963555473e-17
x; -1.841393411159515e-01 -4.240295290946960e-01
y; 4.240295290946960e-01 -1.841393411159515e-01
Paths with the same endpoint, to the prescribed tolerance:
```

```
Multiplicity:  1
-------------------------
Solution 2 (path number 2)
Estimated condition number:  3.092556571960449e+01
Function residual:  5.668214886984848e-15
Latest Newton residual:  1.350278514624130e-15
T value at final sample point:  0.000000000000000e+00
Cycle number:  0
HOM_VAR_0; 4.984982123982650e-17 -1.540749291460877e-17
x; 2.372493028640747e+00 2.534117698669434e-01
y; 2.534117698669434e-01 -2.372493028640747e+00
Paths with the same endpoint, to the prescribed tolerance:
Multiplicity:  1
-------------------------
Solution 3 (path number 3)
Estimated condition number:  4.990838623046875e+01
Function residual:  2.383810117413698e-15
Latest Newton residual:  1.045248913951402e-16
T value at final sample point:  0.000000000000000e+00
Cycle number:  0
HOM_VAR_0; -2.484529614448547e-01 1.530467569828033e-01
x; 8.498945785834275e-18 1.783280632294895e-17
y; 7.027310729026794e-01 -4.328815937042236e-01
Paths with the same endpoint, to the prescribed tolerance:
Multiplicity:  1
-------------------------
At tol=1.000000000000000e-07, there appear to be 4 solutions.
```

For each solution (to the prescribed tolerance), this file records the most recent estimate of the condition number of the Jacobian as well as several other pieces of information that might be of interest to the expert. Note that the cycle number is

only computed if using the power series endgame. After the technical data come the coordinates of the endpoint, beginning with any homogeneous coordinates added on by Bertini. Finally, the path numbers of the other paths ending at the same endpoint (to the prescribed tolerance) are listed and the number of such paths is listed as "Multiplicity." Note that this is not in fact the multiplicity computed by the algorithm of Chapter 4 or any other sophisticated estimate of the multiplicity. It is simply the number of paths leading to the same endpoint.

In this case, there are two solutions (solutions 1 and 2) for which the homogeneous coordinate is zero, meaning that they are at infinity in affine space. The other two solutions (solutions 0 and 3) are clearly finite and and approximates of the roots $\left(0, \pm\sqrt{8}\right)$. To compute the affine coordinates, one can simply divide each non-homogeneous coordinate by the homogeneous coordinate of the same variable group. The affine coordinates are currently displayed in *output*, but they have not yet been incorporated into *refined_solutions*.

A.1.3 Subsequent systems, using parameter homotopies

Given the solution to one system of this class of systems, it is easy to solve other systems in the same class via a parameter homotopy. The idea is to set up parameters depending on the path variable that define a path through the parameter space from the solved system to the target system. Here is an *input* file for doing just that in Bertini for this example:

```
variable x, y;
function f1, f2;
pathvariable t;
constant xc1Start, yc1Start, xc2Start, yc2Start, r1Start, //
r2Start, xc1End, yc1End, xc2End, yc2End, r1End, r2End;
parameter xc1, yc1, xc2, yc2, r1, r2;
```

