MARS: A Maple/Matlab/C Resultant-Based Solver

Aaron Wallack
Cognex Corporation
1 Vision Drive
Natick MA 01760, USA
awallack@cognex.com

Ioannis Z. Emiris
INRIA
B.P. 93
Sophia-Antipolis 06902, France
emiris@sophia.inria.fr
http://www.inria.fr/safir/emiris

Dinesh Manocha
Department of Computer Science, UNC
Chapel Hill NC 27599-3175, USA
dm@cs.unc.edu
http://www.cs.unc.edu/~dm

Abstract

The problem of computing zeros of a system of polynomial equations has been well studied in the computational literature. A number of algorithms have been proposed and many computer algebra and public domain packages provide the capability of computing the roots of polynomial equations. Most of these implementations are based on Gröbner bases which can be slow for even small problems. In this paper, we present a new system, MARS, to compute the roots of a zero dimensional polynomial system. It is based on computing the resultant of a system of polynomial equations followed by eigendecomposition of a generalized companion matrix. MARS includes a robust library of Maple functions for constructing resultant matrices, an efficient library of Matlab routines for numerically solving the eigenproblem, and C code generation routines and a C library for incorporating the numerical solver into applications. We illustrate the usage of MARS on various examples and utilize different resultant formulations.

1 Introduction

Finding the solution of a system of nonlinear polynomial equations over a given field is a classical and fundamental problem in the computational literature which has been extensively studied.

Recently, a great deal of interest in solving nonlinear polynomial systems has come from different applications. It includes computer algebra [Ren92], robotics [MC94, RR95, WC97], computer graphics [Man94], geometric and solid modeling [Hof89, MD95], computer vision [Emi97, WM98], economics and optimization [MM94], and molecular biology [EM96]. The main operations in these applications can be classified into two types. First, the simultaneous elimination of one or more variables from a given set of polynomial equations to obtain a "symbolically smaller" system. This problem arises, for instance, in graphics and modeling applications where the implicit expression of a curve or surface is precisely the resultant polynomial. Second, the computation of all numeric solutions of a system of polynomial equations. Our practical motivation is fast computation of solutions of a zero-dimensional system with 10 variables or less.

Elimination theory, a branch of classical algebraic geometry, investigates the condition under which sets of polynomials have common roots. Its results were known at least a century ago and still appear in modern treatments of algebraic geometry, although often in non-constructive form. The main result is the construction of a single resultant polynomial of n homogeneous polynomial equations in n unknowns, such that the vanishing of the resultant is a necessary but not always sufficient condition for the given system to have a nontrivial solution. This resultant is known as the *multipolynomial resultant* the given system of polynomial equations [AS88, Man94, MC94]. The multipolynomial resultant of the system of polynomial equations can be used for eliminating the variables and computing the numeric solutions of a given system of polynomial equations. The same approach is also valid in the non-homogeneous context, as illustrated later.

Given a zero-dimensional system, the computation of all common solutions can be reduced to an eigenvalue problem. In this resultant-eigendecomposition technique, each eigenvalue corresponds to one variable of a root, and the associated eigenvector characterizes the other variables of the root. This approach is very useful for repeatedly solving similar systems because the symbolic processing is only performed once and numerically solving the system reduces to instantiating coefficients of the resultant matrix followed

 ISSAC'98, Rostock, Germany.

by eigendecomposition.

Main Contributions: The main contribution of our work is a software package consisting of MAPLE, MATLAB and C libraries for solving zero-dimensional systems (a more thorough review can be found in [WEM98]). Given a system, MARS computes the resultant as a matrix polynomial and numerically solving the resultant matrices. MARS simplifies the task of incorporating a numerical multipolynomial solver into a user's application. We present a number of issues in the design and implementation of this library and highlight its performance on a number of examples.

The rest of the paper is organized as follows. The next section discusses alternate approaches, existing implementations and their limitations. Section 3 outlines the main approach in using resultant matrices for reducing system-solving to a problem in linear algebra. In particular, subsection 3.1 mentions the different matrix formulations and how they are constructed, whereas the following subsection shows the matrix operations, typically performed numerically, applied to approximate all common roots. Section 4 describes the main architects of our library and the package's organization, and section 5 discusses implementation details and features of the MARS package. We illustrate the power and adaptability of our library, including the available interfaces and the automatic generation of C code, in section 6 by studying concrete examples. Section 7 reviews the performance and practical complexity of MARS. We summarize and conclude with further work in section 8.

2 Related work

There is a long history of using resultant-based approaches to study and solve systems of polynomial equations. Recently, certain practical results that have established resultants, along with Gröbner bases and continuation techniques, as a method of choice in solving zero-dimensional polynomial systems. For systems of medium size, the applications highlighted earlier illustrate the comparative advantages of resultant-based methods: resultants can strongly exploit polynomial structure, they reduce the nonlinear problem to one in linear algebra, and combine a symbolic with a numeric approach.

Gröbner bases have been studied for a longer time and offer an array of general implementations to efficiently handle zero-dimensional systems. For the purposes of illustration, we mention only very few representatives, namely `GB` [Fau95] and the PoSSo/FRISCO library [FRI97]. Most computer algebra systems, like AXIOM, MATHEMATICA, MAPLE and REDUCE have a package for computing the Gröbner bases of an ideal. One of the main drawbacks of using Gröbner bases is that the method may be slow for even small problems. Motivated by the need for faster implementations, some special systems have been developed exclusively for Gröbner bases computation, including MACAULAY and COCOA.

