National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Polynomial Zerofinding Matrix Algorithms

by

Fadi MALEK

Thesis submitted to the School of Graduate Studies
of the University of Ottawa
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

at the

Ottawa-Carleton Institute of Computer Science.

May 1995

Ottawa, Ontario, Canada K1N 6N5

ISBN 0-612-11576-3

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

In linear algebra, the eigenvalues of a matrix are equivalently defined as the zeros of its characteristic polynomial. Determining the zeros of polynomials by the computation of the eigenvalues of a corresponding companion matrix turns the table on the usual definition. In this dissertation, the work of Newbery has been expanded and a (complex) symmetric or nonsymmetric companion matrix associated with a given characteristic polynomial has been constructed. Schmeisser's technique for the construction of a tridiagonal companion matrix associated with a polynomial with real zeros has been generalized to polynomials with complex zeros. New matrix algorithms based on Schmeisser's and Fiedler's companion matrices are developed. The matrix algorithm which is based on Schmeisser's matrix uses no initial values and computes the simple and multiple zeros with high accuracy. The algorithms based on Fiedler's matrices are applied recursively, and require initial values as approximations to the true zeros of the polynomial. A few techniques concerning the choice of the required initial values are also presented. An important part of this thesis is the design of a new composite three-stage matrix algorithm for finding the real and complex zeros of polynomials. The composite algorithm reduces a polynomial with multiple zeros to another polynomial with simple zeros which are then computed with high accuracy. The exact multiplicities of these zeros are then calculated by means of Lagouanelle's limiting formula. The QR algorithm has been used in all the algorithms to find the eigenvalues of the companion matrices. The effectiveness of these algorithms is illustrated by presenting numerical results based on polynomials taken from the literature and considered to be ill-conditioned, as well as random polynomials with randomly generated zeros in small and large clusters. Polynomials are represented and evaluated in quadruple precision; but it suffices to use a double precision QR algorithm in order to obtain almost double precision in the zeros of polynomials.

Acknowlegments

Most special thanks go to my supervisor, Dr. Rémi Vaillancourt, for whom a proper thank would be larger than this thesis. Many good supervisors care about their students; Dr. Vaillancourt takes a personal interest that represents so much more. I am grateful to him for introducing me to numerical linear algebra, for inspiring me all along the way, for generously making available to me the best numerical analysis books, papers, a SUN workstation through NSERC Equipment Grants (EQP0122258 and EQP0139636), software, and so much more. Dr. Vaillancourt made many useful suggestions that went into this thesis for which I will always be grateful, but it is his friendship that I have valued most.

I am very grateful to NSERC (Natural Sciences and Engineering Research of Canada) and the University of Ottawa for providing me with financial support. Technologically, this thesis would have been impossible without the software packages MATLAB and Mathematica, which made experiments, and computations very easy.

Last, but far from least, I wish to extend my thanks to my parents, sister, and brothers for their love and support. I am grateful to God for having such wonderful parents who have always been supportive of me and surrounded me with their love and care throughout my entire life.

Contents

List of Tables

Chapter 1

Introduction

Polynomial zeros are one of the first nonlinear problems that mathematicians have frequently encountered in their research and in practice. Galois established that a general direct method for calculating zeros in terms of explicit formulas exists only for general polynomials of degree less than five. Therefore, to find the zeros of polynomials of higher degree one must apply numerical methods. Moreover, such methods are already used for polynomials of degree three and four since the corresponding explicit formulas are remarkably complicated.

The great importance of the problem of determining polynomial zeros in theory and in practice (e.g. in distributed control problems, stability of systems, nonlinear circuits, analysis of transfer functions, various mathematical models, differential and difference equations, eigenvalue problems and other disciplines) has led to the development of a great number of numerical methods in this field. A recent paper by McNamee [1] consists of a 3500 item bibliography on $3\frac{1}{2}$ inch diskette of papers related to roots of polynomials. These numerical methods, which generally take the form of iterative procedures, have become practically applicable together with the rapid development of digital c mputers some thirty years ago. It is not easy to choose the best algorithm for a given polynomial equation. Each algorithm possesses its own advantages and disadvantages and so, this appears to be an actual problem at the

present time.

In connection with any implementation of numerical methods on a computer, it is important to note that the selection of a zero-finding algorithm may depend heavily on other mathematical or practical considerations, such as speed and memory of the computing equipment and reliability of the result. Anyone using a computer has surely inquired about the effect of rounding errors and, eventually, propagated errors due to uncertain values of the polynomial coefficients. The computed solution of a polynomial equation is only an approximation to the true solution, since there are errors originating from discretization or truncation and from rounding.

Most of these algorithms compute only *one zero at a time*. If all zeros need to be computed, these algorithms usually work serially as follows: if a zero has been found to sufficient accuracy, the corresponding linear factor is removed from the polynomial by the deflation technique and the process is employed again to determine the remaining zeros of the deflated polynomial whose degree is now lowered by one. In many practical applications, it is necessary to find the zeros to great accuracy. But, if successive deflation is used, the polynomial obtained after division by the early inaccurate linear factors may have contaminated the coefficients to an extent which makes the remaining approximate zeros meaningless. This is a disadvantage of the method of successive removal of linear factors. As Wilkinson [2] pointed out, it is important that the zeros be deflated in increasing order of magnitude in order to reduce the rounding errors of the successive deflated polynomials. This is a further difficulty of the deflation technique.

The approximate zeros, produced as a result of deflation can be employed as initial approximations in some iterative process with the undeflated, original polynomial. However, in certain cases, the accuracy of the initial approximations may be such that this process may fail, particularly in the presence of clusters of zeros (i.e. set of several very close zeros) because the iterative process may converge towards a zero

that has already been determined. In addition, the accuracy of the computed zeros will be greatly affected by their condition regardless of deflation.

> For a given precision of computation the accuracy of determination of a zero is limited by its condition [...]. An ill-conditionned zero will be determined inaccurately even when we iterate with the original polynomial [2].

The above deflation difficulties can be overcome by determining all zeros simultaneously. There are many different approaches to these procedures. Some of the classical methods which exist in the literature can be categorized as follows:

1. *Iterative methods based on search procedures*: They present themselves as three-stage methods. Firstly, an initial region in the complex plane in the form of a circle or rectangle is found which contains all the zeros of the polynomial. Secondly, a slow convergent search procedure is applied to obtain initial approximations to the zeros and then estimate their multiplicities. Thirdly, a rapidly convergent iterative function is used to improve the approximations to any required accuracy. Examples of such methods can be found in [3, 4, 5].

2. *Methods which transform the polynomial equation into an equivalent system of nonlinear equations.* These methods are based on relations among symmetric functions of the roots and the coefficients. The system obtained from these relations is then solved by using Newton's method. This approach was taken by Durand and then Kerner, and allowed the rediscovery of the so-called Weierstrass-Durand-Kerner formulas referred to as WDK formulas. Examples of these methods are the WDK method [6, 7, 8], and the Pasquini-Trigiante method [9]. Because of the use of Newton's method, such methods are potentially dependent on the choice of the starting points and the multiplicities of the roots, unless information about these quantities

is provided.

Hull and Mathon [10] used the WDK formulas with initial values on a circle enclosing all the zeros of the polynomial. They also established mathematical properties of these formulas which suggest ways in which zero-finding algorithms based on these formulas can be speeded up. They also showed that while convergence is quadratic to simple roots and linear to multiple roots, the mean of the individual approximations to a multiple root converges quadratically to that root.

3. *Methods which factor polynomials using Euclid's algorithm.* These methods were used by Dunaway [11] for real polynomials. The original polynomial is factored into polynomials having only simple roots by means of the Euclidean algorithm. The zeros of the factored polynomials are then calculated by using Sturm sequences and Newton–Raphson method.

4. *Methods which transform the problem into an eigenvalue problem.* These methods construct a companion matrix whose characteristic polynomial is the original polynomial. Once the matrix is constructed, its eigenvalues, which are the roots of the original polynomial, can then be determined by some matrix method such as the QR or Jacobi algorithms.

A well-known method which uses this technique is based on the Frobenius companion matrix associated with the polynomial. This method is already used in Matlab [12] (ROOTS command) and gives an approximation to the roots. In the case of multiple roots, the obtainable approximations are not sharp. In a recently published paper by Toh and Trefethen [13], it is shown that the Matlab Roots algorithm which (optionally) balances the companion matrix and then computes its eigenvalues, is a stable algorithm for polynomial zero-finding. This observation came as a result of comparing the ϵ-pseudozero set, $Z_\epsilon(p)$, and the ϵ-pseudospectrum, $\lambda_\epsilon(A)$, of the com-

panion matrix and finding that the two sets are comparable. It is also shown that a modification of the Generalized Rayleigh Quotient Iteration (GRQI) associated with the companion matrix leads to the Jenkins-Traub [14] variable shift iteration for polynomial zero-finding. It is also mentioned in [13] that:

> The Jenkins-Traub iteration can be interpreted as a scheme for taking advantage of the companion matrix structure in Rayleigh quotient iteration so that the work per step is reduced from $O(n^3)$ to $o(n)$ where n is the degree of the polynomial p.

In [15], the backwards normwise stability of ROOTS is established. In [16], it is shown that the QR algorithm of EISPACK has considerable advantages over the standard algorithms of NAG (based on Laguerre's method) and IMSL (based on Jenkins and Traub's method) for finding the zeros of a polynomial.

The research reported here is directed to the problem of solving polynomial equations, having simple or multiple (real or complex) roots, by finding all zeros simultaneously. The approach taken to address this problem is different from the first three types of classical methods mentioned above and resembles the fourth type where it employs matrix methods for the determination of polynomial zeros. The idea is as follows: Given a monic polynomial,

$$p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_0, \qquad a_k \in \mathbf{C}, \quad k = 0, \ldots, n-1, \tag{1.1}$$

in the complex variable $x \in \mathbf{C}$, a matrix $M \in \mathbf{C}^{n \times n}$ is called a companion matrix of p, if:

$$\det(M - \lambda I) = (-1)^n p(\lambda). \tag{1.2}$$

Therefore, the problem of solving the polynomial equation $p(x) = 0$ reduces to the problem of finding the eigenvalues of the matrix M. Several matrix methods have been developed to find the matrix eigenvalues, such as the QR algorithm, Jacobi methods, Lanczos methods, etc.

Chapter 2 reviews some techniques which are used to construct companion matrices associated with a given polynomial. We expand on the work of Newbery and construct a (complex) symmetric or nonsymmetric companion matrix. We also adapt Schmeisser's technique to polynomials with complex zeros. Finally, we present a technique based on Cayley transformation to construct companion matrices [17] associated with polynomials having unimodular zeros of modulus 1.

Chapter 3 presents new algorithms [18] which are based on companion matrices introduced recently by Schmeisser [19] and Fiedler [20]. Schmeisser's matrix is tridiagonal symmetric and Fiedler's matrices are symmetric (full or bordered). Numerical results are also presented to illustrate the effectiveness of these methods and the fast convergence of Fiedler's algorithms to simple roots.

In Chapter 4, a convergent composite algorithm [21] based on a combination of Schmeisser's and Fiedler's matrices is presented. This algorithm mainly focuses on improving the accuracy of multiple zeros in the polynomial by first computing the distinct zeros of the original polynomial and then estimating their multiplicities by means of a modification to Lagouanelle's limiting formula [22]. The accuracy of the small zeros is also improved in the algorithm by using the reciprocal of the original polynomial. Numerical results are presented to demonstrate the effectiveness of the algorithm in improving the accuracy of multiple and small roots.

Chapter 5 shows how the composite algorithm [23], when integrated with the Frobenius companion matrix, can greatly improve the accuracy of multiple roots. Furthermore, a comparison is made between the conditioning of the three companion matrices, namely Frobenius', Schmeisser's and Fiedler's matrices, and between the conditioning of the corresponding matrices of their eigenvectors.

Chapter 6 summarizes the contributions in this thesis. The main weaknesses and strengths of the various methods that are investigated are also presented and followed by a summary of the composite algorithm.

Appendix A provides a description of the Fortran modules (main program and subroutines) along with their input and output arguments.

Finally, a Fortran implementation of all the algorithms is presented in Appendix B.

Chapter 2

Construction of Companion Matrices

2.1 Introduction

Given a monic polynomial of degree n,

$$p(z) = z^n + a_{n-1}z^{n-1} + \cdots + a_0, \qquad a_k \in \mathbf{C}, \quad k = 0, \ldots, n-1, \tag{2.1}$$

of a complex variable $z \in \mathbf{C}$, a matrix $M \in \mathbf{C}^{n\times n}$ such that

$$\det(M - \lambda I) = (-1)^n p(\lambda), \tag{2.2}$$

is called a *companion matrix* of $p(z)$ and $p(z)$ is a characteristic polynomial of M.

By (2.2), the zeros of $p(z)$ are the eigenvalues of M. Therefore, the problem of finding the zeros of $p(z)$ reduces to the problem of finding the n eigenvalues of the $n \times n$ matrix M. This problem is attractive since we have a stable, convergent and fast matrix method, namely the QR algorithm, to solve accurately the determinental equation (2.2), at least for the simple zeros. One could also use Jacobi's method (see [24], [25], [26], [27]) which is parallelizable, for finding the eigenvalues of (real or complex) matrices.

This chapter is concerned with the construction of companion matrices, associated with $p(z)$, which will form the basis of new polynomial zerofinding algorithms presented in the subsequent chapters.

2.2 Frobenius companion matrix

A well-known eigenvalue procedure, for finding zeros of polynomials, uses the Frobenius companion matrix C_p, directly defined in terms of the coefficients of $p(z)$ by:

$$C_p := \begin{bmatrix} -a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}. \tag{2.3}$$

Consider, for example, the monic polynomial

$$p(w) = w^n - 1, \tag{2.4}$$

whose zeros are the n^{th} roots of unity. The Frobenius companion matrix, C_p, associated with p,

$$C_p = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}, \tag{2.5}$$

satisfies (2.2). In this case, C_p is orthogonal since its columns are a permutation of the columns of the identity matrix; hence, it can be diagonalized by a unitary matrix U, $\overline{U}^t C_p U = I$, where the columns of U are the eigenvectors of C_p. Thus every eigenvalue of C_p is well conditioned (see [28], pp. 87–88) since the condition number, $\|U\|_2 \|\overline{U}^t\|_2 = 1$, of U is as small as possible. But, in general C_p is not normal and the condition number, $\kappa(X) = \|X\| \, \|X^{-1}\|$, of the matrix X of its eigenvectors, could be very large. An equivalent matrix-pencil representation for real $p(x)$ was given by J. L. Howland [29] in the form $\det(A - \lambda B) = kp(\lambda)$ with indefinite symmetric matrices A and B and a constant k.

K.-C. Toh and L. N. Trefethen [13] have compared the stability of the roots of a given polynomial, p, with the stability of the eigenvalues of its balanced Frobenius

companion matrix, C_p, used by the Matlab [12] ROOTS command, by comparing the set of pseudozeros, $Z_\epsilon(p)$, of polynomials obtained by ϵ-perturbations of the coefficients of p with the set of pseudoeigenvalues, $\Lambda_\epsilon(C_p)$, of balanced matrices, $C_p + E$, obtained by ϵ-perturbations, $\|E\| \leq \epsilon$, of C_p. Numerical tests showed that these two sets are comparable. Thus the conditioning of the polynomial zerofinding problem and the conditioning of the companion matrix eigenvalue problem are comparable. This conclusion was corroborated by a favorable comparison of ROOTS with the Jenkins-Traub (IMSL) code CPOLY [30] and the Madsen-Reid (Harwell) code PA16 [31].

2.3 Newbery's method

In 1964, A. C. R. Newbery [32] (see also [33], pp. 15–17) proposed a method for the construction of test matrices for which the inverse is known explicitly and the characteristic polynomial can be easily obtained. This procedure can be inverted to construct a matrix with a given characteristic polynomial. We first describe Newbery's construction.

Consider an $n \times n$ bordered matrix Q,

$$Q = \begin{bmatrix} s & r^T \\ c & D \end{bmatrix}, \qquad r = \begin{bmatrix} r_2 \\ \vdots \\ r_n \end{bmatrix}, \qquad c = \begin{bmatrix} c_2 \\ \vdots \\ c_n \end{bmatrix}, \qquad D = \begin{bmatrix} d_2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & d_n \end{bmatrix},$$

where s is a scalar, r and c are $(n-1)$-vectors, and D is an $(n-1) \times (n-1)$ diagonal matrix.

The inverse of Q,

$$Q^{-1} = \begin{bmatrix} s' & r'^T \\ c' & M' \end{bmatrix},$$

is also a bordered matrix, but, in general, the matrix $M' = (m'_{ij})$ is not diagonal.

We remark (see [34], pp. 163–167) that the $n \times n$ bordered matrix Q can be

inverted in $2(n-1)(n+2)+1$ operations by the following algorithm:

$$\begin{aligned} s' &= \left(s - \sum_{i=2}^{n} \frac{r_i c_i}{d_i}\right)^{-1}, \\ c'_i &= \frac{-s' c_i}{d_i}, \\ r'_i &= \frac{-s' r_i}{d_i}, \\ m'_{ij} &= \frac{(\delta_{ij} - c_i r'_j)}{d_i}, \end{aligned}$$

where $\delta_{ij} = 1$ if $i = j$ and 0 if $i \neq j$. We consider the eigenvalue problem for the matrix Q. Let λ be an eigenvalue of Q and x an associated eigenvector:

$$\lambda \in \lambda(Q), \qquad x = \begin{bmatrix} 1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where $\lambda(Q)$ denotes the spectrum of Q. Substituting λ and x in the vector equation

$$Qx = \lambda x,$$

we obtain n scalar equations:

$$\begin{aligned} s + \sum_{i=2}^{n} r_i x_i &= \lambda, \\ c_i + d_i x_i &= \lambda x_i, \qquad i = 2, 3, \ldots, n. \end{aligned}$$

Eliminating x_i from these equations, we have a single scalar equation:

$$s + \sum_{i=2}^{n} \frac{r_i c_i}{\lambda - d_i} - \lambda = 0. \tag{2.6}$$

Now, setting

$$\begin{aligned} v(\lambda) &= \prod_{i=2}^{n} (\lambda - d_i), \\ v_i(\lambda) &= v(\lambda)/(\lambda - d_i), \qquad i = 2, \ldots, n, \end{aligned}$$

and chasing the fractions from (2.6), we obtain the following equation:

$$(\lambda - s)v(\lambda) - \sum_{i=2}^{n} r_i c_i v_i'(\lambda) = 0. \tag{2.7}$$

It is easily seen that (2.7) is the characteristic equation of Q. Moreover, if the $r_i c_i > 0$ and the d_i are distinct, then Q has real and distinct eigenvalues, and these are separated by the d_i.