```
xc1Start = -1.0;
yc1Start = 0.0;
r1Start = 3.0;
xc2Start = 1.0;
yc2Start = 0.0;
r2Start = 3.0;
xc1End = 0.0;
yc1End = 0.5;
r1End = 0.5;
xc2End = 0.0;
yc2End = 1.5;
r2End = 0.5;
xc1 = xc1Start*t+xc1End*(1-t);
xc2 = xc2Start*t+xc2End*(1-t);
yc1 = yc1Start*t+yc1End*(1-t);
yc2 = yc2Start*t+yc2End*(1-t);
r1 = r1Start*t+r1End*(1-t);
r2 = r2Start*t+r2End*(1-t);
f1 = (x-xc1)^2 + (y-yc1)^2 - r1^2;
f2 = (x-xc2)^2 + (y-yc2)^2 - r2^2;
END
```

The constants of the *input* file from the previous section appear again in this file. However, there are also analogous constants for the target circles. In particular, this parameter homotopy is moving to the case of circles of radius 0.5 with centers at $(0, 0.5)$ and $(0, 1.5)$. The parameters $xc1$ through $r2$, declared on line 5 and defined on lines 19 through 24, just move the corresponding constants through linear homotopies.

The *config* file of the previous section was used again for this example. That will often be the case in practice; there is little to change in the *config* file between similar runs. However, since this run is neither m-homogeneous or positive-dimensional, Bertini does not provide the file *start*. Since this is a parameter homotopy and the initial system was solved in the previous section, the endpoints from that run may be used as the start points for this run. Note that only the finite solutions are being followed since affine space is now being used. It would not be hard to use a homogeneous form of the homotopy and all four points, but since the finite coordinates in affine space are of most interest, that is all that was used for this run. Here is the *start* file for this run:

```
2

0.0 0.0;
-2.828427 0.0;

0.0 0.0;
2.828427 0.0;
```

The ending circles for this run meet in one double point, so the condition numbers of the paths swell near the endpoints of the paths. Both solutions are computed by Bertini and reported in *output*, after which the solutions are compared and reported together in *refined_solutions*, which is displayed below.

```
-------------------------
Solution 0 (path number 0)
Estimated condition number:  7.327159680000000e+08
Function residual:  0.000000000000000e+00
Latest Newton residual:  0.000000000000000e+00
T value at final sample point:  0.000000000000000e+00
```

```
Cycle number:  0
x; 9.458371863502180e-10 0.000000000000000e+00
y; 1.000000000000000e+00 0.000000000000000e+00
Paths with the same endpoint, to the prescribed tolerance:
1
Multiplicity:  2
-------------------------
At tol=1.000000000000000e-07, there appears to be one solution.
```

It should be noted that when these files were run a second time using the power series endgame (by changing the endgame number to 3 and the final tolerance for that endgame to 1.0e-8 the *config file*), a cycle number of 2 was computed and displayed in *refined_solutions*.

A.2 Data structures for positive-dimensional solving

The point of this part of Appendix A is to catalog the various data types in Bertini for the computation of the numerical irreducible decomposition via a cascade homotopy. Since there is much that could be said but this information is of interest only to the expert wishing to better understand this part of Bertini, the descriptions given below are kept brief. Unfortunately, the comments from the source code have been removed due to space limitations. Please refer to the Bertini user's manual or the documentation in the source code or contact one of the main developers of Bertini (D. Bates, A. Sommese, or C. Wampler) if further information is desired.

There are three main data types at the highest level. The main data type is cascadeData_d. It contains one copy of each of the other two main types. Here is the definition of this data structure:

```
typedef struct
{
  levelData_d *paths;
  cascade_eval_data_d *ED;
} cascadeData_d;
```

The levelData_d struct holds the witness superset and witness set information for one step of the cascade algorithm, i.e., for one codimension. This is why levelData_d holds an array of this type, one for each possible codimension. The cascade_eval_data_d type holds the straight-line program for the system from *input* as well as all other data that are needed for tracking through the cascade algorithm. Here is the exact definition of the levelData_d data type:

```
typedef struct
{
  int codim;
  int num_paths;
  point_d *startPts;
  point_d *endPts;
  comp_d *finalTs;
  double *condition_nums;
  int *endPt_retVals;
  int *endPt_types;
  point_d *monHalfPts;
  point_d *monEndPts;
  int num_components;
  int *component_nums;
  int num_tmp_gps;
  int *tmp_gp_nums;
  int num_active_tmp_gps;
  int *active_tmp_gp_nums;
```