Other numerical techniques exist based on iterative algorithms and homotopy methods. Iterative techniques, like Newton's method, are good for local analysis and work well if we are given good initial guesses near the solutions. This is a rather difficult prerequisite for most applications. Homotopy methods have a good theoretical background and proceed by following paths in the complex space. In theory, each path converges to a geometrically isolated solution. They have been implemented and tried on a variety of applications. e.g. [MSW94, VVC94]. In practice, however, current homotopy implementations and algorithms suffer from many problems. The different paths being followed may not be geometrically isolated. As a result, each path has to be at times followed with impractically tight tolerances, which slows down the overall algorithm.

Multipolynomial resultant algorithms provide the most efficient methods (as far as asymptotic complexity is concerned) for solving a system of polynomial equations by eliminating variables. One of their main advantages is the fact that the resultant can always be expressed in terms of matrices and determinants. We will later describe different techniques for construction of resultant matrices below. Systems such as AXIOM, MAPLE, MATHEMATICA and REDUCE only offer matrix expressions for the resultant of two univariate polynomials, either as Sylvester's matrix or as Bézout's matrix. Some use of resultants can also be found in other systems, such as CASA, developed at RISC-Linz.

Different specialized modules based on resultant matrices exist for solving systems of polynomial equations, e.g. [CGT97, CP93, Emi97, KM95, KS96, MP97, Reg95]. Typically, these programs would rely on LINPACK, EISPACK, LAPACK, or MATLAB for their numerical calculations. All of these programs implement one or, exceptionally, two kinds of matrices, and are not designed for wide distribution, so they lack in user-friendliness. There is currently a very interesting effort in the context of FRISCO for developing a general library of resultant functions in `C++`, to which the second author is participating.

3 Resultant-based system solving

There is more than one way to solve arbitrary polynomial systems by using resultants, yet here we focus on the one method presenting the strongest practical interest. Namely, we are interested in constructing resultant matrices, whose determinants express nontrivial multiples of the resultant polynomial and which, furthermore, reduce the computation of all common zeros to a problem in linear algebra. The symbolic part of matrix construction can strongly exploit polynomial structure, whereas the manipulation of the matrices benefits from the current state-of-the-art in numerical linear algebra. Below we overview both stages and explain how to reduce the nonlinear problem to the computation of eigenvalues and eigenvectors of a square matrix.

3.1 Symbolic computation

The computation of resultants typically relies on constructing matrices whose determinant is either the exact resultant polynomial or, more generally, a nontrivial multiple of it. In addition, for solving polynomial systems these matrices are sufficient, since they reduce the given nonlinear problem to a question in linear algebra. For details see [KL92, EM97].

Resultant matrices can be classified into two large families. The first type includes Sylvester matrices and their classic generalization by Macaulay. In the context of sparse elimination theory, there are matrices that generalize Sylvester's and Macaulay's formulations and known as sparse, or toric, resultant matrices. The second type of resultant matrices includes Bézout matrices and their generalizations.

3.1.1 Sylvester and Macaulay matrices

The Sylvester resultant is a widely known resultant formulation for systems of two univariate polynomials. In this case, the resultant equals the determinant of the Sylvester matrix. If $d_i = \deg f_i$, for $i = 1, 2$, the rows of the Sylvester matrix express polynomials $f_1(x), xf_1(x), \ldots, x^{d_2-1}f_1(x)$ and $f_2(x), xf_2(x), \ldots, x^{d_1-1}f_2(x)$. The matrix columns are indexed by the monomials $\{1, x, \ldots, x^{d_1+d_2-1}\}$. Example 3.1 below illustrates this approach. This is a widespread tool for variable elimination even in the case of several variables. Then Sylvester's construction is applied repeatedly, albeit with a high overhead, because this technique introduces many superfluous solutions.

Macaulay devised a method that generalizes Sylvester's construction to systems of an arbitrary number of polynomials, under the hypothesis that these polynomials are completely dense [Mac02]. More formally, given $n+1$ dense nonhomogeneous polynomials in n variables, where the total degree of the i-th polynomial is denoted by d_i, $i = 1, \ldots, n+1$, the matrix columns are indexed by all monomials in the input variables whose degree is bounded by $\sum_{i=1}^{n+1} d_i - n$. The matrix rows express monomial multiples of the input polynomials f_i. The entries are either zero or equal to a coefficient of some input polynomial. In the current version of MARS, Macaulay's formulation is used whenever we compute a u-resultant.

3.1.2 Sparse resultant matrices

Resultants in classical elimination theory, as well as Macaulay matrices, are completely defined by the total degrees of the input polynomials. More recently, *sparse elimination* theory has modeled polynomials by the sets of their nonzero monomials, or supports, in order to obtain tighter bounds and exploit sparseness. Polynomials are specified by their support and its convex hull, known as Newton polytope. Sparse elimination defines the sparse, or toric, resultant, whose degree depends on these convex polytopes instead of the total degrees. Canny *et al.* described a construction based on a mixed subdivision of the Newton polytopes of the input polynomial system [CE93, CP93]. A direct incremental method yields smaller matrices [EC95, Emi97].

