POLSYS_PLP: A PARTITIONED LINEAR PRODUCT HOMOTOPY CODE FOR SOLVING POLYNOMIAL SYSTEMS OF EQUATIONS

by

Steven M. Wise

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Mathematics

APPROVED:

Layne T. Watson

Christopher Beattie

John Rossi

August 20, 1998
Blacksburg, Virginia

Key words: Numerical Analysis, Homotopy Methods, Polynomial Systems of Equations, Zeros.

POLSYS_PLP: A PARTITIONED LINEAR PRODUCT HOMOTOPY CODE FOR SOLVING POLYNOMIAL SYSTEMS OF EQUATIONS

by

Steven M. Wise

(ABSTRACT)

Globally convergent, probability-one homotopy methods have proven to be very effective for finding all the isolated solutions to polynomial systems of equations. After many years of development, homotopy path trackers based on probability-one homotopy methods are reliable and fast. Now, theoretical advances reducing the number of homotopy paths that must be tracked, and in the handling of singular solutions, have made probability-one homotopy methods even more practical. This thesis describes the theory behind and performance of the new code POLSYS_PLP, which consists of Fortran 90 modules for finding all isolated solutions of a complex coefficient polynomial system of equations by a probability-one homotopy method. The package is intended to be used in conjunction with HOMPACK90, and makes extensive use of Fortran 90 derived data types to support a partitioned linear product (PLP) polynomial system structure. PLP structure is a generalization of m-homogeneous structure, whereby each component of the system can have a different m-homogeneous structure. POLSYS_PLP employs a sophisticated power series end game for handling singular solutions, and provides support for problem definition both at a high level and via hand-crafted code. Different PLP structures and their corresponding Bezout numbers can be systematically explored before committing to root finding.

ACKNOWLEDGEMENTS

I would like to thank Prof. Layne T. Watson for serving as my committee chairperson, and as my research advisor. We had many fruitful discussions, not all of them on Mathematics. From him I learned many things. I wish also to thank Christopher Beattie and John Rossi for reading this thesis carefully, and for serving on my committee; and Alexander Morgan, Andrew Sommese, and Charles Wampler for kindly helping me with the theory of polynomial systems of equations.

Thank you Nicole for your love, support and patience.

Thank you Jesus.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: INTRODUCTION

Polynomial systems of equations arise in many applications: robotics, computer vision, kinematics, chemical kinetics, truss design, geometric modeling, and many others (see [Morgan 1987] and [Verschelde 1996]). In applications where all the solutions, or a significant number of solutions, must be found, or when locally convergent methods fail, globally convergent, probability-one homotopy methods are preferred. Homotopy methods for polynomial systems were first proposed by Garcia and Zangwill [1977] and Drexler [1977]. While the method in [Garcia and Zangwill 1977] was easily demonstrated by topological techniques and the start system was easily solved, the homotopy produced many more paths than the total degree of the system. Drexler used the powerful results of algebraic geometry to prove his method. Two years later, using differential geometry, Chow, Mallet-Paret, and Yorke [1979] improved on the results of Garcia and Zangwill with a general homotopy which produced the same number of paths as the number of solutions (provided there are a finite number of them), counting multiplicities and solutions at infinity. The start system of this homotopy was difficult to solve. Morgan [1983] solved this problem with a much simpler start system, which had trivially obtained roots, and could be used in a general homotopy. The suggestion by Wright [1985] and Morgan [1986a], [1986b] to track the homotopy zero curves in complex projective space, rather than in Euclidean space, was another fundamental breakthrough—in complex projective space certain paths would no longer diverge to infinity (have infinite arc length), and paths in general were made shorter. Other notable publications, which appear around the end of the first decade of research, are by Meintjes and Morgan [1985], Tsai and Morgan [1985], and Watson, Billups, and Morgan [1987].

In roughly the past decade, since the development of robust, efficient homotopy path tracking algorithms, work has shifted towards lowering the number of paths which must be tracked. In essence, all the methods try to construct a start system for the homotopy map that better models the structure of the given polynomial system, the target system for the homotopy map. Early work includes m-homogeneous theory by Morgan and Sommese [1987a]. In m-homogeneous theory the powerful connection of probability-one homotopy methods for polynomials with the field of algebraic geometry is reestablished (see [Drexler 1977]) with the generalization of the classical theorem of Bezout. Generalizations of m-homogeneous theory appeared in [Verschelde and Haegemans 1993] with the GBQ method, and in [Verschelde and Cools 1993] with set-structure analysis. The methods in both [Verschelde and Haegemans 1993] and [Verschelde and Cools 1993] are derived by modifying slightly the main theorem in [Morgan and Sommese 1987a], but are nonetheless important. The most recent definitive theoretical work is that of Morgan, Sommese, and Wampler [1995]. Though the theorems of [Morgan, Sommese, and Wampler 1995] are very powerful, used in their full generality, they suggest more an approach for exploiting structure than

an algorithm. The method used here in POLSYS_PLP for constructing the start system is essentially the same as that in [Verschelde and Haegemans 1993], but is based on the results of [Morgan, Sommese, and Wampler 1995].

Attention has also been paid to the problem of calculating singular solutions of polynomial systems using homotopy methods. Approaches have been proposed based on Newton's method (see for example [Griewank 1985]), and based on complex analysis as in the work of Morgan, Sommese, and Wampler [1991], [1992a], [1992b]. The most useful approach of those mentioned is found in [Morgan, Sommese, and Wampler 1992b], where the foundation of a reasonable end game is laid. Other work has come from Sosonkina, Stewart, and Watson [1996]. Their approach, which is used in the polynomial system routine POLSYS1H of HOMPACK90 [Watson et al. 1996], is moderately successful on low, odd order singularities. To accurately compute a singular solution of order 30, say, requires a very sophisticated end game as that in [Morgan, Sommese, and Wampler 1992b], which POLSYS_PLP incorporates.

Publically available codes for solving polynomial systems of equations using globally convergent, probability-one homotopy methods do exist: HOMPACK [Watson, Billups, and Morgan 1987], written in FORTRAN 77, and HOMPACK90 [Watson et al. 1996], written in Fortran 90, both have polynomial system solvers. CONSOL in the book by Morgan [1987] is also written in FORTRAN 77. However, neither HOMPACK[90] nor CONSOL have a sophisticated start system that can lower the number of homotopy paths that must be tracked below the total degree. The package PHCPACK by Verschelde [1997], written in Ada, allows a great variety of choices for the start system, and uses an end game like the one proposed in [Morgan, Sommese, and Wampler 1992b]. PHCPACK, based on BKK theory, has a distinctly combinatorial flavor, and tends to be rather slow on large scale production problems.

Polynomial structure is a complicated subject, attacked variously by the combinatorial BKK theory [Verschelde and Cools 1993] and algebraic geometry [Morgan, Sommese, and Wampler 1995]. A design choice of POLSYS_PLP is to strike a balance between the most general structural descriptions (yielding minimal numbers of paths to track, but extremely difficult algorithmically) and no structure at all (where the total degree number of paths must be tracked, algorithmically trivial). The trade-off is moot, because a search for structure may very well cost more than simply tracking the paths a fancier structure would have eliminated. Further, for many industrial problems, an m-homogeneous structure is perfectly adequate, and often even optimal. The structure supported by POLSYS_PLP is called *partitioned linear product*, which in generality lies between m-homogeneous and the arbitrary set-structure supported by PHCPACK.

Chapter 2: POLYNOMIAL SYSTEMS OF EQUATIONS

Let $F(z) = 0$ be a polynomial system of n equations in n unknowns. In symbols

$$F_i(z) = \sum_{j=1}^{n_i} \left[c_{ij} \prod_{k=1}^{n} {z_k}^{d_{ijk}} \right] = 0, \qquad i = 1, \ldots, n, \tag{1}$$

where the c_{ij} are complex (and usually assumed to be different from zero) and the d_{ijk} are nonnegative integers. The *degree* of $F_i(z)$ is

$$d_i = \max_{1 \le j \le n_i} \sum_{k=1}^{n} d_{ijk},$$

and the *total degree* of the system (1) is

$$d = \prod_{i=1}^{n} d_i.$$

Define $F'(w)$ to be the homogenization of $F(z)$:

$$F'_i(w) = {w_{n+1}}^{d_i} \, F_i(w_1/w_{n+1}, \ldots, w_n/w_{n+1}), \qquad i = 1, \ldots, n. \tag{2}$$

Note that, if $F'(w^0) = 0$, then $F'(\alpha w^0) = 0$ for any complex scalar α. Therefore, "solutions" of $F'(w) = 0$ are (complex) lines through the origin in $\mathbf{C}^{n+1}$. The set of all lines through the origin in $\mathbf{C}^{n+1}$ is called complex projective n-space, denoted $\mathbf{P}^n$, and is a compact n-dimensional complex manifold. (Note that we are using complex dimension: $\mathbf{P}^n$ is $2n$-dimensional as a real manifold). The solutions of $F'(w) = 0$ in $\mathbf{P}^n$ are identified with the solutions and solutions at infinity of $F(z) = 0$ as follows: If $L \in \mathbf{P}^n$ is a solution to $F'(w) = 0$ with $w = (w_1, w_2, \ldots, w_{n+1}) \in L$ and $w_{n+1} \neq 0$, then $z = (w_1/w_{n+1}, w_2/w_{n+1}, \ldots, w_n/w_{n+1}) \in \mathbf{C}^n$ is a solution to $F(z) = 0$. On the other hand, if $z \in \mathbf{C}^n$ is a solution to $F(z) = 0$, then the line through $w = (z, 1)$ is a solution to $F'(w) = 0$ with $w_{n+1} = 1 \neq 0$. The standard definition of *solutions to* $F(z) = 0$ *at infinity* is simply *solutions to* $F'(w) = 0$ *(in* $\mathbf{P}^n$*) generated by* w *with* $w_{n+1} = 0$.

A solution $\hat{w} \in \mathbf{P}^n$ is called *geometrically isolated* if there exists an open ball $B \subset \mathbf{P}^n$, with $\hat{w} \in B$ and no other solutions in B. If no such ball exists, then the solution $\hat{w}$ is said to exist on a *positive dimensional solution set*. Suppose $\hat{w}$ is a geometrically isolated solution to (2), and suppose B is a ball which contains it. For almost all perturbations of the coefficients of the polynomial, the perturbed polynomial has only nonsingular solutions. For all such sufficiently small perturbations of the coefficients, there exists a finite number m of solutions inside B to the perturbed system of equations. m is the *multiplicity* of the solution $\hat{w}$ to (2). $\hat{z} \in \mathbf{C}^n$ is a *singular* solution to (1) if the Jacobian matrix at $\hat{z}$, $D_z F(\hat{z})$, is

singular, and *nonsingular* otherwise. Singular solutions at infinity are defined analogously in terms of coordinate patches [Morgan 1987]. A solution has multiplicity greater than one precisely when it is singular [Morgan 1987].

The solution set having been described, the following beautiful result can be stated [van der Waerden 1953]:

BEZOUT'S THEOREM. *There are no more than d isolated solutions to* $F'(w)$ *in* $\mathbf{P}^n$. *If* $F'(w) = 0$ *has only a finite number of solutions in* $\mathbf{P}^n$, *it has exactly d solutions, counting multiplicity.*

In practical problems finite, nonsingular, geometrically isolated solutions are of great importance and interest; they are also the easiest to deal with in the homotopy setting. However, since it is not possible to *a priori* separate the nonsingular solutions from the singular solutions and solutions at infinity, homotopy algorithms are forced to deal with the latter. Solutions that are singular or at infinity can cause serious numerical difficulties and inefficiency—these two types of solutions are discussed in later chapters. More importantly, the problem of handling these "bad" solutions pales in comparison to the potentially huge number of solutions (and homotopy zero curves that must be tracked). d, called the Bezout number or more precisely the 1-homogeneous Bezout number [Morgan and Sommese 1987a], can be overwhelming even for tame-looking problems. For example, 20 cubic equations would have $d = 3^{20} \approx 3.5 \times 10^9$. Consequently, recent research has looked for methods which shrink (in a rigorous sense) the number of solutions that must be computed, while still retaining all the finite isolated solutions. *Reduction*, which seeks to lower the dimension of the system, is one approach which will work, but is not discussed here (see Chapter 7 of [Morgan 1987]). Sophisticated mathematical approaches, generally speaking, seek to "factor out" a significant portion of the nonphysical solutions (typically, including many solutions at infinity and multiplicities). For many important practical problems this is possible, since often systems which arise from physical models have symmetries and redundancies (which spawn solutions at infinity and multiple solutions), yet only a small number (compared to d) of finite, nonsingular solutions [Morgan and Sommese 1987b], [Verschelde 1996], [Verschelde and Cools 1993]. The next section discusses one such approach to reducing the number of homotopy zero curves that must be tracked: the partitioned linear product (PLP) homotopy.

Define a homotopy map $\rho : [0,1) \times \mathbf{C}^n \to \mathbf{C}^n$ by

$$\rho(\lambda, z) = (1-\lambda)G(z) + \lambda\, F(z). \tag{3}$$

$\lambda \in [0,1)$ is the *homotopy parameter*, $G(z) = 0$ is the *start system*, and $F(z) = 0$ is the *target system*. The goal is to find a start system with the same structure as the target system, while possessing the property that $G(z) = 0$ is easily solved. In this chapter a start system with a *partitioned linear product* (PLP) structure will be constructed.

Let $P = (P_1, P_2, \ldots, P_n)$ be an n-tuple of partitions P_i of the set $\{z_1, z_2, \ldots, z_n\}$. That is, for $i = 1, 2, \ldots, n$, $P_i = \{S_{i1}, S_{i2}, \ldots, S_{im_i}\}$, where S_{ij} has cardinality $n_{ij} \neq 0$, $\bigcup_{j=1}^{m_i} S_{ij} = \{z_1, z_2, \ldots, z_n\}$, and $S_{ij_1} \cap S_{ij_2} = \emptyset$ for $j_1 \neq j_2$. For clarity, P is called the *system partition*, and the P_i are the *component partitions*. For $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m_i$ define d_{ij} to be the degree of the component F_i in only the variables of the set S_{ij}, that is, considering the variables of $\{z_1, z_2, \ldots, z_n\} \backslash S_{ij}$ as constants. Thus if $F_2(z_1, z_2, z_3) = z_2^2 + z_3 z_2^3 - z_1$, $S_{21} = \{z_3\}$, and $S_{22} = \{z_1, z_2\}$, then $d_{21} = 1$, $d_{22} = 3$. It is convenient, though only for the definition of the start system, to rename the variables component-by-component. Let $S_{ij} = \{z_{ij1}, z_{ij2}, \ldots, z_{ijn_{ij}}\}$. With all this said, the start system is represented mathematically by $G_i(z) = \prod_{j=1}^{m_i} G_{ij}$, where

$$G_{ij} = \begin{cases} \left(\sum_{k=1}^{n_{ij}} c_{ijk} z_{ijk} \right)^{d_{ij}} - 1, & \text{if } d_{ij} > 0; \\ 1, & \text{if } d_{ij} = 0, \end{cases} \qquad i = 1, 2, \ldots, n, \tag{4}$$

where the numbers $c_{ijk} \in \mathbf{C}_0 = \mathbf{C} \backslash \{0\}$ are chosen at random. The structure defined by the system partition P and manifested in (4) is called the *partitioned linear product* structure. The degree of $G_i(z)$ is

$$\deg(G_i) = \sum_{j=1}^{m_i} d_{ij}.$$

Note that $d_i \leq \deg(G_i)$ always holds—this fact will be important later, when the projective transformation of the homotopy map is defined.

This start system is modeled after the one in [Wampler 1994], and, like its model, is desirable because it is computationally efficient and its solutions, all of which are obtained by the solution of a complex linear system, are nonsingular. To be precise, the linear subsystems into which $G(z) = 0$ decomposes, whether solvable or unsolvable, can be uniquely characterized by two lexicographic vectors. The first, $\Phi = (\Phi_1, \Phi_2, \ldots, \Phi_n)$, is called the *factor lexicographic vector*, and the second, $\Delta = (\Delta_1, \Delta_2, \ldots, \Delta_n)$, is called the *degree lexicographic vector*, where $(1, 1, \ldots, 1) \leq \Phi \leq (m_1, m_2, \ldots, m_n)$, and where, given

Φ and all $d_{j\Phi_j} \neq 0$, $(0, 0, \ldots, 0) \leq \Delta \leq (d_{1\Phi_1} - 1, d_{2\Phi_2} - 1, \ldots, d_{n\Phi_n} - 1)$. For example, suppose that the lexicographic pair (Φ, Δ) with all $d_{j\Phi_j} \neq 0$ is given. Then the linear system this pair uniquely represents is

$$A_\Phi z = \begin{pmatrix} \sum_{k=1}^{n_{1\Phi_1}} c_{1\Phi_1 k} z_{1\Phi_1 k} \\ \sum_{k=1}^{n_{2\Phi_2}} c_{2\Phi_2 k} z_{2\Phi_2 k} \\ \vdots \\ \sum_{k=1}^{n_{n\Phi_n}} c_{n\Phi_n k} z_{n\Phi_n k} \end{pmatrix} = \begin{pmatrix} e^{i(\Delta_1/d_{1\Phi_1})} \\ e^{i(\Delta_2/d_{2\Phi_2})} \\ \vdots \\ e^{i(\Delta_n/d_{n\Phi_n})} \end{pmatrix} \equiv b_\Delta, \tag{5}$$

where the z_{ijk} are as defined above. Either A_Φ is generically nonsingular, that is, nonsingular for almost all choices of the c_{ijk} from $\mathbf{C}_0$, or structurally singular. If A_Φ is structurally singular, then it contributes no solutions to $G(z) = 0$ and may be ignored. If A_Φ is generically nonsingular, then $A_\Phi z = b_\Delta$ has a unique solution for each Δ such that $(0, 0, \ldots, 0) \leq \Delta \leq (d_{1\Phi_1} - 1, d_{2\Phi_2} - 1, \ldots, d_{n\Phi_n} - 1)$. If some $d_{j\Phi_j} = 0$, then the factor $G_{j\Phi_j} = 1$ cannot be zero and A_Φ need not even be considered.

In order to count the number of solutions of $G(z) = 0$, it must be determined for each Φ whether or not A_Φ is generically nonsingular. If A_Φ is nonsingular then $\prod_{i=1}^{n} d_{i\Phi_i}$ is added to the "root count." The final root count is the total number of solutions B_{PLP} to $G(z) = 0$ and is called the PLP Bezout number. There is a combinatorial formula for determining whether or not A_Φ is generically invertible [Verschelde and Cools 1993]. For large problems, however, this rule is expensive. POLSYS_PLP uses numerical linear algebra with random real matrices to determine the generic invertibility of A_Φ. The issues of how to choose the c_{ijk} for such a method, and the likelihood of a generically invertible A_Φ being numerically singular, are discussed in detail in Chapter 8. The numerical algorithm, based on updated Householder reflections, used by POLSYS_PLP for calculating B_{PLP} follows:

begin $index(1:n) := 0$; $B_{PLP} := 0$;
for $\Phi(1:n) := (1, \ldots, 1)$ **step** lexicographically **until** $(m_1, \ldots, m_n)$ **do**
  Suppose that *index* and Φ agree in the first $k-1$ components and disagree in the k-th component.
  for $j := k$ **step** 1 **until** n **do**
    if $d_{j\Phi_j} = 0$ **then**
      $\Phi(j+1:n) := (m_{j+1}, \ldots, m_n)$; **exit do**
    endif
    Form the jth row of A_Φ.

$A_\Phi^T(1:n, 1:j-1)$ has been triangularized from Householder reflections that are saved. Apply the saved Householder reflections in order (from the 1st to the $(j-1)$st) to $A_\Phi^T(1:n, j)$.
Calculate and save the Householder reflection for $A_\Phi^T(j:n, j)$ and apply it to $A_\Phi^T(1:n, j)$, thus continuing the triangulation of A_Φ^T.
if $A_\Phi^T(j,j) \approx 0$ **then**
Declare A_Φ structurally singular.
$\Phi(j+1:n) := (m_{j+1}, \ldots, m_n)$; **exit do**
endif
if $j = n$ **then**
Declare A_Φ generically nonsingular.
$B_{PLP} := B_{PLP} + \prod_{i=1}^{n} d_{i\Phi_i}$
endif
enddo
$index(1:n) := \Phi(1:n)$
enddo

The importance of the number B_{PLP} derives from the next two theorems.

THEOREM 3.1 ([MORGAN, SOMMESE, AND WAMPLER 1995]). *Let $f : \mathbf{C}^n \to \mathbf{C}^n$ be a system of polynomials and $U \subset \mathbf{C}^n$ be open. Define $N(f, U)$ to be the number of nonsingular solutions to $f = 0$ that are in U. Assume that there are positive integers $r_1, \ldots, r_n$ and $m_1, \ldots, m_n$ and finitely generated complex vector spaces V_{ij} of polynomials for $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$, such that*

$$f_i = \sum_{k=1}^{r_i} \prod_{j=1}^{m_i} p_{ijk}, \tag{6}$$

where $p_{ijk} \in V_{ij}$ for $i = 1, \ldots, n$, $j = 1, \ldots, m_i$ and $k = 1, \ldots, r_i$. Let a system g be defined by $g_i = \prod_{j=1}^{m_i} g_{ij}$, with each g_{ij} a generic choice from V_{ij}. Then

$$N(f, U) \le N(g, U), \tag{7}$$

and (7) is equality if, for each i with $1 \le i \le n$, there is a positive integer k_i such that the $p_{ijk_i} \in V_{ij}$ are generic for $j = 1, \ldots, m_i$. Also, $g(z) = 0$ is a suitable start system for the polynomial homotopy $h(t, z) = (1-t)f(z) + tg(z)$ to find all nonsingular solutions to $f(z) = 0$.

REMARK 3.1. The reader will have noted that the homotopy of Theorem 3.1 is different from the one given in (3), but this difference is only a cosmetic change of variables $t = 1 - \lambda$. Moreover, the last line of Theorem 3.1 will be made precise in Chapter 4.

REMARK 3.2. The subsystems of $g = 0$ are the systems $\hat{g} = (g_{1j_1}, \ldots, g_{nj_n}) = 0$ with $1 \leq j_i \leq m_i$ and $i = 1, \ldots, n$ (see Remark 1.1 from [Morgan, Sommese, and Wampler 1995]). In practice, one chooses g so that the subsystems $\hat{g} = 0$ are easy to solve, e.g., so that solving $\hat{g} = 0$ reduces to solving a linear system.

Let $\{e_{ijl} \mid l = 1, \ldots, \ell_{ij}\}$ denote a basis of the vector space V_{ij}. For $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$ define

$$B_{ij} = \{z \mid e_{ijl}(z) = 0 \text{ for } l = 1, \ldots, \ell_{ij}\}.$$

Following [Morgan, Sommese, and Wampler 1995], the *bases have no pairwise intersection with U* if for any choice of j' and j'' with $1 \leq j' < j'' \leq m_i$,

$$B_{ij'} \cap B_{ij''} \cap U = \emptyset.$$

This next theorem relates the nonsingular solutions of $g(z) = 0$ with those of its subsystems.

THEOREM 3.2 ([MORGAN, SOMMESE, AND WAMPLER 1995]). *Let g and V_{ij} be as in Theorem 3.1. Then*

1. *$z_0 \in U$ is a nonsingular solution to $g(z) = 0$ if and only if z_0 is a solution to exactly one subsystem of $g(z) = 0$ and it is a nonsingular solution to this subsystem.*
2. *Assume that the bases for the V_{ij} have no pairwise intersection with U. Then, if $z_0 \in U$ is a nonsingular solution to some subsystem of $g(z) = 0$, it is a solution to exactly one subsystem of $g(z) = 0$.*

REMARK 3.3. It follows from Theorem 3.2 that

$$N(g, U) \leq \sum_{\substack{1 \leq j_1 \leq m_1 \\ 1 \leq j_2 \leq m_2 \\ \vdots \\ 1 \leq j_n \leq m_n}} N\big((g_{1j_1}, g_{2j_2}, \ldots, g_{nj_n}), U\big), \tag{8}$$

with equality if the bases have no pairwise intersection with U. (This is Remark 2.2 from [Morgan, Sommese, and Wampler 1995].)