We prove a converse of the above result.

THEOREM 2.1 *Consider the monic polynomial of degree n,*

$$u(\lambda) = \lambda^n + a_{n-1}\lambda^{n-1} + \cdots + a_0,$$

and choose $n-1$ numbers, $d_2, \ldots, d_n$, such that

$$u(d_i) \neq 0, \qquad i = 2, \ldots, n.$$

Then, Newbery's method yields a bordered matrix Q such that

$$\det(Q - \lambda I) = (-1)^n u(\lambda). \tag{2.8}$$

If the roots of u are real and distinct, and the d_i interlace these roots, then Q is real and symmetric.

Proof. We consider the characteristic polynomial

$$q(\lambda) = (\lambda - s)\prod_{i=2}^{n}(\lambda - d_i) - \sum_{i=2}^{n}\Big[\frac{r_i c_i}{\lambda - d_i}\prod_{j=2}^{n}(\lambda - d_j)\Big] \tag{2.9}$$

of the bordered matrix

$$Q = \begin{bmatrix} s & r_2 & r_3 & \cdots & r_n \\ c_2 & d_2 & 0 & \cdots & 0 \\ c_3 & 0 & \ddots & \cdots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ c_n & 0 & \cdots & 0 & d_n \end{bmatrix}. \tag{2.10}$$

Setting

$$v(\lambda) = \prod_{i=2}^{n}(\lambda - d_i),$$

we have

$$q(\lambda) = (\lambda - s)v(\lambda) - \sum_{i=2}^{n} r_i c_i \frac{v(\lambda)}{\lambda - d_i}.$$

Now we solve the following identity in λ:

$$q(\lambda) = (-1)^n u(\lambda),$$

for the unknown parameters s, r_i and c_i. If $\lambda = d_i$, then

$$q(d_i) = -r_i c_i v'(d_i) = (-1)^n u(d_i), \qquad i = 2, \ldots, n;$$

thus

$$r_i c_i = (-1)^{n+1} \frac{u(d_i)}{v'(d_i)}.$$

From the trace of Q:

$$s + \sum_{i=2}^{n} d_i = -a_{n-1},$$

we have

$$s = -a_{n-1} - \sum_{i=2}^{n} d_i.$$

If we choose $r_i = c_i$, the matrix Q is symmetric. If the zeros of $u(\lambda)$ are real and simple and if the d_i interlace these zeros, one easily sees that $r_i c_i > 0$; thus, if we choose $r_i = c_i$, then Q is real and symmetric. □

2.4 Fiedler's method

In 1990, M. Fiedler [20] proved the following two theorems.

THEOREM 2.2 (Fiedler 2) *Let $u(x)$ be a monic polynomial of degree $n \geq 1$, and $b_1, \ldots, b_n \in \mathbf{C}$ be n distinct numbers such that $u(b_k) \neq 0$ for $k = 1, \ldots, n$. Set*

$$v(x) = \prod_{k=1}^{n}(x - b_k) \quad \textit{and} \quad B = \operatorname{diag}(b_k) \in \mathbf{C}^{n \times n},$$

and define the rank-one perturbation,

$$A = (a_{ij}) = B - \sigma\, d\, d^T \in \mathbf{C}^{n\times n}, \tag{2.11}$$

of the diagonal matrix B by

$$\begin{aligned} a_{ik} &= -\sigma d_i d_k, \qquad \text{if} \quad i \neq k, \\ a_{kk} &= b_k - \sigma d_k^2, \end{aligned}$$

where $i, k = 1, \ldots, n$, the number $\sigma \neq 0$ is fixed and d_k is a root of

$$\sigma v'(b_k) d_k^2 - u(b_k) = 0. \tag{2.12}$$

Then,

$$\det(A - \lambda I) = (-1)^n u(\lambda). \tag{2.13}$$

If the zeros of u are real and distinct, and if the numbers $b_1, \ldots, b_n$ interlace these zeros, σ can be chosen as $+1$ or -1 such that $d_i \in \mathbf{R}$ and A is real symmetric, that is, $A^T = A \in \mathbf{R}^{n\times n}$.

If we take $b_n = \infty$, then the preceding Theorem 2.2 simplifies and Fiedler obtains the next theorem which is similar to Theorem 2.1.

THEOREM 2.3 (Fiedler 3) *Let $u(x)$ be a monic polynomial of degree n, $n \geq 1$,*

$$u(x) = x^n + p\, x^{n-1} + r(x), \qquad \deg r \leq n-2, \text{or } 0, \tag{2.14}$$

and $b_1, \ldots, b_{n-1}$ be $n-1$ distinct complex numbers such that $u(b_k) \neq 0$ for $k = 1, \ldots, n-1$. Set

$$v(x) = \prod_{k=1}^{n-1} (x - b_k), \quad \text{and} \quad B = \operatorname{diag}(b_k) \in \mathbf{C}^{(n-1)\times(n-1)}. \tag{2.15}$$

Let $c = (c_k) \in \mathbf{C}^{n-1}$ be a column vector where c_k satisfy the following equation:

$$v'(b_k) c_k^2 + u(b_k) = 0, \qquad k = 1, \ldots, n-1.$$

Then, the bordered symmetric matrix $A \in \mathbf{C}^{n\times n}$,

$$A = \begin{bmatrix} B & c \\ c^T & d \end{bmatrix}, \qquad where \quad d = -p - \sum_{k=1}^{n-1} b_k, \tag{2.16}$$

has the characteristic polynomial

$$\det(A - \lambda I) = (-1)^n u(\lambda). \tag{2.17}$$

If all the zeros of $u(x)$ *are real and simple and the* b_k *interlace these zeros, then* A *is real and symmetric, that is,* $A^T = A \in \mathbf{R}^{n\times n}$.

Remark 2.1 If one takes $b_1 = \infty$ in Theorem 2.2, one obtains Newbery's symmetric matrix Q of Theorem 2.1.

2.5 Jacobi companion matrices

This section deals with the construction of tridiagonal companion matrics associated with a given polynomial having real or complex zeros. Hermitian tridiagonal inifinite matrices were called Jacobi matrices in early Hilbert space theory (see [35], pp. 282ff and 530ff). Real symmetric Jacobi matrices play a role in the theory of orthogonal polynomials which satisfy a three-point relation. The determinant of a Jacobi matrix of order n can be computed recursively (see [36], Execise 5, p. 35). Of course the construction of these matrices should require only a finite number of numerical operations.

2.5.1 Real symmetric matrices

In 1993, Schmeisser[19] used the following notation and modified Euclidean algorithm to construct a Jacobi companion real matrix associated with a given monic polynomial having only real zeros (simple or multiple).

Notation 2.1 Consider the following polynomial of degree $k \geq 0$,

$$f(x) = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_0, \qquad a_k \neq 0. \tag{2.18}$$

and set

$$c(f) := a_k.$$

Then $f/c(f)$ is a monic polynomial having the same zeros as f.

Modified Euclidean Algorithm 2.1 For the monic polynomial with real coefficients:

$$u(x) = x^n + a_{n-1} x^{n-1} + \cdots + a_0, \qquad a_i \in \mathbf{R}, \quad i = 0, 1, \ldots, n-1, \tag{2.19}$$

define:

$$f_1(x) = u(x), \qquad f_2(x) = \frac{1}{n} u'(x). \tag{2.20}$$

Then, for $\nu = 1, 2, \ldots, n-1$:

(a) If $f_{\nu+1}(x) \not\equiv 1$, divide f_ν by $f_{\nu+1}$ with remainder $-r_\nu$:

$$f_\nu(x) = q_\nu(x) f_{\nu+1}(x) - r_\nu(x).$$

(i) If $r_\nu(x) = 0$, set

$$c_\nu := c(r_\nu), \quad f_{\nu+2}(x) = \frac{r_\nu(x)}{c_\nu}.$$

(ii) Else, set

$$c_\nu := 0, \quad f_{\nu+2}(x) = \frac{f'_{\nu+1}(x)}{c(f'_{\nu+1})}.$$

(b) Else, stop and set

$$q_\nu(x) = f_\nu(x). \quad \square$$

LEMMA 2.1 *Let f and g be monic polynomials of degrees $k+1$ and k, respectively, whose zeros are all real and distinct and separate each other. Then the division transformation*

$$f(x) = q(x)g(x) - r(x), \tag{2.21}$$

yields a polynomial r of degree $k-1$ with $c(r) > 0$ whose zeros are all real and distinct and separate those of g. Furthermore, q is of the form $x - a$ where $a \in \mathbf{R}$.

A converse of the first Lemma (2.1) gives:

LEMMA 2.2 *Let g and h be monic polynomials of degrees k and $k-1$, respectively, whose zeros are all real and distinct and separate each other. For $c > 0$ and $a \in \mathbf{R}$ define*

$$f(x) := (x - a)g(x) - ch(x). \tag{2.22}$$

Then f is a monic polynomial of degree $k+1$ with distinct real zeros which are separated by those of g.

Using the above two Lemmas (2.1, 2.2), Schmeisser proved the following theorem:

THEOREM 2.4 (Schmeisser) *The polynomial* (2.19) *of degree n has only real zeros if and only if the Modified Euclidean Algorithm* 2.1 *yields $n-1$ nonnegative numbers $c_1, \ldots, c_n$; in this case the algorithm yields an $n \times n$ symmetric tridiagonal companion matrix T,*

$$T := \begin{bmatrix} -q_1(0) & \sqrt{c_1} & 0 & 0 & 0 & 0 \\ \sqrt{c_1} & -q_2(0) & \sqrt{c_2} & 0 & 0 & 0 \\ \ddots & \ddots & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & \sqrt{c_{n-2}} & -q_{n-1}(0) & \sqrt{c_{n-1}} \\ 0 & 0 & 0 & 0 & \sqrt{c_{n-1}} & -q_n(0) \end{bmatrix}, \tag{2.23}$$

which satisfies the determinental equation (2.2) *i.e.:*

$$\det(T - \lambda I) = (-1)^n u(\lambda).$$

Moreover, the polynomial (2.19) *has n real and distinct zeros if and only if the numbers* $c_1, \ldots, c_{n-1}$ *are strictly positive.*

Remark 2.2 A block diagonal variation, $\tilde{T}$, of the matrix T is found in [37] in the form

$$\tilde{T} := \begin{bmatrix} T^{(1)} & & \\ & \ddots & \\ & & T^{(s)} \end{bmatrix}. \tag{2.24}$$

The blocks $T^{(j)}$ have simple eigenvalues, and are of the form:

$$T^{(j)} := \begin{bmatrix} -q_1^{(j)}(0) & c_1^{(j)} & 0 & 0 \\ 1 & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & c_{k_j-1}^{(j)} \\ 0 & 0 & 1 & -q_{k_j}^{(j)}(0) \end{bmatrix}, \qquad j = 1, \ldots, s, \tag{2.25}$$

where

$$d = k_1 \geq k_2 \geq \ldots \geq k_s \geq 1,$$

and

$$\sum_{j=1}^{s} k_j = n.$$

Note that d is the number of distinct zeros of $u(x)$ and these zeros are the eigenvalues of the first block $T^{(1)}$. The number, s, of blocks is the maximum multiplicity of the roots of $u(x)$.

2.5.2 Complex symmetric matrices

Schmeisser's Modified Euclidean Algorithm 2.1 can break down for polynomials with nonreal zeros. In such cases, by replacing the polynomial f_2 of (2.20) by another "numerically independent" monic polynomial of degree $n-1$, a (complex) tridiagonal symmetric Jacobi companion matrix can be constructed to satisfy the determinental equation (2.2).

In practice, a random choice of the polynomial f_2 in (2.20), as proposed in the following algorithm, will avoid, in almost all cases, a break down of the Euclidean

algorithm.

Modified Euclidean Algorithm 2.2 Given a monic polynomial,

$$u(z) = z^n + a_{n-1}z^{n-1} + \cdots + a_0, \qquad a_0 \neq 0,\ a_k \in \mathbf{C},\ k = 0, 1, \ldots, n-1, \tag{2.26}$$

define

$$f_1(z) := u(z)$$

and

$$f_2(z) := z^{n-1} + a'_{n-2}z^{n-2} + \cdots + a'_0, \tag{2.27}$$

where $a'_0, \ldots, a'_{n-2}$ are random complex numbers.

Then, for $\nu = 1, 2, \ldots, n-1$, divide f_ν by $f_{\nu+1}$ with remainder $-r_\nu$:

$$f_\nu(z) = q_\nu(z)f_{\nu+1}(z) - r_\nu(z), \tag{2.28}$$

and set

$$c_\nu := c(r_\nu), \qquad f_{\nu+2}(z) := \frac{r_\nu(z)}{c_\nu}.$$

If $f_{\nu+1}(x) = 1$ for some $\nu < n$, restart with another random choice of f_2. Else stop and set $q_\nu(x) = f_\nu(x)$. □

THEOREM 2.5 *Given a polynomial* (2.26) *and the auxiliary polynomial* (2.27), *Euclidean Algorithm* 2.2 *produces the Jacobi companion matrix* (2.23).

Proof. Since only $n-1$ restrictions, $f_\nu \not\equiv 1$, have to be satisfied, in almost all cases Algorithm 2.2 will not break down and a matrix T of dimension n can be constructed. To show inductively that T is a companion matrix, we define $T_1 := T$ and let T_k, for $k > 1$, be the submatrix obtained from T by canceling the first $k-1$ rows and columns. For $\nu = 1, \ldots, n$, denote by g_ν the characteristic polynomial of T_ν. Obviously,

$$\begin{aligned} g_n(z) &= -q_n(0) - z = -q_n(z) = -f_n(z), \\ g_{n-1}(z) &= q_{n-1}(z)f_n(z) - c_{n-1} \\ &= q_{n-1}(z)f_n(z) - c_{n-1}f_{n+1}(z) = f_{n-1}(z). \end{aligned} \tag{2.29}$$

Expanding $\det(T_\nu - zI)$, where I is the identity matrix of the same dimension as T_ν, with respect to the first row, we find

$$g_\nu(z) = [-q_\nu(0) - z]g_{\nu+1}(z) - c_\nu g_{\nu+2}(z) = -q_\nu(z)g_{\nu+1}(z) - c_\nu g_{\nu+2}(z). \qquad (2.30)$$

Comparing (2.29) and (2.30) with (2.28) in the form

$$f_\nu(z) = q_\nu(z)f_{\nu+1}(z) - c_\nu f_{\nu+2}(z),$$

we conclude that

$$f_\nu(z) = (-1)^{n-\nu+1}g_\nu(z),$$

and hence

$$u(z) = f_1(z) = (-1)^n \det(T - zI). \qquad \square$$

2.5.3 A numerical example

It is easy to see that the modified Euclidean Algorithm 2.1 fails for the pair of polynomials

$$f_1(z) = z^4 - 1, \qquad f_2(z) = \frac{1}{4}f_1'(z) = z^3.$$

However, if f_2 is replaced by

$$f_2(z) = z^3 + \sqrt{2}\,z^2 + \sqrt{3}\,z + \sqrt{5},$$

Euclidean Algorithm 2.2 produces a 4×4 matrix, shown to three rounded decimal places,

$$T = \begin{bmatrix} 1.414 & 0.518i & & \\ 0.518i & -0.618 & 2.613 & \\ & 2.613 & -0.394 & 2.813 \\ & & 2.813 & -0.402 \end{bmatrix}, \qquad (2.31)$$

with the correct eigenvalues, $\pm 1, \pm i$, in symbolic computation and to machine precision in floating point arithmetic.

In practice, the coefficients a_k' of f_2 can be obtained by a pseudorandom number generator. Numerical tests indicate that, if sufficiently high precision arithmetic is

used in Algorithms 2.1 and 2.2, then a lower precision use of the QR algorithm, without any rescaling of T, still produces the zeros of the corresponding polynomial to a remarkably high accuracy.

2.6 Jacobi unitary matrices

2.6.1 Tridiagonal unitary matrices

In 1994, P. Dörfler and G. Schmeisser [38] have used a third modified Euclidean algorithm to construct a unitary companion tridiagonal matrix associated with a monic polynomial with unimodular zeros.

Consider a monic polynomial of degree $n \geq 0$,

$$f(z) = z^n + a_{n-1}z^{n-1} + \cdots + a_0, \qquad a_0, \ldots, a_{n-1} \in \mathbf{C}, \tag{2.32}$$

of the complex variable $z \in \mathbf{C}$. Without loss of generality, we assume that $f(1) \neq 0$; otherwise, by a finite number of rational operations, we write

$$f(z) = (z-1)^k g(z),$$

where g is a monic polynomial of degree $n-k$ such as $g(1) \neq 0$. We consider the third modified Euclidean algorithm. We let $\deg f$ stand for the degree of polynomial f.

Modified Euclidean Algorithm 2.3 Given a monic polynomial (2.32) such that $f(1) \neq 0$. Define:

$$f_1(z) := \frac{f(z)}{f(1)}, \qquad f_2(z) := f_1(z) - \frac{1}{n}(z-1)f_1'(z), \qquad \rho_1 := \frac{2}{n}f_1'(1).$$

Then, for $\nu = 1, 2, \ldots$, unless

$$\nu = n, \quad \text{or} \quad \rho_\nu = 0, \quad \text{or} \quad \deg f_{\nu+1} < n - \nu, \tag{2.33}$$

introduce the polynomial

$$s_\nu(z) := f_\nu(z) - \{(z-1)[f_\nu'(1) - f_{\nu+1}'(1)] + 1\} f_{\nu+1}(z).$$

(*a*) If $k_\nu = 2$.