3.1.3 Bézout matrices

The second branch of resultant matrix constructions stems from Bézout's method for the resultant of two univariate polynomials. Let these polynomials be $f_1(x), f_2(x)$, of degrees $d_1 \geq d_2$ respectively, and let y be a new variable. Consider

$$\det \begin{bmatrix} f_1(x) & \frac{f_1(x)-f_1(y)}{x-y} \\ f_2(x) & \frac{f_2(x)-f_2(y)}{x-y} \end{bmatrix} = \sum_{i=0}^{d_1-1} B_i(x) y^i.$$

The $B_i(x)$ are polynomials in x and define the rows of the resultant matrix, whereas the columns are indexed by monomials $\{1, x, \ldots, x^{d_1-1}\}$. Hence, we define a square matrix of dimension d_1, thus smaller than Sylvester's matrix.

Bézout's matrix has been generalized to arbitrary systems. Although the above method does not always yield a square matrix, it is always possible to define a square maximal submatrix whose determinant is a nontrivial multiple of the resultant [CM96, EM97]. There is a rich algebraic theory behind these matrices, based on algebraic residues, which shows that Bézout matrices behave better for several degenerate input systems. Moreover, Bézout's matrix has smaller size than Macaulay's and the sparse resultant matrix. On the downside, its entries are polynomials in the input coefficients. Another difference is that the matrices of Sylvester type are constructed combinatorially, whereas the Bézout matrix construction is based on discrete differentials and requires some polynomial computation.

3.2 Numerical solving

The problem addressed here is to find all the common roots of a system of n non-homogeneous polynomials in n variables. Such a system is known as a square or well-constrained system, and typically has only a finite number of isolated roots. Our method reduces solving a zero-dimensional system to either a regular or a generalized eigenproblem, thus transforming the nonlinear question to a problem in linear algebra. This is a classical technique that enables us to approximate all solutions; see e.g. [Man94, Emi97] and the references thereof. Several extensions to positive-dimensional systems have been explored [KM95] or are currently under investigation.

An overconstrained system is obtained by adding extra polynomial $f_{n+1}(x, u)$ to the given system $f_1(x), \ldots, f_n(x)$, where $x = (x_1, \ldots, x_n)$. We choose $f_{n+1}(x, u)$ to be linear with random coefficients and constant term equal to indeterminate u. Let M be the resultant matrix of the overconstrained system, built by any method discussed above. The vanishing of $\det(M)$ is a necessary condition for the overconstrained system to have common roots. In this case, the resultant of the overconstrained system is a function of u and is known as the *u-resultant*. Partition M so that the upper left square submatrix M_{11} depends on u. By the construction of M and for arbitrary $\alpha \in \mathbf{C}^n$, evaluation of the row polynomials at α is expressed by vector multiplication on the right:

$$\begin{bmatrix} M_{11}(u) & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \begin{bmatrix} \vdots \\ \alpha^q \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ g(\alpha, u) \\ \vdots \end{bmatrix},$$

where $q \in \mathbf{Z}^n$ ranges over all column monomials and $g(x, u)$ ranges over all row polynomials. Clearly, if α is a common root of the input well-constrained system and u takes the values that make $\det M(u)$ vanish, then the vector must lie in the kernel of M and hence in the kernel of $M'(u) = M_{11}(u) - M_{12}M_{22}^{-1}M_{21}$. After suitable transformations the vector corresponds to the eigenvector of another square matrix, hence reducing root-finding to an eigenproblem. In section 5.2, we will describe an algorithm that, given an eigenvector and the column monomials, computes the coordinates of root α. Currently in MARS, u-resultant matrices are always computed by Macaulay's construction.

We have reduced root finding to a problem in linear algebra by adding the u-form to the given well-constrained system. An alternative is to "hide" one of the n variables in the coefficient field. This produces an overconstrained system without increasing the problem dimension. Our experience with systems in robotics and vision suggests that this is preferable in many practical situations. Formally, we consider the given polynomials as

$$f_1, \ldots, f_n \in (\mathbf{Q}[x_n])[x_1, \ldots, x_{n-1}].$$

Variable x_n is chosen so that all roots are separated by projection on x_n, if possible. Otherwise, we have to deal with the case of multiple roots. The construction of M is as before, and any algorithm can be used. Row and column permutations do not affect the matrix properties so we apply them to obtain a minimal M'. Gaussian elimination of all columns which are constant with respect to x_n is now possible, essentially giving $M'(x_n) = M_{11}(x_n) - M_{12}(x_n)M_{22}^{-1}M_{21}(x_n)$. System solving is again reduced to an eigendecomposition of square matrix M'.

Example 3.1 *We illustrate hiding a variable in the case of a system of two polynomials, by the use of Sylvester's matrix hiding y:*

$$\left\{ \begin{array}{ll} f_1(x) = & x^2+6x+3y-4 \\ f_2(x) = & 2x+y^2-7y+5 \end{array} \right\} \text{ then}$$

$$\begin{bmatrix} 1 & 6 & 3y-4 \\ 2 & y^2-7y+5 & 0 \\ 0 & 2 & y^2-7y+5 \end{bmatrix} \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix} = \begin{bmatrix} f_1(x,y) \\ xf_2(x,y) \\ f_2(x,y) \end{bmatrix}$$

If we specialize x and y at the roots, the product vector will be zero. Inversely, to solve the system it suffices to find the values of y for which the matrix is singular and to compute the nonzero vectors in its kernel. Among these vectors we shall restrict attention to those that correspond to a specialization of x. This is equivalent to solving the following problem:

$$\left(y^2 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + y \begin{bmatrix} 0 & 0 & 3 \\ 0 & -7 & 0 \\ 0 & 0 & -7 \end{bmatrix} + \begin{bmatrix} 1 & 6 & -4 \\ 2 & 5 & 0 \\ 0 & 2 & 5 \end{bmatrix} \right) v = 0$$

This can be transformed to an eigenproblem by performing certain matrix operations as explained below. Among the computed eigenvectors we shall restrict attention to those that correspond to a specialization of x in order to extract the roots.