Suppose that P is a system partition as before. The vector spaces of polynomials V_{ij} will be constructed using P: consider both S_{ij} and d_{ij} for $j = 1, 2, \ldots, m_i$ and $i = 1, 2, \ldots, n$, and define V_{ij} to be the complex vector space of polynomials generated by the monomials in the variables from S_{ij} up to degree d_{ij} and the constant 1. For example, suppose that $f(z) = 0$ is a polynomial system in five variables for which $P_2 = \{S_{21}, S_{22}\}$, $S_{21} = \{z_2, z_5, z_1\}$, $S_{22} = \{z_3, z_4\}$, $d_{21} = 2$, and $d_{22} = 3$. Then

$$V_{21} = \mathbf{C}\big\langle z_2^2, z_5^2, z_1^2, z_2z_5, z_2z_1, z_5z_1, z_2, z_5, z_1, 1\big\rangle,$$
$$V_{22} = \mathbf{C}\big\langle z_3^3, z_4^3, z_3^2z_4, z_3z_4^2, z_3^2, z_4^2, z_3z_4, z_3, z_4, 1\big\rangle.$$

Choosing the V_{ij}, albeit implicitly, from P, with $U = \mathbf{C}^n$, ensures that the bases have no pairwise intersection with U. Since $G_{ij} \in V_{ij}$ is generic,

$$N(F, \mathbf{C}^n) \leq N(G, \mathbf{C}^n) = \sum_{\substack{1 \leq j_1 \leq m_1 \\ 1 \leq j_2 \leq m_2 \\ \vdots \\ 1 \leq j_n \leq m_n}} N\big((G_{1j_1}, G_{2j_2}, \ldots, G_{nj_n}), \mathbf{C}^n\big) = B_{PLP}. \qquad (9)$$

This entire discussion would be moot if B_{PLP} were not in many cases smaller than the total degree d. In fact, for many practical problems for well chosen V_{ij}, B_{PLP} is much smaller than the total degree d. The computational implications for the homotopy map (3) with the start system $G(z) = 0$ are clear—only B_{PLP} homotopy zero curves must be tracked.

Since the start system of Theorem 3.1 can result in a lower number of paths to be tracked, while guaranteeing that paths will reach all nonsingular solutions of $f(z) = 0$, the number $N(g, \mathbf{C}^n)$ is commonly referred to as a generalized Bezout number. The PLP method just explained is but one way of arriving at such a number. m-homogeneous theory and the m-homogeneous Bezout number correspond to the special case when each component partition is the same. Morgan, Sommese, and Wampler [1995] show how the m-homogeneous Bezout number is easily derived from Theorem 3.1. The name partitioned linear product (PLP) is essentially a description of the structure of the start system $G(z) = 0$. A generalization of PLP is the linear product decomposition (LPD), where the start system reduces to a product of linear systems, but need not correspond to a system partition ($S_{ij_1} \cap S_{ij_2} \neq \emptyset$ possibly). LPD, which is exactly equivalent to set-structure analysis [Verschelde and Cools 1993], generalizes PLP because it allows for groupings of variables more general than system partitions. The method of greatest generality is the general product decomposition (GPD). GPD is any method which utilizes Theorem 3.1 in more generality than LPD, so the start system does not reduce to a product of linear systems. There is thus a hierarchy of methods, based on start system complexity:

- 1-homogeneous,
- m-homogeneous,
- partitioned linear product,
- linear product decomposition,
- general product decomposition.

As with the the name "m-homogeneous Bezout number," the values of $N(f, \mathbf{C}^n)$ using the the PLP method are called PLP Bezout numbers, and so on for the other methods. Finding the lowest possible PLP Bezout number is a challenging problem. There is no way, short of an exhaustive search through all possible system partitions, of knowing which P will give the lowest value of B_{PLP}. It is possible to construct a heuristic algorithm for picking P. Verschelde [1996] describe one such algorithm for obtaining an m-homogeneous partition, but as pointed out in [Verschelde 1996], the heuristic does not always work. Even if B_{PLP} is not minimized, it may still be small enough so that the path tracking is computationally tractable.

Suppose that P is a system partition for (1) corresponding to the PLP Bezout number B_{PLP}. The following theorem demonstrates the probability-one aspect of the homotopy method in POLSYS_PLP.

THEOREM 4.1. *For almost all choices of c_{ijk} in the start system defined by (4), $\rho^{-1}(0)$ consists of B_{PLP} smooth curves emanating from $\{0\} \times \mathbf{C}^n$, which either diverge to infinity as λ approaches 1 or converge to solutions of $F(z) = 0$. Each nonsingular solution of $F(z) = 0$ will have a curve converging to it.*

Theorem 4.1 is essentially a restatement of the last line of Theorem 3.1, but deserves emphasis; its proof can be found in Section A.5 in the appendix of [Morgan, Sommese, and Wampler 1995]. A noteworthy observation is that since the homotopy map ρ is complex analytic, the homotopy parameter λ is monotonically increasing as a function of arc length along the homotopy zero curves starting at $\lambda = 0$ [Morgan 1987]. Thus, the homotopy zero curves never have turning points with respect to λ.

Though B_{PLP} may be much smaller than the total degree, the possibility of tracking paths of (3) which diverge to infinity still exists. These paths pose significant computational challenges, since time is wasted on divergent paths, and large magnitude solutions may not be found if a path is terminated prematurely. Tracking paths in complex projective space, which was originally proposed in [Morgan 1986a, 1986b], eliminates these concerns. With a suitable "projective transformation" no paths diverge to infinity as λ approaches 1. Moreover, though not guaranteed, paths tend be shorter in projective space.

Constructing the projective transformation is straightforward [Morgan 1986a, 1986b], [Watson, Billups, and Morgan 1987]. As with the homogenization of $F(z)$, define the homogenization of $\rho(\lambda, z)$ to be

$$\rho_i'(\lambda, w) = w_{n+1}^{\deg(G_i)} \rho_i \left(\lambda, \frac{w_1}{w_{n+1}}, \ldots, \frac{w_n}{w_{n+1}} \right), \qquad i = 1, \ldots, n.$$

Define the linear function

$$u(w_1, \ldots, w_{n+1}) = \xi_1 w_1 + \xi_2 w_2 + \ldots, \xi_{n+1} w_{n+1},$$

where the numbers $\xi_i \in \mathbf{C}_0$ are chosen at random. The *projective transformation* of $\rho(\lambda, z)$ is

$$\rho''(\lambda, w) = \begin{pmatrix} \rho_1'(\lambda, w) \\ \rho_2'(\lambda, w) \\ \vdots \\ \rho_n'(\lambda, w) \\ u(w) - 1 \end{pmatrix}.$$

That the projective transformation can be applied to the homotopy map ρ, without changing the essence of Theorem 4.1, follows from Remark 1.4 of [Morgan, Sommese, and Wampler 1995]. Thus

THEOREM 4.2. *For almost all choices of the c_{ijk} in the start system defined by (4) and almost all choices of the ξ in the linear function $u(w)$, $(\rho'')^{-1}(0)$ consists of B_{PLP} smooth curves emanating from $\{0\} \times \mathbf{C}^{n+1}$, which converge to solutions of $F'(w) = 0$. Each nonsingular solution of $F'(w) = 0$ will have a curve converging to it.*

Henceforth, Theorem 4.2 will tacitly be the operative theorem, and references to ρ tacitly assume that the computer implementation actually works with ρ''.

Theorem 4.1 says that in order to reach the nonsingular solutions of $F(z) = 0$, "smooth" (nonintersecting, nonbifurcating) paths in $\rho^{-1}(0)$ must be tracked. There are fast, reliable ways of doing this numerically. Three different path tracking algorithms (ordinary differential equation based, normal flow, and augmented Jacobian matrix) are described in [Watson, Billups, and Morgan 1987] and [Watson et al. 1996]. Simple linear-predictor, Newton-corrector methods are described in [Morgan 1987]and [Verschelde 1997]. There is compelling evidence favoring higher order methods and the normal flow algorithm over simpler schemes [Lundberg and Poore 1991], [Morgan, Sommese, and Watson 1989], [Watson, Billups, and Morgan 1987], [Watson et al. 1996]. POLSYS_PLP uses the sophisticated homotopy zero curve tracking routine `STEPNX` from HOMPACK90.

The normal flow algorithm has essentially three phases: prediction, correction and step size estimation. Once a curve in $\rho^{-1}(0)$ has been tracked to a point for which $\lambda > 1 - \epsilon$, where $0 < \epsilon \ll 1$, the algorithm enters an "end game," discussed in the next chapter. For $\lambda \in [0, 1 - \epsilon]$, based on Theorem 4.1, it can be assumed that the path being tracked is smooth and that the Jacobian matrix $D\rho(\lambda, z)$ has full rank.

Let $\gamma(s)$ denote a path under consideration, where s is the arc length of the path, and $s = 0$ at $\lambda = 0$. For the prediction phase, assume that several points $P^{(1)} = (\lambda(s_1), z(s_1))$, $P^{(2)} = (\lambda(s_2), z(s_2))$ on γ with corresponding tangent vectors $(d\lambda/ds(s_1),\ dz/ds(s_1))$, $(d\lambda/ds(s_2),\ dz/ds(s_2))$ have been found, and h is an estimate of the optimal step (in arc length) to take along γ. The prediction of the next point on γ is

$$Z^{(0)} = p(s_2 + h), \tag{10}$$

where $p(s)$ is the Hermite cubic interpolating $(\lambda(s), z(s))$ at s_1 and s_2. Precisely,

$$\begin{aligned} p(s_1) &= (\lambda(s_1), z(s_1)), \quad & p'(s_1) &= (d\lambda/ds(s_1), dz/ds(s_1)), \\ p(s_2) &= (\lambda(s_2), z(s_2)), \quad & p'(s_2) &= (d\lambda/ds(s_2), dz/ds(s_2)), \end{aligned}$$

and each component of $p(s)$ is a polynomial in s of degree less than or equal to 3.

Starting at the predicted point $Z^{(0)}$, the corrector iteration mathematically is

$$Z^{(k+1)} = Z^{(k)} - \Big[D\rho\big(Z^{(k)}\big)\Big]^{\dagger} \rho\big(Z^{(k)}\big), \qquad k = 0, 1, \ldots, \tag{11}$$

where $\big[D\rho(Z^{(k)})\big]^{\dagger}$ is the Moore-Penrose pseudoinverse of the $n \times (n+1)$ Jacobian matrix $D\rho$. Computationally the corrector step $\Delta Z = Z^{(k+1)} - Z^{(k)}$ is the unique minimum norm solution of the equation

$$\big[D\rho\big]\Delta Z = -\rho. \tag{12}$$

Small perturbations of the c_{ijk} in $G(z)$ produce small changes in the trajectory γ. Geometrically, the iterates given by (11) return to the zero curve γ along the flow normal to the Davidenko flow (the family of trajectories γ for varying c_{ijk}), hence the name "normal flow algorithm." Robust and accurate numerical linear algebra procedures for solving (12) and for computing the kernel of $\left[D\rho\right]$ (tangent vectors required for (10)) are described in detail in [Watson, Billups, and Morgan 1987].

When the iteration (11) converges, the final iterate $Z^{(k+1)}$ is accepted as the next point on γ, and the tangent vector to the integral curve through $Z^{(k)}$ is used for the tangent—this saves a Jacobian matrix evaluation and factorization at $Z^{(k+1)}$. The next phase, step size estimation, attempts to balance progress along γ with the effort expended on the iteration (11), and is a sophisticated blend of mathematics, computational experience, and mathematical software principles. Complete details are given in [Watson et al. 1996].

Chapter 6: THE END GAME

The projective transformation eliminates diverging paths, but in doing so may give paths leading to highly singular solutions at infinity. When the curve being tracked converges to a multiple solution or a positive dimensional solution set of $F(z) = 0$ at $\lambda = 1$, necessarily rank $D\rho(\lambda, z) < n$, which affects both numerical stability and the rate of convergence of the corrector iteration (11). Newton-type algorithms may do very well with nonsingular solutions, but incur a significant expense at singular solutions, an order of magnitude worse than at nonsingular solutions. This chapter describes one way that reasonably accurate estimates of a singular solution may be obtained at a fairly low computational cost (compared to Newton-type algorithms). The algorithm proposed here is based on that in [Morgan, Sommese, and Wampler 1992b], but differs in several important aspects.

Define $h : D_0 \times D \to \mathbf{C}^n$ by $h(t, z) = (1 - t)f(z) + tg(z)$, where $f(z)$ and $g(z)$ are polynomial systems, with $D_0 \times D \subset \mathbf{C} \times \mathbf{C}^n$ open and $D_0 \supset [0, 1]$. Assume

1. z^*,$\bar{z} \in D$ with $h(0, z^*) = f(z^*) = 0$ and $h(1, \bar{z}) = g(\bar{z}) = 0$,
2. $h^{-1}(0)$ contains a connected complex curve $K \subset D_0 \times D$ containing z^* and $\bar{z}$, so that there is a smooth path $z(t)$ with $t \in [0, 1]$ and $z([0, 1]) \subset K$ such that $z(1) = \bar{z}$, $z(0) = z^*$, and $Dh(t, z(t))$ has rank n for $t \in (0, 1]$.

THEOREM 6.1 ([MORGAN, SOMMESE, AND WAMPLER 1992B]). *There is a $\delta > 0$, a smallest positive integer c, and a power series*

$$Z(\sigma) = \sum_{k=0}^{\infty} a_k \sigma^k$$

convergent for $|\sigma| < \delta$ such that

$$Z(\sigma) = z(\sigma^c) \tag{13}$$

for $\sigma \in [0, \delta)$.

REMARK 6.1.1. The integer c is called the *cycle number* of the curve $z(t)$ and is defined in [Morgan, Sommese, and Wampler 1991]. If z^* is a geometrically isolated solution to $f(z) = 0$ that has multiplicity m, then $c \leq m$.

REMARK 6.1.2. As in Theorem 3.1, the change of variables $\lambda = 1 - t$ recasts the theorem into the notation of this paper.

REMARK 6.1.3. The following definition is useful [Morgan, Sommese, and Wampler 1992b]: σ is said to be in the *operating range* if $|\sigma|$ is small enough so that Theorem 6.1 holds and large enough so that the numerical process described below is not overwhelmed by ill conditioning. In many cases this annulus is large enough to work in. It is, however, possible that it will be empty. The size of the operating range annulus depends on the machine precision.

REMARK 6.1.4. It is important to note that even though $z(\sigma^c)$ from Theorem 6.1 is defined only for real values of σ, $Z(\sigma)$ is defined and analytic for all $\sigma \in \mathbf{C}$, $|\sigma| < \delta$. Thus, as is pointed out in [Morgan, Sommese, and Wampler 1992b], the analyticity of h and Z give

$$h(\sigma^c, Z(\sigma)) = 0$$

for all σ such that $|\sigma| < \delta$. This fact suggests that samples of the homotopy zero path $Z(\sigma)$ can be taken at both positive and negative real values of σ, or even at complex σ in a neighborhood of $\sigma = 0$. Morgan, Sommese, and Wampler [1992b] use the latter for an end game based on complex contour integration.

The following example from [Morgan, Sommese, and Wampler 1995] nicely illustrates the theorem: Let $f(z) = z^m$, so that the solution $z^* = 0$ has multiplicity m. Let

$$h(t, z) = t(z^m - 1) + (1 - t)z^m.$$

Then the zero paths of the homotopy map h are given by (here $i = \sqrt{-1}$)

$$z(t) = e^{i(j/m)} \sqrt[m]{t},$$

for $j = 0, 1, 2, \ldots, m - 1$. Here $c = m$, $a_0 = 0$, $a_1 = e^{i(j/m)}$, and $\sigma^m = t$, that is,

$$Z(\sigma) = e^{i(j/m)}\sigma.$$

When speaking of $Z(\sigma)$ it is tacitly assumed that the change of variables $t = \sigma^c$ has taken place. Practical computation with $Z(\sigma)$ is complicated, since neither the value of c nor the size and location of the operating range will be immediately apparent.

The algorithm in POLSYS_PLP is to track a zero curve γ of ρ (using the normal flow algorithm) to $\lambda > 1 - \epsilon$, where $0 < \epsilon \ll 1$, and then enter the end game. Since neither the value of c nor the location of the operating range are known, they must be determined. Once these are found, samples of $Z(\sigma)$ using (13) can be taken at real positive and negative values of σ. Then an interpolant to $Z(\sigma)$ can be used to approximate $z^* = Z(0)$. Morgan, Sommese, and Wampler [1992b] interpolate the even function $A(\sigma) = \big(Z(\sigma) + Z(-\sigma)\big)/2$, and then approximate the root by estimating $z^* = A(0) = Z(0)$; this requires more Jacobian matrix evaluations than using just $Z(\sigma)$ (because Z must be evaluated at *exactly* σ and $-\sigma$), and typically only reduces the error slightly. The algorithm, inspired by one in [Morgan, Sommese, and Wampler 1992b], follows:

Given $\hat{c}_{max}$, tol_1, tol_2, $big \gg 1$, a point $(\lambda_1, z(\lambda_1))$ on γ with $\lambda_1 \leq 1-\epsilon$, and a point $(\lambda_0, z(\lambda_0))$ on γ with $1-\epsilon < \lambda_0 < 1$.

begin $hold := big$;

main loop: **do**

do

The points $P_\lambda^{(i)} = (\lambda_i, z(\lambda_i))$ for $i = 1, 0$ and the corresponding derivatives $dP_\lambda^{(i)} = dz/d\lambda(\lambda_i)$, where $\lambda_1 < \lambda_0$, have been found.

$P_\lambda^{(2)} := P_\lambda^{(1)}$; $dP_\lambda^{(2)} := dP_\lambda^{(1)}$; $P_\lambda^{(1)} := P_\lambda^{(0)}$; $dP_\lambda^{(1)} := dP_\lambda^{(0)}$; $\lambda_2 := \lambda_1$; $\lambda_1 := \lambda_0$;

Using the curve tracker, get one more point $P_\lambda^{(0)}$ and derivative $dP_\lambda^{(0)}$ at λ_0, where $\lambda_1 < \lambda_0 \approx \lambda_1 + .75(1-\lambda_1) < 1$.

if $D_z\rho\big(P_\lambda^{(0)}\big)$ numerically singular **then**

Flag convergence failure.

exit main loop

endif

for $\hat{c} := 1$ **step** 1 **until** $\hat{c}_{max}$ **do**

Change variables from λ to σ via

$P_\sigma^{(i)} = (\sigma_i, Z(\sigma_i)) = ((1-\lambda_i)^{1/\hat{c}}, z(\lambda_i))$ and

$dP_\sigma^{(i)} = dZ/d\sigma(\sigma_i) = -\hat{c}\sigma^{\hat{c}-1} dz/d\lambda(\lambda_i)$.

Construct a fourth order Hermite interpolant (in σ) using $P_\sigma^{(i)}$ and $dP_\sigma^{(i)}$ for $i = 2, 1$, and a sixth order Hermite interpolant using $P_\sigma^{(i)}$ and $dP_\sigma^{(i)}$ for $i = 2, 1, 0$, obtaining the respective approximations z^* and z^{**} at $\sigma = 0$.

$test(\hat{c}) := \|z^* - z^{**}\|_\infty / (1 + \|z^{**}\|_\infty)$;

enddo

$err_1 := \text{minval}(test(1 : \hat{c}_{max}))$;

$c := \text{minloc}(test(1 : \hat{c}_{max}))$;

if $err_1 \leq tol_1 * 10^{c/2}$ **then exit**

enddo

Convert path samples to variable σ, presuming that c is the correct cycle number, and that the samples are taken from the operating range.

for $j := 1$ **step** 1 **until** 3 **do**

Use the samples (of $Z(\sigma)$) $P_\sigma^{(i)}$ and $dP_\sigma^{(i)}$ for $i = 2, \ldots, 1-j$ to construct a $2(2+j)$-th order Hermite interpolant. Use this interpolant to approximate $Z(\sigma)$ at $\sigma = -\sigma_{j-1}$.

Apply the corrector iteration (11) yielding σ_{-j}, $P_\sigma^{(-j)}$, and $dP_\sigma^{(-j)}$ (after appropriate changes of variables).

if $D_z\rho\big(P_\lambda^{(-j)}\big)$ numerically singular **then**

Flag convergence failure.

exit main loop

endif

enddo

Construct a twelfth order interpolant $H_{12}(\sigma)$ using $P_\sigma^{(i)}$ and $dP_\sigma^{(i)}$ for $i = 2, \ldots, -3$, construct a tenth order interpolant $H_{10}(\sigma)$ using $P_\sigma^{(i)}$ and $dP_\sigma^{(i)}$ for $i = 2, \ldots, -2$, construct an eighth order interpolant $H_8(\sigma)$ using $P_\sigma^{(i)}$ and $dP_\sigma^{(i)}$ for $i = 1, \ldots, -2$, construct a sixth order interpolant $H_6(\sigma)$ using $P_\sigma^{(i)}$ and $dP_\sigma^{(i)}$ for $i = 1, \ldots, -1$, and obtain an approximation $z^* = H_{12}(0)$ of the power series at $\sigma = 0$.

$gm := \sqrt{(tol_1 * 10^{c/2}) * (tol_2 * 10^{c-1})}$;

$err_2 := \|H_{10}(0) - H_{12}(0)\|_\infty / (1 + \|z^*\|_\infty)$;

if $\left(\dfrac{\|H_8(0) - H_6(0)\|_\infty}{1 + \|H_{12}(0)\|_\infty} \le tol_1 * 10^{c/2}$ **and** $\dfrac{\|H_{10}(0) - H_8(0)\|_\infty}{1 + \|H_{12}(0)\|_\infty} \le gm$

and $\dfrac{\|H_{12} - H_{10}(0)\|_\infty}{1 + \|H_{12}(0)\|_\infty} \le tol_2 * 10^{c-1} \right)$ **then**

exit main loop

else

if $err_2 \le 1.01 * hold$ **then**

$hold := err_2$

else

Flag convergence failure and **exit** main loop if second occurrence

endif

endif

enddo

The above pseudo code captures the spirit of the algorithm—continue iterating as long as progress is being made—but not the precise details. Difficulty portending failure is measured in several ways: failure of corrector iteration to converge, predicted λ value inconsistent with parity of c, change in predicted c after sampling $Z(\sigma)$ for $\sigma < 0$, increase in err_2 after main loop iteration. These failures are counted, and any two consecutive failures (not separated by a successful, logically consistent iteration) cause the whole algorithm to abort. POLSYS_PLP uses the parameter values $\hat{c}_{max} = 8$, $\epsilon = 0.97$, $tol_1 = 10^{-6}$, and $tol_2 =$ `FINALTOL`, an input parameter indicating the final accuracy desired. For any given machine precision there are theoretical limitations, which Morgan, Sommese, and Wampler [1992b] discuss, on the

maximum cycle number for which the algorithm is still useful. POLSYS_PLP with 64-bit IEEE Standard 754 arithmetic has had success computing solutions with cycle numbers up to 6 and multiplicity 30. As Morgan, Sommese, and Wampler [1992b] demonstrate, the true cycle number c will be evident provided $\hat{c}_{max} \geq c$. ϵ and tol_1 are chosen based solely on computational experience. ϵ should not be so small that the path tracker encounters numerical instability from the singularity, and it should not be so large that the end game requires a large number of iterations before reaching the operating range. tol_1 must be small enough that the correct cycle number will be chosen—time would be wasted by computing points $P_\sigma^{(j)}$ for $\sigma < 0$ with an incorrect cycle number prediction—but not so stringent that the process leaves the operating range because $err_1 < tol_1 * 10^{c/2}$ is never satisfied. tol_2 is the desired accuracy of the solution. For solutions with large cycle numbers, say for $c > 3$, $err_2 < tol_2 * 10^{c-1}$ may not be achievable. In such cases, the algorithm simply returns with the last best estimate of z^*, which is often still very reasonable. Chapter 8 discusses the numerical performance of the end game in POLSYS_PLP.

Chapter 7: ORGANIZATION AND USAGE

The package POLSYS_PLP consists of two Fortran 90 modules (`GLOBAL_PLP`, `POLSYS`). `GLOBAL_PLP` contains the Fortran 90 derived data types which define the target system, the start system, and the system partition. As its name suggests, `GLOBAL_PLP` provides data globally to the routines in POLSYS_PLP. The module `POLSYS` contains three subroutines: `POLSYS_PLP`, `BEZOUT_PLP`, and `SINGSYS_PLP`. `POLSYS_PLP` finds the root count (the Bezout number B_{PLP} for a given system partition P) and the roots of a polynomial system, `BEZOUT_PLP` finds only the root count. `SINGSYS_PLP` checks the singularity of a given start subsystem, and is of interest only to expert users. The package uses the HOMPACK90 modules `REAL_PRECISION`, `HOMPACK90_GLOBAL`, and `HOMOTOPY` [Watson et al. 1996], the HOMPACK90 subroutine `STEPNX`, and numerous LAPACK and BLAS subroutines [Anderson et al. 1995]. The physical organization of POLSYS_PLP into files is described in a README file that comes with the distribution.

Arguments to `POLSYS_PLP` include an input tracking tolerance `TRACKTOL`, an input final solution error tolerance `FINALTOL`, an input singularity tolerance `SINGTOL` for the root counting algorithm, input parameters for curve tracking, various output solution statistics, and four Fortran 90 optional arguments: `NUMRR`, `RECALL`, `NO_SCALING`, and `USER_F_DF`. `NUMRR` is an integer which specifies the number of iterations times 1000 that the path tracker is allowed; if not specified, the default value is 1. The logical variable `RECALL` should be included if, after the first call, `POLSYS_PLP` is being called again to retrack a selected set of curves. The presence of the logical variable `NO_SCALING` (regardless of value) causes `POLSYS_PLP` to *not* scale the target polynomial system. The logical optional argument `USER_F_DF` specifies that the user is supplying hand-crafted code for function and Jacobian matrix evaluation—this option is recommended if efficiency is a concern, or if the original formulation of the system is other than a linear combination of monomials.