(*i*) if $s_\nu(z) \not\equiv 0$, we write

$$s_\nu(z) = (z-1)^{k_\nu} t_\nu(z), \quad \text{with} \quad t_\nu(1) \neq 0,$$

and define

$$\sigma_\nu := \frac{2\sqrt{t_\nu(1)}}{\rho_\nu}, \qquad f_{\nu+2}(z) := \frac{t_\nu(z)}{t_\nu(1)},$$

(*ii*) else, define

$$\sigma_\nu := 0, \qquad f_{\nu+2}(z) := f_{\nu+1}(z) - (z-1)\frac{f'_{\nu+1}(z)}{\deg f_{\nu+1}}, \tag{2.34}$$

and

$$\rho_{\nu+1} := 2[f'_{\nu+1}(1) - f'_{\nu+2}(1)] + \sigma_\nu^2 \rho_\nu. \tag{2.35}$$

(*b*) The algorithm is terminated when $k_\nu \neq 2$ or when one of the relations in (2.33) holds. □

A necessary but not sufficient condition for the polynomial f of degree n to have unimodular zeros, is that

$$z^n f\left(z^{-1}\right) = \varepsilon \overline{f(\bar{z})}, \qquad \text{where} \quad n = \deg f \quad \text{and} \quad |\varepsilon| = 1. \tag{2.36}$$

Dörfler and Schmeisser [38] have proved the following theorem.

THEOREM 2.6 *A polynomial* (2.32) *with* $f(1) \neq 0$, *satisfying* (2.36) *has only unimodular zeros if and only if Algorithm* 2.3 *produces* n *numbers,* $\rho_1, \ldots, \rho_n$, *such that* $\Re\rho_\nu \geq 1$ *for* $\nu = 1, \ldots, n$. *In the latter case, the matrix* U *with entries:*

$$u_{\nu\nu} := 1 - 2\left[\frac{1}{\rho_\nu} + \sum_{k=\nu+1}^{n} \frac{(-1)^{\nu+k}}{\rho_k}(\sigma_\nu \cdots \sigma_{k-1})^2\right], \qquad \nu = 1, 2, \ldots, n,$$

$$u_{\mu\nu} := \begin{cases} (u_{\nu\nu} - 1)\sigma_\mu \cdots \sigma | \nu - 1, & if \quad \mu < \nu, \\ (-1)^{\mu+\nu} u_{\nu\mu}, & if \quad \mu > \nu, \end{cases}$$

is a unitary companion matrix. In addition, U *decomposes into a direct sum of matrices of lower order if and only if* f *has multiple zeros.*

2.6.2 Construction by Cayley transform

This subsection deals with the construction of unitary companion matrices [17], associated with polynomials having unimodular zeros of modulus 1. We use the fact that the Cayley transformation of a real symmetric matrix is unitary and vice versa. Consider a monic polynomial of degree n,

$$p(w) = w^n + a_{n-1}w^{n-1} + \cdots + a_0, \qquad a_k \in \mathbf{C}, \quad k = 0, 1, \ldots, n-1, \tag{2.37}$$

in the complex variable w, having unimodular zeros of modulus 1, $w_j = e^{i\theta_j}$, $j = 1, \ldots, n$. The polynomial $p(w)$ is transformed to another monic polynomial $u(z)$ having real zeros by the following procedure. The unit circle in $\mathbf{C}$, $|w| = 1$, is mapped to the real line, $z = x$, by a Möbius transformation M, for instance:

$$z = M(w) := -i\frac{w + e^{i\theta}}{w - e^{i\theta}}, \qquad p\left(e^{i\theta}\right) \neq 0.$$

The inverse transformation of M is given by:

$$w = M^{-1}(z) := e^{i\theta}\frac{z - i}{z + i}.$$

If we write

$$p(w) = p(M^{-1}(z))$$

and remove the denominators, we obtain the monic real polynomial,

$$u(x) = x^n + \alpha_{n-1}x^{n-1} + \cdots + \alpha_0,$$

having only real zeros, $x_j = M(w_j)$. A real symmetric companion matrix T is constructed by one of the Theorems 2.2, 2.3 or 2.4. Finally, the Cayley transformation of the matrix T, is given by:

$$U = e^{i\theta}(T - iI)(T + iI)^{-1}. \tag{2.38}$$

The matrix U is unitary (see [39], p. 325) and satisfy:

$$\det(U - \lambda I) = (-1)^n p(\lambda). \quad \square \tag{2.39}$$

We mention that Dörfler and Schmeisser [38] use the Cayley transformation and its inverse to prove Theorem 2.6.

2.6.3 A numerical example

Consider the following polynomial:

$$p(w) = w^4 + w^3 - w - 1, \tag{2.40}$$

having unimodular zeros:

$$1, \quad -\frac{1}{2} + \frac{\sqrt{3}}{2}i, \quad -1, \quad -\frac{1}{2} - \frac{\sqrt{3}}{2}i. \tag{2.41}$$

The following Möbius transformation ϕ and its inverse,

$$\phi : w \to z := e^{-i\pi/3}\frac{w - e^{i\pi/3}}{w - e^{-i\pi/3}}, \qquad \phi^{-1} : z \to w := e^{-i\pi/3}\frac{z - e^{i\pi/3}}{z - e^{-i\pi/3}}; \tag{2.42}$$

establish a one-to-one correspondence between the unit circle in $\mathbf{C}$ and the extended real line. Moreover, ϕ maps the unimodular zeros onto the real line to produce:

$$-1, \quad \frac{1}{2}, \quad 1, \quad 2, \tag{2.43}$$

and the ϕ^{-1} produce the polynomial

$$u(x) = x^4 - \frac{5}{2}x^3 + \frac{5}{2}x - 1. \tag{2.44}$$

If b_1, b_2, b_3 are the zeros of $u'(x)$, Theorem 2.3 produce the real symmetric matrix:

$$A = \begin{bmatrix} 1.6437 & 0 & 0 & 0.5977 \\ 0 & 0.7431 & 0 & 0.3480 \\ 0 & 0 & -0.5117 & 0.8328 \\ 0.5977 & 0.3480 & 0.8328 & 0.6250 \end{bmatrix}. \tag{2.45}$$

The Cayley transform of A,

$$U = e^{-i\pi/3}\left(A - e^{i\pi/3}I\right)\left(A - e^{-i\pi/3}I\right)^{-1}, \tag{2.46}$$

is the unitary symmetric matrix:

$$\begin{bmatrix} -0.5886 - 0.5500i & 0.1741 + 0.0614i & 0.2086 - 0.2135i & 0.0313 - 0.4763i \\ 0.1741 + 0.0614i & -0.7143 + 0.4271i & 0.0344 - 0.2750i & -0.2442 - 0.3699i \\ 0.2086 - 0.2135i & 0.0344 - 0.2750i & 0.5530 + 0.1229i & -0.7085 + 0.1064i \\ 0.0313 - 0.4763i & -0.2442 - 0.3699i & -0.7085 + 0.1064i & -0.2500 - 0.0000i \end{bmatrix}.$$

Chapter 3

Polynomial Zerofinding Matrix Algorithms

3.1 Introduction

By a polynomial zerofinding matrix algorithm, we mean the following: given a monic polynomial of degree n,

$$u(z) = z^n + a_{n-1}z^{n-1} + \cdots + a_0, \qquad a_k \in \mathbf{C}, \quad k = 0, \ldots, n-1, \tag{3.1}$$

in the complex variable $z \in \mathbf{C}$, one is to find simultaneously, by some matrix method (like the QR algorithm), the n eigenvalues of an $n \times n$ matrix A, chosen such that

$$\det(A - \lambda I) = (-1)^n u(\lambda). \tag{3.2}$$

where A is one of the companion matrices discussed in Chapter 2. The spectrum of A is denoted by $\lambda(A)$ throughout the chapter.

There is ongoing research on polynomial zerofinding algorithms. The classic books of J. H. Wilkinson [2, 28] have brought to light the fact that no universally good rootfinder exists for arbitrary polynomials. In [2], p. 38, one reads: "In numerical work, polynomials having coefficients which are more or less arbitrary are tiresome to deal with by entirely automatic procedures." In [28], p. 14, it is shown by a simple example that it is necessary to represent the coefficients of a polynomial with high

accuracy on computers and that the zeros of the determinental equation (see 3.2) can be more stable than the zeros of the explicit polynomial (see 3.1). The main purpose of this chapter is to investigate new matrix algorithms based on determinental equations discussed in Chapter 2, for solving polynomial equations. The first matrix algorithm is based on Schmeisser's tridiagonal matrix. The second and third matrix algorithms are of an iterative nature and are based on Fiedler's matrices (see Theorem 2.2 and 2.3) discussed in Chapter 2.

These algorithms aim at finding n approximations, $r_1^*, r_2^*, \ldots, r_n^*$, to the true zeros, $r_1, r_2, \ldots, r_n$, of $u(z)$ which satisfy one of the following two objectives:

1. the computed zeros r_i^* must be "accurate" approximations to the true zeros, r_i, of the polynomial;

2. the computed zeros r_i^* are such that the coefficients of the reconstructed polynomial from these zeros must be "accurate" approximations to the coefficients of the original polynomial.

A polynomial zerofinding algorithm is considered successful if one of the two above objectives is satisfied [40]. Typically, floating-point arithmetic of precision $2t$ was used to evaluate a given polynomial. Then precision t was used to form the matrices T and A given by (2.23) and (2.11 or 2.16), respectively, and their eigenvalues were found by means of a precision-t QR algorithm, although in some cases, different precision ratios have been used.

3.2 Sample polynomials

Several polynomials were collected from the literature and used as test examples in order to illustrate the effectiveness of the new algorithms in computing the (real or

complex) zeros of a polynomial. These polynomials are characterized as being ill-conditioned and therefore provide good test examples for testing certain properties of the algorithms such as convergence difficulties, behaviour for multiple zeros and clustered zeros, etc. The following polynomials were solved by the proposed methods.

JW20: James Wilkinson's polynomial of degree 20 [2, 28]:

$$(x-1)(x-2)\cdots(x-20);$$

TT32: Tho-Trefethen's polynomial of degree 32 [13]:

$$\prod_{k=1}^{32}[(x+2-4(k-1)/31];$$

MR05: The polynomial with a root of multiplicity five (see P_8 in [40]):

$$(x+\pi/3)^5;$$

MR10: The polynomial with a root of multiplicity ten:

$$(z-\pi/3)^{10}(z-5)(z-1)(z+1)^2;$$

MR12: The polynomial with a root of multiplicity twelve:

$$(x-1)^{12};$$

MR14: The polynomial with multiple roots of degree forteen:

$$(z-5-6i)^4(z^2-10^{-6})(z^2+10^{-6})(z-6-i)(z-6+2i)(z-8-7i)$$
$$(z-8+9i)(z-10-15i)(z-10+17i);$$

HM40: The polynomial of degree 40 [10] (see also [40]):

$$\left\{\prod_{k=1}^{19}\left[z-e^{i\pi(k-10)/20}\right]\right\}\left\{\prod_{k=20}^{40}\left[\frac{z-(9}{10)e^{i\pi(k-10)/20}}\right]\right\};$$

DUN1: The polynomial of degree 16. See [11]:

$$(x^2+x+2)^4(x^2+x+3)^4;$$

DUN2: The polynomial of degree 16. See [11]:

$$(x-1.7)^4(x+1.7)^4(x-1.3)^4(x+1.3)^4;$$

DUN3: The polynomial of degree 12. See [11]:

$$(x^6-2^6)(x+2)(x^2+3^2)(x-1)^3;$$

DUN4: The polynomial of degree 5. See [11]:

$$(x+1)(x-1.150016-3.57064i)^2(x-1.150016+3.57064i)^2;$$

BT1: The polynomial of degree 19. See [37]:

$$(x-1)(x-2)^2(x-3)^3(x-4)^4(x-0.25)^4(x-0.5)^5;$$

BT2: The polynomial of degree 17. See [37]:

$$(x-1)^3(x+1)^4(x-0.5+i)^3(x-0.5-i)^3(x-0.5-0.5i)^2(x-0.5+0.5i)^2;$$

BT3: The polynomial of degree 11. See [37]:

$$(x-1)^4(x-1.1)(x-0.9)(x-1+0.1i)(x-1-0.1i)(x-1.05)^3;$$

JTP9: The polynomial of degree 20. See [40]:

$$\left(z^{10}-10^{-20}\right)\left(z^{10}+10^{20}\right);$$

RAC1: The polynomial of degree 9:

$$(x^9-1);$$

RAC2: The polynomial of degree 10:

$$(x-4)^{10}-10;$$

P17: The polynomial of degree 17:

$$(x^5-5)^3(x-2)^2;$$

P27: The polynomial of degree 27:

$$(x^4-4)^5(x^3-6)^2(x-3);$$

SQ09: The polynomial of degree 9. See [10]:

$$(x-10-10i)(x-10-11i)(x-10-12i)(x-11-10i)(x-11-11i)(x-11-12i)$$
$$(x-12-10i)(x-12-11i)(x-12-12i);$$

The *a posteriori* error in the reconstructed polynomial u^*, with coefficients a_k^*, by means of the computed values of the zeros of (3.1), is as follows:

$$\text{Error in } u^* := \max_{k=0,\ldots,n-1} \frac{|a_k^* - a_k|}{1+|a_k|}. \tag{3.3}$$

3.3 Matrix Algorithm I

This algorithm uses the modified Euclidean algorithm (Algorithm 2.1 or 2.2) described in Section 2.5 to construct a tridiagonal symmetric matrix T. If the matrix can be constructed, its eigenvalues are then computed by either the QR algorithm or the Jacobi method. A stable divide-and-conquer algorithm is proposed in [41] as an alternative to the QR algorithm for large (real) symmetric tridiagonal eigenvalue problems. The success of Algorithm I will obviously depend on whether or not the matrix T can be constructed in the n stages of the algorithm.

Table 3.1: For a given polynomial $u(z)$ with real zeros and using Algorithm 2.1, the Table lists the precision used to evaluate u, the precision of the QR algorithm used to compute the eigenvalues of T, the number of correct digits in $\lambda(T)$, and the error in the reconstructed polynomial u^*.

Polyn. $u(z)$	Prec. in $u(z)$	Prec. of QR alg.	No. dig in $\lambda(T)$	Error u^*
JW20	32	16	15	1.E−14
TT32	32	16	15	1.E−09
MR05	32	16	15	1.E−16
MR10	32	16	15	1.E−15
MR12	16	16	16	0
BT1	32	16	15	1.E−15
DUN2	32	16	16	0

3.3.1 Polynomials with real zeros

If the polynomial is known to have only real roots, then Theorem 2.4 guarantees that matrix T can be constructed properly. Table 3.1 illustrates the effectiveness of Algorithm 2.1 for this class of polynomials.

3.3.2 Polynomials with complex zeros

If the polynomial has complex zeros, it was shown in Chapter 2 that Schmeisser's matrix can still be applied to determine zeros of polynomials as long as it can be formed properly. The presence of complex zeros in the polynomial can cause the modified Euclidean Algorithm 2.1 to break down before the $(n-1)^{st}$ stage is reached and therefore T would not be an $n \times n$ matrix. In this case, one would have to use Algorithm 2.2 in which a randomly generated polynomial must be introduced to replace the derivative. Table 3.2 illustrates the results of Algorithm 2.1 for those polynomials having complex zeros, and for which the algorithm does not break down. Table 3.3 illustrates the results of applying Algorithm 2.2 to those polynomials having complex zeros and for which Algorithm 2.1 breaks down.

Table 3.2: For a given polynomial $u(z)$ with complex zeros and using Algorithm 2.1, the Table lists the precision used to evaluate u, the precision of the QR algorithm used to compute the eigenvalues of T, the number of correct digits in $\lambda(T)$, and the error in the reconstructed polynomial u^*.

Polyn. $u(z)$	Prec. in $u(z)$	Prec. of QR alg.	No. dig in $\lambda(T)$	Error u^*
MR14	32	16	11	1.E−05
	64	32	31	1.E−27
HM40	32	16	4	1.E−03
	32	32	26	1.E−23
DUN1	32	16	15	1.E−16
DUN3	32	16	15	1.E−14
DUN4	32	16	15	1.E−15
BT2	32	16	15	1.E−14
BT3	32	16	15	1.E−15

Table 3.3: For a given polynomial $u(z)$ with complex zeros and using Algorithm 2.2, the Table lists the precision used to evaluate u, the precision of the QR algorithm used to compute the eigenvalues of T, the number of correct digits in $\lambda(T)$, and the error in the reconstructed polynomial u^*.

Polyn. $u(z)$	Prec. in $u(z)$	Prec. of QR alg.	No. dig in $\lambda(T)$	Error u^*
JTP9	32	16	1	1.E+06
	32	32	21	1.E−10
RAC1	32	16	14	1.E−14
	32	32	31	0
RAC2	32	16	10	1.E−15
	32	32	30	0
P17	32	16	6	1.E−12
	32	32	12	0

3.3.3 Schmeisser's Tolerance

The most delicate part of the modified Euclidean algorithm is how to decide numerically when the remainder obtained by dividing out two polynomials, is sufficiently close to zero. This decision can have a great impact on the accuracy of the zeros obtained by the algorithm, especially if the polynomial has multiple roots. In ([42], p. 194), one reads

> Theoretically, multiple roots can be eliminated. We determine the common divisor of the given polynomial, say $P(z)$, and its derivative, say $P'(z)$, [...]. The difficulty is that in actual computation we are not able to tell *a priori* whether a remaining polynomial or constant is negligibly small.

For the purpose of this algorithm, a polynomial remainder is considered close to zero if all its coefficients are less than a certain tolerance referred to as *Schmeisser's tolerance*. To illustrate the effects of Schmeisser's tolerance on the accuracy of polynomial zeros, we consider two sample polynomials JTP9 and MR14. The first polynomial JTP9 has two sets of equimodular zeros. The first set has ten roots of modulus 100 and the second set has ten roots of modulus 100^{-1}. When the tolerance is chosen to be less stringent than 10^{-20}, the roots of the second set converge to zero while the first set are more accurate. This polynomial requires a more stringent tolerance (i.e. 10^{-32} for a precision of 32 digits). For MR14 which contains multiple roots, the situation is reversed and one would choose a less stringent tolerance than with polynomials with no multiple roots. The effect of choosing a more stringent tolerance for polynomials with multiple roots, tend to decrease their accuracy considerably. For example in MR14, if the tolerance is chosen to be near the precision used in computation (i.e. 10^{-32} for a precision of 32 digits), the accuracy of the multiple roots drops from 15 to 5. For all the polynomials having multiple roots, the tolerance was set to 10^{-10}.