In contrast to the approach of adding a u-polynomial, the resultant matrix may be nonlinear in the hidden variable, so $M'(x_n)$ is matrix polynomial $A_d x_n^d + \cdots + A_1 x_n + A_0$, for some $d \geq 1$. If A_d is numerically nonsingular, we reduce the equation $(I x_n^d + A_d^{-1}A_{d-1}x_n^{d-1} + \cdots + A_d^{-1}A_0)v = 0$ to the following eigenproblem:

$$\begin{bmatrix} 0 & I & 0 & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 0 & I \\ -A_d^{-1}A_0 & -A_d^{-1}A_1 & \cdots & -A_d^{-1}A_{d-2} & -A_d^{-1}A_{d-1} \end{bmatrix} w = x_n w,$$

where $w = [v, x_n v, \ldots, x_n^{d-1} v]$ and each eigenvalue of the latter matrix corresponds to the y value of a common root.

If A_d is numerically singular, we may change variable x_n to $(t_1 y + t_2)/(t_3 y + t_4)$ for random $t_1, \ldots, t_4$ and new variable y. This *rank-balancing* is used in MARS in order to improve the conditioning of the leading matrix A_d. If the latter is still nonsingular, we have to consider a generalized eigenproblem on the following matrix pencil:

$$C(y) = \begin{bmatrix} I & & & \\ & \ddots & & \\ & & I & \\ & & & A_d \end{bmatrix} y + \begin{bmatrix} 0 & -I & & \\ & & \ddots & \\ & & & -I \\ A_0 & A_1 & \cdots & A_{d-1} \end{bmatrix}.$$

For every eigenvalue λ with associated right eigenvector $w = [v_1, \ldots, v_d]$ of $C(y)$, we have $v_i = \lambda^{i-1} v_1$ for $i = 2, \ldots, d$. Moreover, $A(y)$ has the same eigenvalue λ and has right eigenvector v_1. These are all standard operations in numerical linear algebra.

We can compute the x coordinates (the values of the eliminated variables) from the eigenvector. MARS assumes that each eliminated variable can be computed as the ratio of two eigenvector elements raised to some power. We use the term *extraction recipe* to denote the process of computing the eliminated variables from the eigenvectors. Generating extraction recipes involves determining which eigenvector elements are to be divided and to what power the quotient should be raised in order to extract/compute the values of each eliminated variable.

With respect to example 3.1, each (generalized) eigenvector of C is of the form $[x_y^2, x_y, 1, yx_y^2, yx_y, y, \ldots]^T$, where x_y is a function of the (generalized) eigenvalue y. We can compute the x value corresponding to each y value by dividing the first element of the eigenvector, namely x_y^2, by the second element of the eigenvector, namely x_y.

Special attention is required when there are roots of high multiplicity which give rise to eigenspaces of high dimension. A more serious problem arises when the matrix determinant vanishes for all values of the hidden variable and is, therefore, a trivial multiple of the resultant polynomial. A fast, easy to use package such as MARS will be useful for quickly identifying trivial resultants. Current work is concentrating on numeric methods for transforming the matrix problem in a numerically stable way so that multiple roots are identified and degeneracies are avoided. Another approach is based on perturbation techniques. For more information see [KL92, CE93, Man94, MD95, Mou97] and the references thereof.

4 MARS description

In this section, we present an overview of the MARS package. Then, we discuss the design goals.

4.1 MARS Architecture

The MARS package is implemented in MAPLE, MATLAB, and C. We chose to use three environments because we could not find a standalone package which provided high symbolic, numerical, and application performance. In all three environments, MARS uses resultant objects to characterize resultant matrices.

Although resultant objects are implemented differently in all three environments, all three implementations contain enough information to reconstruct the original system and the resultant matrix polynomial (function array, variable array, hidden variable, resultant matrix polynomial and extraction recipe); consequently, we can reconstruct a resultant object in one environment from a resultant in another environment, simplifying the task of transferring resultant objects between the environments. Furthermore, making the resultant objects self-contained allows these objects to be passed "by-value".

The MARS package works as follows: The MAPLE resultant formulation routines construct resultant objects, which are then passed to the MATLAB and C libraries to be solved numerically.

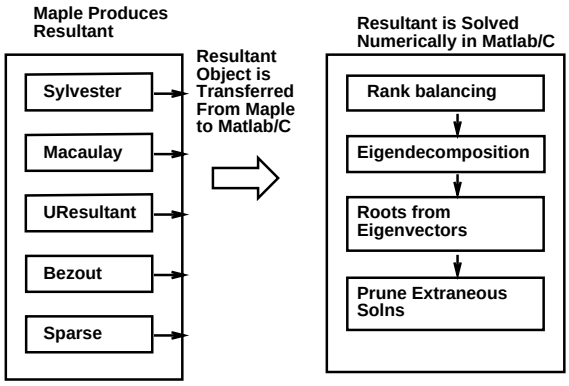

Figure 1: The Architecture of the MARS system

4.2 Design goals

We implemented MARS in order to simplify the process of using resultant eigendecomposition techniques to solve multinomial systems. Consequently, we want MARS to be both easy to use and efficient; this translated into five design goals: Platform independence (achieved by using MATLAB MAPLE and C), Ease of use (see section 6.2), Performance, Functionality, and Utilty.