POLSYS_PLP takes full advantage of Fortran 90 features. For example, all real and complex type declarations use the KIND specification; derived data types are used for storage flexibility and simplicity; array sections, automatic arrays, and allocatable arrays are fully utilized; interface blocks are used consistently; where appropriate, modules, rather than subroutine argument lists, are used for data association; low-level linear algebra is done with Fortran 90 syntax rather than with BLAS routines; internal subroutines are used extensively with most arguments available via host association. POLSYS_PLP is easy to use, with a short argument list, and the target system $F(z)$ defined with a simple tableau format (unless the optional argument `USER_F_DF` is present). The calling program requires the statement

```
USE POLSYS
```

The typical use of POLSYS_PLP is to either call `BEZOUT_PLP` to obtain the root count B_{PLP} of a polynomial system of equations for a specified system partition P, or to call `POLSYS_PLP` to obtain all the roots of the polynomial (and the root count as a byproduct). It is advisable to explore several system partitions with `BEZOUT_PLP` before committing to one and calling `POLSYS_PLP`. Along with the distribution of POLSYS_PLP comes a sample main program `MAIN_TEMPLATE`, which demonstrates how to use POLSYS_PLP as just described. `MAIN_TEMPLATE` uses NAMELIST input for the target system and partition definitions, and allows the user to solve multiple polynomial systems in a single run.

The template `TARGET_SYSTEM_USER` (an external subroutine) is also included with the distribution. This subroutine would contain hand-crafted code for function and Jacobian matrix evaluation if the optional argument `USER_F_DF` to `POLSYS_PLP` were used.

7.1 AN EXAMPLE

The *Boon* problem [Verschelde 1997] from the field of neurophysiology demonstrates the value of exploiting structure and the applicability of POLSYS_PLP. Define $F : \mathbf{C}^6 \rightarrow \mathbf{C}^6$ by

$$F(z) = \begin{pmatrix} z_1^2 + z_3^2 - 1.0 \\ z_2^2 + z_4^2 - 1.0 \\ z_5 z_3^3 + z_6 z_4^3 - 1.2 \\ z_5 z_1^3 + z_6 z_2^3 - 1.2 \\ z_5 z_3^2 z_1 + z_6 z_4^2 z_2 - 0.7 \\ z_5 z_3 z_1^2 + z_6 z_4 z_2^2 - 0.7 \end{pmatrix}.$$

Thus the total degree of F is $d = 1024$. Consider the following system partition for F:

$$P = \big\{\{\{1,3\},\{2,4,5,6\}\}, \{\{1,3,5,6\},\{2,4\}\}, \{\{1,2\},\{3,4\},\{5,6\}\}^4\big\},$$

which would be input for `POLSYS_PLP` along with the coefficients and degrees of the variables for each term in each component of $F(z)$. The subroutine `POLSYS_PLP` then computes the degree structure

$$\begin{aligned} d_{11} = 2, d_{12} = 0; \\ d_{21} = 0, d_{22} = 2; \\ d_{31} = 0, d_{32} = 3, d_{33} = 1; \\ d_{41} = 3, d_{42} = 0, d_{43} = 1; \\ d_{51} = 1, d_{52} = 2, d_{53} = 1; \\ d_{61} = 2, d_{62} = 1, d_{63} = 1. \end{aligned}$$

These degrees are used with P to compute the start system $G(z) = 0$ defined in (4), which has exactly 216 nonsingular solutions (the PLP Bezout number B_{PLP}). Then $F(z)$ has at most 216 isolated solutions, found by tracking 216 homotopy zero curves starting at the roots of G. For comparison, the best m-homogeneous Bezout number, derived from the partition $\big\{\{1,2\},\{3,4\},\{5,6\}\big\}$, is 344.

Chapter 8: PERFORMANCE

This chapter discusses implementation issues and numerical performance of the root counting algorithm of Chapter 3, and of the end game algorithm in Chapter 6. In POLSYS_PLP root counting is done in the subroutines `BEZOUT_PLP` and `SINGSYS_PLP`, the former calling the latter. The end game is housed in an internal subroutine `ROOT_PLP`.

8.1 ROOT COUNTING

The danger in implementing a root counting algorithm with floating point arithmetic, rather than as an exact integer computation, is that a generically nonsingular A_Φ may be classified singular, and vice versa. Structurally singular A_Φ will always be ill conditioned, so the issue reduces to the likelihood that a generically invertible A_Φ will also be ill conditioned, and hence misclassified. Different ways for choosing the start system coefficients c_{ijk} have been tested by accumulating statistics on cond A_Φ.

Observe that the literature on random dense matrices is not directly applicable here, because the matrices A_Φ are typically very sparse and very structured (which of course is the whole point of the PLP structure). Thus rather than generating random sparse matrices, a better test is to take a few nontrivial PLP structures, generate large numbers of A_Φ for those structures, and observe the distribution of cond A_Φ. Results for three representative target polynomial systems $F^{(1)}$, $F^{(2)}$, $F^{(3)}$ are reported here; two of these are *cyclic4* and *cyclic8* from Björk and Fröberg [1991]. Relevant data for these three systems follows. For $F^{(1)}$: $n = 8$,

$$P = \big\{\big\{\{1,2\},\{3,4\},\{5,6\},\{7,8\}\big\}^8\big\};$$

for $F^{(2)}$ (*cyclic4*): $n = 4$,

$$P = \big\{\big\{\{1,2,3,4\}\big\},\big\{\{1,3\},\{2,4\}\big\},\big\{\{1\},\{2\},\{3\},\{4\}\big\}^2\big\};$$

for $F^{(3)}$ (*cyclic8*): $n = 8$,

$$P = \big\{\big\{\{1,2,3,4,5,6,7,8\}\big\},\big\{\{1,3,5,7\},\{2,4,6,8\}\big\},$$
$$\big\{\{1,5\},\{2,6\},\{3,7\},\{4,8\}\big\}^2,\big\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\}\big\}^4\big\};$$

Random c_{ijk} were chosen from the unit square in $\mathbf{C}$ and $[-1,-1/2]\cup[1/2,1] \subset \mathbf{R}$. Using only real c_{ijk} yielded the same qualitative results as complex c_{ijk}, and of course is considerably cheaper. From a PLU factorization of each A_Φ, $\mu = \min_{1\le i\le n} |U_{ii}|$ was computed. Let $r = \lceil \log_{10} \mu \rceil$ for $\mu \neq 0$ and $r = -100$ for $\mu = 0$. $r = -7$ corresponds to μ being the square root of machine precision for 64-bit IEEE arithmetic. Declaring A_Φ

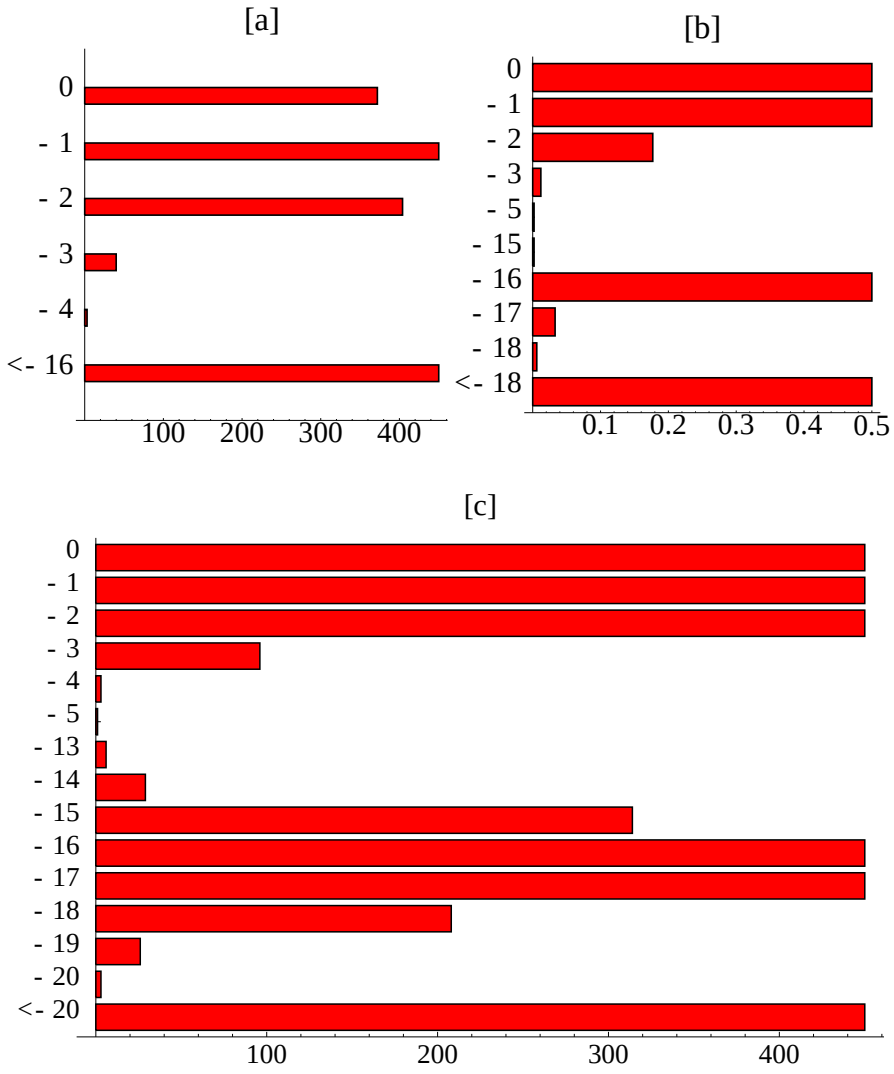

Fig. 1. Histograms of r frequencies. (a) $F^{(1)}$, 4^8 matrices; (b) $F^{(2)}$, 32 matrices (frequencies averaged over 1000 runs); (c) $F^{(3)}$, 131072 matrices.

singular if $r < -7$, A_Φ was correctly identified as either structurally singular or generically nonsingular in every single instance.

Figure 1 shows histograms for r for $F^{(1)}$, $F^{(2)}$, $F^{(3)}$. The point made emphatically by Figure 1 is that the invertible and singular A_Φ are clearly separated, with only a handful of matrices even approaching the cutoff value. Though not guaranteed, it is extremely unlikely that POLSYS_PLP will misclassify a matrix A_Φ.

For consistency with HOMPACK90, and because for poorly scaled systems accuracy is important, POLSYS_PLP uses QR factorizations for both the root count algorithm (`BEZOUT_PLP`) and the solution of the start subsystems to get the start points for the homotopy map (internal subroutine `START_POINTS_PLP`). Since both of these calculations

Table I. Path Tracking Statistics

	Path Characteristics	Average number of function evaluations Tracking	ROOT_PLP	ROOTNX	ROOTNX failures
PB601	42 paths, $\hat{c} = 6$	88	154	102	26
	3 paths, $\hat{c} = 3$	39	98	36	0
	15 paths, $\hat{c} = 1$	68	93	50	0
PB801	113 paths, $\hat{c} = 2$	117	48	2191	82
	15 paths, $\hat{c} = 1$	85	47	19	0
PB803	32 paths, $\hat{c} = 2$	114	35	59	32
	64 paths, $\hat{c} = 1$	121	22	14	16

proceed in lexicographic order, the QR factorization for one lexicographic vector can be efficiently updated for the next lexicographic vector. Here updating means simply replacing the tail of a sequence of Householder reflections with different reflections. Using QR rather than PLU to classify A_Φ can only improve the classification algorithm. The pseudo code

if $A_\Phi^T(j,j) \approx 0$ **then**

in the root counting algorithm is actually the test

```
IF (ABS(A_PHI(J,J)) <= SINGTOL) THEN
```

where `SINGTOL` is an argument to `SINGSYS_PLP`, and is set to the square root of machine precision by default.

8.2 END GAME

This section discusses the performance of the internal subroutine `ROOT_PLP`, the end game algorithm of Chapter 6. Three problems having singular solutions are considered: PB601 [Morgan and Watson 1989] is very hard, and PB801 and PB803 [Morgan, Sommese, and Wampler 1992b] are of moderate difficulty. Performance of the end game subroutine `ROOTNX` of HOMPACK90 [Watson et al. 1996] is compared to that of `ROOT_PLP` (cf. Table I). For problems PB601, PB801, and PB803, `TRACKTOL` was 10^{-4}, 10^{-4}, 10^{-3} respectively, and `FINALTOL` was 10^{-14}, 10^{-12}, 10^{-12} respectively. Scaling was enabled for all problems, and the polynomials were evaluated from their coefficient tableaus rather than hand-written code. Recall for the ensuing discussion that there is no way to know *a priori* which curves will lead to nonsingular solutions.

There are three possibilities: use `ROOTNX` always, use a hybrid scheme applying `ROOTNX` to nonsingular solutions and `ROOT_PLP` to singular solutions, or use `ROOT_PLP` always. `ROOTNX` performs better at nonsingular solutions than `ROOT_PLP`. Used alone, `ROOTNX` can be much more expensive overall than `ROOT_PLP`, its good performance at nonsingular solutions overshadowed by its extremely poor performance at singular solutions. `ROOT_PLP` handles singular solutions

much better than ROOTNX. The cost of ROOTNX can be much higher than ROOT_PLP at singular solutions (PB801 in Table I). Moreover, whereas ROOTNX could rarely find even a poor approximation of a singular root, ROOT_PLP obtained good approximations, even for the multiplicity 30 solution of PB601. ROOTNX failed completely on many paths (even on some paths for which $\hat{c} = 1$), but ROOT_PLP never failed. Data for PB803 and PB801 show that the overall cost of using ROOT_PLP alone can be far less than the cost of using ROOTNX alone. Thus using ROOTNX alone (as POLSYS1H of HOMPACK90 does) is not viable.

Early versions of POLSYS_PLP used a hybrid end game: if a curve appeared to be going to a singular solution, ROOT_PLP was chosen, otherwise ROOTNX was used, with reversion to ROOT_PLP if ROOTNX failed. Various criteria such as condition number estimates and step size reduction were tried. The hybrid schemes proved unsuccessful since curves were frequently misdiagnosed: ROOTNX would be called but would then fail because the curve was actually going to a singular solution, and ROOT_PLP was sometimes called when a curve was converging to a nonsingular solution. Essentially, any failure of ROOTNX is more costly than using ROOT_PLP alone on that root. Despite the appeal of hybrid methods, the evidence seems to support using ROOT_PLP only on all roots.

BIBLIOGRAPHY

Anderson, E., Bai, Z., Bishof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. 1995. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 2nd ed.

Björk, G., and Fröberg, R. 1991. A faster way to count the solutions of inhomogeneous systems of algebraic equations, with applications to cyclic n-roots. *J. Symbolic Computation 12*, 329–336.

Chow, S. N., Mallet-Paret, J., and Yorke, J. A. 1979. A homotopy method for locating all zeros of a system of polynomials. In *Functional Differential Equations and Approximation of Fixed Points*, Lecture Notes in Math. #730, H. O. Peitgen and H. O. Walther, Eds., Springer-Verlag, 228–237.

Drexler, F. J. 1979. Eine methode zur berechung sämtlicher lösunger von polynomgleichungessystemen. *Numer. Math. 29*, 45–58.

Garcia, C. B., and Zangwill, W. I. 1977. Global continuation methods for finding all solutions to polynomial systems of equations in N variables. Center for Math Studies in Business and Economics Report No. 7755, Univ. of Chicago, Chicago, IL.

Griewank, A. 1985. On solving nonlinear equations with simple singularities or nearly singular solutions. *SIAM Rev. 27*, 537–563.

Lundberg, B. N., and Poore, A. B. 1991. Variable order Adams-Bashforth predictors with an error-stepsize control for continuation methods. *SIAM J. Sci. Stat. Comput. 12*, 695–723.

Meintjes, K., and Morgan, A. P. 1985. A methodology for solving chemical equilibrium systems. Tech. Rep. GMR-4971, GM Res. Lab., Warren, MI.

Morgan, A. P. 1983. A method for computing all solutions to systems of polynomial equations. *ACM. Trans. Math. Software 9*, 1–17.

Morgan, A. P. 1986a. A transformation to avoid solutions at infinity for polynomial systems. *Appl. Math. Comput. 18*, 77–86.

Morgan, A. P. 1986b. A homotopy for solving polynomial systems. *Appl. Math. Comput. 18*, 87–92.

Morgan, A. P. 1987. *Solving polynomial systems using continuation for engineering and scientific problems*. Prentice-Hall, Englewood Cliffs, NJ.

Morgan, A. P., and Sommese, A. J. 1987a. A homotopy for solving general polynomial systems that respects m-homogeneous structures. *Appl. Math. Comput. 24*, 101–113.

Morgan, A. P., and Sommese, A. J. 1987b. Computing all solutions to polynomial systems using homotopy continuation. *Appl. Math. Comput. 24*, 115–138.

Morgan, A. P., Sommese, A. J., and Wampler, C. W. 1991. Computing singular solutions to nonlinear analytic systems. *Numer. Math. 58*, 669–684.

Morgan, A. P., Sommese, A. J., and Wampler, C. W. 1992a. Computing singular solutions to polynomial systems. *Advances Appl. Math. 13*, 305–327.

Morgan, A. P., Sommese, A. J., and Wampler, C. W. 1992b. A power series method for computing singular solutions to nonlinear analytic systems. *Numer. Math. 63*, 391–409.

Morgan, A. P., Sommese, A. J., and Wampler, C. W. 1995. A product-decomposition bound for Bezout numbers. *SIAM J. Numer. Anal. 32*, 1308–1325.

Morgan, A. P., Sommese, A. J., and Watson, L. T. 1989. Finding all isolated solutions to polynomial systems using HOMPACK. *ACM Trans. Math. Software 15*, 93–122.

Morgan, A. P., and Watson, L. T. 1989. A globally convergent parallel algorithm for zeros of polynomial systems. *Nonlinear Anal. 13*, 1339–1350.

Sosonkina, M., Watson, L. T., and Stewart, D. E. 1996. Note on the end game in homotopy zero curve tracking. *ACM Trans. Math. Software 22*, 281–287.

Tsai, L.-W., and Morgan, A. P. 1985. Solving the kinematics of the most general six- and five-degree-of-freedom manipulators by continuation methods. *ASME J. Mechanisms, Transmissions, Aut. Design 107*, 48–57.

van der Waerden, B. L. 1953. *Modern Algebra*. Vols. 1, 2, Frederick Ungar Pub. Co., New York.

Verschelde, J. May, 1996. Homotopy continuation methods for solving polynomial systems. Ph.D. Thesis, Dept. of Computer Sci., Katholieke Univ. Lueven, Lueven, Belgium.

Verschelde, J. 1997. PHCPACK: A general-purpose solver for polynomial systems by homotopy continuation. Preprint.

Verschelde, J., and Cools, R. 1993. Symbolic homotopy construction. *Appl. Algebra Engrg. Comm. Comput. 4*, 169–183.

Verschelde, J., and Haegemans, A. 1993. The GBQ-algorithm for constructing start systems of homotopies for polynomial systems. *SIAM J. Numer. Anal. 30*, 583–594.

Wampler, C. W. 1994. An efficient start system for multi-homogeneous polynomial continuation. *Numer. Math. 66*, 517–523.

Watson, L. T., Billups, S. C., and Morgan, A. P. 1987. Algorithm 652: HOMPACK: A suite of codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software 13*, 281–310.

Watson, L. T., Sosonkina, M., Melville, R. C., Morgan, A. P., and Walker, H. F. 1997. Algorithm 777: HOMPACK90: A suite of Fortran 90 codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software 23*, 514–549.

Wright, A. H. 1985. Finding all solutions to a system of polynomial equations. *Math. Comp. 44*, 125–133.