```
  int *multiplicities;
  vec_d linear_for_trace_test;
  comp_d *full_traces;
  comp_d *tmp_gp_full_traces;
} levelData_d;
```

The levelData_d structure consists of instances and arrays of several more basic data types. In particular, a comp_d is simply a complex number (a struct consisting of two doubles, one for the real part, one for the imaginary part), a point_d is an array of type comp_d plus an integer indicating the size of the array, and a vec_d is just an alias for a point_d. Bertini also has a few other very basic data types which will be described below. The endPt_types array contains a code for each path indicating the type of end point that it is.

The first few fields of the levelData_d struct are obviously needed. They incdicate the codimension of the data, the number of paths, the set of start points, the set of endpoints, and the final value of the path tracking variable for each path (which is nonzero if the path fails). The struct also contains the final estimated condition number of each path as well as the return value from the tracker for each path. Together, these two pieces of information help to classify the endpoints in several ways, e.g., singular versus nonsingular.

The next two data types contain monodromy information. In particular, Bertini tracks halfway around each monodromy loop and stores the endpoints for that half in monHalfPts. Bertini then tries several different paths to complete each loop, storing the endpoints (to be compared to the original endpoints of the slice) in monEndPts. These data will be overwritten for each different monodromy loop.

The levelData_d data structure also contains a field storing the number of

irreducible components discovered at the given codimension and an array of size num_paths that stores the component number of each witness point. The data types related to "temporary groups" (there are four) are very technical in nature, and one should refer to the source code to better understand their use. The idea is that monodromy results in the grouping of endpoints that may not constitute complete components. These partial components must then be considered together when using the trace test to complete the pure-dimensional decomposition. When Bertini begins the pure-dimensional decomposition of a given codimension, it first stores endpoints with zero traces as complete components and endpoints with nonzero traces as temporary groups (one point per group).

The multiplicities field stores the multiplicity of each endpoint, and the final three fields listed above store trace information. Although coordinate-wise traces provide more security than just the total trace (which is the sum of the coordinate-wise traces), they have not yet been implemented in Bertini.

Here is the specification of the cascade_eval_data_ data type:

```
typedef struct {
  prog_t *Prog;
  mat_d A;
  mat_d B;
  mat_d T;
  int codim;
  int codim_high;
  point_d targetPoint;
  mat_d coeffs_for_junk_removal;
  mat_d rands_for_junk_removal;
  comp_d random_gamma_for_initial_step;
  point_d random_point_for_traces;
  mat_d linear_for_monodromy;
```