4.2.1 Performance

In order for MARS to be useful, must perform symbolic computation (to construct resultants) and numerical computation (to "solve" resultants) quickly and accurately. Unfortunately, we could not find one standalone mathematical package which simultaneously provided high performance for both symbolic processing and numerical computation. Consequently, we decided to use a combination of three packages, MAPLE to perform symbolic computation, MATLAB to perform numerical computations, and C for programming applications. One reason we selected this particular combination is that MATLAB and MAPLE already provide an easy-to-use interface between the two systems. MATLAB currently includes a MAPLE kernel to do symbolic processing, and also provides a top-level MATLAB command (`maple()`) to execute MAPLE function calls. Furthermore, MAPLE provides a function for generating C code.

4.2.2 Functionality

In addition to providing top-level MATLAB commands, we also provide lower-level commands and functionality, such as computing the Newton polytope. MARS was designed to simplify the task of incorporating resultant techniques into applications. Many applications involve repeatedly solving a multipolynomial system, and these applications are best handled by creating a generic multipolynomial systems in terms of generic coefficients, constructing a generic matrix resultant, and then instantiating and solving the generic matrix resultant. For these applications, MARS resultant construction functions can handle symbolic variables; MARS also provides functions for automatically generating C code corresponding to a matrix resultant.

4.2.3 Utility

Using MARS to implement a multipolynomial solver into an application involves two steps:

1. Prototyping the resultant strategy (determining which system formulation and resultant construction to use; this step often involves trying different formulations)

2. Generating "production code" to be incorporated into an application

In order to simplify the prototyping step, we designed all of the resultant construction functions to use a consistent interface, so that they could be easily interchanged. In order to simplify the production step, we also designed all of the numerical solver routines (in MATLAB and C) to use a consistent interface, so that they can also be easily interchanged. In addition, we implemented MAPLE routines for automatically generating C source code corresponding to a resultant matrix polynomial.

5 Implementation

In this section, we describe the implementation details of the MARS package: common interface, root extraction, changing of variables, avoiding superfluous roots, and genericity.

5.1 Common resultant interface

One of the design goals was to provide a consistent API for the resultant formulation routines (to facilitate switching formulations). Each of the MAPLE resultant construction functions adheres to the following function prototype

```
"Resultant-constructor"
("function-array",
 "eliminated-variable-array",
 "hidden-variable",
 "optional-debugging-flag")
```

5.2 Generating extraction recipes

We generate the extraction recipes using the resultant columns (*i.e.*, $[x^2, x, 1]^T$ in example 3.1). We consider each pair of column elements and check if the ratio corresponds to an exponent of the desired eliminated variable; if so, then the extraction recipe corresponds to the ratio of these two eigenvector elements raised to the inverse power.

In example 3.1, x can be computed by raising the first column (or first vector element) x^2 to the first power and dividing that exponent by the second column (or second vector element) x to the first power ($\frac{x^2}{x}$).

For homogeneous systems, we need to employ a "trick" because all of the monomials have the same total degree; the trick is to specify that one of the variables is identically 1, and then we enforce the trick by scaling the eigenvector at runtime.

5.3 Generic linear change of variables

For certain "degenerate" multipolynomial inputs, resultant techniques can be susceptible to numerical imprecision, and it is well known that a random change of variables ameliorates these precision problems. As such, MARS includes routines for applying a generic linear change of variables.

MARS takes the original multipolynomial system $F(X)$ defined in terms of a variable set X and constructs a new multipolynomial system $F'(X')$ defined in terms of another variable set X' where X' and X are related according to the following linear transformation (M is a random rotation matrix).

$$\left[X'\right] = [M]\,[X] \Rightarrow F(X) = F'(X')$$

Then, MARS numerically solves the new multipolynomial system $F'(X')$ and returns a set of common roots U'. Finally, we compute the roots U of the original system $F(X)$ by inverting the map M.

$$[U] = [M]^{-1}\left[U'\right]$$

5.4 Pruning extraneous solutions

Since resultant techniques usually correspond to necessary but not sufficient conditions for common roots, resultant techniques often generate a superset of the desired roots of the multipolynomial system. In addition, the numerical eigendecomposition routines compute roots over the complex plane, whereas most users only care about the real roots. For this reason, we provide two functions for pruning non-roots and complex roots: MATLAB functions `removecomplex()` and `removenonroot()`. Furthermore, we allow the user to set two global MATLAB variables, `only_real` and `only_root`, to specify whether they always want extraneous solutions to be pruned.

`removecomplex()` uses the following criteria: `abs(Im(x_{i,j}))>1e-6*max(1,abs(Re(x_{i,j})))`.

`removenonroot()` uses the following criteria: `abs(fn(x_{i,j}))>1e-6*max(1,abs(|x_{i,j}|))`.

Notice that these naive approaches may incorrectly prune complex conjugate solutions corresponding to double roots; we are currently investigating extensions the implementation to overcome this limitation.