Appendix A: Fortran 90 Source Code

The Fortran 90 modules REAL_PRECISION, GLOBAL_PLP and POLSYS which are discussed in Chapter 7 are listed here.

```
     MODULE REAL_PRECISION
! This is for 64-bit arithmetic.
     INTEGER, PARAMETER:: R8=SELECTED_REAL_KIND(13)
     END MODULE REAL_PRECISION
MODULE GLOBAL_PLP
!
! The module GLOBAL_PLP contains derived data types, arrays, and
! functions used in POLSYS_PLP and related subroutines.  GLOBAL_PLP uses
! the HOMPACK90 module REAL_PRECISION for 64-bit arithmetic.
USE REAL_PRECISION, ONLY: R8
INTEGER, PARAMETER:: LARGE=SELECTED_INT_KIND(15)
REAL (KIND=R8), PARAMETER:: PI=3.14159265358979323846264433_R8
!
!
! TARGET SYSTEM: Let X be a complex N-dimensional vector.  POLSYS_PLP
! is used to solve the polynomial system, called the target system,
! F(X)=0, where F is represented by the following derived data types:
!
TYPE TERM_TYPE
   COMPLEX (KIND=R8):: COEF
   INTEGER, DIMENSION(:), POINTER:: DEG
END TYPE TERM_TYPE
TYPE POLYNOMIAL_TYPE
   TYPE(TERM_TYPE), DIMENSION(:), POINTER:: TERM
   INTEGER:: NUM_TERMS
END TYPE POLYNOMIAL_TYPE
TYPE(POLYNOMIAL_TYPE), DIMENSION(:), ALLOCATABLE:: POLYNOMIAL
!
! The mathematical representation of the target system F is, for I=1,...,N,
!
! F_I(X) = SUM_J=1^POLYNOMIAL(I)%NUM_TERMS
!         POLYNOMIAL(I)%TERM(J)%COEF *
!         PRODUCT_K=1^N  X(K)**POLYNOMIAL(I)%TERM(J)%DEG(K).
!
! Any program calling POLSYS_PLP (such as the sample main program
! MAIN_TEMPLATE) must aquire data and allocate storage for the target
! system as illustrated below:
!
! ALLOCATE(POLYNOMIAL(N))
! DO I=1,N
!   READ (*,*) POLYNOMIAL(I)%NUM_TERMS
!   ALLOCATE(POLYNOMIAL(I)%TERM(POLYNOMIAL(I)%NUM_TERMS))
!   DO J=1,POLYNOMIAL(I)%NUM_TERMS
!     ALLOCATE(POLYNOMIAL(I)%TERM(J)%DEG(N+1))
!     READ (*,*) POLYNOMIAL(I)%TERM(J)%COEF,POLYNOMIAL(I)%TERM(J)%DEG(1:N)
!   END DO
! END DO
!
! START SYSTEM/PARTITION:  In a partitioned linear product (PLP)
! formulation the start system G(X)=0 and the variable partition
```

```
! P have the same structure.  G and P are represented by the derived data
! types:
!
INTEGER, DIMENSION(:), ALLOCATABLE:: PARTITION_SIZES
TYPE SET_TYPE
   INTEGER, DIMENSION(:), POINTER:: INDEX
   INTEGER:: NUM_INDICES
   INTEGER:: SET_DEG
   COMPLEX (KIND=R8), DIMENSION(:), POINTER:: START_COEF
END TYPE SET_TYPE
TYPE PARTITION_TYPE
   TYPE(SET_TYPE), DIMENSION(:), POINTER:: SET
END TYPE PARTITION_TYPE
TYPE(PARTITION_TYPE), DIMENSION(:), ALLOCATABLE:: PARTITION
!
! The mathematical representation of the start system G is, for I=1,...,N,
!
! G_I(X) = PRODUCT_J=1^PARTITION_SIZES(I)
!         ( L(I,J)**PARTITION(I)SET(J)%SET_DEG - 1.0 ),
!
! where the linear factors L(I,J) are
!
! L(I,J) = SUM_K=1^PARTITION(I)%SET(J)%NUM_INDICES
!   PARTITION(I)%SET(J)%START_COEF(K) * X(PARTITION(I)%SET(J)%INDEX(K)).
!
! The system partition P=(P(1),...,P(N)) is comprised of the component
! partitions P(I) = S(I,1),...S(I, PARTITION_SIZES(I)), where the sets
! of variables S(I,J) are defined by
!
! S(I,J) = UNION_K=1^PARTITION(I)%SET(J)%NUM_INDICES
!          X(PARTITION(I)%SET(J)%INDEX(K)) .
!
! The calling program must acquire data and allocate storage as
! illustrated below:
!
! ALLOCATE(PARTITION_SIZES(N))
! READ (*,*) PARTITION_SIZES(1:N)
! ALLOCATE(PARTITION(N))
! DO I=1,N
!   ALLOCATE(PARTITION(I)%SET(PARTITION_SIZES(I))
!   DO J=1, PARTITION_SIZES(I)
!     READ (*,*) PARTITION(I)%SET(J)%NUM_INDICES
!     ALLOCATE(PARTITION(I)%SET(J)%INDEX(PARTITION(I)%SET(J)%NUM_INDICES))
!     READ (*,*) PARTITION(I)%SET(J)%INDEX
!   END DO
! END DO
!
! SET_DEG and START_COEF are calculated by POLSYS_PLP.
!
!
!
CONTAINS
! INDEXING FUNCTIONS FOR THE TARGET SYSTEM:
!
! C(I,J) retrieves the coefficient of the Jth term of the Ith polynomial
! component of the target system.
```

```
COMPLEX (KIND=R8) FUNCTION C(I,J)
  INTEGER:: I,J
  C=POLYNOMIAL(I)%TERM(J)%COEF
END FUNCTION C
!
! D(I,J,K) retrieves the degree of the Kth variable in the Jth term of
! the Ith polynomial component of the target system.
INTEGER FUNCTION D(I,J,K)
  INTEGER:: I,J,K
  D=POLYNOMIAL(I)%TERM(J)%DEG(K)
END FUNCTION D
!
! NUMT(I) retrieves the number of terms in the Ith polynomial component of
! the target system F(X).
INTEGER FUNCTION NUMT(I)
  INTEGER:: I
  NUMT=POLYNOMIAL(I)%NUM_TERMS
END FUNCTION NUMT
!
! The target system is succinctly specified with the retrieval functions:
!
! F_I(X) = SUM_J=1^NUMT(I) C(I,J) * PRODUCT_K=1^N  X(K)**D(I,J,K).
!
! INDEXING FUNCTIONS FOR THE START SYSTEM/PARTITION:
!
! PAR(I,J,K) retrieves the index of the Kth variable in the Jth set
! S(I,J) of the Ith partition P(I).
INTEGER FUNCTION PAR(I,J,K)
  INTEGER:: I,J,K
  PAR=PARTITION(I)%SET(J)%INDEX(K)
END FUNCTION PAR
!
! SC(I,J,K) retrieves the coefficient of the variable with index
! PAR(I,J,K) in the Jth factor of the Ith component of the start system
! G(X).
COMPLEX (KIND=R8) FUNCTION SC(I,J,K)
  INTEGER:: I,J,K
  SC=PARTITION(I)%SET(J)%START_COEF(K)
END FUNCTION SC
!
! SD(I,J) retrieves the set degree of the Jth set S(I,J) in the Ith
! partition P(I).
INTEGER FUNCTION SD(I,J)
  INTEGER:: I,J
  SD=PARTITION(I)%SET(J)%SET_DEG
END FUNCTION SD
!
! NUMV(I,J) retrieves the number of variables in the Jth set S(I,J) of
! the Ith partition P(I).
INTEGER FUNCTION NUMV(I,J)
  INTEGER:: I,J
  NUMV=PARTITION(I)%SET(J)%NUM_INDICES
END FUNCTION NUMV
!
! Both the start system and the partition are succinctly specified with
! retrieval functions:
```

```
!
! G_I(X) = PRODUCT_J=1^PARTITION_SIZES(I)
! ( [ SUM_K=1^NUMV(I,J) SC(I,J,K)*X(PAR(I,J,K)) ]**SD(I,J) - 1.0 ),
!
! and P(I) =  S(I,1),...,S(I,PARTITION_SIZES(I)) , where
!
! S(I,J) = UNION_K=1^NUMV(I,J)  X(PAR(I,J,K)) .
!
END MODULE GLOBAL_PLP
MODULE POLSYS
! This module contains the subroutines POLSYS_PLP (finds all or some of
! the roots of a polynomial system defined in the module GLOBAL_PLP),
! BEZOUT_PLP (computes the generalized Bezout number), and SINGSYS_PLP
! (checks the nonsingularity of a generic start point).  Typically a
! user would only call POLSYS_PLP, and thus include in their main
! program the statements:
!   USE GLOBAL_PLP
!   USE POLSYS, ONLY: POLSYS_PLP
! An expert user might want to call BEZOUT_PLP or SINGSYS_PLP
! separately, and thus these routines are also provided as module
! procedures.
!
USE GLOBAL_PLP
CONTAINS
SUBROUTINE POLSYS_PLP(N,TRACKTOL,FINALTOL,SINGTOL,SSPAR,BPLP,IFLAG1,   &
          IFLAG2,ARCLEN,LAMBDA,ROOTS,NFE,SCALE_FACTORS,             &
          NUMRR,RECALL,NO_SCALING,USER_F_DF)
!
! Using a probability-one globally convergent homotopy method,
! POLSYS_PLP finds all finite isolated complex solutions to a system
! F(X) = 0 of N polynomial equations in N unknowns with complex
! coefficients.  A partitioned linear product (PLP) formulation is used
! for the start system of the homotopy map.
!
! POLSYS_PLP uses the module GLOBAL_PLP, which contains the definition
! of the polynomial system to be solved, and also defines the notation
! used below.  The user may also find it beneficial at some point to
! refer to the documentation for STEPNX in the HOMPACK90 package.
!
! The representation of F(X) is stored in the module GLOBAL_PLP.  Using
! the same notation as GLOBAL_PLP, F(X) is defined mathematically by
!
! F_I(X)=SUM_J=1^NUMT(I) C(I,J) * PRODUCT_K=1^N X(K)**D(I,J,K),
!
! for I=1,...,N.
!
! POLSYS_PLP features target system scaling, a projective
! transformation so that the homotopy zero curves are tracked in complex
! projective space, and a partitioned linear product (PLP) formulation of
! the start system.  Scaling may be disabled by the optional argument
! NO_SCALING.  Whatever the case, the roots of F(X) are always returned
! unscaled and untransformed.  The PLP partition (an m-homogeneous
! partition of the variables, possibly different for each component
! F_I(X)) is defined in the module GLOBAL_PLP.
!
! Scaling is carried out in the internal subroutine SCALE_PLP, and is
```

```
! an independent preprocessing step.  SCALE_PLP modifies the polynomial
! coefficients and creates and stores unscaling factors SCALE_FACTORS
! for the variables X(I).  The problem is solved with the scaled
! coefficients and scaled variables.  The coefficients of the target
! polynomial system, which are contained in the global structure
! POLYNOMIAL, remain in modified form on return from POLSYS_PLP.
!
! With the projective transformation, the system is essentially recast in
! homogeneous coordinates, Z(1),...,Z(N+1), and solved in complex
! projective space.  The resulting solutions are untransformed via
! X(I) = Z(I)/Z(N+1), I=1,...N, unless this division would cause
! overflow, in which case Re(X(I)) = Im(X(I)) = HUGE(1.0_R8).
! On return, for the Jth path, ROOTS(I,J) = X(I) for I=1,...,N, and
! ROOTS(N+1,J) = Z(N+1), the homogeneous variable.
!
! In the PLP scheme the number of paths that must be tracked can be
! less, and commonly far less, than the "total degree" because of the
! specialized start system G(X) = 0.  The structure of the start system
! is determined by the system partition P.  The representations of both
! are stored in the module GLOBAL_PLP, and following the comments there,
! are defined mathematically as follows:
!
! The system partition P=(P(1),...,P(N)) is comprised of the component
! partitions P(I)=S(I,1),...,S(I,PARTITION_SIZES(I)), where the sets of
! variables S(I,J) are defined by
!
! S(I,J) = UNION_K=1^NUMV(I,J) X(PAR(I,J,K)).
!
! The only restriction on the system partition P is that each component
! partition P(I) should be a partition of the set X(1),...,X(N), that
! is, the three following properties should hold for each I=1,...,N:
!
! i)    each set S(I,J) has cardinality NUMV(I,J) > 0,
!
! ii)   S(I,J1) INTERSECTION S(I,J2) =  , for J1 /= J2, and
!
! iii)  UNION_J=1^PARTITION_SIZES(I) S(I,J) = X(1),...,X(N).
!
! The start system is defined mathematically, for I=1,...,N, by
!
! G_I(X) = PRODUCT_J=1^PARTITION_SIZES(I) ( L(I,J)**SD(I,J)-1.0 ),
!
! where the linear factors L(I,J) are
!
! L(I,J) = SUMK=1^NUMV(I,J) SC(I,J,K)*X(PAR(I,J,K)).
!
! Contained in this module (POLSYS) is the routine BEZOUT_PLP.  This
! routine calculates the generalized PLP Bezout number, based on the
! system partition P provided by the user, by counting the number of
! solutions to the start system.  The user is encouraged to explore
! several system partitions with BEZOUT_PLP before calling POLSYS_PLP.
! See the sample calling program MAIN_TEMPLATE and the comments in
! BEZOUT_PLP.
!
! Internal routines:  INIT_PLP, INTERP, OUTPUT_PLP, RHO, ROOT_OF_UNITY,
!   ROOT_PLP, SCALE_PLP, START_POINTS_PLP, START_SYSTEM, TANGENT_PLP,
```

```
!   TARGET_SYSTEM.
!
! External routines called:  BEZOUT_PLP, SINGSYS_PLP, STEPNX.
!
!
! On input:
!
! N is the dimension of the target polynomial system.
!
! TRACKTOL is the local error tolerance allowed the path tracker along
!   the path.  ABSERR and RELERR (of STEPNX) are set to TRACKTOL.
!
! FINALTOL is the accuracy desired for the final solution.  It is used
!   for both the absolute and relative errors in a mixed error criterion.
!
! SINGTOL is the singularity test threshold used by SINGSYS_PLP.  If
!   SINGTOL <= 0.0 on input, then SINGTOL is reset to a default value.
!
! SSPAR(1:8) = (LIDEAL, RIDEAL, DIDEAL, HMIN, HMAX, BMIN, BMAX, P) is a
!   vector of parameters used for the optimal step size estimation. If
!   SSPAR(I) <= 0.0 on input, it is reset to a default value by STEPNX.
!   See the comments in STEPNX for more information.
!
! Optional arguments:
!
! NUMRR is the number of multiples of 1000 steps that will be tried before
!   abandoning a path.  If absent, NUMRR is taken as 1.
!
! RECALL is used to retrack certain homotopy paths.  It's use assumes
!   BPLP contains the Bezout number (which is not recalculated),
!   SCALE_FACTORS contains the variable unscaling factors, and that
!   IFLAG2(1:BPLP) exists.  The Ith homotopy path is retracked if
!   IFLAG2(I) = -2, and skipped otherwise.
!
! NO_SCALING indicates that the target polynomial is not to be scaled.
!   Scaling is done by default when NO_SCALING is absent.
!
! USER_F_DF indicates (when present) that the user is providing a subroutine
!   TARGET_SYSTEM_USER to evaluate the (complex) target system F(XC) and
!   its (complex) N x N Jacobian matrix DF(XC).  XC(1:N+1) is in
!   complex projective coordinates, and the homogeneous coordinate XC(N+1)
!   is explicitly eliminated from F(XC) and DF(XC) using the projective
!   transformation (cf. the comments in START_POINTS_PLP).
!
!
! The following objects must be allocated and defined as described in
! GLOBAL_PLP:
!
! POLYNOMIAL(I)%NUM_TERMS is the number of terms in the Ith component
!   F_I(X) of the target polynomial system, for I=1,...,N.
!
! POLYNOMIAL(I)%TERM(J)%COEF is the coefficient of the Jth term in the Ith
!   component of the target polynomial system, for J=1,...,NUMT(I), and
!   I=1,...,N.
!
! POLYNOMIAL(I)%TERM(J)%DEG(K) is the degree of the Kth variable in the
```

```
!   Jth term of the Ith component of the target polynomial system, for
!   K=1,...,N, J=1,...NUMT(I), and I=1,...,N.
!
! PARTITION_SIZES(I) is the number of sets in the Ith component
!   partition P(I), for I=1,...,N.
!
! PARTITION(I)%SET(J)%NUM_INDICES is the number of indices stored in the
!   Jth set S(I,J) of the Ith component partition P(I), for
!   J=1,...,PARTITION_SIZES(I), and I=1,...,N.
!
! PARTITION(I)SET(J)%INDEX(K) is the index of the Kth variable stored
!   in the Jth set S(I,J) of the Ith component partition P(I).
!
!
! On output:
!
! BPLP is the generalized Bezout number corresponding to the
!   partitioned linear product (PLP) formulation defined by the system
!   partition P.  This is the number of paths tracked and the number of
!   roots returned (counting multiplicity).
!
! IFLAG1
!   = 0  for a normal return.
!
!   = -1 if either POLYNOMIAL or PARTITION was improperly allocated.
!
!   = -2 if any POLYNOMIAL(I)%TERM(J)%DEG(K) is less than zero.
!
!   = -3 if F_I(X) = CONSTANT for some I.
!
!   = -4 if SUM_J=1^PARTITION_SIZES(I)
!         PARTITION(I)SET(J)%NUM_INDICES /= N, for some I.
!
!   = -5 if UNION_J=1^PARTITION_SIZES
!              S(I,J) /= 1,2,...,N-1,N, for some I.
!
!   = -6 if the optional argument RECALL was present but any of BPLP
!       or the arrays ARCLEN, IFLAG2, LAMBDA, NFE, ROOTS are
!       inconsistent with the previous call to POLSYS_PLP.
!
!   = -7 if the array SCALE_FACTORS is too small.
!
! IFLAG2(1:BPLP) is an integer array which returns information about
!   each path tracked.  Precisely, for each path I that was tracked,
!   IFLAG2(I):
!   = 1 + 10*C, where C is the cycle number of the path, for a normal return.
!
!   = 2 if the specified error tolerance could not be met.  Increase
!      TRACKTOL and rerun.
!
!   = 3 if the maximum number of steps allowed was exceeded.  To track
!      the path further, increase NUMRR and rerun the path.
!
!   = 4 if the Jacobian matrix does not have full rank.  The algorithm has
!      failed (the zero curve of the homotopy map cannot be followed any
!      further).
```

```
!
!   = 5 if the tracking algorithm has lost the zero curve of the homotopy
!       map and is not making progress.  The error tolerances TRACKTOL and
!       FINALTOL were too lenient.  The problem should be restarted with
!       smaller error tolerances.
!
!   = 6 if the normal flow Newton iteration in STEPNX or ROOT_PLP failed
!       to converge.  The error error tolerances TRACKTOL or FINALTOL may
!       be too stringent.
!
!   = 7 if ROOT_PLP failed to find a root in 10*NUMRR iterations.
!
! ARCLEN(I) is the approximate arc length of the Ith path, for I=1,...,BPLP.
!
! LAMBDA(I), if MOD(IFLAG2(I),10) = 1, contains an error estimate of
!   the normalized residual of the scaled, transformed polynomial
!   system of equations at the scaled, transformed root for the Ith path
!   (LAMBDA for this path is assumed to be 1). Otherwise LAMBDA(I) is the
!   final value of the homotopy parameter lambda on the Ith path, for
!   I=1,...,BPLP.
!
! ROOTS(1:N,I) are the complex roots (untransformed and unscaled) of
!   the target polynomial corresonding to the Ith path, for I=1,...,BPLP.
!
! ROOTS(N+1,I) is the homogeneous variable of the target polynomial
!   system in complex projective space corresponding to ROOTS(1:N,I).
!
! NFE(I) is the number of Jacobian matrix evaluations required to track
!   the Ith path, for I=1,...,BPLP.
!
! SCALE_FACTORS(1:N) contains the unscaling factors for the variables X(I).
!   These are needed only on a recall when scaling was done on the original
!   call to POLSYS_PLP (NO_SCALING was absent).
!
!
USE GLOBAL_PLP
INTEGER, INTENT(IN):: N
REAL (KIND=R8), INTENT(IN):: TRACKTOL, FINALTOL
REAL (KIND=R8), INTENT(IN OUT):: SINGTOL
REAL (KIND=R8), DIMENSION(8), INTENT(IN OUT):: SSPAR
INTEGER, INTENT(IN OUT):: BPLP, IFLAG1
INTEGER, DIMENSION(:), POINTER:: IFLAG2
REAL (KIND=R8), DIMENSION(:), POINTER:: ARCLEN, LAMBDA
COMPLEX (KIND=R8), DIMENSION(:,:), POINTER:: ROOTS
INTEGER, DIMENSION(:), POINTER:: NFE
REAL (KIND=R8), DIMENSION(:), INTENT(IN OUT):: SCALE_FACTORS
INTEGER, OPTIONAL, INTENT(IN):: NUMRR
LOGICAL, OPTIONAL, INTENT(IN):: RECALL, NO_SCALING, USER_F_DF
!
INTERFACE
  SUBROUTINE STEPNX(N,NFE,IFLAG,START,CRASH,HOLD,H,RELERR,   &
     ABSERR,S,Y,YP,YOLD,YPOLD,A,TZ,W,WP,RHOLEN,SSPAR)
    USE REAL_PRECISION
    INTEGER, INTENT(IN):: N
    INTEGER, INTENT(IN OUT):: NFE,IFLAG
    LOGICAL, INTENT(IN OUT):: START,CRASH
```

```
    REAL (KIND=R8), INTENT(IN OUT):: HOLD,H,RELERR,ABSERR,S,RHOLEN, &
      SSPAR(8)
    REAL (KIND=R8), DIMENSION(:), INTENT(IN):: A
    REAL (KIND=R8), DIMENSION(:), INTENT(IN OUT):: Y,YP,YOLD,YPOLD, &
      TZ,W,WP
    REAL (KIND=R8), DIMENSION(:), ALLOCATABLE, SAVE:: Z0,Z1
  END SUBROUTINE STEPNX
END INTERFACE
!
! Local variables.
INTEGER:: BTEMP, I, IFLAG, II, ITER, J, JJ, K, KK, L, LIMIT, M, MAXPS, &
  MAXT, NNFE, NUM_RERUNS, ROOT_COUNT
INTEGER, SAVE:: BPLP_SAVE
INTEGER, DIMENSION(N):: CHECK_PAR, DLEX_NUM, DLEX_SAVE, FLEX_NUM, FLEX_SAVE
INTEGER, DIMENSION(2*N+1):: PIVOT
REAL (KIND=R8):: ABSERR, H, HOLD, RELERR, RHOLEN, S
REAL (KIND=R8), DIMENSION(2*N):: A, DRHOL, RHOV, Z
REAL (KIND=R8), DIMENSION(2*N+1):: Y, YP, YOLD, YOLDS, YPOLD, TZ, W, WP
REAL (KIND=R8), DIMENSION(3*(2*N+1)):: ALPHA
REAL (KIND=R8), DIMENSION(2*N+1,12):: YS
REAL (KIND=R8), DIMENSION(N,N):: RAND_MAT
REAL, DIMENSION(N,N):: RANDNUMS
REAL (KIND=R8), DIMENSION(N+1,N):: MAT
REAL (KIND=R8), DIMENSION(2*N,2*N):: DRHOX
REAL (KIND=R8), DIMENSION(2*N,2*N+2):: QR
COMPLEX (KIND=R8), DIMENSION(N-1):: TAU
COMPLEX (KIND=R8), DIMENSION(N):: B, F, G, V
COMPLEX (KIND=R8), DIMENSION(N+1):: PROJ_COEF, XC
COMPLEX (KIND=R8), DIMENSION(N,N):: AA
COMPLEX (KIND=R8), DIMENSION(N,N+1):: DF, DG
COMPLEX (KIND=R8), DIMENSION(:,:), ALLOCATABLE:: TEMP1G, TEMP2G
LOGICAL:: CRASH, NONSING, START
!
! Begin input data check.
!
IFLAG1=0  ! Normal return.
! Check that dimensions are valid.
IF ((N <= 0) .OR. (SIZE(POLYNOMIAL) /= N)         &
           .OR. ANY((/(NUMT(I),I=1,N)/) <= 0)    &
           .OR. (SIZE(PARTITION) /= N)          &
           .OR. ANY(PARTITION_SIZES <=0)) THEN
  IFLAG1=-1
  RETURN
END IF
DO I=1,N
  IF ((SIZE(POLYNOMIAL(I)%TERM) /= NUMT(I))                &
    .OR. (SIZE(PARTITION(I)%SET) /= PARTITION_SIZES(I))      &
    .OR. ANY((/(NUMV(I,J),J=1,PARTITION_SIZES(I))/) <= 0)) THEN
    IFLAG1=-1
    RETURN
  END IF
END DO
DO I=1,N
  DO J=1,NUMT(I)
    IF (SIZE(POLYNOMIAL(I)%TERM(J)%DEG) /= N+1) THEN
      IFLAG1=-1
```

```
      RETURN
    END IF
  END DO
  DO J=1,PARTITION_SIZES(I)
    IF (SIZE(PARTITION(I)%SET(J)%INDEX) /= NUMV(I,J)) THEN
      IFLAG1=-1
      RETURN
    END IF
  END DO
END DO
! Check that the target system has no negative powers.
DO I=1,N
  DO J=1,NUMT(I)
    IF (ANY(POLYNOMIAL(I)%TERM(J)%DEG(1:N) < 0)) THEN
      IFLAG1=-2
      RETURN
    END IF
  END DO
END DO
! Check that the target system has no constant-valued components.
DO I=1,N
  IF (ALL( (/( SUM(POLYNOMIAL(I)%TERM(J)%DEG(1:N)),   &
                            J=1,NUMT(I) )/) == 0)) THEN
    IFLAG1=-3
    RETURN
  END IF
END DO
! Check that the system partition is valid.
DO I=1,N
  IF (SUM( (/(NUMV(I,J),J=1,PARTITION_SIZES(I))/) ) /= N) THEN
    IFLAG1=-4
    RETURN
  END IF
  CHECK_PAR(1:N)=0
  DO J=1,PARTITION_SIZES(I)
    DO K=1,NUMV(I,J)
      CHECK_PAR(PAR(I,J,K))=CHECK_PAR(PAR(I,J,K))+1
    END DO
  END DO
  IF (ANY(CHECK_PAR /= 1)) THEN
    IFLAG1=-5
    RETURN
  END IF
END DO
! Check consistency on a recall.
IF (PRESENT(RECALL)) THEN
  IF ( (BPLP /= BPLP_SAVE) .OR. (SIZE(ARCLEN) < BPLP)            &
                        .OR. (SIZE(IFLAG2) < BPLP)            &
                        .OR. (SIZE(LAMBDA) < BPLP)            &
                        .OR. (SIZE(NFE) < BPLP)               &
                        .OR. (SIZE(ROOTS,DIM=2) < BPLP) ) THEN
    IFLAG1=-6
    RETURN
  END IF
END IF
! Check SCALE_FACTORS array size.
```

```
IF (SIZE(SCALE_FACTORS) < N) THEN
  IFLAG1=-7
  RETURN
END IF
!
! End input data check.
!
! Initialize the POINTER aguments of POLSYS_PLP.
MAXT=MAXVAL((/(NUMT(I),I=1,N)/))
IF ( .NOT. PRESENT(RECALL)) THEN
  CALL BEZOUT_PLP(N,MAXT,SINGTOL,BPLP)
  BPLP_SAVE=BPLP  ! Save Bezout number for recall check.
  IF (ASSOCIATED(ARCLEN)) THEN
    IF (SIZE(ARCLEN) < BPLP) THEN
      DEALLOCATE(ARCLEN); ALLOCATE(ARCLEN(BPLP))
    END IF
  ELSE
    ALLOCATE(ARCLEN(BPLP))
  END IF
  IF (ASSOCIATED(IFLAG2)) THEN
    IF (SIZE(IFLAG2) < BPLP) THEN
      DEALLOCATE(IFLAG2); ALLOCATE(IFLAG2(BPLP))
    END IF
  ELSE
    ALLOCATE(IFLAG2(BPLP))
  END IF
  IFLAG2=-2
  IF (ASSOCIATED(NFE)) THEN
    IF (SIZE(NFE) < BPLP) THEN
      DEALLOCATE(NFE); ALLOCATE(NFE(BPLP))
    END IF
  ELSE
    ALLOCATE(NFE(BPLP))
  END IF
  IF (ASSOCIATED(LAMBDA)) THEN
    IF (SIZE(LAMBDA) < BPLP) THEN
      DEALLOCATE(LAMBDA); ALLOCATE(LAMBDA(BPLP))
    END IF
  ELSE
    ALLOCATE(LAMBDA(BPLP))
  END IF
  IF (ASSOCIATED(ROOTS)) THEN
    IF (SIZE(ROOTS,DIM=2) < BPLP .OR. SIZE(ROOTS,DIM=1) < N+1) THEN
      DEALLOCATE(ROOTS); ALLOCATE(ROOTS(N+1,BPLP))
    END IF
  ELSE
    ALLOCATE(ROOTS(N+1,BPLP))
  END IF
END IF
!
! Allocate storage for the start system.
DO I=1,N
  DO J=1,PARTITION_SIZES(I)
    ALLOCATE(PARTITION(I)%SET(J)%START_COEF(NUMV(I,J)))
  END DO
END DO
```

```
!
! Allocate working space for homotopy map derivative calculation.
MAXPS=MAXVAL(PARTITION_SIZES)
ALLOCATE(TEMP1G(N,MAXPS), TEMP2G(N,MAXPS))
!
! Get real random numbers uniformly distributed in [-1,-1/2] union
! [1/2,1] for RAND_MAT, which is used in SINGSYS_PLP.
CALL RANDOM_NUMBER(HARVEST=RANDNUMS)
RANDNUMS=RANDNUMS-0.5+SIGN(0.5,RANDNUMS-0.5)
RAND_MAT=REAL(RANDNUMS,KIND=R8)
!
! Set default value for singularity threshold SINGTOL in SINGSYS_PLP.
IF (SINGTOL <= REAL(N,KIND=R8)*EPSILON(1.0_R8))  &
    SINGTOL=SQRT(EPSILON(1.0_R8))
!
! Scale the target polynomial system as requested.
IF (PRESENT(NO_SCALING)) THEN
  SCALE_FACTORS=0.0_R8
ELSE IF (.NOT. PRESENT(RECALL)) THEN
  CALL SCALE_PLP
END IF
!
! Initialize the start system for the homotopy map.
CALL INIT_PLP
!
! Set main loop initial values.
FLEX_NUM(1:N-1)=1
FLEX_NUM(N)=0
FLEX_SAVE=0
ROOT_COUNT=0
IF (PRESENT(NUMRR)) THEN
  NUM_RERUNS=MAX(NUMRR,1)
ELSE
  NUM_RERUNS=1
END IF
!
! Main loop over all possible lexicographic vectors FLEX_NUM(1:N)
! corresponding to linear factors.
MAIN_LOOP: DO
  DO J=N,1,-1
    IF(FLEX_NUM(J) < PARTITION_SIZES(J)) THEN
      K=J
      EXIT
    END IF
  END DO
  FLEX_NUM(K)=FLEX_NUM(K)+1
  IF(K+1 <= N) FLEX_NUM(K+1:N)=1
!
! Check if the subsystem of the start system defined by the
! lexicographic vector FLEX_NUM is singular.
  CALL SINGSYS_PLP(N,FLEX_NUM,FLEX_SAVE,SINGTOL,RAND_MAT,MAT,NONSING)
!
! If the subsystem is nonsingular, track a path.
  NONSING_START_POINT: IF (NONSING) THEN
  BTEMP=PRODUCT( (/(SD(I,FLEX_NUM(I)),I=1,N)/) )
  DLEX_NUM(1:N-1)=1
```

```
  DLEX_NUM(N)=0
  DLEX_SAVE=0
!
! Cycle through all lexicographic vectors DLEX_NUM(1:N) corresponding
! to roots of unity, defined by the set degrees specified in
! (/(SD(I,FLEX_NUM(I)),I=1,N)/).
  SD_LEX_LOOP: DO II=1,BTEMP
    DO JJ=N,1,-1
      IF(DLEX_NUM(JJ) < SD(JJ,FLEX_NUM(JJ))) THEN
        KK=JJ
        EXIT
      END IF
    END DO
    DLEX_NUM(KK)=DLEX_NUM(KK)+1
    IF(KK+1 <= N) DLEX_NUM(KK+1:N)=1
    ROOT_COUNT=ROOT_COUNT+1
    IF (IFLAG2(ROOT_COUNT) /= -2) CYCLE SD_LEX_LOOP
!
! Get the start point for the homotopy path defined by FLEX_NUM and
! DLEX_NUM.
    CALL START_POINTS_PLP
!
    NNFE=0
    IFLAG=-2
    Y(1)=0.0_R8; Y(2:2*N+1)=Z(1:2*N)
    YP(1)=1.0_R8;  YP(2:2*N+1)=0.0_R8
    YOLD=Y;  YPOLD=YP
    HOLD=1.0_R8; H=0.1_R8
    S=0.0_R8
    LIMIT=1000*NUM_RERUNS
    START=.TRUE.
    CRASH=.FALSE.
!
! Track the homotopy path.
    TRACKER: DO ITER=1,LIMIT
      IF (Y(1) < 0.0_R8) THEN
        IFLAG=5
        EXIT TRACKER
      END IF
!
! Set different error tolerance if the trajectory Y(S) has any high
! curvature components.
      RELERR=TRACKTOL
      ABSERR=TRACKTOL
      IF (ANY(ABS(YP-YPOLD) > 10.0_R8*HOLD)) THEN
        RELERR=FINALTOL
        ABSERR=FINALTOL
      END IF
!
! Take a step along the homotopy zero curve.
      CALL STEP_PLP
      IF (IFLAG > 0) EXIT TRACKER
      IF (Y(1) >= .97_R8) THEN
        RELERR = FINALTOL
        ABSERR = FINALTOL
! Enter end game.
```

```
       CALL ROOT_PLP
       EXIT TRACKER
      END IF
!
! D LAMBDA/DS >= 0 necessarily.  This condition is forced here.
     IF (YP(1) < 0.0_R8) THEN
! Reverse the tangent direction so D LAMBDA/DS = YP(1) > 0.
       YP=-YP
       YPOLD=YP
! Force STEPNX to use the linear predictor for the next step only.
       START=.TRUE.
     END IF
   END DO TRACKER
!
! Set error flag if limit on number of steps exceeded.
   IF (ITER >= LIMIT) IFLAG=3
!
   ARCLEN(ROOT_COUNT)=S
   NFE(ROOT_COUNT)=NNFE
   IFLAG2(ROOT_COUNT)=IFLAG
   LAMBDA(ROOT_COUNT)=Y(1)
!
! Convert from real to complex arithmetic.
   XC(1:N)=CMPLX(Y(2:2*N:2),Y(3:2*N+1:2),KIND=R8)
!
! Untransform and unscale solutions.
   CALL OUTPUT_PLP
   ROOTS(1:N,ROOT_COUNT)=XC(1:N)
   ROOTS(N+1,ROOT_COUNT)=XC(N+1)
  END DO SD_LEX_LOOP
  END IF NONSING_START_POINT
!
  IF(ALL(FLEX_NUM == PARTITION_SIZES)) EXIT MAIN_LOOP
END DO MAIN_LOOP
!
! Clean up working storage in STEPNX.
IFLAG=-42
CALL STEPNX (2*N,NNFE,IFLAG,START,CRASH,HOLD,H,RELERR, &
  ABSERR,S,Y,YP,YOLD,YPOLD,A,TZ,W,WP,RHOLEN,SSPAR)
!
! Deallocate the storage for the start system and working storage.
DO I=1,N
  DO J=1,PARTITION_SIZES(I)
    DEALLOCATE(PARTITION(I)%SET(J)%START_COEF)
  END DO
END DO
DEALLOCATE(TEMP1G,TEMP2G)
RETURN
!
CONTAINS
!
!
SUBROUTINE SCALE_PLP
! SCALE_PLP scales the complex coefficients of a polynomial system of N
! equations in N unknowns, F(X)=0, where the Jth term of the Ith equation
! looks like
```

```
!
!   C(I,J) * X(1)**D(I,J,1) ... X(N)**D(I,J,N).
!
! The Ith equation is scaled by 10**FACE(I).  The Kth variable is scaled
! by 10**FACV(K).  In other words, X(K)=10**FACV(K)*Y(K), where Y solves
! the scaled equation.  The scaled equation has the same form as the
! original, except that CSCL(I,J) replaces POLYNOMIAL(I)%TERM(J)%COEF,
! where
!
! CSCL(I,J)=C(I,J)*10**(FACE(I)+FACV(1)*D(I,J,1)+...+FACV(N)*D(I,J,N)).
!
! The criterion for generating FACE and FACV is that of minimizing the
! sum of squares of the exponents of the scaled coefficients.  It turns
! out that this criterion reduces to solving a single linear system,
! ALPHA*X=BETA, as defined in the code below.  See Meintjas and Morgan,
! "A methodology for solving chemical equilibrium problems," General
! Motors Research Laboratories Technical Report GMR-4971.
!
! Calls the LAPACK routines DGEQRF, DORMQR, and the BLAS routines
! DTRSV and IDAMAX.
!
! On exit:
!
! SCALE_FACTORS(K) = FACV(K) is the scale factor for X(K), K=1,...,N.
!  Precisely, the unscaled solution
!  X(K) = 10**FACV(K) * (computed scaled solution).
!
! POLYNOMIAL(I)%TERM(J)%COEF = CSCL(I,J) is the scaled complex
!  coefficient, for J=1,...,NUMT(I), and I=1,...,N.
!
! Local variables.
INTEGER:: COUNT, I, ICMAX, IRMAX, J, K, L, LENR
INTEGER, DIMENSION(N):: NNUMT
INTEGER, DIMENSION(N,MAXT,N):: DDEG
REAL (KIND=R8):: DUM, RTOL, TUM
REAL (KIND=R8), DIMENSION(:), POINTER:: FACE, FACV
REAL (KIND=R8), DIMENSION(2*N), TARGET:: BETA, RWORK, XWORK
REAL (KIND=R8), DIMENSION(2*N,2*N):: ALPHA
REAL (KIND=R8), DIMENSION(N,MAXT):: CMAG
INTERFACE
  INTEGER FUNCTION IDAMAX(N,X,STRIDE)
    USE REAL_PRECISION
    INTEGER:: N,STRIDE
    REAL (KIND=R8), DIMENSION(N):: X
  END FUNCTION IDAMAX
END INTERFACE
!
LENR=N*(N+1)/2
SCALE_FACTORS(1:N)=0.0_R8 ! This corresponds to no scaling.
!
! Delete exact zero coefficients, just for scaling.
NNUMT = 0
DO I=1,N
  COUNT=0
  DO J=1,NUMT(I)
    IF (ABS(C(I,J)) > 0.0_R8) THEN
```

```
      COUNT=COUNT+1
      NNUMT(I)=NNUMT(I)+1
      CMAG(I,COUNT)=LOG10(ABS(C(I,J)))
      DDEG(I,COUNT,1:N)=(/(D(I,J,K),K=1,N)/)
    END IF
  END DO
END DO
!
! Generate the matrix ALPHA.
ALPHA(1:N,1:N)=0.0_R8
DO I=1,N
  ALPHA(I,I)=REAL(NNUMT(I),KIND=R8)
END DO
DO I=1,N
  ALPHA(N+1:2*N,I)=REAL(SUM(DDEG(I,1:NNUMT(I),1:N),DIM=1),KIND=R8)
END DO
DO L=1,N
  DO K=1,L
    ICMAX=0
    DO I=1,N
      ICMAX=ICMAX+DOT_PRODUCT(DDEG(I,1:NNUMT(I),L),DDEG(I,1:NNUMT(I),K))
    END DO
    ALPHA(N+L,N+K)=REAL(ICMAX,KIND=R8)
    ALPHA(N+K,N+L)=ALPHA(N+L,N+K)
  END DO
END DO
ALPHA(1:N,N+1:2*N)=TRANSPOSE(ALPHA(N+1:2*N,1:N))
!
! Compute the QR-factorization of the matrix ALPHA.
CALL DGEQRF(2*N,2*N,ALPHA,2*N,XWORK,BETA,2*N,I)
!
! Check for ill-conditioned scaling matrix.
IRMAX=1
ICMAX=1
DO J=2,N
  I=IDAMAX(J,ALPHA(1,J),1)
  IF (ABS(ALPHA(I,J)) > ABS(ALPHA(IRMAX,ICMAX))) THEN
    IRMAX=I
    ICMAX=J
  END IF
END DO
RTOL=ABS(ALPHA(IRMAX,ICMAX))*EPSILON(1.0_R8)*REAL(N,KIND=R8)
DO I=1,N
  IF (ABS(ALPHA(I,I)) < RTOL) THEN  ! ALPHA is ill conditioned.
    RETURN  ! Default to no scaling at all.
  END IF
END DO
!
! Generate the column BETA.
DO K=1,N
  BETA(K)=-SUM(CMAG(K,1:NNUMT(K)))
  TUM=0.0_R8
  DO I=1,N
    TUM=TUM+SUM(CMAG(I,1:NNUMT(I))*REAL(DDEG(I,1:NNUMT(I),K),KIND=R8))
  END DO
  BETA(N+K)=-TUM
```

```
END DO
!
! Solve the linear system ALPHA*X=BETA.
CALL DORMQR('L','T',2*N,1,2*N-1,ALPHA,2*N,XWORK,BETA,2*N,RWORK,2*N,I)
CALL DTRSV('U','N','N',2*N,ALPHA,2*N,BETA,1)
!
! Generate FACE, FACV, and the scaled coefficients CSCL(I,J).
FACE => BETA(1:N)
FACV => BETA(N+1:2*N)
DO I=1,N
  DO J=1,NUMT(I)
    DUM=ABS(C(I,J))
    IF (DUM /= 0.0) THEN
      TUM = FACE(I) + LOG10(DUM) + DOT_PRODUCT(FACV(1:N), &
                                POLYNOMIAL(I)%TERM(J)%DEG(1:N))
      POLYNOMIAL(I)%TERM(J)%COEF = (10.0_R8**TUM) * (C(I,J)/DUM)
    ENDIF
  END DO
END DO
!
SCALE_FACTORS(1:N)=FACV(1:N)
RETURN
END SUBROUTINE SCALE_PLP
!
!
SUBROUTINE INIT_PLP
! INIT_PLP homogenizes the homotopy map, and harvests random complex
! numbers which define the start system and the projective transformation.
!
! On exit:
!
! POLYNOMIAL(I)%TERM(J)%DEG(N+1) is the degree of the homogeneous variable
!  in the Jth term of the Ith component of the target system.
!
! PARTITION(I)%SET(J)%START_COEF(K) is the coefficient of X(PAR(I,J,K)) in
!  the linear factor L(I,J).  (L(I,J) is defined in GLOBAL_PLP.)
!
! PROJ_COEF(I) is the coefficient of X(I) in the projective transformation,
!  when I=1,...,N, and the constant term in the projective transformation,
!  when I=N+1.
!
! Local variables.
INTEGER:: COUNT, I, J, K, SEED_SIZE
INTEGER, DIMENSION(:), ALLOCATABLE:: SEED
REAL, DIMENSION(N*N+N+1,2):: RANDS
REAL (KIND=R8), DIMENSION(N*N+N+1,2):: RANDSR8
!
! Construct the homogenization of the homotopy map.  Note:
! Homogenization of the start system is implicit.
DO I=1,N
  DO J=1,NUMT(I)
    POLYNOMIAL(I)%TERM(J)%DEG(N+1)=SUM((/(SD(I,K),K=1, &
      PARTITION_SIZES(I))/)) - SUM(POLYNOMIAL(I)%TERM(J)%DEG(1:N))
  END DO
END DO
!
```

```
! Get the random coefficients START_COEF which define the start system
! and the random coefficients PROJ_COEF which define the projective
! transformation.
CALL RANDOM_SEED(SIZE=SEED_SIZE)
ALLOCATE(SEED(SEED_SIZE))
SEED(1:SEED_SIZE)=194917317
CALL RANDOM_SEED(PUT=SEED(1:SEED_SIZE))
CALL RANDOM_NUMBER(HARVEST=RANDS)
RANDS=2.0*RANDS-1.0
RANDSR8=REAL(RANDS,KIND=R8)
COUNT=1
DO I=1,N
  DO J=1,PARTITION_SIZES(I)
    DO K=1,NUMV(I,J)
      PARTITION(I)%SET(J)%START_COEF(K)=CMPLX(RANDSR8(COUNT,1),  &
        RANDSR8(COUNT,2),KIND=R8)
      COUNT=COUNT+1
    END DO
  END DO
END DO
PROJ_COEF(1:N+1)=CMPLX(RANDSR8(COUNT:COUNT+N,1),  &
  RANDSR8(COUNT:COUNT+N,2),KIND=R8)
!
DEALLOCATE(SEED)
RETURN
END SUBROUTINE INIT_PLP
!
!
SUBROUTINE START_POINTS_PLP
! START_POINTS_PLP finds a starting point for the homotopy map
! corresponding to the lexicographic vector FLEX_NUM (defining the
! variable sets) and the lexicographic vector DLEX_NUM (defining the
! particular start point among all those defined by FLEX_NUM).  The
! (complex) start point z is the solution to a nonsingular linear system
! AA z = B, defined by (cf. the notation in the module GLOBAL_PLP)
!
! L(1,FLEX_NUM(1)) - R(DLEX_NUM(1)-1,SD(1,FLEX_NUM(1))) * X(N+1) = 0,
! .
! .
! .
! L(N,FLEX_NUM(N)) - R(DLEX_NUM(N)-1,SD(N,FLEX_NUM(N))) * X(N+1) = 0,
! X(N+1) = SUM_J=1^N PROJ_COEF(J)*X(J) + PROJ_COEF(N+1),
!
! where the last equation is the projective transformation, X(N+1) is
! the homogeneous coordinate, and R(K,M)=e**(i*2*PI*K/M) is an Mth root
! of unity.  The homogeneous variable X(N+1) is explicitly eliminated,
! resulting in an N x N complex linear system for z=(X(1),...,X(N)).
!
! START_POINTS_PLP calculates a start point in an efficient way:  For each
! fixed lexicographic number LEX_NUM, the routine reuses, if possible,
! previous Householder reflections in the LQ decomposition of AA.
!
! Calls the LAPACK routines ZLARFG, ZLARFX, the BLAS routine ZTRSV, and the
! internal function ROOT_OF_UNITY.
!
! On exit:
```

```
!
! Z(1:2N) is a real vector representing the (complex) start point z.
!
! Local variables.
INTEGER:: I, INFO, J, K
COMPLEX (KIND=R8):: ROOT, WORK
!
! (Re)set the coefficient matrix AA, and set B.
DO I=1,N
  IF (DLEX_SAVE(I) /= DLEX_NUM(I)) THEN
    DLEX_SAVE(I+1:N)=0
    DO J=1,N
      ROOT=ROOT_OF_UNITY(DLEX_NUM(J)-1,SD(J,FLEX_NUM(J)))
      B(J) = ROOT * PROJ_COEF(N+1)
      IF (J >= I) THEN
        AA(J,1:N)=(0.0_R8,0.0_R8)
        K=NUMV(J,FLEX_NUM(J))
        AA(J,PARTITION(J)%SET(FLEX_NUM(J))%INDEX(1:K)) = &
          PARTITION(J)%SET(FLEX_NUM(J))%START_COEF(1:K)
        AA(J,1:N) = AA(J,1:N) - PROJ_COEF(1:N) * ROOT
      END IF
    END DO
    EXIT
  END IF
END DO
!
! Special code for the case N=1.
IF (N == 1) THEN
  WORK = B(1)/AA(1,1)
  Z(1) = REAL(WORK)
  Z(2) = AIMAG(WORK)
  DLEX_SAVE=DLEX_NUM
  RETURN
END IF
!
! Update the LQ factorization of AA.
IF (DLEX_SAVE(1) /= DLEX_NUM(1)) THEN
  AA(1,1:N)=CONJG(AA(1,1:N))
  CALL ZLARFG(N,AA(1,1),AA(1,2:N),1,TAU(1))
END IF
DO I=2,N
  IF (DLEX_SAVE(I) /= DLEX_NUM(I)) THEN
    DO J=1,I-1
      V(J)=(1.0_R8,0.0_R8)
      V(J+1:N)=AA(J,J+1:N)
      CALL ZLARFX('R',1,N-J+1,V(J:N),TAU(J),AA(I,J:N),1,WORK)
    END DO
    IF (I < N) THEN
      AA(I,I:N)=CONJG(AA(I,I:N))
      CALL ZLARFG(N-I+1,AA(I,I),AA(I,I+1:N),1,TAU(I))
    END IF
  END IF
END DO
DLEX_SAVE=DLEX_NUM
!
! Solve the linear system AA Z = B, by solving L Q Z = B.
```

```
! L W = B.
CALL ZTRSV('L','N','N',N,AA(1:N,1:N),N,B(1:N),1)
! Z = CONJG(Q') W.
DO I=N-1,1,-1
  V(I)=(1.0_R8,0.0_R8)
  V(I+1:N)=AA(I,I+1:N)
  CALL ZLARFX('L',N-I+1,1,V(I:N),TAU(I),B(I:N),N,WORK)
END DO
!
! Convert the complex start point to a real vector.
Z(1:2*N:2) = REAL(B)
Z(2:2*N:2) = AIMAG(B)
RETURN
END SUBROUTINE START_POINTS_PLP