3.4 Matrix Algorithm II

The second matrix algorithm uses Theorem 2.2 of Chapter 2 to construct a full symmetric matrix. The purpose of this section is to present a numerical study of the convergence properties of this algorithm which is used iteratively to find the zeros of polynomials. As this algorithm requires initial values $b_1, \ldots, b_n$, to construct Fiedler's matrix A (see 2.11), The following three choices are considered:

1. Initial values are obtained from matrix Algorithm I;
2. Initial values are chosen equidistantly on a large circle;
3. Initial values are chosen randomly in the complex plane.

Matrix Algorithm II is summarized as follows:

1. Use initial values b_i to construct Fiedler's matrix A (see 2.11);
2. Compute the spectrum $\lambda(A)$ of the matrix A by means of the QR algorithm or any other matrix method;
3. Use the spectrum obtained in step 2 as initial values b_i for the next iteration;
4. Goto step 1.

3.4.1 The combined Schmeisser–Fiedler method

The underlying idea behind this method is to use matrix Algorithm I to obtain reasonable initial values required for the construction of Fiedler's matrix (see 2.11). Fiedler's algorithm (see Theorem 2.2) is then applied recursively, along with the QR algorithm, to determine real and complex zeros of polynomials to high accuracy. The recursion implied by the procedure requires that the the eigenvalues obtained by one iteration are used as initial values for the next iteration. Different floating-point

Table 3.4: For a given polynomial $u(z)$, the Table lists the precision used to evaluate u, the precision of the QR algorithm used to compute the eigenvalues of T and A, the number of iterations of Algorithm II, the number of correct digits in $\lambda(T)$ and $\lambda(A)$, and the error in the reconstructed polynomial u^*.

Polyn. $u(z)$	Prec. in $u(z)$	Prec. of QR alg.	No. of iter.	No. dig in $\lambda(T)$	No. dig in $\lambda(A)$	Error u^*
JW20	32	16	1	15	15	1.E−14
TT32	32	16	1	15	15	1.E−09
MR05	32	16	1	15	15	1.E−16
MR10	32	16	1	15	15	1.E−15
MR12	16	16	1	16	16	0
MR14	32	16	1	11	15	1.E−06
HM40	32	16	1	4	15	1.E−08
JTP9	32	16	1	01	13	1.E+04
	32	16	2	01	15	1.E+03
RAC1	32	16	1	14	15	1.E−15
RAC2	32	16	1	10	15	1.E−15
P17	32	16	1	6	9	1.E−12
	32	16	2	6	11	1.E−12
	32	16	3	6	12	1.E−12

arithmetic precisions were used for different steps of the algorithm. Typically, higher precision was used for the non-recursive purely algebraic steps, such as evaluating the given polynomial, and lower precision for the iterative processes, such as computing the spectrum, $\lambda(T)$ and $\lambda(A)$, of the matrices T (see 2.23) and A (see 2.11), respectively, by the EISPACK QR algorithm (for example, by the Mathematica [43] command "Eigenvalues" for floating point matrices). Consequently the elements of T and A need only be computed to lower precision. In the sequel, precision s refers to s (decimal) digits.

The precision used in different steps, the number of iterations of Algorithm II, the number of correct digits in $\lambda(T)$ and $\lambda(A)$, and the error in u^* for some of the above polynomials are listed in Table 3.4.

It seems that Algorithm II can be applied also to polynomials u with multiple roots if the number d_k given by (2.12),

$$d_k = \sqrt{\frac{u(b_k)}{\sigma v'(b_k)}}, \tag{3.4}$$

is set to zero if $v'(b_k)$ is sufficiently near zero. In fact, if b_k is a good approximation to a root of multiplicity μ of u and is a root of multiplicity $\mu - 1$ of v', then it can be practically assumed that $u(z)/v'(z)$ has a simple root at $z = b_k$.

3.4.2 Fiedler's method with initials values on a circle

This subsection investigates how matrix Algorithm II can be applied with initial values taken to be equidistant on a large circle of radius R [10]. It is observed numerically that the algorithm will always converge to zeros of polynomials as long as the radius of the circle is taken sufficiently large. This method could be used as an alternative to solve those polynomials for which the modified Euclidean Algorithm 2.1 breaks down. For example, as a second method for finding the zeros of polynomial JTP9, it was found that Algorithm II could be applied with starting values $b_1, \ldots, b_{20}$ equal to the 20^{th} roots of unity multiplied by a positive number R. It is to be noted that the iterates converge more rapidly to the roots of the larger modulus, $|z| = 100$, than to the roots of the smaller modulus, $|z| = 100^{-1}$, but, finally, they do converge to the smaller roots, provided R is taken sufficiently large, say $R > 10$; otherwise, say if $R < 1$, the method converges to the ten large roots, and to zero instead of converging to the ten small roots.

This second method is illustrated for some polynomials and the results are listed in Table 3.5.

3.4.3 Fiedler's method with random starting values

Polynomials were also solved iteratively by means of Algorithm II with the initial values b_k taken to be complex random numbers in the square of side $2R$, with lower

Table 3.5: For a given polynomial $u(z)$, the Table lists the modulus R of the starting values, the precision in the value of $u(z)$, the precision of the QR algorithm, the number of iterations of Algorithm II, the number of correct digits in $\lambda(A)$, and the error in the reconstructed polynomial u^*.

Polyn. $u(z)$	Radius R	Prec. in $u(z)$	Prec. of QR alg.	No. of iter.	No. dig in $\lambda(A)$	Error in u^*
JW20	25	32	16	01	00	1.E−02
	25	32	16	02	13	1.E−15
	25	32	16	03	15	1.E−14
TT32	05	32	16	01	01	1.E+07
	05	32	16	02	08	1.E−08
	05	32	16	03	15	1.E−09
MR05	05	32	16	02	05	1.E−16
MR10	25	64	32	02	04	1.E−37
MR12	05	64	32	02	05	1.E−37
MR14	25	32	16	03	08	1.E−04
HM40	25	32	16	04	10	1.E−08
	25	32	16	05	15	1.E−08
JTP9	25	32	16	03	10	1.E+06
	25	32	16	04	14	1.E+06
	110	32	16	05	14	1.E+06
	110	32	32	03	14	1.E−33

Table 3.6: For a given polynomial $u(z)$, the Table lists the half-length R of the square containing random starting values, the precision in the value of $u(z)$, the precision of the QR algorithm, the number of iterations of Algorithm II, the number of correct digits in $\lambda(A)$, and the error in the reconstructed polynomial u^*.

Polyn. $u(z)$	Side/2 R	Prec. in $u(z)$	Prec. of QR alg.	No. of iter.	No. dig in $\lambda(A)$	Error in u^*
JW20	20	32	16	03	15	1.E−15
TT32	04	32	16	03	15	1.E−08
MR05	02	32	16	03	07	1.E−15
MR10	10	64	32	03	06	1.E−37
MR12	02	64	64	02	05	1.E−65
MR14	14	32	16	03	08	1.E−04
HM40	02	32	16	03	15	1.E−07
JTP9	40	32	16	04	14	1.E+05
	40	32	32	03	15	1.E−33

left and upper right corners at $z = -R(1+i)$ and $z = R(1+i)$, respectively. The results are listed in Table 3.6.

3.4.4 Fast convergence to simple roots

Iteration of Algorithm II produces fast convergence to the roots of polynomials JW20 and TT32, which are all simple, as shown in Tables 3.5 and 3.6.

Table 3.7 lists the number of correct digits in the two simple roots of polynomial MR10 and the six simple roots of polynomial MR14 after the indicated number of iterations of Algorithm II with starting values for $b_1, \ldots, b_n$ obtained by matrix Algorithm I and also taken to be R times the n^{th} roots of unity, where n is the degree of the polynomial. Hence, simple roots can be deflated, with due care (see [2], pp. 55–65), from a given polynomial with both simple and multiple roots, or multiple roots can be extrapolated as shown in the following subsection.

Table 3.7: For a given polynomial $u(z)$ with simple and multiple roots and starting procedure matrix Algorithm I or b_k on a circle of radius R, the Table lists the precisions used in u and the QR algorithm, the number of iterations of Algorithm II, and the number of correct digits in the *simple* eigenvalues of A.

Polyn. $u(z)$	Alg. I or rad. R	Prec. in $u(z)$	Prec. of QR alg.	No. of iter.	No. dig. in simple $\lambda(A)$
MR10	Alg. I	32	16	01	15
	10	32	16	01	00
	10	32	16	02	06
	10	32	16	03	15
MR14	Alg. I	32	16	01	15
	25	32	16	01	00
	25	32	16	02	11
	25	32	16	03	15

3.4.5 Convergence to multiple roots

It has been noticed that the approximate multiple eigenvalues of a matrix, obtained by the QR algorithm, lie equidistantly on a circle with center at the multiple root. By applying the Hull–Mathon procedure [10], higher accuracy can be obtained for these roots. This is verified numerically for the multiple roots of polynomials MR05, MR10, MR12 and MR14, obtained by iterating Algorithm II with starting values for $b_1, \ldots, b_n$, supplied by matrix Algorithm I and on a circle of radius R, respectively. The precisions in the polynomial $u(z)$ and the QR algorithm are 32 and 16, respectively. The results are shown in Table 3.8.

It is to be noticed that, for polynomial MR05, matrix Algorithm I followed by one iteration of Algorithm II produces one of the quintuple roots to an accuracy of 10^{-15}, while the other four roots are on a circle of radius 1.9×10^{-5} with center at the average of the five roots. This explains the last two entries in line one of Table 3.8. In this case, the Hull-Mathon procedure has to be applied with care.

Table 3.8: The Table lists the multiple roots with their multiplicities, μ, the starting procedure (Algorithm I, or b_k on a circle of radius R), the number of iterations of Algorithm II, the number of correct digits in the average of the approximates to the multiple root, the smallest and largest distances of the approximates to the average, and the largest absolute deviation, from $2\pi/\mu$, in the angle between two successive approximates in radian measures.

Polyn. $u(z)$	Root & multip. μ	Alg. I or rad. R	No. of iter.	No. dig. in aver.	[min,max] modulus	Angular dev.
MR05	$\pi/3, 5$	Alg. I	1	16	$[4 \times 10^{-16}, 2 \times 10^{-5}]$	0.051
	$\pi/3, 5$	02	3	15	$[8.4, 9.1] \times 10^{-7}$	0.058
MR10	$\pi/3, 10$	Alg. I	1	16	$[1.80, 1.84] \times 10^{-3}$	0.007
	$\pi/3, 10$	10	3	16	$[6.2, 6.3] \times 10^{-4}$	0.005
	$-1, 2$	Alg. I	1	17	$[3.3, 3.3] \times 10^{-13}$	0.000
	$-1, 2$	10	2	16	$[3.42, 3.42] \times 10^{-10}$	0.000
MR12	$1, 12$	Alg. I	1	16	0	0
	$1, 12$	05	3	15	$[1.43, 1.44] \times 10^{-3}$	0.004
MR14	$5 + 6i, 4$	Alg. I	1	16	$[2.07, 2.07] \times 10^{-6}$	0.0002
	$5 + 6i, 4$	25	3	16	$[3.87, 3.89] \times 10^{-8}$	0.003

3.5 Matrix Algorithm III

The third matrix algorithm is based on Theorem 2.3 of Chapter 2, and uses Fiedler's bordered matrix A (see 2.16). This algorithm is very similar to matrix Algorithm II and both of them exhibit similar convergence properties concerning the choice of the initial values $b_1, b_2, \cdots, b_n$. As an illustration of the convergence of this algorithm, some sample polynomials are solved with the initial values taken to be equidistant on a circle of radius R. The results are listed in Table 3.9.

3.6 Eigenvalues by Jacobi's method

As an alternative to the QR algorithm, Jacobi's method (see [24], [25], [26], [27]) can be used for finding the eigenvalues of a matrix. Algorithms based on this method exist for real as well as complex matrices. One advantage of this method, apart from its stability, is that it can be easily adapted to parallelism [44]. However, for large matrices $n \times n$, the Jacobi method, which requires $O(n^4)$ operations, tends to get much slower than the QR algorithm which has a complexity of $O(n^3)$ for general matrices and $O(n^2)$ for tridiagonal matrices. Jacobi's algorithm is used in conjunction with Matrix Algorithm I to find the eigenvalues of Schmeisser's matrix, and the results are listed in Table 3.10.

Table 3.9: For a given polynomial $u(z)$, the Table lists the modulus R of the starting values, the precision in the value of $u(z)$, the precision of the QR algorithm, the number of iterations of Algorithm III, the number of correct digits in $\lambda(A)$, and the error in the reconstructed polynomial u^*.

Polyn. $u(z)$	Radius R	Prec. in $u(z)$	Prec. of QR alg.	No. of iter.	No. dig in $\lambda(A)$	Error in u^*
JW20	25	32	16	01	00	1.E−03
	25	32	16	02	14	1.E−15
	25	32	16	03	15	1.E−15
TT32	05	32	16	01	01	1.E+07
	05	32	16	02	07	1.E−08
	05	32	16	03	14	1.E−08
MR05	05	32	16	02	05	1.E−16
	05	32	16	03	07	1.E−16
MR10	25	32	16	02	02	1.E−13
	25	32	16	03	03	1.E−14
	25	64	32	02	05	0
MR14	25	32	16	03	08	1.E−04
HM40	25	32	16	04	06	1.E−07
	25	32	16	05	15	1.E−08
JTP9	25	32	16	03	12	1.E+06
	25	32	16	04	14	1.E+06
	110	32	16	05	14	1.E+06
	110	32	32	03	14	1.E−11
SQ09	20	16	16	02	05	1.E−14
	20	32	16	02	15	1.E−15

Table 3.10: For a given polynomial $u(z)$, the Table lists the precision used to evaluate u, the precision of the Jacobi's algorithm used to compute the eigenvalues of T, the number of correct digits in $\lambda(T)$.

Polyn. $u(z)$	Prec. in $u(z)$	Prec. of QR alg.	No. dig in $\lambda(T)$
JW20	32	16	15
TT32	32	16	15
MR05	32	16	15
MR10	32	16	15
BT1	32	16	15
BT2	32	16	15
DUN1	32	16	15
DUN4	32	16	15

Chapter 4

Composite Polynomial Zerofinding Matrix Algorithm

4.1 Introduction

Let $p(x)$ be a monic polynomial of degree n with distinct roots, $r_1, r_2, \ldots, r_N$, having multiplicities, $\mu_1, \mu_2, \ldots, \mu_N$, respectively, such that $\mu_1 + \cdots + \mu_N = n$; then

$$p(x) = (x - r_1)^{\mu_1}(x - r_2)^{\mu_2} \ldots (x - r_N)^{\mu_N}. \tag{4.1}$$

In this chapter, we are concerned with finding approximations to distinct zeros of polynomials $p(x)$ and estimating their multiplicities. It was shown in Chapters 2 and 3 that polynomials with complex zeros, may sometimes cause the modified Euclidean algorithm to break down and therefore Schmeisser's matrix cannot be formed properly; furthermore, if these polynomials have multiple roots, matrix Algorithms II or III of Sections 3.4 and 3.5, respectively, may not produce highly accurate results. With this in mind, it is necessary to find a general purpose algorithm which would solve polynomial equations, including those with multiple complex zeros, with high accuracy. This chapter presents a new three-stage convergent matrix algorithm which would compute simple and multiple roots of a given polynomial (with real or complex zeros) to high accuracy and determine their multiplicities. Sample numerical results are given to demonstrate the effectiveness of the algorithm. A summary of

this algorithm is outlined below.

Algorithm 4.1 (Summary of the composite algorithm)

1. Find the greatest common divisor, $\gcd(p(x), p'(x))$, of $p(x)$ and $p'(x)$ where $p'(x)$ is the derivative of $p(x)$. Reduce the polynomial $p(x)$ to a polynomial $q(x)$ having only simple roots by:

$$q(x) = \frac{p(x)}{\gcd(p(x), p'(x))}.$$

2. Compute the simple roots of $q(x)$ either by finding the eigenvalues of $T^{(1)}$ (the first block of Schmeisser's matrix given by (2.24), if it can be constructed properly), or by using matrix Algorithms II or III (see Section 3.4 and 3.5).

3. Calculate the multiplicity of each root of $p(x)$ by means of Lagouanelle's modified limiting formula (4.7).

Note that other methods which deal with multiple roots can be found, for example, in [4, 5, 10, 11, 37].

4.2 The composite algorithm

The new three-stage Algorithm 4.1 is now described in detail.

4.2.1 First stage: polynomial reduction

The first stage of the algorithm consists of reducing a polynomial with multiple roots to a polynomial with simple roots. Let $p(x)$ be a monic (complex) polynomial of degree n defined as in (4.1), and let $p'(x)$ be the derivative of $p(x)$. Let

$$g(x) = \gcd(p(x), p'(x)) \tag{4.2}$$

denote the greatest common divisor of $p(x)$ and $p'(x)$. The following algorithm is used to find $g(x)$.

Algorithm 4.2 (Euclidean Algorithm) Let $p(x)$ be a monic polynomial and set

$$f_0(x) = p(x), \qquad f_1(x) = p'(x)/n. \tag{4.3}$$

For $\nu = 0, 1, \ldots$, set:

$$f_\nu(x) = q_{\nu+1}(x) f_{\nu+1}(x) - c_\nu f_{\nu+2}(x). \tag{4.4}$$

If $f_{\nu+2}(x) = 0$, then
stop and set

$$s = \nu + 1, \qquad g(x) = f_{\nu+1}(x). \quad \square \tag{4.5}$$

The coefficients c_ν are introduced to ensure that the sequence of polynomials, $f_\nu(x)$, produced by the algorithm are monic. Note that Algorithm 4.2 is similar to Algorithm 2.1, except that it stops as soon as $g(x)$ is found, that is, only the first block, $T^{(1)}$, of Schmeisser's matrix (2.23) or (2.24) is constructed. From the algorithm it is easy to see that:

$$g(x) = (x - r_1)^{\mu_1 - 1}(x - r_2)^{\mu_2 - 1} \cdots (x - r_N)^{\mu_N - 1}.$$

Thus, the reduced polynomial,

$$q(x) = \frac{p(x)}{g(x)} = (x - r_1)(x - r_2) \cdots (x - r_N),$$

contains all the simple roots of the original polynomial $p(x)$ (see [11, 37]).