5.5 Function Genericitization

In order to efficiently repeatedly solve the similar systems using resultant techniques, one must construct a generic multipolynomial which will be specialized for each instantiation to be solved. We (the authors) usually genericitize the multipolynomials by replacing the monomial coefficients with symbolic variables. For example, we would transform the function $13x^2 + 7xy + 6y^2 + 4x - 2y + 9$ into `a_X0Y0 + a_X1Y0 * X + a_X0Y1 * Y + a_X1Y1 * X * Y + a_X2Y0 * X * X + a_X0Y2 * Y * Y`. We use this genericitization technique because the resultant matrices are functions of monomial coefficients.

Multipolynomial systems are often formulated in terms of other symbolic variables (not the monomial coefficients). For this reason, MARS includes a routine for converting from a system of multipolynomials defined with respect to one variable set into a system of multipolynomials with symbolic monomial coefficients; furthermore, this routine computes and outputs the relationship between the original symbolic variables and the new monomial coefficient variables.

6 Usage

This section illustrated the use of MARS by a series of small-size examples.

6.1 User Interface

MARS provides easy-to-use interfaces to top-level MATLAB routines for solving multipolynomial systems using all four resultant constructions (Sylvester, u-resultant, Dixon/Bézout, and Sparse) for systems of two and three equations in two and three unknowns respectively. These top-level MATLAB routines take as inputs the polynomials and the variables, and output a matrix were each row corresponds to a computed root; they are illustrated below.

6.2 Examples

For illustration, we use MARS to find the common roots of the system of two equations studied above and introduced in example 3.1: `f1(x,y) = x*x+6*x+3*y-4` and `f2(x,y) = y*y+2*x-7*y+5`. Here we use MARS functions `uresultantglt2()` with the appropriate arguments.

Example 6.1 *Using u-resultant formulation and generic linear transforms to solve a system in two variables*

```
% to print out all of the results
>> format long g
>> uresultantglt2('x*x+6*x+3*y-4',
                  'y*y+2*x-7*y+5','x','y','u')
ans =
 0.214777517761955          0.888401837097428
-7.04453580733046          -1.11942329
```

We can find the common roots of the system of three equations: `f1(x,y,z) = x*x+6*x+3*y+6*z-4`, `f2(x,y,z) =y*y+2*x-7*y+5+2*z`, `f3(x,y,z) = x*x+y*y+z*z-1`, by calling `uresultantglt3()`

Example 6.2 *Using u-resultant formulation and generic linear transforms to solve a system in three variables*

```
>> uresultantglt3('x*x+6*x+3*y+6*z-4','y*y+2*x-7*y+5+2*z',
                  'x*x+y*y+z*z-1','x','y','z','u')
ans =
-0.197730522175298   0.888778516434436   0.413491704058126
 0.417407913885499   0.881547960575403  -0.220553455268931
```

6.3 Using Resultants of Generic Functions

In most resultant applications, constructing the resultant via symbolic processing takes much longer than numerically solving the resultant. Many applications involve repeatedly solving the same type of multipolynomial system. In this case, we can construct a resultant of generic polynomials, instantiate the generic resultant for each set of parameters, and then numerically solve the instantiated resultant. The MARS package supports this methodology in two ways. First, the resultant constructions can handle unresolved variables, and second, MARS uses MAPLE's `subs()` function to instantiate variables. Note that in this case, we need to use lower-level MARS functions (`mapleresultant()`, `solveresultant()`) rather than the top-level MATLAB interface (`bezout2()`, `sparse2()`, ...).

In Example 6.3, the multipolynomial system characterizes two generic axis-aligned ellipses at unspecified locations with unspecified axis lengths. First we construct a generic resultant using the MAPLE Bezout resultant formulation. Then, we instantiate this generic resultant using MAPLE's `subs()` function. Next, we use the `mapleresultant()` routine to convert the MAPLE resultant data structure to a MATLAB resultant data structure. Finally, the function `solveresultant()` numerically computes the roots.

Example 6.3 *Constructing a generic resultant corresponding to a generic multipolynomial system for intersecting two ellipses, and then instantiating the resultant (so that we don't have to recompute the resultant every time).*

```
>> format long g

% initialize the Maple functions and variables
>> maple('Asym:= expand(square((x-Ax)/Arx) +
                         square((y-Ay/Ary)) - 1);');
>> maple('Bsym:= expand(square((x-Bx)/Brx) +
                         square((y-By/Bry)) - 1);');
>> maple('fnarray:=array(1..2,[Asym,Bsym]);');
>> maple('vararray:=array(1..1,[y]);');
% construct the generic resultant
>> maple('ellipseResultant:=Bezout(eval(fnarray),
                                   eval(vararray),x);');
% instantiate the generic resultant
% Note that Maple resultant objects permanently
% reside in the Maple kernel and that Matlab
% uses only their names to refer to them
>> maple('thisEllRes:=
   map (proc(x) expand(subs({Ax=3,Ay=10,Arx=7,Ary=4,
                             Bx=8,By=-4,Brx=8,Bry=16},x));
         end, ellipseResultant);');
% convert the Maple resultant data structure
% into a Matlab resultant data structure using
% the Matlab function ``mapleresultant'' which
% takes as input the name of a Maple resultant
% object
>> matlabEllipseRes=mapleresultant('thisEllRes')
>> intersections=solveresultant(matlabEllipseRes)
intersections =
 9.24740034171016        11.8043022481223
 1.77964742680292        6.06125516327268
```

6.4 Automatically Generating C Code

One of MARS' key features is the ability to automatically generate C code for the resultant matrix polynomial and computing the eliminated variables from the eigenvectors. The generated C code interfaces with MARS' C library which numerically solves matrix polynomials via the eigendecomposition approach. MARS' C library expects the matrix polynomial to be characterized by a three dimensional matrix where the first index represents the term degree, and the second and third elements characterize the matrix indices. We implemented separate functions in MATLAB and MAPLE for generating C code: MARS includes functions for generating C code for instantiating the coefficients of resultant matrix polynomials, namely `resultantccode()`, and for generating C code for extraction recipes `resultantextractrecipeccode()`.