!
!
COMPLEX (KIND=R8) FUNCTION ROOT_OF_UNITY(K,N) RESULT(RU)
! RU = e**(i*2*PI*K/N).
  INTEGER:: K, N
  REAL (KIND=R8):: ANGLE
  ANGLE=2.0_R8*PI*(REAL(K,KIND=R8)/REAL(N,KIND=R8))
  RU=CMPLX(COS(ANGLE),SIN(ANGLE),KIND=R8)
  RETURN
END FUNCTION ROOT_OF_UNITY
!
!
SUBROUTINE STEP_PLP
! Driver for reverse call external subroutine STEPNX from HOMPACK90.
INTEGER:: FAIL=0,IFLAGS
STEP: DO
  CALL STEPNX(2*N,NNFE,IFLAG,START,CRASH,HOLD,H,RELERR, &
    ABSERR,S,Y,YP,YOLD,YPOLD,A,TZ,W,WP,RHOLEN,SSPAR)
  IF (CRASH) THEN
    IFLAG = 2
    EXIT
  END IF
  IFLAGS = IFLAG
  SELECT CASE (IFLAGS)
    CASE (-2)     ! Successful step.
      EXIT
    CASE (-12)    ! Compute tangent vector.
      RHOLEN = 0.0_R8
      CALL TANGENT_PLP
      IF (IFLAG == 4) THEN
        IFLAG = IFLAGS - 100
        FAIL = FAIL + 1
      ENDIF
    CASE (-32,-22) ! Compute tangent vector and Newton step.
      RHOLEN = -1.0_R8
      CALL TANGENT_PLP(NEWTON_STEP=.TRUE.)
      IF (IFLAG == 4) THEN
        IFLAG = IFLAGS - 100
        FAIL = FAIL + 1
      ENDIF
    CASE (4,6) ! STEPNX failed.
      EXIT
```

```
  END SELECT
  IF (FAIL == 2) THEN
    IFLAG = 4; RETURN
  ENDIF
END DO STEP
RETURN
END SUBROUTINE STEP_PLP
!
!
SUBROUTINE TANGENT_PLP(NEWTON_STEP)
! This subroutine builds the Jacobian matrix of the homotopy map,
! computes a QR decomposition of that matrix, and then calculates the
! (unit) tangent vector and (if NEWTON_STEP is present) the Newton
! step.
!
! On input:
!
! NEWTON_STEP is a logical optional argument which, if present,
!    indicates that the Newton step is also to be calculated.
!
! RHOLEN < 0 if the norm of the homotopy map evaluated at
!    (LAMBDA, X) is to be computed.  If  RHOLEN >= 0  the norm is not
!    computed and RHOLEN is not changed.
!
! W(1:2*N+1) = current point (LAMBDA(S), X(S)).
!
! YPOLD(1:2*N+1) = unit tangent vector at previous point on the zero
!    curve of the homotopy map.
!
! On output:
!
! RHOLEN = ||RHO(LAMBDA(S), X(S))|| if  RHOLEN < 0  on input.
!    Otherwise RHOLEN is unchanged.
!
! WP(1:2*N+1) = dW/dS = unit tangent vector to integral curve of
!    d(homotopy map)/dS = 0  at  W(S) = (LAMBDA(S), X(S)) .
!
! TZ = the Newton step = -(pseudo inverse of  (d RHO(W(S))/d LAMBDA ,
!    d RHO(W(S))/dX)) * RHO(W(S)) .
!
! IFLAG  is unchanged, unless the QR factorization detects a rank < N,
!    in which case the tangent and Newton step vectors are not computed
!    and TANGENT_PLP returns with  IFLAG = 4 .
!
!
! Calls  DGEQPF, DNRM2, DORMQR, RHO.
!
LOGICAL, INTENT(IN), OPTIONAL:: NEWTON_STEP
REAL (KIND=R8):: LAMBDA, SIGMA, WPNORM
INTEGER:: I, J, K
INTERFACE
  FUNCTION DNRM2(N,X,STRIDE)
    USE REAL_PRECISION
    INTEGER:: N,STRIDE
    REAL (KIND=R8):: DNRM2,X(N)
  END FUNCTION DNRM2
```

```
END INTERFACE
!
! Compute the Jacobian matrix, store it and homotopy map in QR.
!
!  QR = ( D RHO(LAMBDA,X)/D LAMBDA , D RHO(LAMBDA,X)/DX ,
!                                        RHO(LAMBDA,X) )  .
!
! Force LAMBDA >= 0 for tangent calculation.
IF (W(1) < 0.0_R8) THEN
  LAMBDA = 0.0_R8
ELSE
  LAMBDA = W(1)
END IF
!
! RHO(W) evaluates the homotopy map and its Jacobian matrix at W,
! leaving the results in the arrays RHOV, DRHOL, and DRHOX.
CALL RHO(LAMBDA,W(2:2*N+1))
QR(1:2*N,1) = DRHOL(1:2*N)
QR(1:2*N,2:2*N+1) = DRHOX(1:2*N,1:2*N)
QR(1:2*N,2*N+2) = RHOV(1:2*N)
!
! Compute the norm of the homotopy map if it was requested.
IF (RHOLEN < 0.0_R8) RHOLEN=DNRM2(2*N,QR(:,2*N+2),1)
!
! Reduce the Jacobian matrix to upper triangular form.
PIVOT = 0
CALL DGEQPF(2*N,2*N+1,QR,2*N,PIVOT,WP,ALPHA,K)
IF (ABS(QR(2*N,2*N)) <= ABS(QR(1,1))*EPSILON(1.0_R8)) THEN
  IFLAG=4
  RETURN
ENDIF
!
! Apply Householder reflections to last column of QR (which contains
! RHO(A,W)).
CALL DORMQR('L','T',2*N,1,2*N-1,QR,2*N,WP,QR(:,2*N+2),2*N,  &
  ALPHA, 3*(2*N+1),K)
!
! Compute kernel of Jacobian matrix, yielding WP=dW/dS.
TZ(2*N+1)=1.0_R8
DO I=2*N,1,-1
  J=I+1
  TZ(I)= -DOT_PRODUCT(QR(I,J:2*N+1),TZ(J:2*N+1))/QR(I,I)
END DO
WPNORM=DNRM2(2*N+1,TZ,1)
WP(PIVOT)=TZ/WPNORM
IF (DOT_PRODUCT(WP,YPOLD) < 0.0_R8) WP = -WP
!
! WP is the unit tangent vector in the correct direction.
IF (.NOT. PRESENT(NEWTON_STEP)) RETURN
!
! Compute the minimum norm solution of [d RHO(W(S))] V = -RHO(W(S)).
! V is given by  P - (P,Q)Q  , where P is any solution of
! [d RHO] V = -RHO  and Q is a unit vector in the kernel of [d RHO].
!
ALPHA(2*N+1)=1.0_R8
DO I=2*N,1,-1
```

```
  J=I+1
  ALPHA(I)= -(DOT_PRODUCT(QR(I,J:2*N+1),ALPHA(J:2*N+1)) + QR(I,2*N+2)) &
           /QR(I,I)
END DO
TZ(PIVOT)=ALPHA(1:2*N+1)
!
! TZ now contains a particular solution P, and WP contains a vector Q
! in the kernel (the unit tangent).
SIGMA=DOT_PRODUCT(TZ,WP)
TZ = TZ - SIGMA*WP
! TZ is the Newton step from the current point W(S) = (LAMBDA(S), X(S)).
RETURN
END SUBROUTINE TANGENT_PLP
!
!
SUBROUTINE ROOT_PLP
! In a deleted neighborhood of a solution (1,X(SBAR)), the homotopy zero
! curve (LAMBDA(S),X(S)) is assumed to safisfy X = X(LAMBDA), a consequence
! of the Implicit Function Theorem and the fact that the Jacobian matrix
! D RHO(A,LAMBDA(S),X(S))/DX is nonsingular in a sufficiently small
! deleted neighborhood of an isolated solution.  Let
!        TAU = 1 - LAMBDA = SIGMA**C,
! where the positive integer C is the cycle number of the root.  Then
!   X(LAMBDA) = X(1 - TAU) = X(1 - SIGMA**C) = Z(SIGMA)
! is an analytic function of SIGMA in a neighborhood of SIGMA=0.  This fact
! is exploited by guessing C and interpolating Z(SIGMA) within its
! Maclaurin series' radius of convergence, but far enough away from 0 to
! avoid numerical instability.  This annulus is called the "operating
! range" of the algorithm.  The interpolant to analytic Z(SIGMA) is then
! evaluated at SIGMA=0 to estimate the root X(1)=Z(0).
!
! Local variables.
INTEGER, PARAMETER:: CHAT_MAX=8, LITFH = 7
INTEGER:: C, CHAT(1), CHAT_BEST, CHAT_OLD, GOING_BAD, I, &
  II, J, ML_ITER, N2P1, RETRY
REAL (KIND=R8):: ACCURACY, FV(12), GM, H_SAVE, HC, HQ, HQ_BEST,    &
  HQMHC(CHAT_MAX), L(-3:2), S_SAVE, SIGMA(-3:2), SHRINK, T, TOL_1, &
  TOL_2, V(12)
LOGICAL:: EVEN, FIRST_JUMP, REUSE
INTERFACE
  FUNCTION DNRM2(N,X,STRIDE)
    USE REAL_PRECISION
    INTEGER:: N, STRIDE
    REAL (KIND=R8):: DNRM2, X(N)
  END FUNCTION DNRM2
END INTERFACE
!
N2P1 = 2*N + 1
ACCURACY = MAX(FINALTOL,SQRT(EPSILON(1.0_R8))*10.0_R8**2)
HQ_BEST = 10.0_R8*ACCURACY
CHAT_BEST = 0 ; CHAT_OLD = 0 ; GOING_BAD = 0
FIRST_JUMP = .TRUE. ; REUSE = .FALSE.
YOLDS = 0.0_R8
!
! Save the first point.
H_SAVE = HOLD
```

```
S_SAVE = S - HOLD
YS(:,1) = YOLD ; YS(:,2) = YPOLD
!
! If Y(1) >= 1 or if YP(1) <= 0 back up to YOLD and generate another point.
REFINE_Y: DO
  IF ((Y(1) >= 1.0_R8) .OR. (YP(1) <= 0.0_R8)) THEN
    SHRINK = 1.0_R8
! Try 3 times to get a point.
    DO I=1,3
      SHRINK = SHRINK*.75_R8
      S = S_SAVE
      H = MIN(H_SAVE,SHRINK*(1.0_R8-YS(1,1))/YS(1,2))
! If Y(1)>=1 increase RELERR and ABSERR to prevent STEPNX from making
! the stepsize too small.
      IF (Y(1) >= 1.0_R8) THEN
        RELERR = TRACKTOL ; ABSERR = TRACKTOL
      END IF
      Y = YS(:,1) ; YP = YS(:,2)
      START = .TRUE.
      CALL STEP_PLP
      RELERR = FINALTOL ; ABSERR = FINALTOL
      IF (IFLAG > 0) THEN
        IFLAG = 4 ; RETURN
      ELSE IF ((Y(1) < 1.0_R8) .AND. (YP(1) > 0.0_R8)) THEN
        ITER = ITER + 1
        EXIT REFINE_Y
      ELSE IF (I == 3) THEN
        IFLAG = 4 ; RETURN
      END IF
    END DO
  ELSE
! Refine the second point Y to FINALTOL accuracy.  If the refinement
! fails, back up and get another point.
    W = Y
    RHOLEN = 0.0_R8
    DO J=1,LITFH
      CALL TANGENT_PLP(NEWTON_STEP=.TRUE.)
      NNFE = NNFE + 1
      IF (IFLAG > 0) THEN
        IFLAG = -2
        YP(1) = -1.0_R8 ; CYCLE REFINE_Y
      END IF
      W = W + TZ
! Test for erratic LAMBDA.
      IF (W(1) >= 1.0_R8 .OR. WP(1) <= 0.0_R8) THEN
        YP(1) = -1.0_R8 ; CYCLE REFINE_Y
      END IF
      IF (DNRM2(N2P1,TZ,1) <= FINALTOL*(DNRM2(N2P1,W,1) + 1.0_R8)) EXIT
! Test for lack of convergence.
      IF (J == LITFH) THEN
        YP(1) = -1.0_R8 ; CYCLE REFINE_Y
      END IF
    END DO
    Y = W ; YP = WP
    S = S - HOLD
    W = Y - YOLD
```

```
    HOLD = DNRM2(N2P1,W,1)
    S = S + HOLD
    EXIT REFINE_Y
  END IF
END DO REFINE_Y
!
! Save the second point.
YS(:,3) = Y ; YS(:,4) = YP
H_SAVE = H ; S_SAVE = S
!
! Try entire end game interpolation process RETRY=10*NUMRR times.
RETRY = 10*NUM_RERUNS
MAIN_LOOP: DO ML_ITER=1,RETRY
!
! Enforce LIMIT on the number of steps.
IF (ITER == LIMIT) THEN
  IFLAG = 3 ; EXIT MAIN_LOOP
END IF
! Get close enough to SIGMA=0 (LAMBDA=1) so that a Hermite cubic
! interpolant is accurate to within TOL_1 (defined by CHAT).
OPERATING_RANGE: DO
  SHRINK = 1.0_R8
  DO J=1,3
    SHRINK = .75_R8*SHRINK
! Get a third point Y with Y(1) < 1.
    H = MIN(H_SAVE,SHRINK*(1.0_R8-Y(1))/YP(1))
    CALL STEP_PLP
    IF (IFLAG > 0) THEN
      IFLAG = 4 ; EXIT MAIN_LOOP
    ELSE IF ((Y(1) >= 1.0_R8) .OR. (YP(1) <= 0.0_R8)) THEN
! Back up and try again with a smaller step.
      Y = YS(:,3) ; YP = YS(:,4) ; YOLD = YS(:,1) ; YPOLD = YS(:,2)
      S = S_SAVE
    ELSE
      ITER=ITER+1
      EXIT
    END IF
    IF (J == 3) THEN
      IFLAG = 4 ; EXIT MAIN_LOOP
    END IF
  END DO
!
! Save the third point.
  YS(:,5)=Y; YS(:,6)=YP
  H_SAVE = H ; S_SAVE = S
! L(2) < L(1) < L(0) < 1.
  L(2) = YS(1,1) ; L(1) = YS(1,3) ; L(0) = YS(1,5)
! Test approximation quality for each cycle number C = 1,...,CHAT_MAX.
  SHRINK = 1.0_R8/(1.0_R8 + MAXVAL(ABS(YS(2:N2P1,5))))
  DO C=1,CHAT_MAX
    SIGMA(0:2) = (1.0_R8 - L(0:2))**(1.0_R8/REAL(C,KIND=R8))
! 0 < SIGMA(0) < SIGMA(1) < SIGMA(2).
! Compute difference between Hermite quintic HQ(SIGMA) interpolating at
! SIGMA(0:2) and Hermite cubic HC(SIGMA) interpolating at SIGMA(0:1).
! The interpolation points for the Newton form are (SIGMA(0), SIGMA(0),
! SIGMA(1), SIGMA(1), SIGMA(2), SIGMA(2)).  The function values are in
```

```
! YS(:,5:1:-2) and the derivatives YS(:,6:2:-2) = dX/dS have to be
! converted to dX/dSIGMA.
   T = 0.0_R8
   V(1:6) = (/ (SIGMA(J),SIGMA(J),J=0,2) /)
   DO J=2,N2P1
     FV(1:5:2) = YS(J,5:1:-2)
     FV(2:6:2) = (YS(J,6:2:-2)/YS(1,6:2:-2))*(-REAL(C,KIND=R8))* &
       SIGMA(0:2)**(C-1)
     CALL INTERP(V(1:6),FV(1:6))
     T = MAX(T,ABS(FV(5) - SIGMA(2)*FV(6)))
   END DO
! T*(SIGMA(1)*SIGMA(0))**2 = ||HQ(0) - HC(0)||_infty.
   HQMHC(C) = T*((SIGMA(1)*SIGMA(0))**2)*SHRINK
 END DO
! Find best estimate CHAT of cycle number.
 CHAT = MINLOC(HQMHC)
! If there has been one successful jump across the origin (with
! CHAT_BEST) and the cycle number prediction changes, then the process
! may be leaving the operating range.
 IF (( .NOT. FIRST_JUMP) .AND. (CHAT(1) /= CHAT_BEST)) THEN
   GOING_BAD = GOING_BAD + 1
   IF (GOING_BAD == 2) EXIT MAIN_LOOP
 END IF
 TOL_1 = ACCURACY*10.0_R8**(REAL(CHAT(1),KIND=R8)/2.0_R8)
 IF (HQMHC(CHAT(1)) <= TOL_1) THEN
   EXIT OPERATING_RANGE
 ELSE IF ( .NOT. FIRST_JUMP) THEN
   GOING_BAD = GOING_BAD + 1
   IF (GOING_BAD == 2) EXIT MAIN_LOOP
 END IF
! Shift point history, and try to get closer to SIGMA=0.
 YS(:,1:2) = YS(:,3:4) ; YS(:,3:4) = YS(:,5:6) ; REUSE = .FALSE.
END DO OPERATING_RANGE
!
! Add 3 new points past SIGMA=0 such that
! SIGMA(2) > SIGMA(1) > SIGMA(0) > 0 > SIGMA(-1) > SIGMA(-2) > SIGMA(-3).
! If CHAT is odd then the corresponding LAMBDA are such that
!   L(2)   <   L(1)   <   L(0)   < 1 <   L(-1)   <   L(-2)   <   L(-3),
! and if CHAT is even then
!   L(2)   <   L(1)   <   L(0)   < 1
!                              1 >   L(-1)   >   L(-2)   >   L(-3).
!
SIGMA(0:2) = (1.0_R8 - L(0:2))**(1.0_R8/REAL(CHAT(1),KIND=R8))
DO I=1,3
 V(1:4+2*I) = (/ (SIGMA(J),SIGMA(J),J=2,1-I,-1) /)
 DO J=2,N2P1
   FV(1:3+2*I:2) = YS(J,1:3+2*I:2)
   FV(2:4+2*I:2) = (YS(J,2:4+2*I:2)/YS(1,2:4+2*I:2))* &
     (-REAL(CHAT(1),KIND=R8))*SIGMA(2:1-I:-1)**(CHAT(1)-1)
   CALL INTERP(V(1:4+2*I),FV(1:4+2*I))
   CALL INTERP(V(1:4+2*I),FV(1:4+2*I),-SIGMA(I-1),W(J))
 END DO
 IF (MOD(CHAT(1),2) == 0) THEN
   EVEN = .TRUE.
   W(1) = L(I-1)
 ELSE
```

```
    EVEN = .FALSE.
    W(1) = 2.0_R8 - L(I-1)
  END IF
! W now contains the (predicted) point symmetric to SIGMA(I-1) with
! respect to SIGMA=0.
  RHOLEN = 0.0_R8
! Correct the prediction.  If there has been one successful jump across
! the origin, correction failures may indicate that the process is
! leaving the operating range.
  DO J=1,LITFH
    CALL TANGENT_PLP(NEWTON_STEP=.TRUE.)
    NNFE = NNFE + 1
! Test for singular Jacobian matrix.
    IF (IFLAG > 0) EXIT MAIN_LOOP
    W = W + TZ
! Test for erratic LAMBDA.
    IF ((( .NOT. EVEN) .AND. (W(1) <= 1.0_R8)) .OR.  &
        (EVEN .AND. (W(1) >= 1.0_R8))) THEN
      IF ( .NOT. FIRST_JUMP) THEN
        GOING_BAD = GOING_BAD + 1
        IF (GOING_BAD == 2) EXIT MAIN_LOOP
      END IF
      YS(:,1:2) = YS (:,3:4) ; YS(:,3:4) = YS(:,5:6)
      REUSE = .FALSE. ; CYCLE MAIN_LOOP
    END IF
    IF (DNRM2(N2P1,TZ,1) <= FINALTOL*(DNRM2(N2P1,W,1) + 1.0_R8)) EXIT
! Test for lack of convergence.
    IF (J == LITFH) THEN
      IF ( .NOT. FIRST_JUMP) THEN
        GOING_BAD = GOING_BAD + 1
        IF (GOING_BAD == 2) EXIT MAIN_LOOP
      END IF
      YS(:,1:2) = YS (:,3:4) ; YS(:,3:4) = YS(:,5:6)
      REUSE = .FALSE. ; CYCLE MAIN_LOOP
    END IF
  END DO
! Ensure that the tangent vector has the correct direction.
  IF (EVEN) THEN
    IF (WP(1) > 0.0_R8) WP = -WP
  ELSE
    IF (WP(1) < 0.0_R8) WP = -WP
  END IF
! Update the lambda (L), sigma (SIGMA), and history (YS) arrays.
  L(-I) = W(1)
  SIGMA(-I) =  -(ABS(L(-I)-1.0_R8))**(1.0_R8/REAL(CHAT(1),KIND=R8))
  YS(:,5+2*I) = W ; YS(:,6+2*I) = WP
! Reuse old points if the cycle number estimation has not changed from the
! last iterration, and the origin was successfully jumped in the last
! iteration.
  IF (REUSE .AND. (CHAT(1) == CHAT_OLD)) EXIT
END DO
!
! Construct 12th order interpolant and estimate the root at SIGMA=0.
HC = 0.0_R8 ; HQ = 0.0_R8 ; T = 0.0_R8
V(1:12) = (/ (SIGMA(J),SIGMA(J),J=-3,2) /)
DO J=2,N2P1
```

```
  FV(1:11:2) = YS(J,11:1:-2)
  FV(2:12:2) = (YS(J,12:2:-2)/YS(1,12:2:-2))* &
    (-REAL(CHAT(1),KIND=R8))*SIGMA(-3:2)**(CHAT(1)-1)
  CALL INTERP(V(1:12),FV(1:12))
  CALL INTERP(V(1:12),FV(1:12),0.0_R8,W(J))
! Difference between 8th and 6th order Hermite interpolants.
  T  = MAX(T ,ABS(FV( 7) - SIGMA(0)*FV( 8)))
! Difference between 10th and 8th order Hermite interpolants.
  HC = MAX(HC,ABS(FV( 9) - SIGMA(1)*FV(10)))
! Difference between 12th and 10th order Hermite interpolants.
  HQ = MAX(HQ,ABS(FV(11) - SIGMA(2)*FV(12)))
END DO
SHRINK = 1.0_R8/(1.0_R8 + MAXVAL(ABS(W(2:N2P1))))
T  =  T*((PRODUCT(SIGMA(-3:-1)))**2)*SHRINK  ! ||H_7  - H_5||/(1+||W||)
HC = HC*((PRODUCT(SIGMA(-3: 0)))**2)*SHRINK  ! ||H_9  - H_7||/(1+||W||)
HQ = HQ*((PRODUCT(SIGMA(-3: 1)))**2)*SHRINK  ! ||H_11 - H_9||/(1+||W||)
!
! Check both accuracy and consistency of Hermite interpolants.
TOL_2 = FINALTOL*(10**(CHAT(1)-1))
GM = SQRT(TOL_1*TOL_2)
IF ((T <= TOL_1) .AND. (HC <= GM) .AND. (HQ <= TOL_2)) THEN
! Full convergence.
  IF (FIRST_JUMP) FIRST_JUMP = .FALSE.
  YOLDS(2:N2P1) = W(2:N2P1); HQ_BEST = HQ
  CHAT_BEST = CHAT(1)
  EXIT MAIN_LOOP
ELSE IF (HQ > 1.01_R8*HQ_BEST) THEN
  IF ( .NOT. FIRST_JUMP) THEN
    GOING_BAD = GOING_BAD + 1
    IF (GOING_BAD == 2) EXIT MAIN_LOOP
  END IF
ELSE
! Progress has been made.
  IF (FIRST_JUMP) FIRST_JUMP = .FALSE.
  GOING_BAD = 0
  YOLDS(2:N2P1) = W(2:N2P1); HQ_BEST = HQ
  CHAT_BEST = CHAT(1)
END IF
! Shift point history.
YS(:,1:2) = YS(:,3:4) ; YS(:,3:4) = YS(:,5:6)
! If the cycle number estimate does not change in the next iteration, the
! points found across the origin can be reused.
REUSE = .TRUE. ; CHAT_OLD = CHAT(1)
SIGMA(-3)   = SIGMA(-2)  ; SIGMA(-2)  = SIGMA(-1)
YS(:,11:12) = YS(:,9:10) ; YS(:,9:10) = YS(:,7:8)
END DO MAIN_LOOP
!
IF (ML_ITER >= RETRY) IFLAG = 7
!
! Return final solution in Y.
IF ( .NOT. FIRST_JUMP) THEN
  Y(1) = HQ_BEST; Y(2:N2P1) = YOLDS(2:N2P1)
  IFLAG = 1 + 10*CHAT_BEST
END IF
RETURN
END SUBROUTINE ROOT_PLP
```

```
!
!
SUBROUTINE INTERP(T,FT,X,FX)
! Given data points T(:) and function values FT(:)=f(T(:)), INTERP
! computes the Newton form of the interpolating polynomial to f at T(:).
! T is assumed to be sorted, and if
! T(I-1) < T(I) = T(I+1) = ... = T(I+K) < T(I+K+1) then
! FT(I)=f(T(I)), FT(I+1)=f'(T(I)), ..., FT(I+K)=f^(K)(T(I)).
! On return FT(K) contains the divided difference f[T(1),...,T(K)], and
! FX contains the interpolating polynomial evaluated at X.  If X and FX
! are present, the divided differences are not calculated.
!
REAL (KIND=R8), DIMENSION(:):: T, FT
REAL (KIND=R8), OPTIONAL:: X, FX
! Local variables.
REAL (KIND=R8):: FOLD,SAVE
INTEGER:: I,K,N
!
N=SIZE(T)
IF (.NOT. PRESENT(X)) THEN ! Calculate divided differences.
  DO K=1,N-1
    FOLD = FT(K)
    DO I=K+1,N
      IF (T(I) == T(I-K)) THEN
        FT(I) = FT(I)/REAL(K,KIND=R8)
      ELSE
        SAVE = FT(I)
        FT(I) = (FT(I)-FOLD)/(T(I)-T(I-K))
        FOLD = SAVE
      END IF
    END DO
  END DO
  RETURN
END IF
FX = FT(N) ! Evaluate Newton polynomial.
DO K=N-1,1,-1
  FX = FX*(X -T(K)) + FT(K)
END DO
RETURN
END SUBROUTINE INTERP
!
!
SUBROUTINE RHO(LAMBDA,X)
! RHO evaluates the (complex) homotopy map
!
!   RHO(A,LAMBDA,X) = LAMBDA*F(X) + (1 - LAMBDA)*GAMMA*G(X),
!
! where GAMMA is a random complex constant, and the Jacobian
! matrix [ D RHO(A,LAMBDA,X)/D LAMBDA, D RHO(A,LAMBDA,X)/DX ] at
! (A,LAMBDA,X), and updates the global arrays RHOV (the homotopy map),
! DRHOX (the derivative of the homotopy with repect to X) , and DRHOL
! (the derivative with respect to LAMBDA).  The vector A corresponds
! mathematically to all the random coefficients in the start system, and
! is not explicitly referenced by RHO.  X, on entry, is real, but since
! arithmetic in RHO is complex, X is converted to complex form.  Before
! return RHO converts the homotopy map and the two derivatives back to
```

```
! real.  Precisely, suppose XC is the complexification of X, i.e.,
!
! XC(1:N)=CMPLX(X(1:2*N-1:2),X(2:2*N:2)).
!
! Let CRHOV(A,LAMBDA,XC) be the (complex) homotopy map.  Then RHOV
! is just
!
! RHOV(1:2*N-1:2) = REAL( CRHOV(1:N)),
! RHOV(2:2*N  :2) = AIMAG(CRHOV(1:N)).
!
! Let CDRHOXC = D CRHOV(A,LAMBDA,XC)/D XC denote the (complex) derivative
! of the homotopy map with respect to XC, evaluated at (A,LAMBDA,XC).
! DRHOX is obtained by
!
! DRHOX(2*I-1,2*J-1) = REAL(CDRHOXC(I,J)),
! DRHOX(2*I  ,2*J  ) = DRHOX(2*I-1,2*J-1),
! DRHOX(2*I  ,2*J-1) = AIMAG(CDRHOXC(I,J)),
! DRHOX(2*I-1,2*J  ) = -DRHOX(2*I  ,2*J-1),
!
! for I, J = 1,...,N.  Let CDRHOL = D CRHOV(A,LAMBDA,XC)/D LAMBDA denote
! the (complex) derivative of the homotopy map with respect to LAMBDA,
! evaluated at (A,LAMBDA,XC).  Then DRHOL is obtained by
!
! DRHOL(1:2*N-1:2) = REAL( CDRHOL(1:N)),
! DRHOL(2:2*N  :2) = AIMAG(CDRHOL(1:N)).
!
! (None of CRHOV, CDRHOXC, or CDRHOL are in the code.)
!
! Internal subroutines:  START_SYSTEM, TARGET_SYSTEM.
! External (optional, user written) subroutine:  TARGET_SYSTEM_USER.
!
! On input:
!
! LAMBDA is the continuation parameter.
!
! X(1:2*N) is the real 2*N-dimensional evaluation point.
!
! On exit:
!
! LAMBDA and X are unchanged.
!
! RHOV(1:2*N) is the real (2*N)-dimensional representation of the
!   homotopy map RHO(A,LAMBDA,X).
!
! DRHOX(1:2*N,1:2*N) is the real (2*N)-by-(2*N)-dimensional
!   representation of D RHO(A,LAMBDA,X)/DX evaluated at (A,LAMBDA,X).
!
! DRHOL(1:2*N) is the real (2*N)-dimensional representation of
!   D RHO(A,LAMBDA,X)/D LAMBDA evaluated at (A,LAMBDA,X).
!
REAL (KIND=R8), INTENT(IN):: LAMBDA
REAL (KIND=R8), DIMENSION(2*N), INTENT(IN):: X
!
INTERFACE
SUBROUTINE TARGET_SYSTEM_USER(N,PROJ_COEF,XC,F,DF)
  USE REAL_PRECISION
```

```
  INTEGER, INTENT(IN):: N
  COMPLEX (KIND=R8), INTENT(IN), DIMENSION(N+1):: PROJ_COEF,XC
  COMPLEX (KIND=R8), INTENT(OUT):: F(N), DF(N,N+1)
END SUBROUTINE TARGET_SYSTEM_USER
END INTERFACE
!
! Local variables.
INTEGER:: I, J
REAL (KIND=R8):: ONEML
COMPLEX (KIND=R8):: GAMMA
!
ONEML=1.0_R8-LAMBDA
GAMMA=(.0053292102547824_R8,.9793238462643383_R8)
!
! Convert the real-valued evaluation point X to a complex vector.
XC(1:N)=CMPLX(X(1:2*N-1:2),X(2:2*N:2),KIND=R8)
!
! Calculate the homogeneous variable.
XC(N+1)=SUM(PROJ_COEF(1:N)*XC(1:N))+PROJ_COEF(N+1)
!
CALL START_SYSTEM             ! Returns G and DG.
IF (PRESENT(USER_F_DF)) THEN   ! Returns F and DF.
  CALL TARGET_SYSTEM_USER(N,PROJ_COEF,XC,F,DF) ! User written subroutine.
ELSE
  CALL TARGET_SYSTEM ! Internal subroutine.
END IF
!
! Convert complex derivatives to real derivatives via the Cauchy-Riemann
! equations.
DO I=1,N
  DO J=1,N
    DRHOX(2*I-1,2*J-1) = LAMBDA*REAL(DF(I,J)) + ONEML*REAL(DG(I,J)*GAMMA)
    DRHOX(2*I  ,2*J  ) = DRHOX(2*I-1,2*J-1)
    DRHOX(2*I  ,2*J-1) = LAMBDA*AIMAG(DF(I,J)) + ONEML*AIMAG(DG(I,J)*GAMMA)
    DRHOX(2*I-1,2*J  ) = -DRHOX(2*I,2*J-1)
  END DO
END DO
  DRHOL(1:2*N-1:2) = REAL(F) - REAL(G*GAMMA)
  DRHOL(2:2*N:2  ) = AIMAG(F) - AIMAG(G*GAMMA)
  RHOV(1:2*N-1:2)  = LAMBDA*REAL(F) + ONEML*REAL(G*GAMMA)
  RHOV(2:2*N:2  )  = LAMBDA*AIMAG(F) + ONEML*AIMAG(G*GAMMA)
RETURN
END SUBROUTINE RHO
!
!
SUBROUTINE START_SYSTEM
! START_SYSTEM evaluates the start system G(XC) and the Jacobian matrix
!   DG(XC).  Arithmetic is complex.
!
! On exit:
!
! G(:) contains the complex N-dimensional start system evaluated at XC(:).
!
! DG(:,:) contains the complex N-by-N-dimensional Jacobian matrix of
!   the start system evaluted at XC(:).
!
```

```
! Local variables.
INTEGER:: I, J, K, L, M
COMPLEX (KIND=R8):: TEMP
!
! TEMP1G AND TEMP2G are employed to reduce recalculation in G and DG.
! Note:  If SD(I,J)=0, then the corresponding factor is 1, not 0.
TEMP1G=(0.0_R8,0.0_R8)
TEMP2G=(0.0_R8,0.0_R8)
DO I=1,N
  DO J=1,PARTITION_SIZES(I)
    IF (PARTITION(I)%SET(J)%SET_DEG == 0) THEN
      TEMP2G(I,J)=(1.0_R8,0.0_R8)
    ELSE
      K=PARTITION(I)%SET(J)%NUM_INDICES
      TEMP1G(I,J) = SUM( PARTITION(I)%SET(J)%START_COEF(1:K)*  &
                      XC(PARTITION(I)%SET(J)%INDEX(1:K)) )
      TEMP2G(I,J) = TEMP1G(I,J)**PARTITION(I)%SET(J)%SET_DEG - &
                     XC(N+1)**PARTITION(I)%SET(J)%SET_DEG
    END IF
  END DO
  G(I)=PRODUCT(TEMP2G(I,1:PARTITION_SIZES(I)))
END DO
!
! Calculate the derivative of G with respect to XC(1),...,XC(N)
!   in 3 steps.
! STEP 1:  First treat XC(N+1) as an independent variable.
DG=(0.0_R8,0.0_R8)
DO I=1,N
  DO J=1,PARTITION_SIZES(I)
    IF (PARTITION(I)%SET(J)%SET_DEG == 0) CYCLE
    K=PARTITION(I)%SET(J)%NUM_INDICES
    DG(I,PARTITION(I)%SET(J)%INDEX(1:K)) = PARTITION(I)%SET(J)%SET_DEG * &
      PARTITION(I)%SET(J)%START_COEF(1:K) * &
      (TEMP1G(I,J)**(PARTITION(I)%SET(J)%SET_DEG - 1))
    TEMP = (1.0_R8,0.0_R8)
    DO L=1,PARTITION_SIZES(I)
      IF (L == J) CYCLE
      TEMP = TEMP * TEMP2G(I,L)
    END DO
    DG(I,PARTITION(I)%SET(J)%INDEX(1:K)) = &
      DG(I,PARTITION(I)%SET(J)%INDEX(1:K)) * TEMP
  END DO
END DO
!
! STEP 2:  Now calculate the N-by-1 Jacobian matrix of G with
!   respect to XC(N+1) using the product rule.
DO I=1,N
  DO J=1,PARTITION_SIZES(I)
    IF (PARTITION(I)%SET(J)%SET_DEG == 0) CYCLE
    TEMP = -PARTITION(I)%SET(J)%SET_DEG * &
      (XC(N+1)**(PARTITION(I)%SET(J)%SET_DEG - 1))
    DO K=1,PARTITION_SIZES(I)
      IF (K == J) CYCLE
      TEMP=TEMP*TEMP2G(I,K)
    END DO
    DG(I,N+1)=DG(I,N+1)+TEMP
```

```
  END DO
END DO
!
! STEP 3:  Use the chain rule with XC(N+1) considered as a function
!   of XC(1),...,XC(N).
DO I=1,N
  DG(I,1:N)=DG(I,1:N)+DG(I,N+1)*PROJ_COEF(1:N)
END DO
RETURN
END SUBROUTINE START_SYSTEM
!
!
SUBROUTINE TARGET_SYSTEM
! TARGET_SYSTEM calculates the target system F(XC) and the Jacobian matrix
!   DF(XC).  Arithmetic is complex.
!
! On exit:
!
! F(:) contains the complex N-dimensional target system evaluated
!   at XC(:).
!
! DF(:,:) is the complex N-by-N-dimensional Jacobian matrix of the
!   target system evaluated at XC(:).
!
! Local variables.
INTEGER:: I, J, K, L
COMPLEX (KIND=R8):: T, TS
!
! Evaluate F(XC).  For efficiency, indexing functions and array sections
! are avoided.
DO I=1,N
  TS = (0.0_R8, 0.0_R8)
  DO J=1,POLYNOMIAL(I)%NUM_TERMS
    T = POLYNOMIAL(I)%TERM(J)%COEF
    DO K=1,N+1
      IF (POLYNOMIAL(I)%TERM(J)%DEG(K) == 0) CYCLE
      T = T * XC(K)**POLYNOMIAL(I)%TERM(J)%DEG(K)
    END DO
    TS = TS + T
  END DO
  F(I)=TS
END DO
!
! Calulate the Jacobian matrix DF(XC).
DF=(0.0_R8,0.0_R8)
DO I=1,N
  DO J=1,N+1
    TS = (0.0_R8,0.0_R8)
    DO K=1,POLYNOMIAL(I)%NUM_TERMS
      IF (POLYNOMIAL(I)%TERM(K)%DEG(J) == 0) CYCLE
      T = POLYNOMIAL(I)%TERM(K)%COEF * POLYNOMIAL(I)%TERM(K)%DEG(J) * &
          (XC(J)**(POLYNOMIAL(I)%TERM(K)%DEG(J) - 1))
      DO L=1,N+1
        IF ((L == J) .OR. (POLYNOMIAL(I)%TERM(K)%DEG(L) == 0)) CYCLE
        T = T * (XC(L)**POLYNOMIAL(I)%TERM(K)%DEG(L))
      END DO
```

```
      TS = TS + T
    END DO
    DF(I,J) = TS
  END DO
END DO
!
! Convert DF to partials with respect to XC(1),...,XC(N) by
!   applying the chain rule with XC(N+1) considered as a function
!   of XC(1),...,XC(N).
DO I=1,N
  DF(I,1:N) = DF(I,1:N) + PROJ_COEF(1:N) * DF(I,N+1)
END DO
RETURN
END SUBROUTINE TARGET_SYSTEM
!
!
SUBROUTINE OUTPUT_PLP
! OUTPUT_PLP first untransforms (converts from projective to affine
! coordinates) and then unscales a root.
!
! On entry:
!
! XC(1:N) contains a root in projective coordinates, with the (N+1)st
!   projective coordinate XC(N+1) implicitly defined by the
!   projective transformation.
!
! On exit:
!
! XC(1:N) contains the untransformed (affine), unscaled root.
!
! XC(N+1) is the homogeneous coordinate of the root of the scaled
!   target system, if scaling was performed.
!
INTEGER:: I
REAL (KIND=R8), PARAMETER:: BIG=HUGE(1.0_R8)
!
! Calculate the homogeneous coordinate XC(N+1) using the vector XC(1:N)
! with the projective transformation, then untransform XC(1:N) (convert
! to affine coordinates).
XC(N+1)=SUM(PROJ_COEF(1:N)*XC(1:N))+PROJ_COEF(N+1)
!
! Deal carefully with solutions at infinity.
IF (ABS(XC(N+1)) < 1.0_R8) THEN
  DO I=1,N
    IF (ABS(XC(I)) >= BIG*ABS(XC(N+1))) THEN
      XC(I)=CMPLX(BIG,BIG,KIND=R8)  ! Solution at infinity.
    ELSE
      XC(I)=XC(I)/XC(N+1)
    END IF
  END DO
ELSE
  XC(1:N)=XC(1:N)/XC(N+1)
END IF
!
! Unscale the variables.
IF (.NOT. PRESENT(NO_SCALING)) THEN
```

```
  DO I=1,N
    IF (REAL(XC(I)) /= BIG) XC(I)=XC(I)*(10.0_R8**SCALE_FACTORS(I))
  END DO
END IF
!
RETURN
END SUBROUTINE OUTPUT_PLP
END SUBROUTINE POLSYS_PLP
!
!
SUBROUTINE BEZOUT_PLP(N,MAXT,TOL,BPLP)
!
! BEZOUT_PLP calculates and returns only the generalized Bezout number
! BPLP of the target polynomial system, based on the variable partition
! P defined in the module GLOBAL_PLP.  BEZOUT_PLP finds BPLP very
! quickly, which is useful for exploring alternative partitions.
!
! Calls SINGSYS_PLP.
!
! On input:
!
! N is the dimension of the target system.
!
! MAXT is the maximum number of terms in any component of the target
!   system.  MAXT = MAX((/(NUMT(I),I=1,N)/)).
!
! TOL is the singularity test threshold used by SINGSYS_PLP.  If
!   TOL <= 0.0 on input, TOL is reset to the default value
!   SQRT(EPSILON(1.0_R8)).
!
! GLOBAL_PLP allocatable objects POLYNOMIAL, PARTITION_SIZES, and
!   PARTITION (see GLOBAL_PLP documentation) must be allocated and
!   defined in the calling program.
!
! On output:
!
! N and MAXT are unchanged, and TOL may have been changed as described
!   above.
!
! BPLP is the generalized Bezout number for the target system based on
!   the variable partition P defined in the module GLOBAL_PLP.
!
USE GLOBAL_PLP
INTEGER, INTENT(IN):: N, MAXT
REAL (KIND=R8), INTENT(IN OUT):: TOL
INTEGER, INTENT(OUT):: BPLP
!INTERFACE
!  SUBROUTINE SINGSYS_PLP(N,LEX_NUM,LEX_SAVE,TOL,RAND_MAT,MAT,NONSING)
!  USE GLOBAL_PLP
!  INTEGER, INTENT(IN):: N
!  INTEGER, DIMENSION(N), INTENT(IN OUT):: LEX_NUM,LEX_SAVE
!  REAL (KIND=R8), INTENT(IN):: TOL
!  REAL (KIND=R8), DIMENSION(N,N), INTENT(IN):: RAND_MAT
!  REAL (KIND=R8), DIMENSION(N+1,N), INTENT(IN OUT):: MAT
!  LOGICAL, INTENT(OUT):: NONSING
!  END SUBROUTINE SINGSYS_PLP
```

```
!END INTERFACE
!
! Local variables.
INTEGER:: I, J, K, L
INTEGER, DIMENSION(MAXT):: DHOLD
INTEGER, DIMENSION(N):: LEX_NUM, LEX_SAVE
REAL (KIND=R8), DIMENSION(N+1,N):: MAT
REAL (KIND=R8), DIMENSION(N,N):: RAND_MAT
REAL, DIMENSION(N,N):: RANDNUMS
LOGICAL:: NONSING
!
! Set default value for singularity threshold TOL.
IF (TOL <= REAL(N,KIND=R8)*EPSILON(1.0_R8)) TOL=SQRT(EPSILON(1.0_R8))
!
! Initialize RAND_MAT with random numbers uniformly distributed in
! [-1,-1/2] union [1/2,1].
CALL RANDOM_SEED
CALL RANDOM_NUMBER(HARVEST=RANDNUMS)
RANDNUMS=RANDNUMS - 0.5 + SIGN(0.5, RANDNUMS - 0.5)
RAND_MAT=REAL(RANDNUMS,KIND=R8)
!
! Calculate set degrees of the variable partition P.
DHOLD=0
DO I=1,N
  DO J=1,PARTITION_SIZES(I)
    DO K=1,NUMV(I,J)
      DHOLD(1:NUMT(I))=(/(D(I,L,PAR(I,J,K)),L=1,NUMT(I))/)+DHOLD(1:NUMT(I))
    END DO
    PARTITION(I)%SET(J)%SET_DEG=MAXVAL(DHOLD(1:NUMT(I)))
    DHOLD=0
  END DO
END DO
!
! Compute Bezout number using lexicographic ordering.
!
BPLP=0
LEX_NUM(1:N-1)=1
LEX_NUM(N)=0
LEX_SAVE=0
MAIN_LOOP: DO
  DO J=N,1,-1
    IF (LEX_NUM(J) < PARTITION_SIZES(J)) THEN
      L=J
      EXIT
    END IF
  END DO
  LEX_NUM(L)=LEX_NUM(L)+1
  IF (L+1 <= N) LEX_NUM(L+1:N)=1
!
! Test singularity of start subsystem corresponding to lexicographic
! vector LEX_NUM.
  CALL SINGSYS_PLP(N,LEX_NUM,LEX_SAVE,TOL,RAND_MAT,MAT,NONSING)
  IF (NONSING) BPLP=BPLP+PRODUCT((/(SD(K,LEX_NUM(K)),K=1,N)/))
  IF (ALL(LEX_NUM == PARTITION_SIZES)) EXIT
END DO MAIN_LOOP
!
```

```
RETURN
END SUBROUTINE BEZOUT_PLP
!
!
SUBROUTINE SINGSYS_PLP(N,LEX_NUM,LEX_SAVE,TOL,RAND_MAT,MAT,NONSING)
!
! SINGSYS_PLP determines if the subsystem of the start system
! corresponding to the lexicographic vector LEX_NUM is nonsingular,
! or if a family of subsystems of the start system defined by
! LEX_NUM and LEX_SAVE is singular, by using Householder reflections and
! tree pruning.  Using the notation defined in the module GLOBAL_PLP,
! the vector LEX_NUM defines a linear system of equations
!    L(1,LEX_NUM(1)) = constant_1
!                  .
!                  .
!                  .
!    L(N,LEX_NUM(N)) = constant_N
! which, if nonsingular for generic coefficients, defines
! PRODUCT((/ (SD(K,LEX_NUM(K)), K=1,N) /)) nonsingular starting points
! for homotopy paths.  Nonsingularity of a generic coefficient matrix is
! checked by computing a QR decomposition of the transpose of the
! coefficient matrix.  Observe that if the first J rows are rank
! deficient, then all lexicographic vectors (LEX_NUM(1:J), *) also
! correspond to singular systems, and thus the tree of all possible
! lexicographic orderings can be pruned.
!
! The QR factorization is maintained as a product of Householder
! reflections, and updated based on the difference between LEX_SAVE
! (the value of LEX_NUM returned from the previous call to SINGSYS_PLP)
! and the current input LEX_NUM.  LEX_SAVE and LEX_NUM together
! implicitly define a family of subsystems, namely, all those
! corresponding to lexicographic orderings with head LEX_NUM(1:J),
! where J is the smallest index such that LEX_SAVE(J) /= LEX_NUM(J).
!
! Calls LAPACK subroutines DLARFX and DLARFG.
!
! On input:
!
! N is the dimension of the start and target systems.
!
! LEX_NUM(1:N) is a lexicographic vector which specifies a particular
!   subsystem (and with LEX_SAVE a family of subsystems) of the start
!   system.
!
! LEX_SAVE(1:N) holds the value of LEX_NUM returned from the previous
!   call, and should not be changed between calls to SINGSYS_PLP.  Set
!   LEX_SAVE=0 on the first call to SINGSYS_PLP.
!
! TOL is the singularity test threshold.  The family of subsystems
!   corresponding to lexicographic vectors (LEX_NUM(1:J), *) is declared
!   singular if ABS(R(J,J)) < TOL for the QR factorization of a generic
!   start system coefficient matrix.
!
! RAND_MAT(N,N) is a random matrix with entries uniformly distributed
!   in [-1,-1/2] union [1/2,1], used to seed the random generic
!   coefficient matrix MAT.  RAND_MAT should not change between calls to
```

```
!   SINGSYS_PLP.
!
! On output:
!
! LEX_NUM is unchanged if NONSING=.TRUE.  If NONSING=.FALSE.,
!   LEX_NUM(1:J) is unchanged, and
!   LEX_NUM(J+1:N) = PARTITION_SIZES(J+1:N), where J is the smallest
!   index such that ABS(R(J,J)) < TOL for the QR factorization of the
!   generic start system coefficient matrix corresponding to LEX_NUM
!   (on input).
!
! LEX_SAVE = LEX_NUM.
!
! NONSING = .TRUE. if the subsystem of the start system defined by
!   LEX_NUM is nonsingular.  NONSING = .FALSE. otherwise, which means that
!   the entire family of subsystems corresponding to lexicographic vectors
!   (LEX_NUM(1:J), *) is singular, where J is the smallest index such that
!   ABS(R(J,J)) < TOL for the QR factorization of the generic start system
!   coefficient matrix corresponding to LEX_NUM (on input).
!
! Working storage:
!
! MAT(N+1,N) is updated on successive calls to SINGSYS_PLP, and should
! not be changed by the calling program.  MAT can be undefined on the
! first call to SINGSYS_PLP (when LEX_SAVE = 0).  Define J as the
! smallest index where LEX_SAVE(J) /= LEX_NUM(J).  Upon exit after a
! subsequent call, for some M >= J, MAT contains, in the first M columns,
! a partial QR factorization stored as a product of Householder
! reflections, and, in the last N-M columns, random numbers that define
! the subsystem of the start system corresponding to the lexicographic
! vector LEX_NUM.  For 1<=K<=M, V(2:N+1-K)=MAT(K+1:N,K), V(1)=1, together
! with TAU=MAT(N+1,K), define a Householder reflection of dimension
! N+1-K.
!
USE GLOBAL_PLP
INTEGER, INTENT(IN):: N
INTEGER, DIMENSION(N), INTENT(IN OUT):: LEX_NUM, LEX_SAVE
REAL (KIND=R8), INTENT(IN):: TOL
REAL (KIND=R8), DIMENSION(N,N), INTENT(IN):: RAND_MAT
REAL (KIND=R8), DIMENSION(N+1,N), INTENT(IN OUT):: MAT
LOGICAL, INTENT(OUT):: NONSING
!
! Local variables.
INTEGER:: I, J, K
REAL (KIND=R8), DIMENSION(N):: V
REAL (KIND=R8):: WORK
IF (N == 1) THEN
  LEX_SAVE=LEX_NUM
  NONSING=.TRUE.
  RETURN
END IF
!
! (Re)set MAT (in column form) from LEX_NUM.
DO I=1,N
  IF (LEX_SAVE(I) /= LEX_NUM(I)) THEN
    LEX_SAVE(I+1:N)=0
```

```
    DO K=I,N
      MAT(1:N+1,K)=0.0_R8
      DO J=1,NUMV(K,LEX_NUM(K))
        MAT(PAR(K,LEX_NUM(K),J),K)=RAND_MAT(PAR(K,LEX_NUM(K),J),K)
      END DO
    END DO
    EXIT
  END IF
END DO
!
! Recompute QR factorization of MAT starting where first change in
! LEX_NUM occurred.
NONSING=.FALSE.
IF (LEX_SAVE(1) /= LEX_NUM(1)) THEN
! Skip QR factorization and prune tree if this set degree = 0.
  IF (SD(1,LEX_NUM(1)) == 0) THEN
    LEX_NUM(2:N)=PARTITION_SIZES(2:N)
    LEX_SAVE=LEX_NUM
    RETURN
  ELSE
    CALL DLARFG(N,MAT(1,1),MAT(2:N,1),1,MAT(N+1,1))
  END IF
END IF
DO J=2,N
  IF (LEX_SAVE(J) /= LEX_NUM(J)) THEN
! Skip rest of QR factorization and prune tree if this set degree = 0.
    IF (SD(J,LEX_NUM(J)) == 0) THEN
      IF (J < N) LEX_NUM(J+1:N)=PARTITION_SIZES(J+1:N)
      EXIT
    END IF
    DO K=1,J-1
      V(K)=1.0_R8
      V(K+1:N)=MAT(K+1:N,K)
      CALL DLARFX('L',N-K+1,1,V(K:N),MAT(N+1,K),MAT(K:N,J),N-K+1,WORK)
    END DO
    IF (J < N) CALL DLARFG(N-J+1,MAT(J,J),MAT(J+1:N,J),1,MAT(N+1,J))
! Check singularity of subsystem corresponding to lexicographic
! vector (LEX_NUM(1:J), *).
    IF (ABS(MAT(J,J)) < TOL) THEN
      IF (J < N) LEX_NUM(J+1:N)=PARTITION_SIZES(J+1:N)
      EXIT
    END IF
  END IF
! Subsystem corresponding to LEX_NUM is nonsingular when J==N here.
  IF (J == N) NONSING=.TRUE.
END DO
! Save updated LEX_NUM for next call.
LEX_SAVE=LEX_NUM
RETURN
END SUBROUTINE SINGSYS_PLP
!
!
END MODULE POLSYS
```