When the number, N, of distinct roots is equal to the number of iterations necessary to find $g(x)$, i.e. $N = s$, we say that Algorithm 4.2 terminates regularly; otherwise, it breaks down. A more practical definition of regular and breakdown terminations of the algorithm is given below in the second stage.

The first stage has a dual purpose: the first one is to compute $g(x)$ and the second one is to use Algorithm I to form the first block of Schmeisser's matrix (2.23) or (2.24).

The greatest common divisor of $p(x)$ and $p'(x)$ can also be found by applying some matrix operations to the Frobenius companion matrix associated with the polynomial. This method is due to Barnett [45].

4.2.2 Second stage: computing distinct roots

When performing Algorithm 4.2, it is necessary to record the degree of each polynomial $p_\nu(x)$ generated by the algorithm in order to determine which procedure to use for finding the simple roots. The following notation is due to [37].

Notation 4.1: If $\deg p_i = n - i$ for $i = 0, 1, \ldots, s$, then Algorithm 4.2 terminates regularly; otherwise it breaks down.
There are two cases to be considered:

Case (a) If $g(x) = 1$, then $p(x)$ and $p'(x)$ are coprime and $p(x)$ has simple roots. We consider the following two sub-cases:

1. If Algorithm 4.2 breaks down, then the matrix $T^{(1)}$ cannot be formed properly unless another random polynomial is introduced to replace the derivative $p'(x)$ of $p(x)$. In this case, matrix Algorithm II or III is used iteratively to compute the simple roots of $q(x)$.

2. If Algorithm 4.2 terminates regularly, then $T^{(1)}$ is formed properly and satisfies the determinental equation (2.2) for $q(x)$. The simple roots are computed by finding the eigenvalues of $T^{(1)}$. One could also use Algorithm II or III iteratively to compute these simple roots.

Case (b) If $\deg g > 0$, then $p(x)$ has multiple roots. To compute the simple roots of $q(x)$, we consider the following two sub-cases:

1. If Algorithm 4.2 breaks down, then the matrix $T^{(1)}$ cannot be formed properly and Algorithm II or III is used iteratively to find the simple roots of $q(x)$.

2. If Algorithm 4.2 terminates regularly, then the matrix $T^{(1)}$ contains the simple roots of $q(x)$. These roots can be computed by solving the determinental equation (2.2) for $q(x)$. As an alternative, one can use Algorithm II or III iteratively.

Algorithm II and III require initial values for b_k, $k = 1, \ldots, n$. The composite algorithm uses initial values on a large circle of radius R, given by (4.6) below, which encloses the n zeros, r_i, of $p(x)$.

$$|r_i| \leq R, \qquad i = 1, \ldots, n.$$

The following estimate for R is found in Henrici [3].

THEOREM 4.1 *Let $\lambda_1, \ldots, \lambda_n$ be positive numbers such that*

$$\lambda_1 + \cdots + \lambda_n \leq 1,$$

and let

$$R = \max_{1 \leq k \leq n} \lambda_k^{-1/k} |a_{n-k}/a_n|^{1/k}.$$

Then R is an inclusion radius for $p(x)$, i.e. $|r_i| \leq R$, for all $i = 1, \ldots, n$.

In particular, it follows from Theorem 4.1 with $\lambda_k = 1/2^k$ that the circle of radius R,

$$R = 2 \max_{1 \leq k \leq n} |a_{n-k}|^{1/k}, \tag{4.6}$$

with centre at the origin, contains all the zeros of $p(x)$.

For Algorithm II or III one can also choose random initial values inside a square centered at the origin with sides of length $2R$, where R is given by (4.6).

4.2.3 Third stage: computing root multiplicities

Let x_1 be a root of multiplicity $\mu_1 > 1$, of a given polynomial $u(x)$ of degree n. Then, Lagouanelle's limiting formula [22], proved in the following theorem, computes the multiplicity of x_1.

THEOREM 4.2 (Lagouanelle's limiting formula) *Let* $u(x) = (x - x_1)^{\mu_1} h(x)$ *where* $h(x_1) \neq 0$. *Then,*

$$\mu_1 = \lim_{x \to x_1} \frac{1}{[u(x)/u'(x)]'}. \tag{4.7}$$

Proof. Taking the logarithm of $|u(x)|$, we have

$$\ln |u(x)| = \mu_1 \ln |x - x_1| + \ln |h(x)|. \tag{4.8}$$

Differentiating (4.8) with respect to x, we get:

$$\frac{u'(x)}{u(x)} = \frac{\mu_1}{x - x_1} + \frac{h'(x)}{h(x)}.$$

Hence,

$$\begin{aligned} \lim_{x \to x_1} \left[\frac{u(x)}{u'(x)}\right]' &= \lim_{x \to x_1} \left[\frac{(x - x_1)h(x)}{\mu_1 h(x) + (x - x_1)h'(x)}\right]' \\ &= \frac{1}{\mu_1}. \quad \square \end{aligned} \tag{4.9}$$

While formula (4.7) gives the multiplicity of the roots, it can cause numerical difficulty since both $u(x)$ and $u'(x)$ approach zero as x approaches a multiple root. To overcome this difficulty, the following modification of formula (4.7) is proposed.

Corollary 4.1 (Lagouanelle's modified limiting formula) Let

$$g(x) = \gcd(u(x), u'(x))$$

be the greatest common divisor of $u(x)$ and $u'(x)$ and set

$$v(x) = \frac{u(x)}{g(x)}, \qquad w(x) = \frac{u'(x)}{g(x)}. \tag{4.10}$$

Then μ_1 is given by Lagouanelle's modified limiting formula:

$$\mu_1 = \lim_{x \to x_1} \frac{w(x)}{v'(x)}. \tag{4.11}$$

By (4.10), we have

$$\frac{u(x)}{u'(x)} = \frac{v(x)}{w(x)}. \tag{4.12}$$

Differentiating (4.12), we obtain

$$\left[\frac{u(x)}{u'(x)}\right]' = \frac{v'(x)w(x) - v(x)w'(x)}{w(x)^2}.$$

Therefore,

$$\lim_{x \to x_1} \left[\frac{u(x)}{u'(x)}\right]' = \lim_{x \to x_1} \frac{v'(x)}{w(x)},$$

since $v(x_1) \to 0$ as $x \to x_1$. Thus

$$\mu_1 = \lim_{x \to x_1} \frac{w(x)}{v'(x)}. \quad \square$$

In numerical computation, μ_1, given by (4.11), is rounded to the nearest (real) integer.

4.3 Improving the accuracy of small roots

In Chapter 3 it is observed numerically that the eigenvalues of A produced by iterating Algorithm II or III converge more rapidly to the large than to the small roots of a given polynomial.

Wilkinson [2, 28] has shown that large roots cannot, in general, be stably deflated. In fact, he indicated that, if a polynomial equation is deflated, the deflation should be done by dividing out first the smallest root, then the next smallest and so on. Now since Euclid's algorithm is a deflation process, and we might possibly divide out large roots, the remaining low-order polynomial would be quite distorted. For this reason, and for the purpose of our composite algorithm which attempts to simultaneously find

all the roots of a given polynomial, we present an additional stage to this algorithm for improving the accuracy of the small roots.

In the next section, the following practical criterion will be used to distinguish numerically between small and large roots.

Criterion 4.1 A root r_i is considered to be numerically small if $|r_i| < 1/100$ and large otherwise.

To accelerate the convergence to small roots, we shall make use of the reciprocal polynomial defined as follows.

Definition 4.1 Given a real or complex polynomial,

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0,$$

where $a_n a_0 \neq 0$, with roots r_i, the reciprocal polynomial, $p^*(x)$, of $p(x)$ is defined by

$$p^*(x) = x^n p(1/x),$$

with roots $s_i = 1/r_i$. We note that the small roots of $p(x)$ are the large roots of $p^*(x)$. The iterative process of the second stage can be re-applied to the reciprocal polynomial to get its large roots to high accuracy.

4.4 Error analysis

When applying Algorithm II, either by itself or in the composite algorithm, it is necessary to know just when to terminate the iteration process. It is desirable to terminate the process when all the zeros of the polynomial are known to within round-off accuracy. Various stopping criteria have been used in different methods found in the literature (see [46, 47]). Algorithm II uses the running error analysis described in [47], to test if convergence has occurred. If so, the calculated zeros

are accepted as approximations to the true zeros of the original polynomial. This criteria takes into account particular properties of the polynomial being evaluated, including its conditioning, multiple zeros and clusters of zeros. As outlined in [2], the conditioning of the polynomial is determined by the sensitivity of the zeros with respect to the changes in the coefficients of the polynomial. For this algorithm, the convergence is assumed if, for each calculated zero, the error bound in evaluating the polynomial is greater than the corresponding calculated value of the polynomial. If x_i is an approximate zero after a certain number of iterations and $u(x_i)$ is the computed polynomial at x_i, then convergence is assumed when

$$|a_i| \geq |u(x_i)|, \qquad i = 1, \ldots, n \tag{4.13}$$

where a_i is a realistic error bound for the computed $u(x_i)$. The recurrence relation which is used in the running error analysis is described as follows:
Let s_0 be the computed polynomial at a zero x; s_0 is computed by nested multiplication using the Horner scheme. Then, relation (4.13) can be rewritten as

$$|s_0| \leq 2^{-t} g_0 \tag{4.14}$$

where g_i is defined by the relations:

$$g_n = 0, \qquad g_i = |x|(g_{i+1} + |s_{i+1}|) + |s_i| \tag{4.15}$$

and t is the precision used in the computation.
A posteriori error bounds can also be obtained using a technique outlined in [47] which is based on the reconstructed polynomial from the approximate zeros. As indicated in [47], the success of this technique depends on the closeness of the reconstructed polynomial to the original polynomial.

4.5 Numerical results

The composite algorithm described in this chapter has been tested for accuracy with many polynomials found in the literature and considered to be ill-conditioned, without

Table 4.1: For a given polynomial $p(x)$, the Table lists the precisions used to evaluate p and to compute the elements of the matrix $T^{(1)}$, the number of iterations of Algorithm II, and the number of correct digits in the computed roots.

Polyn. $p(x)$	Prec. in $p(x)$	Prec. in QR	No. of iter.	Min. no. of correct dig.
JW20	32	16	0	15
TT32	32	16	0	15
MR05	32	16	0	16
MR10	32	16	0	15
HM40	32	16	0	15
DUN1	32	16	0	15

any consideration to speed or memory allocation. The results have been successful. The algorithm has been implemented in Mathematica [43] and then converted to Fortran 77 on a Sun Microsystems SPARCstation 10/30. Quadruple precision was used in all the computations except for the QR algorithm which used double precision arithmetic. Schmeisser's tolerance was set at 10^{-10}, except for JTP9 where it was set at 10^{-25}.

4.5.1 Polynomials without breakdown in the first stage

In this subsection, we consider a set of polynomials for which the greatest common divisor (4.2) was calculated in the 1^{st} stage without any breakdown of Algorithm 4.2. Therefore, the matrix $T^{(1)}$ was constructed properly. The roots of these polynomials were computed in the 2^{nd} stage by applying the QR algorithm to $T^{(1)}$. Finally, the 3^{rd} stage of the algorithm found the multiplicities of these roots by means of Lagouanelle's modified limiting formula. The results are listed in Table 4.1.

Table 4.2: For a given polynomial $p(x)$, the Table lists the precisions used to evaluate p and to compute the elements of the matrix A, the number of iterations of Algorithm II, and the number of correct digits in the computed roots.

Polyn. $p(x)$	Prec. in $p(x)$	Prec. in QR	No. of iter.	Min. no. of correct dig.
JTP9	32	16	5	15
P17	32	16	3	15
P27	32	16	3	15

Table 4.3: Calculated values and multiplicities of the roots of P17.

Roots	Multiplicity
2	2
1.379729661461215	3
$-1.116224743765316 - 0.810984747157389\ i$	3
$-1.116224743765316 + 0.810984747157389 i$	3
$0.426359913034708 + 1.312200885258394\ i$	3
$0.426359913034708 - 1.312200885258394\ i$	3

4.5.2 Polynomials with breakdown in the first stage

In this subection, another set of polynomials are considered for which Algorithm 4.2 broke down while calculating the greatest common divisor (4.2). Since the matrix $T^{(1)}$ could not be constructed, in the second stage, Algorithm II was used iteratively, with the QR algorithm, to compute the distinct roots of these polynomials. The results are listed in Table 4.2.

Tables 4.3 and 4.4 list the computed values of the roots of P17 and P27, respectively, and their multiplicities.

Table 4.4: Calculated values and multiplicities of the roots of P27.

Roots	Multiplicity
3	1
$-0.90856029641607 + 1.573672595132474i$	2
$-0.90856029641607 - 1.573672595132472i$	2
1.817120592832139	2
$1.414213562373095i$	5
1.414213562373095	5
-1.414213562373095	5
$-1.414213562373095i$	5

4.5.3 Small roots

The last polynomial, MR14, contains four small roots of modulus 1/1000. Even though Algorithm 4.2 did not break down at the 1st stage, the small roots were accurate to only 10 digits. The reciprocal polynomial was used to enhance the accuracy of these roots to 15 digits. The second stage of the algorithm plays a role of predicting the small roots and correcting them with the reciprocal of the original polynomial. The results are listed in Table 4.5.

4.5.4 Multiple zeros and clusters of zeros

The problem of multiple zeros and clusters of zeros (i.e. a set of several zeros that either coincide or are very close) is a very difficult one and polynomials containing such zeros are known to be ill-conditionned. It is well known that most algorithms break down as the distance between any zeros becomes comparable in size to errors of the corresponding approximations if any zero is multiple. Most frequently, a distinction between multiple zero and a zero cluster by numerical computation is very cumbersome or even impossible. The question of distinguishing a zero cluster from a multiple zero has been tackeled theoretically by Ostrowski [48]. As usual, in finite

Table 4.5: Calculated values and multiplicities of the roots of MR14.

Roots	Multiplicity
$10. - 17.i$	1
$10. + 15.00000000000001i$	1
$8. - 8.99999999999999i$	1
$7.999999999999996 + 7.000000000000001i$	1
$5.000000000000001 + 6.000000000000001i$	1
$6. - 2.000000000000001i$	1
$6.000000000000002 + 1.000000000000002i$	1
0.001	1
-0.001	1
$0.001i$	1
$-0.001i$	1

precision arithmetic, one must distinguish the following two cases:

1. Coefficients of the polynomial are exactly represented;

2. Coefficients of the polynomial are not exactly represented;

In the last case, the represented polynomial may have different multiplicities with respect to the exact one. A typical example would be the polynomial equation $x^n = 0$, which may be represented in a computing machine as $x^n + c = 0$. In this case, the zero of multiplicity n would "explode" into a cluster of n very close zeros. The composite algorithm would fail in this case in the determination of the multiplicities. To avoid the effects of rounding errors, numerical computations must be done in high precision arithmetic.

The purpose of this subsection is to study the behaviour of the composite algorithm for multiple zeros and clusters of zeros. As an illustration of the above problem, the following examples are taken from the literature, except the last polynomial, P28, which contains both clustered and multiple zeros. The results are illustrated in Tables 4.6, 4.7 and 4.8, respectively.

Table 4.6: Calculated values and multiplicities of the roots of JTP5.

Roots	Multiplicity
0.4000000000000003	1
0.3	2
0.1999999999999999	3
0.1000000000000001	4

Table 4.7: Calculated values and multiplicities of the roots of JTP6.

Roots	Multiplicity
1.001	1
1.000019999999999	1
0.99999	1
0.998	1
0.1000000000000001	1

JTP5: The polynomial of degree 10. See [40]:

$$(x-0.1)^4(x-0.2)^3(x-0.3)^2(x-0.4);$$

JTP6: The polynomial of degree 5. See [40]:

$$(x-0.1)(x-1.001)(x-0.998)(x-1.00002)(x-0.99999);$$

P28: The polynomial of degree 28:

$$(x-\pi/3)^{10}(x-\pi/3+0.001)^6(x-1)(x-5)(x+1)^2$$
$$(x-5-6i)^4(x-6-i)(x-6+2i)(x-8-7i)(x-8+9i);$$

Table 4.8: Calculated values and multiplicities of the roots of P28.

Roots	Multiplicity
$8.000000000000017 - 9.i$	1
$8.000000000000003 + 7.000000000000002i$	1
$4.999999999999994 + 6.i$	4
$6.000000000000005 - 2.000000000000001i$	1
$6. + 1.i$	1
4.999999999999996	1
1.048197551196596	6
1.047197551196595	10
−1.000000000000005	2
1.	1

4.5.5 Random polynomials

The composite algorithm has been tested with random polynomials under the criteria of reliability, accuracy and speed on a SUN Microsystems SPARCstation 10/30. These random polynomials are of degree 10 (R100–R109), 20 (R200–R209), and 50 (R500–R509). Each polynomial of degree $5t$ (where t=2,4 and 10) has t random random zeros in a square of side 2×10^i, for $i = -2, -1, 0, 1, 2$ centered at the origin. The performance results of this algorithm are listed in Table 4.9. As the composite algorithm uses either Algorithm I (in the first stage) or Algorithm II or III (in the second stage), these results have been obtained for both of Algorithms I and II. Time is indicated in milliseconds, and error is the maximum error observed. The number of iterations used in Algorithm II is also indicated in the last column.

One clearly sees from Table 4.9 that the complexity of Algorithms I and II is $O(n^2)$ for polynomials of degree n.

Table 4.9: Results using the composite algorithm on random polynomials of degree 10 (R100-R109), 20 (R200-R209), and 50 (R500-R509).