```
  comp_d random_gamma_for_outgoing_monodromy;
  comp_d random_gamma_for_incoming_monodromy;
  int *degrees;
} cascade_eval_data_d;
```

The prog_t data type is the straight-line program structure in Bertini, as described in Chapter 6. The mat_d type is the matrix data type. It consists of a two-dimensional array of type comp_d and two integers which store the dimensions of the matrix. Prog stores the straight-line program from *input*.

The matrices A, B, and T are described in Chapter 6. They store the information needed to turn the start system into the cascade homotopy. In particular, codim is changed throughout the cascade algorithm to indicate the current codimension, which in turn indicates which diagonal entries of T are set to 0 and which are set to 1. On the other hand, codim_high stores the maximal codimension for the polynomial system. The field targetPoint is only used when removing junk. The coordinates of targetPoint are used to define a homotopy from the slice of each higher-dimensional witness set to a random slice through targetPoint. The other two fields obviously related to junk removal store the coefficients of the linears and the random numbers for combining the linears with the rest of the homotopy in a generic way. Please refer to [56] or the source code for further details of how to remove junk.

The cascade algorithm begins with a basic homotopy continuation run. Bertini solves at the first stage using a total degree homotopy. It needs a random complex number in the homotopy (the "random gamma trick" of [56]), which is stored in random_gamma_for_initial_step. The degrees of the polynomials are needed to form this total degree homotopy, so they are discovered by the parser and stored in the field named degrees.

The other fields of cascade_eval_data_d are used to gather the trace and monodromy information described in Chapter 6. In particular, random_point_for_traces determines a direction along which to move the slices in parallel to gather trace data. The linear_for_monodromy field stores the target for the first half of the monodromy loop and the other two fields store the "random gammas" used to form the homotopy in each direction. A number of these data types are overwritten repeatedly throughout the cascade algorithm.

REFERENCES

1. William W. Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 1994.

2. Götz Alefeld and Jürgen Herzberger. *Introduction to interval computations*. Computer Science and Applied Mathematics. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], New York, 1983. Translated from the German by Jon Rokne.

3. Eugene L. Allgower. On a discretization of $y'' + \lambda y^k = 0$. In *Topics in numerical analysis,* II *(Proc. Roy. Irish Acad. Conf., Univ. College, Dublin, 1974)*, pages 1–15. Academic Press, London, 1975.

4. Eugene L. Allgower. A survey of homotopy methods for smooth mappings. In *Numerical solution of nonlinear equations (Bremen, 1980)*, volume 878 of *Lecture Notes in Math.*, pages 1–29. Springer, Berlin, 1981.

5. Eugene L. Allgower, Daniel J. Bates, Andrew J. Sommese, and Charles W. Wampler. Solution of polynomial systems derived from differential equations. *Computing*, 76(1):1–10, 2006.

6. Eugene L. Allgower and Kurt Georg. *Introduction to numerical continuation methods*, volume 45 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2003. Reprint of the 1990 edition [Springer-Verlag, Berlin; MR1059455 (92a:65165)].

7. M. F. Atiyah and I. G. Macdonald. *Introduction to commutative algebra*. Addison-Wesley Publishing Co., Reading, Mass.-London-Don Mills, Ont., 1969.

8. Raymond G. Ayoub. Paolo Ruffini's contributions to the quintic. *Arch. Hist. Exact Sci.*, 23(3):253–277, 1980/81.

9. Daniel Bates, Chris Peterson, and Andrew Sommese. A numeric-symbolic algorithm for computing the multiplicity of a component of an algebraic set. Manuscript, 2006.

10. Daniel Bates, Andrew Sommese, and Charles Wampler. Multiprecision path tracking. Manuscript, 2006.

11. Dave Bayer and David Mumford. What can be computed in algebraic geometry? In *Computational algebraic geometry and commutative algebra (Cortona, 1991)*, Sympos. Math., XXXIV, pages 1–48. Cambridge Univ. Press, Cambridge, 1993.

12. David Bayer and Michael Stillman. A criterion for detecting m-regularity. *Invent. Math.*, 87(1):1–11, 1987.