Appendix B: The Sample Calling Program MAIN_TEMPLATE

```
PROGRAM MAIN_TEMPLATE
!
! MAIN_TEMPLATE is a template for calling BEZOUT_PLP and POLSYS_PLP.
! There are two options provided by MAIN_TEMPLATE:  (1) MAIN_TEMPLATE
! returns only the generalized PLP Bezout number ("root count") of the
! target polynomial system based on a system partition provided by the
! user (calls BEZOUT_PLP) or (2) MAIN_TEMPLATE returns the root count,
! homotopy path tracking statistics, error flags, and the roots (calls
! POLSYS_PLP).  For the first option set the logical switch
! ROOT_COUNT_ONLY = .TRUE., and for the second option set ROOT_COUNT_ONLY
! = .FALSE..
!
! The file INPUT.DAT contains data for several sample target systems
! and system partitions.  This main program illustrates how to find the
! root count for several different partitions for the same polynomial
! system, and also how to solve more than one polynomial system in the
! same run.  The data is read in using NAMELISTs, which makes the data
! file INPUT.DAT self-explanatory.  The problem definition is given in
! the NAMELIST /PROBLEM/ and the PLP system partition is defined in the
! NAMELIST /SYSPARTITION/.  A new polynomial system definition is
! signalled by setting the variable NEW_PROBLEM=.TRUE. in the /PROBLEM/
! namelist.  Thus a data file describing several different polynomial
! systems to solve, and exploring different system partitions for the
! same polynomial system, might look like
!
! &PROBLEM NEW_PROBLEM=.TRUE.  data  /
! &SYSPARTITION ROOT_COUNT_ONLY=.FALSE.  data  /  finds roots
!
! &PROBLEM NEW_PROBLEM=.TRUE.  data  /
! &SYSPARTITION ROOT_COUNT_ONLY=.TRUE.  data  /  finds root count only
! &PROBLEM NEW_PROBLEM=.FALSE.  /
! &SYSPARTITION ROOT_COUNT_ONLY=.TRUE.  data  /  a different root count
! &PROBLEM NEW_PROBLEM=.FALSE.  /
! &SYSPARTITION ROOT_COUNT_ONLY=.TRUE.  data  /  another root count
!
! Note that static arrays are used below only to support NAMELIST input;
! the actual storage of the polynomial system and partition information
! in the data structures in the module GLOBAL_PLP is very compact.
!
!
USE POLSYS
!
! Local variables.
INTEGER, PARAMETER:: NN=30, MMAXT=50
INTEGER:: BPLP, I, IFLAG1, J, K, M, MAXT, N, NUMRR=1
INTEGER, DIMENSION(NN):: NUM_TERMS, NUM_SETS
INTEGER, DIMENSION(NN,NN):: NUM_INDICES
INTEGER, DIMENSION(NN,NN,NN):: INDEX
INTEGER, DIMENSION(NN,MMAXT,NN):: DEG
INTEGER, DIMENSION(:), POINTER:: IFLAG2, NFE
REAL (KIND=R8):: TRACKTOL, FINALTOL, SINGTOL
REAL (KIND=R8), DIMENSION(8):: SSPAR
REAL (KIND=R8), DIMENSION(NN):: SCALE_FACTORS
```

```
REAL (KIND=R8), DIMENSION(:), POINTER:: ARCLEN, LAMBDA
COMPLEX (KIND=R8), DIMENSION(NN,MMAXT):: COEF
COMPLEX (KIND=R8), DIMENSION(:,:), POINTER:: ROOTS
CHARACTER (LEN=80):: TITLE
CHARACTER (LEN=80), DIMENSION(NN):: P
LOGICAL:: NEW_PROBLEM, NO_SCALING, RECALL, ROOT_COUNT_ONLY
!
NAMELIST /PROBLEM/ COEF, DEG, FINALTOL, NEW_PROBLEM, N, NUMRR, NUM_TERMS,  &
  SINGTOL, SSPAR, TITLE, TRACKTOL
NAMELIST /SYSPARTITION/ INDEX, NUM_INDICES, NUM_SETS, P, ROOT_COUNT_ONLY
!
!
! MAIN_TEMPLATE reads the target polynomial system definition and the
! system partition specification from the file INPUT.DAT.
OPEN (UNIT=3,FILE='INPUT.DAT',ACTION='READ',POSITION='REWIND',   &
    DELIM='APOSTROPHE',STATUS='OLD')
!
! All output is to the file OUTPUT.DAT, which is overwritten.
OPEN (UNIT=7,FILE='OUTPUT.DAT',ACTION='WRITE',STATUS='REPLACE',DELIM='NONE')
!
SSPAR(1:8)=0.0_R8
DEG=0
COEF=(0.0_R8,0.0_R8)
MAIN_LOOP: DO
READ (3,NML=PROBLEM,END=500)
IF (NEW_PROBLEM) THEN
  WRITE (7,190) TITLE,TRACKTOL,FINALTOL,SINGTOL,SSPAR(5),N
  190 FORMAT(///A80//'TRACKTOL, FINALTOL =',2ES22.14, &
    /,'SINGTOL (0 SETS DEFAULT) =',ES22.14, &
    /,'SSPAR(5) (0 SETS DEFAULT) =',ES22.14, &
    /,'NUMBER OF EQUATIONS =',I3)
  WRITE (7,200)
  200 FORMAT(/'****** COEFFICIENT TABLEAU ******')
  DO I=1,N
    WRITE (7,210) I,NUM_TERMS(I)
    210 FORMAT(/,'POLYNOMIAL(',I2,')%NUM_TERMS =',I3)
    DO J=1,NUM_TERMS(I)
      WRITE (7,220) (I,J,K,DEG(I,J,K), K=1,N)
      220 FORMAT('POLYNOMIAL(',I2,')%TERM(',I2,')%DEG(',I2,') =',I2)
      WRITE (7,230) I,J,COEF(I,J)
      230 FORMAT('POLYNOMIAL(',I2,')%TERM(',I2,')%COEF = (',ES22.14, &
          ',',ES22.14,')')
    END DO
  END DO
!
! Allocate storage for the target system in POLYNOMIAL.
  CALL CLEANUP_POL
  ALLOCATE(POLYNOMIAL(N))
  DO I=1,N
    POLYNOMIAL(I)%NUM_TERMS=NUM_TERMS(I)
    ALLOCATE(POLYNOMIAL(I)%TERM(NUM_TERMS(I)))
    DO J=1,NUM_TERMS(I)
      ALLOCATE(POLYNOMIAL(I)%TERM(J)%DEG(N+1))
      POLYNOMIAL(I)%TERM(J)%COEF=COEF(I,J)
      POLYNOMIAL(I)%TERM(J)%DEG(1:N)=DEG(I,J,1:N)
    END DO
```

```
  END DO
END IF
READ (3,NML=SYSPARTITION)
!
! Allocate storage for the system partition in PARTITION.
CALL CLEANUP_PAR
ALLOCATE(PARTITION_SIZES(N))
PARTITION_SIZES(1:N)=NUM_SETS(1:N)
ALLOCATE(PARTITION(N))
DO I=1,N
  ALLOCATE(PARTITION(I)%SET(PARTITION_SIZES(I)))
  DO J=1,PARTITION_SIZES(I)
    PARTITION(I)%SET(J)%NUM_INDICES=NUM_INDICES(I,J)
    ALLOCATE(PARTITION(I)%SET(J)%INDEX(NUM_INDICES(I,J)))
    PARTITION(I)%SET(J)%INDEX(1:NUM_INDICES(I,J)) = &
      INDEX(I,J,1:NUM_INDICES(I,J))
  END DO
END DO
!
IF (ROOT_COUNT_ONLY) THEN
! Compute only the PLP Bezout number BPLP for this partition.
  MAXT=MAXVAL(NUM_TERMS(1:N))
  CALL BEZOUT_PLP(N,MAXT,SINGTOL,BPLP)
ELSE
! Compute all BPLP roots of the target polynomial system.
  CALL POLSYS_PLP(N,TRACKTOL,FINALTOL,SINGTOL,SSPAR,BPLP,IFLAG1,IFLAG2, &
    ARCLEN,LAMBDA,ROOTS,NFE,SCALE_FACTORS)
END IF
!
WRITE (7,240) BPLP, (K,TRIM(P(K)),K=1,N)
240 FORMAT(//,'GENERALIZED PLP BEZOUT NUMBER (BPLP) =',I10, &
  /'BASED ON THE FOLLOWING SYSTEM PARTITION:',/('P(',I2,') = ',A))
!
IF (.NOT. ROOT_COUNT_ONLY) THEN
  DO M=1,BPLP
    WRITE (7,260) M,ARCLEN(M),NFE(M),IFLAG2(M)
    260 FORMAT(/'PATH NUMBER =',I10//'ARCLEN =',ES22.14/'NFE =',I5/ &
      'IFLAG2 =',I3)
!
! Designate solutions as "REAL" or "COMPLEX."
    IF (ANY(ABS(AIMAG(ROOTS(1:N,M))) >= 1.0E-4_R8)) THEN
      WRITE (7,270,ADVANCE='NO')
      270 FORMAT('COMPLEX, ')
    ELSE
      WRITE (7,280,ADVANCE='NO')
      280 FORMAT('REAL, ')
    END IF
!
! Designate solutions as "FINITE" or "INFINITE."
    IF (ABS(ROOTS(N+1,M)) < 1.0E-6_R8) THEN
      WRITE (7,290)
      290 FORMAT('INFINITE SOLUTION')
    ELSE
      WRITE (7,300)
      300 FORMAT('FINITE SOLUTION')
    END IF
```

```
    IF (MOD(IFLAG2(M),10) == 1) THEN
      WRITE (7,310) 1.0_R8,LAMBDA(M)
      310 FORMAT('LAMBDA =',ES22.14,', ESTIMATED ERROR =',ES22.14/)
    ELSE
      WRITE (7,315) LAMBDA(M)
      315 FORMAT('LAMBDA =',ES22.14/)
    END IF
    WRITE (7,320) (J,ROOTS(J,M),J=1,N)
    320 FORMAT(('X(',I2,') = (',ES22.14,',',ES22.14,')'))
    WRITE (7,330) N+1,ROOTS(N+1,M)
    330 FORMAT(/,'X(',I2,') = (',ES22.14,',',ES22.14,')')
  END DO
END IF
END DO MAIN_LOOP
500 CALL TEST_OPTIONS ! This tests various options, and is not part of a
                      ! typical main program.
CLOSE (UNIT=3); CLOSE (UNIT=7)
CALL CLEANUP_POL
CALL CLEANUP_PAR
STOP
!
CONTAINS
SUBROUTINE CLEANUP_POL
! Deallocates structure POLYNOMIAL.
IF (.NOT. ALLOCATED(POLYNOMIAL)) RETURN
DO I=1,SIZE(POLYNOMIAL)
  DO J=1,NUMT(I)
    DEALLOCATE(POLYNOMIAL(I)%TERM(J)%DEG)
  END DO
  DEALLOCATE(POLYNOMIAL(I)%TERM)
END DO
DEALLOCATE(POLYNOMIAL)
RETURN
END SUBROUTINE CLEANUP_POL
!
SUBROUTINE CLEANUP_PAR
! Deallocates structure PARTITION.
IF (.NOT. ALLOCATED(PARTITION)) RETURN
DO I=1,SIZE(PARTITION)
  DO J=1,PARTITION_SIZES(I)
    DEALLOCATE(PARTITION(I)%SET(J)%INDEX)
  END DO
  DEALLOCATE(PARTITION(I)%SET)
END DO
DEALLOCATE(PARTITION)
DEALLOCATE(PARTITION_SIZES)
RETURN
END SUBROUTINE CLEANUP_PAR
!
SUBROUTINE TEST_OPTIONS
! Illustrate use of optional arguments NUMRR, NO_SCALING, USER_F_DF:
TRACKTOL=1.0E-6_R8;  FINALTOL=1.0E-8_R8
CALL POLSYS_PLP(N,TRACKTOL,FINALTOL,SINGTOL,SSPAR,BPLP,IFLAG1,IFLAG2, &
  ARCLEN,LAMBDA,ROOTS,NFE,SCALE_FACTORS, NUMRR=1, NO_SCALING=.TRUE.,  &
  USER_F_DF=.TRUE.)
M = 3
```

```
WRITE (7,FMT="(//'Testing optional arguments.')")
WRITE (7,260) M,ARCLEN(M),NFE(M),IFLAG2(M)
IF (MOD(IFLAG2(M),10) == 1) THEN
  WRITE (7,310) 1.0_R8,LAMBDA(M)
ELSE
  WRITE (7,315) LAMBDA(M)
END IF
WRITE (7,320) (J,ROOTS(J,M),J=1,N)
WRITE (7,330) N+1,ROOTS(N+1,M)
!
! Now retrack one of these paths (#3) using the RECALL option:
IFLAG2(3) = -2
TRACKTOL=1.0E-10_R8;  FINALTOL=1.0E-14_R8
CALL POLSYS_PLP(N,TRACKTOL,FINALTOL,SINGTOL,SSPAR,BPLP,IFLAG1,IFLAG2, &
  ARCLEN,LAMBDA,ROOTS,NFE,SCALE_FACTORS, NUMRR=3, NO_SCALING=.TRUE.,  &
  USER_F_DF=.TRUE., RECALL=.TRUE.)
M = 3
WRITE (7,FMT="(//'Statistics for retracked path.')")
WRITE (7,260) M,ARCLEN(M),NFE(M),IFLAG2(M)
IF (MOD(IFLAG2(M),10) == 1) THEN
  WRITE (7,310) 1.0_R8,LAMBDA(M)
ELSE
  WRITE (7,315) LAMBDA(M)
END IF
WRITE (7,320) (J,ROOTS(J,M),J=1,N)
WRITE (7,330) N+1,ROOTS(N+1,M)
RETURN
260 FORMAT(/'PATH NUMBER =',I10//'ARCLEN =',ES22.14/'NFE =',I5/ &
    'IFLAG2 =',I3)
310 FORMAT('LAMBDA =',ES22.14,', ESTIMATED ERROR =',ES22.14/)
315 FORMAT('LAMBDA =',ES22.14/)
320 FORMAT(('X(',I2,') = (',ES22.14,',',ES22.14,')'))
330 FORMAT(/,'X(',I2,') = (',ES22.14,',',ES22.14,')')
END SUBROUTINE TEST_OPTIONS
!
END PROGRAM MAIN_TEMPLATE
```