Example 6.4 MARS *includes routines for automatically generating C code for instantiating the coefficients of the resultant matrix polynomial*

```
>> resultantccode('ellipseResultant')
      t0 = -2.0/(Arx*Arx)/(Bry*Bry)*Ax*Ax*By-
        2.0/(Ary*Ary)/(Bry*Bry)*Ay*Ay*By+
        2.0/(Bry*Bry)*By+2.0/(Ary*Ary)/
        (Brx*Brx)*Bx*Bx*Ay+
        2.0/(Ary*Ary)/(Bry*Bry)*By*By*Ay-2.0/
        (Ary*Ary)*Ay;
resultant[0][0][0]=t0;
      t0 = 4.0/(Arx*Arx)/(Bry*Bry)*Ax*By-4.0/
        (Ary*Ary)/(Brx*Brx)*Bx*Ay;
...
```

7 Performance

Formulation	Degree	Symbolic time (s)	Communication time (s)	Root Solver time (s)
Sylvester	2	0.05	0.88	6.26
Bezout	2	0.22	0.82	0.11
u-resultant	2	1.38	2.2	0.05
Sparse	2	7.97	1.1	6.59
Bezout	3	0.16	1.48	0.17
u-resultant	3	14.1	23.0	0.10
Sparse	3	22.96	1.81	6.82
Bezout	4	26.86	1.48	24.38

Table 1: Running times (seconds) for constructing, converting, and numerically solving systems of two, three and four polynomial equations using MARS on a Cyrix-686 166MHz PC-compatible computer w/48Meg

Table 1 presents times for solving multipolynomial systems using MARS via three different formulations (Sylvester, Bezout, u-resultant, and Sparse). The times are broken down into symbolic computation (constructing the resultant polynomial), communication time (converting the resultant data structure from MAPLEto MATLAB), and for numerically computing roots (via eigendecomposition). These preliminary timings are not meant as a comparison of the different algorithms; instead, they serve as evidence for the diversity and functionality of MARS(more performance results appear in [WEM98]. These measurements were computed using the multipolynomial systems `f1(x,y)=x*x+6*x+3*y-4`, and `f2(x,y)=y*y+2*x-7*y+5` and `f1(x,y,z)=x*x+6*x+3*y+6*z-4`, `f2(x,y,z)=y*y+2*x-7*y+5+2*z`, and `f3(x,y,z) = x*x+y*y+z*z-1`, both studied above. Bezout4 used the `f1(x,y,z,w)=5*x*x+6*x+3*y+6*z+3+2*w`, `f2(x,y,z,w)=4*y*y+4*x+3*z+w*2+4`, `f3(x,y,z,w)=x+3*z-9+w*7-11*z*z`, and `f4(x,y,z,w)=x+3*y+2*z-3*w*w+13.` .

8 Conclusion

Resultant matrices reduce polynomial system solving in the zero-dimensional case to a linear algebra problem, at the heart of which lies an eigenvalue/eigenvector computation. We have designed and implemented a MAPLE/MATLAB/C package of resultant-based methods for solving arbitrary systems whose roots form a set of zero dimension. There are three main components to our package, baptized MARS: (a) the symbolic manipulation in MAPLE to construct a variety of different resultant matrices, (b) the eigendecomposition technique, coupled with techniques for improving precision, for numerically computing all common solutions in MATLABand, (c) C code generation routines and a library of C functions for incorporating the numerical solver into real-world engineering and scientific applications.

In its current preliminary state, MARS has proven, through the examples of this paper, its diversity and user-friendliness, which make it suitable for educational purposes and for exploring different approaches to system solving. Moreover, the performance of MARS is reasonable and current work in improving the implementation should lead to a practical competitive system. Directions of further algorithmic work include the use of matrix structure for reducing

the time as well as space complexity of our methods, and the study of genericity conditions, as in e.g. [Gon91].

Acknowledgments

I.E. was partially supported by European ESPRIT project FRISCO (LTR 21.024) and acknowledges enlightening car commutes with Bernard Mourrain. D.M. has been supported by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Grant CCR-9319957, NSF Career Award CCR-9625217, ONR Young Investigator Award (N00014-97-1-0631) and Intel. Work partially conducted while A.W. and I.E. were visiting D.M. at the Department of Computer Science of UNC, Chapel Hill.

References

[AS88] W. Auzinger and H.J. Stetter, An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations, In *Proc. Intern. Conf. on Numerical Math., Intern. Series of Numerical Math., 86*, pages 12–30, 1988. Birkhäuser, Basel.

[CE93] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In G. Cohen, T. Mora, and O. Moreno, editors, *Proc. Intern. Symp. on Applied Algebra, Algebraic Algor. and Error-Corr. Codes, Lect. Notes in Comp. Science 263*, pages 89–104, Puerto Rico, 1993. Springer.