	Algorithm I		Algorithm II		
$u(z)$	time (ms)	error	time (ms)	error	# it.
R100	0.15 E+03	1. E−15	1.10 E+03	1. E−16	4
R101	0.17 E+03	1. E−13	1.12 E+03	1. E−16	4
R102	0.17 E+03	1. E−15	1.11 E+03	1. E−16	4
R103	0.18 E+03	1. E−15	1.12 E+03	1. E−16	4
R104	0.17 E+03	1. E−15	1.12 E+03	1. E−15	4
R105	0.17 E+03	1. E−15	1.12 E+03	1. E−16	4
R106	0.17 E+03	1. E−15	1.11 E+03	1. E−16	4
R107	0.18 E+03	1. E−14	1.12 E+03	1. E−16	4
R108	0.18 E+03	1. E−14	1.13 E+03	1. E−16	4
R109	0.18 E+03	1. E−14	1.12 E+03	1. E−16	4
R200	0.64 E+03	1. E−13	4.45 E+03	1. E−15	4
R201	0.64 E+03	1. E−15	4.38 E+03	1. E−14	4
R202	0.67 E+03	1. E−15	4.44 E+03	1. E−15	4
R203	0.64 E+03	1. E−15	4.34 E+03	1. E−15	4
R204	0.67 E+03	1. E−14	4.39 E+03	1. E−15	4
R205	0.66 E+03	1. E−15	4.37 E+03	1. E−15	4
R206	0.66 E+03	1. E−15	4.49 E+03	1. E−15	4
R207	0.67 E+03	1. E−14	4.40 E+03	1. E−15	4
R208	0.66 E+03	1. E−14	4.47 E+03	1. E−14	4
R209	0.68 E+03	1. E−15	4.48 E+03	1. E−15	4
R500	4.30 E+03	1. E−14	60.54 E+03	1. E−15	8
R501	4.11 E+03	1. E−14	60.07 E+03	1. E−16	8
R502	4.13 E+03	1. E−15	60.61 E+03	1. E−16	8
R503	4.13 E+03	1. E−13	60.85 E+03	1. E−15	8
R504	4.28 E+03	1. E−14	60.80 E+03	1. E−15	8
R505	4.27 E+03	1. E−14	60.41 E+03	1. E−14	8
R506	4.21 E+03	1. E−14	60.73 E+03	1. E−14	8
R507	4.33 E+03	1. E−13	60.92 E+03	1. E−14	8
R508	4.04 E+03	1. E−14	59.23 E+03	1. E−16	8
R509	4.60 E+03	1. E−14	61.88 E+03	1. E−15	8

Chapter 5

On the Conditioning of a Composite Polynomial Zerofinding Matrix Algorithm

5.1 Introduction

The Frobenius companion matrix C (see 2.6) associated with a given polynomial $p(x)$, is used by MATLAB (ROOTS command) to find zeros (real or complex) of a polynomial with simple or multiple roots. This matrix, when balanced, produces good approximations to the true zeros, with multiple roots being less accurate than simple ones. In Chapter 4, both Schmeisser's and Fiedler's matrices were used in the composite algorithm to compute the distinct zeros of a polynomial p. In this chapter, the Frobenius companion matrix is integrated in the composite algorithm, and it is shown how such integration can have a great effect on improving the accuracy of multiple roots. In addition, A comparison is made between the conditioning of Frobenius', Schmeisser's and Fiedler's companion matrices, and between the conditioning of their eigenvalues. The first and third stage of the composite algorithm are the same as in Chapter 4, and they are included here for completeness only.

5.2 The modified composite algorithm

The modified composite algorithm is divided into three stages.

5.2.1 Polynomial reduction

The first stage of the algorithm consists in reducing a polynomial with multiple zeros to a polynomial with simple zeros. Let $p(x)$ be a complex polynomial of degree n with distinct zeros, $r_1, r_2, \ldots, r_N$, having multiplicities, $\mu_1, \mu_2, \ldots, \mu_N$, respectively, such that $\sum_{i=1}^{N} \mu_i = n$; then

$$p(x) = (x - r_1)^{\mu_1}(x - r_2)^{\mu_2} \ldots (x - r_N)^{\mu_N}.$$

The monic greatest common divisor,

$$g(x) := \gcd(p(x), p'(x)) = (x - r_1)^{\mu_1 - 1}(x - r_2)^{\mu_2 - 1} \cdots (x - r_N)^{\mu_N - 1}, \qquad (5.1)$$

of $p(x)$ and $p'(x)$ is obtained by means of the Euclidean algorithm (4.3)–(4.4) as soon as the remainder $f_{\nu+2}$ in (4.4) is zero. Numerically, the remainder is set to zero if all its coefficients are smaller than a chosen tolerance. Then, the zeros of the reduced polynomial,

$$q(x) := \frac{p(x)}{g(x)} = (x - r_1)(x - r_2) \cdots (x - r_N),$$

are simple and coincide with the distinct zeros of the original polynomial $p(x)$ (see [11, 37]).

5.2.2 Computing distinct roots

Once the polynomial $q(x)$ is formed, the Frobenius companion matrix, C_q, associated with q, with no multiple eigenvalues, is constructed, and its eigenvalues, which coincide with the distinct zeros of $p(x)$, are computed by the QR algorithm after balancing C_q.

5.2.3 Computing root multiplicities

If x_1 is a zero of multiplicity $\mu_1 > 1$, of a given polynomial $u(x) = (x - x_1)^{\mu_1} h(x)$, where $h(x_1) \neq 0$, then, by Lagouanelle's limiting formula [22],

$$\mu_1 = \lim_{x \to x_1} \frac{1}{[u(x)/u'(x)]'}. \tag{5.2}$$

To overcome the numerical difficulty caused by the vanishing of both u and u' at a multiple zero of u, one needs only set $v(x) = u(x)/g(x)$ and $w(x) = u'(x)/g(x)$, where $g(x) = \gcd(u(x), u'(x))$, and rewrite Lagouanelle's limiting formula in the non-indeterminate form:

$$\mu_1 = \lim_{x \to x_1} \frac{w(x)}{v'(x)}. \tag{5.3}$$

In numerical computation, μ_1, given by (5.3), is rounded to the nearest (real) integer.

5.3 Numerical results

The above modified composite algorithm has been tested with all the polynomials presented in the previous chapters. Results have been extremely successful. In order to illustrate the effect of the first stage of the modified algorithm on improving the accuracy of multiple roots, Tables 5.1 and 5.2 list the computed values of the roots of P17 and P27, respectively, and their multiplicities.

As a mean of comparing the performance of the three companion matrices, namely Frobenius (C), Schmeisser's (T) and Fiedler's (A) matrices, Table 5.3 shows the accuracy obtained when applying each of them separately in the second stage of the composite algorithm. When Fiedler's matrix is used, the initial values for b_i were chosen on a circle of radius 25 centered at the origin and the number of iterations was set at five. Companion matrices are denoted by C_p, T_p and A_p when they are associated with the original polynomial p (i.e. in the absence of the first stage) and by C_q, T_q and A_q when they are associated with the reduced polynomial q after the first stage. For JW20 with $C_q = C_p$, the 16-digit QR algorithm produced complex

Table 5.1: Calculated values and multiplicities of the roots of P17, when using the Frobenius companion matrix.

Roots	Multiplicity
2	2
1.379729661461215	3
$-1.116224743765316 - 0.810984747157388i$	3
$-1.116224743765316 + 0.810984747157388i$	3
$0.426359913034708 7 + 1.312200885258395i$	3
$0.426359913034708 7 - 1.312200885258395i$	3

Table 5.2: Calculated values and multiplicities of the roots of P27, when using the Frobenius companion matrix.

Roots	Multiplicity
3	1
$-0.908560296416069 + 1.573672595132471i$	2
$-0.908560296416069 - 1.573672595132471i$	2
1.81712059283214	2
$1.414213562373095i$	5
1.414213562373095	5
-1.414213562373096	5
$-1.414213562373095i$	5

Table 5.3: For given p, the Table lists the precisions used in evaluating p and in the QR algorithm, and the minimum number of correct digits in the computed eigenvalues of the companion matrices for p and q.

Polynomial	Precision in		Min. no. of corr. dig. in eigenval. of					
$p(z)$	$p(z)$	QR	C_p	C_q	T_p	T_q	A_p	A_q
JW20	32	16	0	0	15	15	15	15
MR05	32	16	3	16	16	16	7	16
MR10	32	16	1	13	15	15	3	16
MR14	32	16	2	7	10	10	8	7
DUN1	32	16	3	14	15	15	7	15
DUN2	32	16	4	15	16	16	8	15
DUN3	32	16	6	15	15	15	11	15
DUN4	32	16	8	15	15	15	15	16
BT1	32	16	3	14	15	15	7	15
BT2	32	16	5	15	15	15	10	15
BT3	32	16	2	11	16	16	6	16
P17	32	16	6	15	–	–	11	15
P27	32	16	3	15	–	–	7	15

conjugates zeros for $x = 14$ and $x = 15$. For MR14, the low precision of the four small zeros can be improved either by using the reciprocal polynomial (see [21]) or by using the combined Schmeisser–Fiedler technique (see Chapter 3). Finally, the two polynomials P17 and P27 caused a breakdown in the Euclidean algorithm and Schmeisser's matrix could not be constructed; this explains the dashed entries in the columns of T_p and T_q in Table 5.3.

5.4 Conditioning of companion matrices

The composite algorithm was used with the companion matrices, C, T and A, for the above polynomials. Although the prime interest of this section lies in the condition number of the eigenvalues of a companion matrix, the condition number of the companion matrix itself was also computed since, in practice, these two numbers are often

Table 5.4: For p with real zeros, the Table lists the condition numbers of C_q, T_q and A_q, and of the corresponding matrices of eigenvectors, VC_q, VT_q and VA_q, respectively.

$p(x)$	$\kappa(C_q)$	$\kappa(T_q)$	$\kappa(A_q)$	$\kappa(VC_q)$	$\kappa(VT_q)$	$\kappa(VA_q)$
JW20	2.2×10^{19}	20	20	7.4×10^{13}	1	1
TT32	1.2×10^{13}	31	31	1.5×10^{9}	1	1
MR05	1	1	1	1	1	1
MR10	22.7	5	5	90.0	1	1
DUN2	9.2	3	1.3	14.8	1	1
BT1	4292	16	16	1323.8	1	1

both large or both small. The condition number of a matrix was computed by means of its singular value decomposition. It could have been estimated by an algorithm found in [49]. The results are illustrated in Tables 5.4 and 5.5 for polynomials with real and complex zeros, respectively.

5.4.1 Polynomials with only real zeros

If $q(x)$ has only real zeros, then T_q is a real symmetric tridiagonal matrix with distinct eigenvalues. Therefore it is diagonalizable by a unitary matrix VT_q, whose condition number is $\kappa(VT_q) = 1$. Since Fiedler's matrix, A_q, eventually converges to a diagonal matrix after a certain number of iterations (see [18, 21]), then finally $\kappa(VA_q) = 1$. Therefore, in this case, T_q and A_q are usually better conditioned than C_q, as seen in Table 5.4.

5.4.2 Polynomials with complex zeros

If $q(x)$ has complex zeros, then T_q is a complex symmetric tridiagonal matrix, whose condition number, in general, compares favourably with the condition number of C_q. Since A_q converges to a diagonal matrix after a certain number of iterations, its final conditioning is superior to both the conditioning of T_q and C_q. The results are shown in Table 5.5.

Table 5.5: For p with complex zeros, the Table lists the condition numbers of C_q, T_q and A_q, and of the corresponding matrices of eigenvectors, VC_q, VT_q and VA_q, respectively.

$p(x)$	$\kappa(C_q)$	$\kappa(T_q)$	$\kappa(A_q)$	$\kappa(VC_q)$	$\kappa(VT_q)$	$\kappa(VA_q)$
HM40	31.5	1.1×10^7	1.1	44.0	5.0×10^{13}	1
MR14	1.5×10^7	2.0×10^8	19724.1	1.5×10^9	5.9×10^6	1
DUN1	16.9	1.2	1.2	29.8	1	1
DUN3	1166.5	13.4	3	507.0	15.5	1
DUN4	24.4	6.5	3.8	5.6	1.8	1
BT2	25.7	109.3	1.5	13.6	23.6	1
BT3	924.6	1.7	1.2	3.0×10^7	18.2	1

Chapter 6

Conclusion

The main purpose of this research project was to present a new approach for solving polynomial equations based on matrix methods. New matrix algorithms based on companion matrices were developed and numerical results were presented to illustrate their effectiveness in finding real and complex zeros of polynomials. Some of the companion matrices which were used, including the well-known Frobenius matrix, are found in the literature, while others are constructed (see Chapter 2). All these matrices satisfy the property that their characteristic polynomial is the original polynomial of which we wish to find the zeros.

The first matrix algorithm (Algorithm I) was based on Schmeisser's tridiagonal matrix. This algorithm has the advantages that:

1. no initial values are required;

2. multiple and simple roots are determined with the same precision which is the same as that of the QR algorithm.

3. if the polynomial has only real zeros (simple or multiple), then Schmeisser's matrix can be formed properly and the algorithm computes all zeros to high accuracy.

Moreover, given the tridiagonal form of the matrix, the rate of convergence of the QR

algorithm is fast [requiring $O(n^2)$ operations]. However, the algorithm has some disadvantages related to the construction of the matrix. As noted previously in Chapters 2 and 3, the algorithm may sometimes fail to complete the construction of Schmeisser's matrix if the polynomial has complex roots. Another disadvantage of the method is that, when performing Euclid's algorithm, it is numerically difficult to decide when the remainder is sufficiently close to zero. This decision can have a great impact on the accuracy of the zeros obtained by the algorithm, especially if the polynomial has multiple roots. For the purpose of the algorithm, a polynomial remainder is considered close to zero if all its coefficients are less than a chosen tolerance. If the algorithm breaks down when the polynomial has complex roots, the problem can be remedied by replacing the derivative of the original polynomial with a randomly generated polynomial of the same degree as the derivative.

The second and third matrix algorithms (Algorithm II and III) were based on Fiedler's symmetric matrices, and both of them require initial values in order to form the matrices. The following three choices concerning the initial values were considered:

1. Initial values are obtained from matrix Algorithm I;

2. Initial values are chosen equidistantly on a large circle;

3. Initial values are chosen randomly in the complex plane.

One advantage exhibited by these two algorithms is their fast convergence to the simple roots of the polynomial. As pointed out in Chapter 3, these algorithms converge faster to the large roots than to the small ones. The main weakness exhibited by these two algorithms is their behaviour towards multiple zeros. Both of these two algorithms have the disadvantage of not providing high accuracy for multiple roots.

A fourth matrix algorithm (the composite algorithm) was based on either one of the companion matrices (i.e. Schmeisser's, Fiedler's or Frobenius's), or on a combina-

tion of them. The composite algorithm used the greatest common divisor to eliminate the multiple roots and took advantage of the fast convergence of matrix Algorithm II or III to simple roots. In addition, Lagouanelle's limiting formula was introduced in the algorithm to estimate the multiplicity of each root. Multiple zeros which had proved troublesome for most algorithms received special attention. The composite algorithm that has been produced has shown great merit as a general-purpose zero-finding algorithm, and more particularly, as an algorithm that is especially effective in the numerical calculation of multiple zeros. Multiple zeros have been computed with great accuracy in problems previously thought to be very ill-conditioned. In addition to proving this algorithm's effectiveness, it has been shown that the occurrence of multiple zeros does not automatically imply ill-conditioning. The composite algorithm has proved to be insensitive to the ill-conditioning exhibited by other zero-finding algorithms in calculating multiple zeros.

An essential component of the composite algorithm centres around finding the greatest common divisor of two polynomials. The idea of greatest common divisors goes several years back and was used strictly in integer arithmetic in order to avoid some problems that could result from round-off errors introduced in the process when using floating-point arithmetic. Since the process of finding the greatest common divisor is known to have stability problems due to round-off errors, care must be exercised in order to reduce this type of errors by using high precision arithmetic. As pointed out by Wilkinson [2],

> In attempting to devise a procedure for an automatic computer which will find zeros to a prescribed accuracy, it should be appreciated that computation of a very high precision may sometimes be necessary.

The composite algorithm used quadruple precision in calculating the greatest common divisor as well as in some other parts of the algorithm.

The composite algorithm is almost always convergent, and can solve numerically polynomial equations having simple or multiple real and complex zeros with high accuracy. A summary of the composite algorithm along with its main features is presented in the next section.

The effectiveness of these algorithms has been illustrated by presenting numerical results based on polynomials taken from the literature and considered to be ill-conditioned, as well as random polynomials with randomly generated zeros in small and large clusters.

Finally, a comparison was made between the conditioning of the companion matrices of Frobenius, Schmeisser and Fiedler, and between the conditioning of their eigenvalues. If a (reduced) polynomial has only real simple zeros, the eigenvalues of Schmeisser's matrix are better conditioned than those of Frobenius' matrix; otherwise their conditioning is comparable. Since Fiedler's matrices converge to a diagonal matrix, their eigenvalues eventually becomes very well conditioned. A companion matrix of a reduced polynomial q can be used to find the multiple zeros of a given polynomial p to higher accuracy than with the corresponding companion matrix of p.

6.1 Summary of the composite algorithm

The composite algorithm is implemented in three stages:

1. **First stage:** In this stage, the greatest common divisor, $\gcd(u(x), u'(x))$, of $u(x)$ and $u'(x)$ is found where $u'(x)$ is the derivative of $u(x)$. The original polynomial $u(x)$ is reduced to another polynomial $q(x)$ having only simple roots by:

$$q(x) = \frac{u(x)}{\gcd(u(x), u'(x))}.$$

Finally, the first block of Schmeisser's matrix is constructed.

2. **Second stage:** Compute the simple roots of $q(x)$ either by finding the eigenvalues of $T^{(1)}$ (the first block of Schmeisser's matrix), or by using matrix Algorithms II or III.

3. **Third stage:** Calculate the multiplicity of each root of $u(x)$ by means of Lagouanelle's modified limiting formula. If x_1 is a root of $u(x)$ and $\mu(x_1)$ denotes its multiplicity, then:

$$\mu(x_1) = \lim_{x \to x_1} \frac{1}{[u(x)/u'(x)]'}.$$

The main features of this algorithm are:

1. Exact information about multiplicities of the zeros can be obtained;

2. Simple and multiple zeros are computed with the same precision;

3. If the first stage succeeds in forming the first block of Schmeisser's matrix, then neither Algorithm II nor Algorithm III is required and consequently, there is no need for initial values to be introduced in the algorithm.

Bibliography

[1] J. E. McNamee, *A bibliography on roots of polynomials*, J. Comp. Appl. Math., **47** (1993), pp. 391–394.

[2] J. H. Wilkinson, *Rounding errors in algebraic processes*, Prentice-Hall, Englewood Cliffs, NJ, 1963, pp. 56, 77.

[3] P. Henrici, *Applied and computational complex analysis*, Vol. I, John Wiley and Sons Inc., New York, 1974, p. 513.

[4] M. R. Farmer and G. Loizou, *An algorithm for the total, or partial, factorization of a polynomial*, Math. Proc. Camb. Phil. Soc., **82** (1977), pp. 427–437.