Appendix C: Sample Input for MAIN_TEMPLATE

```
&PROBLEM  NEW_PROBLEM=.TRUE.
TITLE='TWO QUADRICS, NO SOLUTIONS AT INFINITY, TWO REAL SOLUTIONS.'
TRACKTOL = 1.0D-4  FINALTOL = 1.0D-14  SINGTOL = 0.0  SSPAR(5) = 1.0D0
NUMRR = 1
N = 2
NUM_TERMS(1) = 6
COEF(1,1) = (-9.80D-04,0.0)   DEG(1,1,1) = 2
COEF(1,2) = ( 9.78D+05,0.0)   DEG(1,2,2) = 2
COEF(1,3) = (-9.80D+00,0.0)   DEG(1,3,1) = 1 DEG(1,3,2) = 1
COEF(1,4) = (-2.35D+02,0.0)   DEG(1,4,1) = 1
COEF(1,5) = ( 8.89D+04,0.0)   DEG(1,5,2) = 1
COEF(1,6) = (-1.00D+00,0.0)
NUM_TERMS(2) = 6
COEF(2,1) = (-1.00D-02,0.0)   DEG(2,1,1) = 2
COEF(2,2) = (-9.84D-01,0.0)   DEG(2,2,2) = 2
COEF(2,3) = (-2.97D+01,0.0)   DEG(2,3,1) = 1 DEG(2,3,2) = 1
COEF(2,4) = ( 9.87D-03,0.0)   DEG(2,4,1) = 1
COEF(2,5) = (-1.24D-01,0.0)   DEG(2,5,2) = 1
COEF(2,6) = (-2.50D-01,0.0)                  /
&SYSPARTITION  ROOT_COUNT_ONLY = .FALSE.
P(1) = '{{x1,x2}}'
P(2) = '{{x1,x2}}'
NUM_SETS(1) = 1  NUM_INDICES(1,1) = 2
INDEX(1,1,1) = 1 INDEX(1,1,2) = 2
NUM_SETS(2) = 1  NUM_INDICES(2,1) = 2
INDEX(2,1,1) = 1 INDEX(2,1,2) = 2   /
```