[CGT97] R.M. Corless, P.M. Gianni, and B.M. Trager. A reordered Schur factorization method for zero-dimensional polynomial systems with multiple roots. In *Proc. ACM Intern. Symp. on Symbolic and Algebraic Computation*, pages 133–140, 1997.

[CM96] J.-P. Cardinal and B. Mourrain. Algebraic approach of residues and applications. In J. Renegar, M. Shub, and S. Smale, editors, *The Mathematics of Numerical Analysis*, volume 32 of *Lectures in Applied Math.*, pages 189–210. AMS, 1996.

[CP93] J. Canny and P. Pedersen. An algorithm for the Newton resultant. Technical Report 1394, Comp. Science Dept., Cornell University, 1993.

[EC95] I.Z. Emiris and J.F. Canny. Efficient incremental algorithms for the sparse resultant and the mixed volume. *J. Symbolic Computation*, 20(2):117–149, August 1995.

[EM96] I.Z. Emiris and B. Mourrain. Polynomial system solving: The case of a 6-atom molecule. Technical Report 3075, INRIA Sophia-Antipolis, France, 1996.

[EM97] I.Z. Emiris and B. Mourrain. Matrices in elimination theory. *J. Symbolic Computation, Special Issue on Elimination*, 1997. Submitted.

[Emi97] I.Z. Emiris. A general solver based on sparse resultants: Numerical issues and kinematic applications. Technical Report 3110, INRIA Sophia-Antipolis, France, 1997.

[Fau95] J.-C. Faugère. State of GB and tutorial. In *Proc. PoSSo (Polynomial System Solving) Workshop on Software*, pages 55–71, Paris, March 1995.

[FRI97] FRISCO. First year report, February 1997. `http://extweb.nag.co.uk/projects/FRISCO.html`.

[Gon91] L. González-Vega. Determinantal formulae for the solution set of zero-dimensional ideals, *J. Pure Applied Algebra*, 76:57–80, 1991.

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

[KL92] D. Kapur and Y.N. Lakshman. Elimination methods: An introduction. In B. Donald, D. Kapur, and J. Mundy, editors, *Symbolic and Numerical Computation for Artificial Intelligence*, pages 45–88. Academic Press, 1992.

[KM95] S. Krishnan and D. Manocha. Numeric-symbolic algorithms for evaluating one-dimensional algebraic sets. In *Proc. ACM Intern. Symp. on Symbolic and Algebraic Computation*, pages 59–67, 1995.

[KS96] D. Kapur and T. Saxena. Sparsity considerations in Dixon resultants. In *Proc. ACM Symp. Theory of Computing*, pages 184–191, 1996.

[Mac02] F.S. Macaulay On Some Formula in Elimination *Proceedings of London Mathematical Society*, pages 3–27, May 1902.

[Man92] D. Manocha. *Algebraic and Numeric Techniques for Modeling and Robotics.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

[Man94] D. Manocha. Solving systems of polynomial equations. *IEEE Comp. Graphics and Appl., Special Issue on Solid Modeling*, pages 46–55, 1994.

[MC94] D. Manocha and J.F. Canny. Efficient inverse kinematics for general 6R manipulators. *IEEE Trans. on Robotics and Automation*, 10(5):648–657, 1994.

[MD95] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves II: Multiple intersections. *Graphical Models and Image Proc.*, 57(2):81–100, 1995.

[MM94] R.D. McKelvey and A. McLennan. The maximal number of regular totally mixed Nash equilibria. Technical Report 865, Div. of the Humanities and Social Sciences, California Institute of Technology, Pasadena, Calif., July 1994.

[Mou97] B. Mourrain. Solving polynomial systems by matrix computations. Manuscript. INRIA Sophia-Antipolis, France. Submitted for publication, 1997.

[MP97] B. Mourrain and V.Y. Pan. Solving special polynomial systems by using structured matrices and algebraic residues. In F. Cucker and M. Shub, editors, *Proc. Workshop on Foundations of Computational Mathematics*, pages 287–304, Berlin, 1997. Springer.

[MSW94] A.P. Morgan, A.J. Sommese, and C.W. Wampler. A product-decomposition bound for Bézout numbers. *SIAM J. Numerical Analysis*, 32(4), 1994.

[Reg95] A. Rege. A complete and practical algorithm for geometric theorem proving. In *Proc. ACM Symp. on Computational Geometry*, pages 277–286, Vancouver, June 1995.

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

[RR95] M. Raghavan and B. Roth. Solving polynomial systems for the kinematics analysis and synthesis of mechanisms and robot manipulators. *Trans. ASME, Special 50th Annivers. Design Issue*, 117:71–79, June 1995.

[VVC94] J. Verschelde, P. Verlinden, and R. Cools. Homotopies exploiting Newton polytopes for solving sparse polynomial systems. *SIAM J. Numerical Analysis*, 31(3):915–930, 1994.

[WC97] A. Wallack and J. Canny. Planning for modular and hybrid fixtures. *Algorithmica*, 19:40–60, 1997.

[WEM98] A. Wallack and I. Emiris and D. Manocha MARS: A Maple/Matlab/C Resultant-based Solver Technical Report TR98-020, Dept. of Computer Science, University of North Carolina, Chapel Hill, April 1998.

[WM98] A. Wallack and D. Manocha. Robust algorithms for object localization. *Intern. J. Comp. Vision*, 27(3):243–262, 1998.