13. Doug Brown, John Levine, and Tony Mason. *lex & yacc (A Nutshell Handbook)*. O'Reilly Media, Sebastopol, CA, 1992.

14. Lothar Collatz. *Differentialgleichungen. Eine Einführung unter besonderer Berücksichtigung der Anwendungen*. Dritte überarbeitete und erweiterte Auflage. Leitfäden der angewandten Mathematik und Mechanik, Band 1. B. G. Teubner, Stuttgart, 1966.

15. David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms: An introduction to computational algebraic geometry and commutative algebr*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, second edition, 1997.

16. David Cox, John Little, and Donal O'Shea. *Using algebraic geometry*, volume 185 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1998.

17. D. F. Davidenko. On a new method of numerical solution of systems of nonlinear equations. *Doklady Akad. Nauk SSSR (N.S.)*, 88:601–602, 1953.

18. Harold T. Davis. *Introduction to nonlinear differential and integral equations*. Dover Publications Inc., New York, 1962.

19. Barry H. Dayton and Zhonggang Zeng. Computing the multiplicity structure in solving polynomial systems. Unpublished manuscript, 2005.

20. James W. Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

21. David Eisenbud and Shiro Goto. Linear free resolutions and minimal multiplicity. *J. Algebra*, 88(1):89–133, 1984.

22. David Eisenbud, Daniel R. Grayson, and Michael Stillman, editors. *Computations in algebraic geometry with Macaulay 2*, volume 8 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 2002.

23. William Fulton. *Algebraic curves. An introduction to algebraic geometry*. W. A. Benjamin, Inc., New York-Amsterdam, 1969. Notes written with the collaboration of Richard Weiss, Mathematics Lecture Notes Series.

24. Kurt Georg. Improving the efficiency of exclusion algorithms. *Adv. Geom.*, 1(2):193–210, 2001.

25. Gert-Martin Greuel and Gerhard Pfister. *A* **Singular** *introduction to commutative algebra*. Springer-Verlag, Berlin, 2002. With contributions by Olaf Bachmann, Christoph Lossen and Hans Schönemann, With 1 CD-ROM (Windows, Macintosh, and UNIX).

26. Andreas Griewank. *Evaluating derivatives: Principles and techniques of algorithmic differentiation*, volume 19 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.

27. Andreas Griewank and Michael R. Osborne. Analysis of Newton's method at irregular singularities. *SIAM J. Numer. Anal.*, 20(4):747–773, 1983.

28. Phillip Griffiths and Joseph Harris. *Principles of algebraic geometry*. Wiley Classics Library. John Wiley & Sons Inc., New York, 1994. Reprint of the 1978 original.

29. Robin Hartshorne. *Algebraic geometry*. Graduate Texts in Mathematics, No. 52. Springer-Verlag, New York, 1977.

30. Jens Høyrup. The Babylonian cellar text BM 85200+ VAT 6599. Retranslation and analysis. In *Amphora*, pages 315–358. Birkhäuser, Basel, 1992.

31. Thomas W. Hungerford. *Algebra*, volume 73 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1980. Reprint of the 1974 original.

32. R. Baker Kearfott. Empirical evaluation of innovations in interval branch and bound algorithms for nonlinear systems. *SIAM J. Sci. Comput.*, 18(2):574–594, 1997.

33. Herbert B. Keller. *Numerical methods for two-point boundary value problems*. Dover Publications Inc., New York, 1992. Corrected reprint of the 1968 edition.

34. Martin Kreuzer and Lorenzo Robbiano. *Computational commutative algebra. 1*. Springer-Verlag, Berlin, 2000.

35. Martin Kreuzer and Lorenzo Robbiano. *Computational commutative algebra. 2*. Springer-Verlag, Berlin, 2005.

36. Theodore Laetsch. On the number of solutions of boundary value problems with convex nonlinearities. *J. Math. Anal. Appl.*, 35:389–404, 1971.

37. Anton Leykin, Jan Verschelde, and Ailing Zhao. Evaluation of jacobian matrices for newton's method with deflation for isolated singularities of polynomial systems. In *SNC 2005 Proc.*, pages 19–28, 2005.

38. Tien-Yien Li. Numerical solution of multivariate polynomial systems by homotopy continuation methods. In *Acta numerica, 1997*, volume 6 of *Acta Numer.*, pages 399–436. Cambridge Univ. Press, Cambridge, 1997.

39. Ye Lu, Daniel Bates, Andrew Sommese, and Charles Wampler. Computing all real solutions of polynomial systems: II the surface case. Manuscript, 2006.