Appendix D: Sample Output from MAIN_TEMPLATE

```
TWO QUADRICS, NO SOLUTIONS AT INFINITY, TWO REAL SOLUTIONS.
TRACKTOL, FINALTOL =  1.00000000000000E-04  1.00000000000000E-14
SINGTOL (0 SETS DEFAULT) =  0.00000000000000E+00
SSPAR(5) (0 SETS DEFAULT) =  1.00000000000000E+00
NUMBER OF EQUATIONS =  2
****** COEFFICIENT TABLEAU ******
POLYNOMIAL( 1)%NUM_TERMS =  6
POLYNOMIAL( 1)%TERM( 1)%DEG( 1) = 2
POLYNOMIAL( 1)%TERM( 1)%DEG( 2) = 0
POLYNOMIAL( 1)%TERM( 1)%COEF = ( -9.80000000000000E-04,  0.00000000000000E+00)
POLYNOMIAL( 1)%TERM( 2)%DEG( 1) = 0
POLYNOMIAL( 1)%TERM( 2)%DEG( 2) = 2
POLYNOMIAL( 1)%TERM( 2)%COEF = (  9.78000000000000E+05,  0.00000000000000E+00)
POLYNOMIAL( 1)%TERM( 3)%DEG( 1) = 1
POLYNOMIAL( 1)%TERM( 3)%DEG( 2) = 1
POLYNOMIAL( 1)%TERM( 3)%COEF = ( -9.80000000000000E+00,  0.00000000000000E+00)
POLYNOMIAL( 1)%TERM( 4)%DEG( 1) = 1
POLYNOMIAL( 1)%TERM( 4)%DEG( 2) = 0
POLYNOMIAL( 1)%TERM( 4)%COEF = ( -2.35000000000000E+02,  0.00000000000000E+00)
POLYNOMIAL( 1)%TERM( 5)%DEG( 1) = 0
POLYNOMIAL( 1)%TERM( 5)%DEG( 2) = 1
POLYNOMIAL( 1)%TERM( 5)%COEF = (  8.89000000000000E+04,  0.00000000000000E+00)
POLYNOMIAL( 1)%TERM( 6)%DEG( 1) = 0
POLYNOMIAL( 1)%TERM( 6)%DEG( 2) = 0
POLYNOMIAL( 1)%TERM( 6)%COEF = ( -1.00000000000000E+00,  0.00000000000000E+00)
POLYNOMIAL( 2)%NUM_TERMS =  6
POLYNOMIAL( 2)%TERM( 1)%DEG( 1) = 2
POLYNOMIAL( 2)%TERM( 1)%DEG( 2) = 0
POLYNOMIAL( 2)%TERM( 1)%COEF = ( -1.00000000000000E-02,  0.00000000000000E+00)
POLYNOMIAL( 2)%TERM( 2)%DEG( 1) = 0
POLYNOMIAL( 2)%TERM( 2)%DEG( 2) = 2
POLYNOMIAL( 2)%TERM( 2)%COEF = ( -9.84000000000000E-01,  0.00000000000000E+00)
POLYNOMIAL( 2)%TERM( 3)%DEG( 1) = 1
POLYNOMIAL( 2)%TERM( 3)%DEG( 2) = 1
POLYNOMIAL( 2)%TERM( 3)%COEF = ( -2.97000000000000E+01,  0.00000000000000E+00)
POLYNOMIAL( 2)%TERM( 4)%DEG( 1) = 1
POLYNOMIAL( 2)%TERM( 4)%DEG( 2) = 0
POLYNOMIAL( 2)%TERM( 4)%COEF = (  9.87000000000000E-03,  0.00000000000000E+00)
POLYNOMIAL( 2)%TERM( 5)%DEG( 1) = 0
POLYNOMIAL( 2)%TERM( 5)%DEG( 2) = 1
POLYNOMIAL( 2)%TERM( 5)%COEF = ( -1.24000000000000E-01,  0.00000000000000E+00)
POLYNOMIAL( 2)%TERM( 6)%DEG( 1) = 0
POLYNOMIAL( 2)%TERM( 6)%DEG( 2) = 0
POLYNOMIAL( 2)%TERM( 6)%COEF = ( -2.50000000000000E-01,  0.00000000000000E+00)
GENERALIZED PLP BEZOUT NUMBER (BPLP) =        4
BASED ON THE FOLLOWING SYSTEM PARTITION:
P( 1) = {{x1,x2}}
P( 2) = {{x1,x2}}
PATH NUMBER =        1
ARCLEN =  4.88774990521209E+00
NFE =   60
IFLAG2 = 11
REAL, FINITE SOLUTION
```

```
LAMBDA =  1.00000000000000E+00, ESTIMATED ERROR =  3.38846533606060E-16
X( 1) = (  2.34233851959124E+03,  1.65647002775734E-11)
X( 2) = ( -7.88344824094128E-01, -8.79747864489673E-15)
X( 3) = (  1.46585064047300E-03,  3.95169260376393E-03)
PATH NUMBER =        2
ARCLEN =  1.05721592852627E+01
NFE =   86
IFLAG2 = 11
COMPLEX, FINITE SOLUTION
LAMBDA =  1.00000000000000E+00, ESTIMATED ERROR =  5.25603639602664E-19
X( 1) = (  1.61478579234358E-02,  1.68496955498881E+00)
X( 2) = (  2.67994739614461E-04,  4.42802993973661E-03)
X( 3) = (  6.28839380529929E-01, -6.23770253460440E-01)
PATH NUMBER =        3
ARCLEN =  1.40888746921829E+00
NFE =   54
IFLAG2 = 11
REAL, FINITE SOLUTION
LAMBDA =  1.00000000000000E+00, ESTIMATED ERROR =  2.39884416065940E-18
X( 1) = (  9.08921229615392E-02, -1.58321340069559E-16)
X( 2) = ( -9.11497098197500E-02, -6.00374957769670E-17)
X( 3) = (  1.78804341487788E-01, -9.61729429034368E-02)
PATH NUMBER =        4
ARCLEN =  3.06763846239533E+00
NFE =   73
IFLAG2 = 11
COMPLEX, FINITE SOLUTION
LAMBDA =  1.00000000000000E+00, ESTIMATED ERROR =  2.32172816672818E-15
X( 1) = (  1.61478579234664E-02, -1.68496955498883E+00)
X( 2) = (  2.67994739614542E-04, -4.42802993973661E-03)
X( 3) = (  2.52977008145125E-01, -8.41936532701388E-01)
Testing optional arguments.
PATH NUMBER =        3
ARCLEN =  1.37807382637515E+00
NFE =   50
IFLAG2 = 11
LAMBDA =  1.00000000000000E+00, ESTIMATED ERROR =  3.62025597748724E-12
X( 1) = (  8.30518610084354E-03, -7.06597379977770E-12)
X( 2) = ( -3.06685914636607E+00, -2.10576255707730E-11)
X( 3) = (  1.78804341485080E-01, -9.61729429030812E-02)
Statistics for retracked path.
PATH NUMBER =        3
ARCLEN =  1.39789758479940E+00
NFE =   77
IFLAG2 = 11
LAMBDA =  1.00000000000000E+00, ESTIMATED ERROR =  8.31859886381831E-18
X( 1) = (  8.30518610365687E-03,  2.43299727056826E-17)
X( 2) = ( -3.06685914631600E+00, -6.73348433800512E-16)
X( 3) = (  1.78804341487788E-01, -9.61729429034368E-02)
```

Appendix E: The TARGET_SYSTEM_USER Template

```
SUBROUTINE TARGET_SYSTEM_USER(N,PROJ_COEF,XC,F,DF)
! Template for user written subroutine to evaluate the (complex) target
! system F(XC) and its (complex) N x N Jacobian matrix DF(XC).  XC(1:N+1)
! is in complex projective coordinates, and the homogeneous coordinate
! XC(N+1) is explicitly eliminated from F(XC) and DF(XC) using the
! projective transformation (cf. the comments in START_POINTS_PLP).  The
! comments in the internal subroutine TARGET_SYSTEM should be read before
! attempting to write this subroutine; pay particular attention to the
! handling of the homogeneous coordinate XC(N+1).  DF(:,N+1) is not
! referenced by the calling program.
!
USE REAL_PRECISION
USE GLOBAL_PLP
INTEGER, INTENT(IN):: N
COMPLEX (KIND=R8), INTENT(IN), DIMENSION(N+1):: PROJ_COEF,XC
COMPLEX (KIND=R8), INTENT(OUT):: F(N), DF(N,N+1)
!
! For greater efficiency, replace the following code (which is just the
! internal POLSYS_PLP subroutine TARGET_SYSTEM) with hand-crafted code.
! # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
INTEGER:: I, J, K, L
COMPLEX (KIND=R8):: T, TS
DO I=1,N
  TS = (0.0_R8, 0.0_R8)
  DO J=1,POLYNOMIAL(I)%NUM_TERMS
    T = POLYNOMIAL(I)%TERM(J)%COEF
    DO K=1,N+1
      IF (POLYNOMIAL(I)%TERM(J)%DEG(K) == 0) CYCLE
      T = T * XC(K)**POLYNOMIAL(I)%TERM(J)%DEG(K)
    END DO
    TS = TS + T
  END DO
  F(I)=TS
END DO
DF=(0.0_R8,0.0_R8)
DO I=1,N
  DO J=1,N+1
    TS = (0.0_R8,0.0_R8)
    DO K=1,POLYNOMIAL(I)%NUM_TERMS
      IF (POLYNOMIAL(I)%TERM(K)%DEG(J) == 0) CYCLE
      T = POLYNOMIAL(I)%TERM(K)%COEF * POLYNOMIAL(I)%TERM(K)%DEG(J) * &
          (XC(J)**(POLYNOMIAL(I)%TERM(K)%DEG(J) - 1))
      DO L=1,N+1
        IF ((L == J) .OR. (POLYNOMIAL(I)%TERM(K)%DEG(L) == 0)) CYCLE
        T = T * (XC(L)**POLYNOMIAL(I)%TERM(K)%DEG(L))
      END DO
      TS = TS + T
    END DO
    DF(I,J) = TS
  END DO
END DO
DO I=1,N
  DF(I,1:N) = DF(I,1:N) + PROJ_COEF(1:N) * DF(I,N+1)
```

```
END DO
! # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
RETURN
END SUBROUTINE TARGET_SYSTEM_USER
```

VITA

Steven M. Wise was born in Butler, Pennsylvania on April 4, 1974. He earned a bachelors degree in Applied Mathematics in May 1996 from Clarion University of Pennsylvania. In August 1998, he received the Master of Science degree in Mathematics from Virginia Polytechnic Institute and State University. He will attend The Pennsylvania State University in pursuit of the Doctor of Philosophy degree in Mathematics, and hopes to teach at university in the future. Steven is married to Nicole M. (Mooney) Wise.