[5] M. S. Petković and L. V. Stefanović, *On some iteration functions for the simultaneous computation of multiple complex polynomial zeros*, BIT, **27** (1987), pp. 111–222.

[6] K. Weierstrass, *Neuer Beweis des Satzes, dass jede ganze rationale Function einer Veränderlichen dargestellt werden kann als ein Product aus linearen Functionen derselben Veränderlichen (1891)* In Gesammelte Werke, 2^{nd} ed., **3**, (1969), Chelsea Publ., New York, pp. 251–269.

[7] E. Durand, *Solution numérique des équations algébriques, Tome I*, Massons, Paris, 1968, p. 278

[8] I. O. Kerner, *Ein Gesamtschrittfahren zur Berechnung der Nullstellen von Polynomen*, Numer. Math., **8** (1966), pp. 290–294.

[9] L. Pasquini, D. Trigiante, *A globally convergent method for simultaneously finding polynomial roots*, Math. of Comp., **44** (1985), pp. 135–149.

[10] T. E. Hull and R. Mathon, *The mathematical basis and implementation of a new polynomial rootfinder with quadratic convergence*, preprint, Department of Computer Science, University of Toronto, Ontario, Canada M5S 1A4, 1995, pp. 1–18.

[11] D. K. Dunaway, *Calculation of zeros of a real polynomial through factorization using Euclid's algorithm*, SIAM J. Numer. Anal., **11**(6) (1974), pp. 1087–1104.

[12] The MathWorks, Inc., *MATLAB User's guide*, The MathWorks Inc., Natick, MA, 1992.

[13] K.-C. Toh and L. N. Trefethen, *Pseudozeros of polynomials and pseudospectra of companion matrices*, Numer. Math., **68**(3) (1994), pp. 403–425.

[14] M. A. Jenkins and J. F. Traub, *A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration*, Numer. Math., **14** (1970), pp. 252–263.

[15] A. Edelman and H. Murakami, *Polynomial roots from companion matrix eigenvalues*, Math. Comp., to appear.

[16] S. Goedecker, *Remarks on algorithms to find roots of polynomials*, SIAM J. Sci. Comput., **15**(5) (1994) pp. 1059–1063.

[17] F. Malek and R. Vaillancourt, *Sur les matrices compagnons*, Gazette Sci. Math. Québec, (1994), to appear.

[18] F. Malek and R. Vaillancourt, *Polynomial zerofinding iterative matrix algorithms*, Comput. Math. Applic., **29**(1) (1995), pp. 1–13.

[19] G. Schmeisser, *A real symmetric tridiagonal matrix with a given characteristic polynomial*, Linear Algebra and its Applications, **193** (1993), pp. 11–18.

[20] M. Fiedler, *Expressing a polynomial as the characteristic polynomial of a symmetric matrix*, Linear Algebra and its Applications, **141** (1990), pp. 265–270.

[21] F. Malek and R. Vaillancourt, *A composite polynomial zerofinding matrix algorithm*, Comput. Math. Applic., (1995), to appear.

[22] J.L. Lagouanelle, *Sur une méthode de calcul de l'ordre de multiplicité des zéros d'un polynôme* C. R. Acad. Sci. Paris Sér. A. **262** (1966), pp. 626–627.

[23] F. Malek and R. Vaillancourt, *On the conditioning of a composite polynomial zerofinding matrix algorithm*, submitted to C. R. Acad. Sci. Canada.

[24] P. J. Eberlein, *Solution to the complex eigenproblem by a norm reducing Jacobi type method*, Numer. Math., **14** (1970), pp. 232–245.

[25] P. J. Anderson and G. Loizou, *A Jacobi type method for complex symmetric matrices*, Numer. Math., **25**(4) (1976), pp. 347–363.

[26] H. Rutishauser, *The Jacobi method for real symmetric symmetric matrices*, Numer. Math., **9** (1966), pp. 1–10.

[27] P. J. Eberlein, J. Boothroyd, *Solution to the eigenproblem by a norm-reducing Jacobi-type method*, Numer. Math., **11**(1) (1968), pp. 1–12.

[28] J. H. Wilkinson, *The algebraic eigenvalue problem*, Clarendon Press, Oxford, 1965.

[29] J. L. Howland, *On some methods for computing the roots of polynomials*, In Information Processing 1962, Proceedings of IFIP Congress 62, Munich, Aug. 27 to Sept. 1, 1962, pp. 116–121, North-Holland, Amsterdam, (1962).

[30] M. A. Jenkins and J. F. Traub, *Algorithm 419—Zeros of a complex polynomial*, Comm. ACM, **15**(2) (1972), pp. 97–99.

[31] K. Madsen and J. Reid, *Fortran subroutines for finding polynomial zeros*, In Report HL.75/1172(C.13), Computer Science and Systems Divisions, A.E.R.E. Harwell, Oxford, (February 1975).

[32] A. C. R. Newbery, *A family of test matrices*, Communications of the ACM, **7**(12) (1964), p. 724.

[33] R. T. Gregory and D. L. Karney, *A collection of matrices for testing computational algorithms*, Wiley-Interscience, New York, (1969).

[34] D. K. Faddeev and V. N. Faddeeva, *Computational methods of linear algebra*, (translated from the Russian by R. C. Williams), Freeman, San Francisco, London, (1963).

[35] M. H. Stone, *Linear transformations in Hilbert space and their applications to analysis*, Colloquium Publications, **15**, Amer. Math. Soc. New York, 1932.

[36] P. Lancaster and M. Tismenetsky, *The theory of matrices*, Second edition with applications, Academic Press, Orlando, FL, 1985.

[37] L. Brugnano and D. Trigiante, *Polynomial roots: The ultimate answer*, preprint, 1993, to appear in Linear Algebra and its Applications.

[38] P. Dörfler and G. Schmeisser, *Construction of unitary and normal companion matrices*, Linear Algebra and its Applications, **202** (1994), pp. 193–220.

[39] F. Riesz et B. Sz.-Nagy, *Functional analysis*, traduit de la 2è éd. française par L. F. Boron, F. Ungar, New York, (1955).

[40] M. A. Jenkins and J. F. Traub, *Principles for testing polynomial zerofinding programs*, ACM Trans. Math. Softw., **1**(1) (1975), pp. 26–34.

[41] Ming Gu and S. C. Eisenstat, *A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem*, SIAM J. Matrix Anal. Applic., **16**(1) (1995) pp. 171–191.

[42] E. H. Bareiss, *The numerical solution of polynomial equations and the resultant procedures*, Mathematical Methods for Digital Computers, **2**, Anthony Ralston and Hebert S. Wilf, eds., John Wiley, New York, 1967, p. 194.

[43] S. Wolfram, *Mathematica. A system for doing mathematics by computer*, 2nd ed., Addison-Wesley, Reading MA, 1992.

[44] U. Schendel, *Introduction to numerical methods for parallel computers*, (Ellis Horwood series in mathematics and its applications), John Wiley and Sons, Canada, 1984, p. 83.

[45] S. Barnett, *Greatest common divisor of two polynomials*, Linear Algebra and its Applications, **3** (1970), pp. 7–9.

[46] D. A. Adams, *A stopping criterion for polynomial root finding*, Communications of the ACM, **10**(10) (1967), pp. 655–658.

[47] G. Peters and J. H. Wilkinson, *Practical problems arising in the solution of polynomial equations*, J. Inst. Math. Appl., **8** (1971), pp. 16–35.

[48] A. M. Ostrowski, *A theorem on clusters of roots of polynomial equations*, SIAM J. Numer. Anal., **7** (1970) pp. 567–570.

[49] N. Doual and R. Vaillancourt, *Estimation du conditionnement spectral d'une matrice*, Ann. Sci. Math. Québec, **16**(1) (1992), pp. 92–107.

Appendix A

Program Description

This appendix contains a description of the main program and all the subroutines which together make up a complete system for solving polynomial equations. Most of the algorithms described in the previous chapters, were implemented in the program. These algorithms were first coded in Mathematica on a SUN workstation and then translated to FORTRAN 77 on the same workstation. The Jacobi subroutine which is used in Section (3.6) is a translation of the Algol procedure found in [24]. Futhermore, three subroutines from EISPACK were incorporated in the program; these subroutines were acquired through the NETLIB library. Quadruple precision arithmetic was used throughout in the program except in the EISPACK routines which use double precision. A description of the main program and its subroutines along with their input and output arguments is summarized below.

PROGRAM PROOTS

DESCRIPTION

PROOTS is the main program designed to solve polynomial equations (real or complex) having simple or multiple zeros. The polynomial will be entered either in coefficient-form or in zero-form (where the zeros are entered and the coefficients are calculated from these zeros). This program calls one the algorithms previously dis-

cussed to form a companion matrix whose characteristic polynomial is either the original polynomial or the reduced polynomial in the case of the composite algorithm. Either the QR algorithm, represented by the three EISPACK routines (CBAL, COMHES, COMQR), or the complex Jacobi procedure are then called to obtain the eigenvalues of the matrix which are the zeros of the polynomial.

INPUTS

p: array containing the coefficients of a polynomial in coefficient-form (the coefficients are known or calculated)

np: degree of the polynomial

maxroot: maximum number of roots a polynomial can have. Its value was chosen arbitrarily.

OUTPUTS

a: companion matrix generated by one of the previously discussed algorithms

b: zeros of the polynomial *p*

SUBROUTINE GETPOLY

DESCRIPTION

This procedure is used to get the polynomial equation to be solved. The input polynomial can be presented to the program in two forms. The zero-form where the zeros of the polynomial are known, in which case the polynomial coefficients are obtained from these zeros. The coefficient-form where the polynomial coefficients are read in from a given file. In either case, the output will be an array containing the coefficients of the polynomial.

INPUTS

roots: array containing roots of the input polynomial

nr: number of roots of the input polynomial

filename: name of the file containing the coefficients of the input polynomial

OUTPUTS

p: array containing the coefficients of the input polynomial

np: degree of the input polynomial

SUBROUTINE SETINIT

DESCRIPTION

This procedure determines the initial values required for the construction of Fiedler's matrix. These initial values are taken to be either equidistant on a circle of radius R or generated randomly in a square region of radius $R = \text{Side}/2$ in the complex plane.

INPUTS

np: degree of the input polynomial

radius: radius of a circle containing the initial values, $b_i = \text{radius}\, e^{2\pi i/np}$

side: side of a square enclosing the initial values, $b_i, i = 1, \ldots, np$; $b_i = (\alpha, \beta)$, where $-side/2 \leq \alpha, \beta \leq side/2$

procno: identifier of the method to be chosen for the initial values (circle or square)

OUTPUTS

b_i: array containing the initial values required for Fiedler's matrix

SUBROUTINE FORMPOLY

DESCRIPTION

This procedure constructs a polynomial from its roots. If $r_i, i = 1, \ldots, np$, are the roots of a polynomial p of degree np, the procedure forms $p = \prod_{i=1}^{n} p(x - r_i)$.

INPUTS

root: array containing the roots of the input polynomial

nr: number of elements in *root*

OUTPUTS

p: array containing the coefficients of the input polynomial

np: degree of polynomial *p*

SUBROUTINE POLYMULT

DESCRIPTION

This procedure multiplies two polynomials. If u and v are two polynomials in coefficient-form, the procedure yields another polynomial $uv = u \cdot v$.

INPUTS

u: array of size $nu + 1$ containing the coefficients of the first polynomial

nu: degree of the first polynomial

v: array of size $nv + 1$ containing the coefficients of the second polynomial

nv: degree of the second polynomial

OUTPUTS

uv: array of size $nuv + 1$ containing the coefficients of the resulting polynomial

nuv: degree of the resulting polynomial

SUBROUTINE READCOEF

DESCRIPTION

This procedure reads the coefficients of a polynomial from a file. The coefficients are sorted from highest to lowest exponent.

INPUTS

filename: name of the file which contains the coefficients of the input polynomial

OUTPUTS

p: array of size $np + 1$ containing the coefficients of the input polynomial

np: degree of polynomial *p*

SUBROUTINE EIGENV

DESCRIPTION

This procedure computes the eigenvalues of a matrix constructed by one of the algorithms discussed previously. The two methods which are used for computing the eigenvalues are the QR and the Jacobi methods.

INPUTS

np: degree of the input polynomial p

A: companion matrix of order $np \times np$ associated with the polynomial of degree np

method: matrix method to be used for the calculation of the eigenvalues (i.e. QR or Jacobi)

OUTPUTS

b: eigenvalues of the matrix A which are the zeros of the polynomial p

SUBROUTINE SCMSUB

DESCRIPTION

Given a polynomial p of degree np, this procedure uses the modified Euclidean algorithm to form a tridiagonal symmetric matrix t. When using Euclid's algorithm, it is important to set to zero a remainder when all its coefficients are smaller than a chosen tolerance *stol*.

INPUTS

p: array containing the coefficients of the input polynomial

np: degree of polynomial p

stol: Schmeisser's tolerance used to determine when a remainder is sufficiently close to zero

OUTPUTS

t: tridiagonal symmetrix matrix of order $np \times np$. This procedure has not been optimized, and one could only pass the diagonal and codiagonal elements of t

SUBROUTINE POLYCOPY

DESCRIPTION

This procedure copies the coefficients of one polynomial multiplied by a complex number into another polynomial.

INPUTS

$p1$: array containing the coefficients of the first polynomial

$np1$: degree of polynomial $p1$

factor: a complex number to be multiplied by all the elements of $p1$

OUTPUTS

$p2$: array containing the coefficients of the output polynomial

$np2$: degree of polynomial $p2$

SUBROUTINE POLYDIV

DESCRIPTION

This procedure divides two polynomials u and v and yields two other polynomials q and rem where q is the quotient polynomial and rem is the remainder polynomial, according to $u = v \cdot q + rem$.

INPUTS

u: array containing the coefficients of the first polynomial

nu: degree of u

v: array containing the coefficients of the second polynomial

nv: degree of polynomial v

OUTPUTS

q: array containing the coefficients of the quotient polynomial

nq: degree of the quotient polynomial

rem: array containing coefficients of the remainder polynomial

$nrem$: degree of rem

SUBROUTINE POLYDERIV

DESCRIPTION

This procedure finds the derivative of a polynomial.

INPUTS

u: array containing the coefficients of the input polynomial

nu: degree of *u*

OUTPUTS

du: array containing the coefficients of the differentiated polynomial

SUBROUTINE POLYADJUST

DESCRIPTION

This procedure sets to zero all the coefficients of a polynomial which are smaller than a chosen tolerance. It also checks whether a polynomial is zero or not by checking all its coefficients.

INPUTS

p: array containing the coefficients of the polynomial to be adjusted

np: degree of *p*

stol: chosen tolerance

OUTPUTS

p: array containing the coefficients of the adjusted polynomial

np: degree of *p*

polzero: Boolean flag which is set to TRUE if *p* is zero and FALSE otherwise

SUBROUTINE FIED2SUB

DESCRIPTION

This procedure implements Fiedler's method (Theorem 2.2) for the construction of a companion symmetric matrix associated with a given polynomial. This algorithm

is applied recursively until all the off-diagonal elements of the matrix are sufficiently close to zero, or close to the corresponding elements of the previous matrix. As mentioned in Chapter 3, the convergence is very fast for polynomials with simple zeros. In order to accelerate the convergence to multiple zeros, it was noticed that one could set to zero all the off-diagonal elements of the same column and row of the corresponding multiple root. A zero is considered multiple if the derivative of the auxilliary polynomial, $e(x)$, when evaluated at a multiple root, b_k, is smaller than a chosen tolerance referred to as Fiedler's tolerance ($ftol$).

INPUTS

p: array containing the coefficients of an input polynomial

np: degree of polynomial p

$ftol$: Fiedler's tolerance

b: array containing the initial values required for the construction of Fiedler's full matrix. These initial values are taken to be equidistant on a circle or randomly generated inside a square centered at the origin

OUTPUTS a: $np \times np$ complex symmetric matrix associatred with p

SUBROUTINE POLYEVAL

DESCRIPTION

This procedure evaluates a polynomial at a certain value.

INPUTS

u: array containing the coefficients of the input polynomial

nu: degree of u

v: value at which polynomial u is to be evaluated

OUTPUTS

pv: value of the polynomial at v (i.e. $pv = p(v)$)

SUBROUTINE FIED3SUB

DESCRIPTION

This procedure implements Fiedler's method (Theorem 2.3) for the construction of a bordered symmetric matrix associated with a given polynomial. This algorithm is applied recursively until all the off-diagonal elements of the matrix are sufficiently close to zero, or close to the corresponding elements of the previous matrix. For convergence properties see FIED2SUB and Chapter 3.

INPUTS

p: array containing the coefficients of the input polynomial

np: degree of polynomial p

$ftol$: Fiedler's tolerance

b: array containing the initial values required for the construction of Fiedler's bordered matrix. These initial values are taken to be equidistant on a circle or randomly generated inside a square centered at the origin

OUTPUTS

a: $np \times np$ complex symmetric matrix associated with p

SUBROUTINE COMPASUB

DESCRIPTION

This procedure constructs a Frobenius companion matrix associated with a given polynomial.

INPUTS

p: array containing the coefficients of the input polynomial

np: degree of polynomial p

OUTPUTS

c: $np \times np$ Frobenius matrix associated with p

SUBROUTINE COMPALG

DESCRIPTION

This procedure implements the three stages of the composite algorithm discussed in Chapter 4. For a polynomial with multiple zeros, it forms a reduced polynomial having simple zeros. Then, in the second stage, it computes the simple zeros by one of the algorithms discussed in Chapter 3. Finally, the multiplicities of zeros are estimated by the modified Lagouanelle's formula.

INPUTS

p: array containing the coefficients of the input polynomial

np: degree of polynomial p

scmtol: Schmeisser's tolerance required to determine when a remainder is to be set to zero

Fiedtol: Fiedler's tolerance required to accelarate convergence to multiple roots

OUTPUTS

b: array containig the simple zeros of the polynomial p

m: array containing the multiplicity of each simple zero

SUBROUTINE MULTIPLIC

DESCRIPTION

This procedure is based on the modified Lagouanelle's limiting formula and is used to estimate the multiplicities of the distinct zeros in a polynomial.