40. Ye Lu, Andrew Sommese, and Charles Wampler. Computing all real solutions of polynomial systems: I the curve case. Manuscript, 2006.

41. F. S. Macaulay. *The algebraic theory of modular systems*. Cambridge Mathematical Library. Cambridge University Press, Cambridge, 1994. Revised reprint of the 1916 original, With an introduction by Paul Roberts.

42. I. G. Macdonald. *Algebraic geometry. Introduction to schemes*. W. A. Benjamin, Inc., New York-Amsterdam, 1968.

43. Alexander Morgan. *Solving polynomial systems using continuation for engineering and scientific problems*. Prentice Hall Inc., Englewood Cliffs, NJ, 1987.

44. Alexander Morgan and Andrew Sommese. Computing all solutions to polynomial systems using homotopy continuation. *Appl. Math. Comput.*, 24(2):115–138, 1987.

45. Alexander P. Morgan, Andrew J. Sommese, and Charles W. Wampler. Computing singular solutions to nonlinear analytic systems. *Numer. Math.*, 58(7):669–684, 1991.

46. Alexander P. Morgan, Andrew J. Sommese, and Charles W. Wampler. Computing singular solutions to polynomial systems. *Adv. in Appl. Math.*, 13(3):305–327, 1992.

47. Alexander P. Morgan, Andrew J. Sommese, and Charles W. Wampler. A power series method for computing singular solutions to nonlinear analytic systems. *Numer. Math.*, 63(3):391–409, 1992.

48. David Mumford. *Lectures on curves on an algebraic surface*. With a section by G. M. Bergman. Annals of Mathematics Studies, No. 59. Princeton University Press, Princeton, N.J., 1966.

49. David Mumford. *Algebraic geometry. I.* Classics in Mathematics. Springer-Verlag, Berlin, 1995. Complex projective varieties, Reprint of the 1976 edition.

50. Sanford M. Roberts and Jerome S. Shipman. *Two-point boundary value problems: shooting methods.* American Elsevier Publishing Co., Inc., New York, 1972. Modern Analytic and Computational Methods in Science and Mathematics, No. 31.

51. Jean-Pierre Serre. Faisceaux algébriques cohérents. *Ann. of Math. (2)*, 61:197–278, 1955.

52. Andrew J. Sommese and Jan Verschelde. Numerical homotopies to compute generic points on positive dimensional algebraic sets. *J. Complexity*, 16(3):572–602, 2000. Complexity theory, real machines, and homotopy (Oxford, 1999).

53. Andrew J. Sommese, Jan Verschelde, and Charles W. Wampler. Numerical decomposition of the solution sets of polynomial systems into irreducible components. *SIAM J. Numer. Anal.*, 38(6):2022–2046 (electronic), 2001.

54. Andrew J. Sommese, Jan Verschelde, and Charles W. Wampler. Numerical irreducible decomposition using projections from points on the components. In *Symbolic computation: solving equations in algebra, geometry, and engineering (South Hadley, MA, 2000)*, volume 286 of *Contemp. Math.*, pages 37–51. Amer. Math. Soc., Providence, RI, 2001.

55. Andrew J. Sommese, Jan Verschelde, and Charles W. Wampler. Homotopies for intersecting solution components of polynomial systems. *SIAM J. Numer. Anal.*, 42(4):1552–1571 (electronic), 2004.

56. Andrew J. Sommese and Charles W. Wampler, II. *The numerical solution of systems of polynomials arising in engineering and science.* World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, 2005.

57. Hans J. Stetter. *Numerical polynomial algebra.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2004.

58. G. W. Stewart. *Matrix algorithms. Vol. I: Basic decompositions.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

59. Bernd Sturmfels. *Solving systems of polynomial equations*, volume 97 of *CBMS Regional Conference Series in Mathematics.* Published for the Conference Board of the Mathematical Sciences, Washington, DC, 2002.

60. Françoise Tisseur. Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 22(4):1038–1057 (electronic), 2001.

61. Lloyd N. Trefethen and David Bau, III. *Numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

62. Jan Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM T. Math. Software*, 25(2):251–276, 1999.

63. David S. Watkins. *Fundamentals of matrix computations*. John Wiley & Sons Inc., New York, 1991.

64. James H. Wilkinson. *Rounding errors in algebraic processes*. Dover Publications Inc., New York, 1994. Reprint of the 1963 original [Prentice-Hall, Englewood Cliffs, NJ; MR0161456 (28 #4661)].