INPUTS

u: array containing the coefficients of the reduced polynomial $u = p/g$

nu: degree of u

v: array containing the coefficients of the reduced differentiated polynomial $v = dp/g$

nv: degree of v

b: array containing the distinct zeros of the polynomial p

OUTPUTS

m: array containing the estimated multiplicity of each distinct zero in b

SUBROUTINE GCD

DESCRIPTION

Given two polynomials $p1$ and $p2$, this procedure finds 1) the greatest common divisor of $p1$ and $p2$, 2) whether or not the Euclidean algorithm breaks down (see Chapter 4), 3) the first block of Schmeisser's matrix in the case where Euclid's algorithm does not break down.

INPUTS

$p1$: array containing the coefficients of the first polynomial

$np1$: degree of $p1$

$p2$: array containing the coefficients of the second polynomial

$np2$: degree of $p2$

$stol$: Schmeisser's tolerance

OUTPUTS

g: array containing the coefficients of the greatest common divisor of $p1$ and $p2$

ng: degree of g

t: first block of Schmeisser's matrix

Appendix B

Fortran Programs

```
C * ************************************************************
C * Program Proots: This program finds the roots of any polynomial
C * of degree np by computing the eigenvalues of np*np matrix
C * determined by Schmeisser or Fiedler
C *
C * p: array which contains the coefficients of the polynomial
C *    we wish to solve
C * bp: degree of the polynomial p
C * maxroot: maximum number of roots contained in a polynomial
C * b: initial values required for the construction of
C *    Fiedler's matrix
C * a: matrix which contains either Schmeisser's or
C *    Fiedler's matrix
C * ************************************************************
      PROGRAM proots

        IMPLICIT COMPLEX*32 (A-Z)
        INTEGER MAXROOT
        PARAMETER (MAXROOT=200)
        DIMENSION p(0:MAXROOT)   /201*0.0Q0/
        DIMENSION b(MAXROOT)     /200*0.0Q0/
        DIMENSION a(MAXROOT,MAXROOT) /40000*0.0Q0/

        INTEGER i,np,choice, iter, method
        REAL*16 scmtol,fiedtol

C Determine the method for computing the eigenvalues
        write(*,*) '\nComputing Eigenvalues by:'
        write(*,*) '1. Jacobi non symmetric'
```

```
      write(*,*) '2. QR'
      write(*,10)
10     FORMAT (' \nEnter method to be used: ',$)
      read(*,*) method

C Allow users to choose tolerance for Schemeisser and Fiedler
      write(*,15)
15     FORMAT('Enter Tolerance for Schmeisser:',$)
      read(*,*) scmtol

      write(*,16)
16     FORMAT('Enter Tolerance for Fiedler:',$)
      read(*,*) fiedtol

C Get the polynomial
      CALL getpoly(p,np)

C Choose the roots's solver technique

      write(*,*) '\n\nHere are the following procedures:'
      write(*,*) '1. Schmeisser'
      write(*,*) '2. Schmeisser-Fiedler2'
      write(*,*) '3. Schmeisser-Fiedler3\n'
      write(*,*) '4. Fiedler2 with initial values on a circle'
      write(*,*) '5. Fiedler2 with initial values in a square\n'
      write(*,*) '6. Fiedler3 with initial values on a circle'
      write(*,*) '7. Fiedler3 with initial values in a square\n'
      write(*,*) '8. Companion'
      write(*,*) '9. Companion-Fiedler2'
      write(*,*) '10.Companion-Fiedler3'
      write(*,*) '11.Composite Algorithm\n'

      write(*,20)
20     FORMAT (' \nSelect procedure number (1-11): ',$)
      read(*,*) choice

      iter = 1
      IF (choice .NE. 1 .AND. choice .NE. 8 .AND.
     &     choice .NE. 11) THEN
        write(*,30)
30      FORMAT (' \nEnter number of iterations for Fiedler: ',$)
        read(*,*) iter
      END IF
```

```
C Call the procedure selected

      IF (Choice .EQ. 1) THEN
       CALL scmsub(p,np,a,scmtol)
       CALL eigenv(a,np,b,method)

      ELSE IF (choice .EQ. 2) THEN
        CALL scmsub(p,np,a,scmtol)
        CALL eigenv(a,np,b,method)
        DO i=1,iter
          CALL fied2sub(p,np,b,a,fiedtol)
          CALL eigenv(a,np,b,method)
        END DO

      ELSE IF (choice .EQ. 3) THEN
        CALL scmsub(p,np,a,scmtol)
        CALL eigenv(a,np,b,method)
        DO i=1,iter
          CALL fied3sub(p,np,b,a,fiedtol)
          CALL eigenv(a,np,b,method)
        END DO

C Fiedler 2 method
      ELSE IF (choice .EQ. 4 .OR. choice .EQ. 5) THEN
        CALL setinit(choice,b,np)
        DO i=1, iter
          CALL fied2sub(p,np,b,a,fiedtol)
          CALL eigenv(a,np,b,method)
        END DO

C Fiedler 3 method
      ELSE IF (choice .EQ. 6 .OR. choice .EQ. 7) THEN
        CALL setinit(choice,b,np)
        DO i=1, iter
          CALL fied3sub(p,np,b,a,fiedtol)
          CALL eigenv(a,np,b,method)
        END DO

C Companion method
      ELSE IF (Choice .EQ. 8) THEN
       CALL compasub(p,np,a)
       CALL eigenv(a,np,b,method)

      ELSE IF (choice .EQ. 9) THEN
```

```
          CALL compasub(p,np,a)
          CALL eigenv(a,np,b,method)
          DO i=1,iter
            CALL fied2sub(p,np,b,a,fiedtol)
            CALL eigenv(a,np,b,method)
          END DO

        ELSE IF (choice .EQ. 10) THEN
          CALL compasub(p,np,a)
          CALL eigenv(a,np,b,method)
          DO i=1,iter
            CALL fied3sub(p,np,b,a,fiedtol)
            CALL eigenv(a,np,b,method)
          END DO

C Composite algorithm
        ELSE IF (choice .EQ. 11) THEN
          CALL compalg(p,np,scmtol,fiedtol)

        ELSE
          write(*,*) '\nwrong choice'
        END IF

      END

C End of proots's program


C ********************************************************
C *
C * Procedure Getpoly
C * This procedure determines the polynomial to be solved
C * with Fiedler/Schmeisser/composite algorithm
C * roots: will contain the roots of the polynomial
C * nr: will contain the number of roots in the polynomial
C *
C ********************************************************
      SUBROUTINE getpoly(p,np)
      IMPLICIT COMPLEX*32 (A-Z)
      INTEGER MAXROOT
      PARAMETER (MAXROOT=200,
     &          PI=0.31415926535897932384626433832795028Q001)
      DIMENSION p(0:MAXROOT)
      INTEGER np
```

```
C Local Variables
      DIMENSION roots(MAXROOT)
      INTEGER nr,i, polyno

      write(*,*) '\nPolynomials to be tested:'
      write(*,*) '1.  JW20'
      write(*,*) '2.  DUN2'
      write(*,*) '3.  TT32'
      write(*,*) '4.  MR05'
      write(*,*) '5.  MR10'
      write(*,*) '6.  MR12'
      write(*,*) '7.  MR14'
      write(*,*) '8.  HM40'
      write(*,*) '9.  RAC1'
      write(*,*) '10. DUN4'
      write(*,*) '11. BT1'
      write(*,*) '12. BT2'
      write(*,*) '13. rob38'
      write(*,*) '14. rob101'
      write(*,30)
30    FORMAT (' \nEnter polynomial number: ',$)
      read (*,*) polyno

C  JW20
C  zero=N[Table[k,{k,1,20}],precis];

      IF (polyno .EQ. 1) THEN
        nr=20
        DO i=1,nr
          roots(i) = i*(1.0Q0,0.0Q0)
        END DO
        CALL formpoly(roots,nr,p,np)
        RETURN
      END IF

C  Dun2
C  n=16
C  p[x_] := ((x-pre[1.7]) (x+pre[1.7])(x-pre[1.3]) (x+pre[1.3]))^4;

      IF (polyno .EQ. 2) THEN
        nr=16
        DO i=1,4
          roots(i) = 1.7Q0
        END DO
```

```
      DO i=5,8
        roots(i) = -1.7Q0
      END DO
      DO i=9,12
        roots(i) = 1.3Q0
      END DO
      DO i=13,16
        roots(i) = -1.3Q0
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C TT32
C n=32;
C zero=N[Table[-2+4(i-1)/(n-1),{i,n}],precis];

    IF (polyno .EQ. 3) THEN
      nr=32
      DO i=1,nr
        roots(i) = (-2.0Q0+(4.0Q0*(i-1))/(nr-1))*(1.0Q0,0.Q0)
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C MR05
C zero=N[{-Pi/3, -Pi/3, -Pi/3, -Pi/3, -Pi/3},precis];

    IF (polyno .EQ. 4) THEN
      nr=5
      DO i=1,nr
        roots(i) = -PI/3.0Q0
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C MR10
C zero=N[{Pi/3, Pi/3, Pi/3, Pi/3, Pi/3, Pi/3, Pi/3, Pi/3,
C   Pi/3, Pi/3,1, 5,-1,-1},precis];

    IF (polyno .EQ. 5) THEN
      nr=14
```

```
      DO i=1,10
        roots(i) = PI/3.0Q0
      END DO
      roots(11) = 1.0Q0
      roots(12) = 5.0Q0
      roots(13) = -1.0Q0
      roots(14) = -1.0Q0
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C MR12
C zero=N[Table[1.0,{i,1,12}],precis];

    IF (polyno .EQ. 6) THEN
      nr=12
      DO i=1,nr
        roots(i) = (1.0Q0,0.0Q0)
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C MR14
C zero=N[{5+6 I, 5+6 I,5+6 I, 5+6 I, 6+I, 6-2 I, 8+7 I,
C    8-9 I, 10+15 I,10-17 I,10^(-3),-10^(-3),10^(-3) I,
C    -10^(-3)I},precis];

    IF (polyno .EQ. 7) THEN
      nr=14
      roots(1) = (5.0Q0,6.0Q0)
      roots(2) = (5.0Q0,6.0Q0)
      roots(3) = (5.0Q0,6.0Q0)
      roots(4) = (5.0Q0,6.0Q0)
      roots(5) = (6.0Q0,1.0Q0)
      roots(6) = (6.0Q0,-2.0Q0)
      roots(7) = (8.0Q0,7.0Q0)
      roots(8) = (8.0Q0,-9.0Q0)
      roots(9) = (10.0Q0,15.0Q0)
      roots(10) = (10.0Q0,-17.0Q0)
      roots(11) = (0.1Q-2,0.0Q0)
      roots(12) = (-0.1Q-2,0.0Q0)
      roots(13) = (0.0Q0,0.1Q-2)
      roots(14) = (0.0Q0,-0.1Q-2)
```

```
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C  HM40
C  zero=N[Join[Table[E^(I Pi (k-10)/20),{k,1,19}],
C   Table[(9/10) E^(I Pi (k-10)/20),{k,20,40}]],precis];

    IF (polyno .EQ. 8) THEN
      nr=40
      DO i=1,19
        radians = QCMPLX(PI*(i-10)/20.0Q0)
        roots(i) = CQEXP(radians*(0.0Q0,1.0Q0))
      END DO
      DO i=20,40
        radians = QCMPLX(PI*(i-10)/20.0Q0)
        roots(i) = (9.0Q0/10.0Q0) * CQEXP(radians*(0.0Q0,1.0Q0))
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C  RAC1
C  n=9;
C  zero=N[Table[E^(I 2Pi (k-1)/n),{k,1,n}],precis];

    IF (polyno .EQ. 9) THEN
      nr=9
      DO i=1,nr
        radians = 2.0Q0*PI*(i-1)/nr
        roots(i) = CQEXP(radians*(0.0Q0,1.0Q0))
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C Dun4
C p[x_] := (x+1) (x-(1.150016+3.57064 I))^2
C        (x-(1.50016-3.57064 I))^2;
C n= 5;
    IF (polyno .EQ. 10) THEN
      nr=5
      roots(1) = -1.0Q0
      DO i=2,3
```

```
        roots(i) = (1.150016Q0,3.57064Q0)
      END DO
      DO i=4,5
        roots(i) = (1.150016Q0,-3.57064Q0)
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C BT1
C p[x_]:=(x-1) (x-2)^2 (x-3)^3 (x-4)^4 (x-1/4)^4 (x-1/2)^5;
C n=19;
    IF (polyno .EQ. 11) THEN
      nr=19
      roots(1) = 1.0Q0
      DO i=2,3
        roots(i) = (2.0Q0,0.0Q0)
      END DO
      DO i=4,6
        roots(i) = (3.0Q0,0.0Q0)
      END DO
      DO i=7,10
        roots(i) = (4.0Q0,0.0Q0)
      END DO
      DO i=11,14
        roots(i) = (0.25Q0,0.0Q0)
      END DO
      DO i=15,19
        roots(i) = (0.5Q0,0.0Q0)
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C BT2
C p[x_]:=(x-1)^3 (x+1)^4 (x-[0.5-I])^3 (x-[0.5+I])^3
C (x-[0.5-0.5 I])^2 C(x-[0.5+0.5 I])^2;
C n=17;
    IF (polyno .EQ. 12) THEN
      nr=17
      DO i=1,3
        roots(i) = (1.0Q0,0.0Q0)
      END DO
      DO i=4,7
```

```
        roots(i) = (-1.0Q0, 0.0Q0)
      END DO
      DO i=8,10
        roots(i) = (0.5Q0, -1.0Q0)
      END DO
      DO i=11,13
        roots(i) = (0.5Q0, 1.0Q0)
      END DO
      DO i=14,15
        roots(i) = (0.5Q0, -0.5Q0)
      END DO
      DO i=16,17
        roots(i) = (0.5Q0, 0.5Q0)
      END DO
      CALL formpoly(roots,nr,p,np)
      RETURN
    END IF

C  Rob38 or Rob100

    IF (polyno .GE. 13) THEN
      CALL readcoef(polyno,p,np)
      RETURN
    END IF


    END


C ***************************************************************
C * Procedure Setinit: This procedure sets the initial values
C *    for the b's used in Fiedler's method
C *
C ***************************************************************
    SUBROUTINE setinit(procno,b,np)
    INTEGER MAXROOT
    REAL*16 PI
    PARAMETER (MAXROOT=200,
   &        PI=0.31415926535897932384626433832795028Q001)

    COMPLEX*32 b(MAXROOT)
    INTEGER np, procno, seed/0/

C Local variables
```

```
      INTEGER i
      REAL*16 Radius, Side, coord1, coord2, rd

      IF (procno .EQ. 4 .OR. procno .EQ. 6 ) THEN
        write(*,40)
40    FORMAT (' \nEnter Radius of the circle: ',$)
        read(*,*) radius
        DO i=1,np
          b(i) = Radius * (QCOS(2*PI*i/np) * (1.0Q0,0.0Q0) +
     &                QSIN(2*PI*i/np) * (0.0Q0,1.0Q0))
        END DO
      ELSE IF (procno .EQ. 5 .OR. procno .EQ. 7) THEN
        write(*,50)
50    FORMAT (' \nEnter Side of the square: ',$)
        read (*,*) side
        DO i=1,np
          rd = rand(seed)
          coord1 = Side * QREAL(2*rd - 1)
          rd = rand(seed)
          coord2 = Side * QREAL(2*rd -1)
          b(i) = coord1*(1.0Q0,0.0Q0) + coord2*(0.0Q0,1.0Q0)
        END DO

      END IF

C Display the initial values for b
        write(1,*) '\nInitial values b are\n'
        DO i=1,np
          write(1,*) i,': ',b(i)
        END DO

      END

C ***************************************************************
C * Procedure Formpoly: This procedure constructs a polynomial
C * from its roots
C *
C ***************************************************************

      SUBROUTINE formpoly(root,nr,p,np)
        IMPLICIT COMPLEX*32 (A-Z)
        PARAMETER (MAXROOT=200)
        DIMENSION root(MAXROOT),p(0:MAXROOT)
        INTEGER nr,np
```

```
C Local variables
      DIMENSION u(0:MAXROOT) /1.0Q0,200*0.0Q0/
      DIMENSION v(0:1)
      DIMENSION uv(0:MAXROOT)
      INTEGER nuv
      INTEGER nu
      INTEGER nv
      INTEGER i,j

C Degree of polynomial u initially
      nu=0
C Degree of polynomial v initially
      nv=1

      DO i=1,nr

C v=x-root(i) and x is the variable of the polynomial p(x)
        v(0)= -root(i)
        v(1)= 1

        CALL polymult(u,nu,v,nv,uv,nuv)

C Copy nuv elements of product uv to u
        DO j=0,nuv
          u(j) = uv(j)
        END DO

        nu=nuv
      END DO

C Form polynomial p
      DO i=0,nu
        p(i) = u(i)
        np = nu
      END DO
    END


C ***************************************************************
C * Procedure  Polymult: This procedure multiplies two
C *    polynomials u and v
C * and yields another polynomial uv.
C *
C * u: 1st poly of degree nu
```

```
C * v: 2nd poly of degree nv
C * uv: uv=u*v poly of degree nu+nv returned by the procedure
C ****************************************************************

      SUBROUTINE polymult(u,nu,v,nv,uv,nuv)
      IMPLICIT COMPLEX*32 (A-Z)
      PARAMETER (MAXROOT=200)
      DIMENSION u(0:MAXROOT),v(0:MAXROOT),uv(0:MAXROOT)
      INTEGER nu,nv,nuv

C  Local variables
      INTEGER i,j,k

C  Initialize uv
      DO i=0,nu+nv
        uv(i) = 0.0Q0
      END DO

      DO i=0,nu+nv
        j=0
        DO WHILE (j .LE. i .AND. j .LE. nu)
          k=0
          DO WHILE (k .LE. nv)
            IF (j+k .EQ. i) THEN
              uv(i) = uv(i) + u(j) * v(k)
              k = nv+1
            ELSE
              k=k+1
            END IF
          END DO
          j=j+1
        END DO
      END DO
      nuv = nu+nv
      END


C * ****************************************************************
C * Procedure Readcoef: This procedure reads the coefficients
C *    of a polynomial from  a file passed in the argument
C *
C * Filename: is the name of the file containing the coefficients
C *
C ****************************************************************
```

```
      SUBROUTINE readcoef (polyno,p,np)
      INTEGER MAXROOT
      PARAMETER(MAXROOT=200)

      COMPLEX*32 p(0:MAXROOT)
      INTEGER polyno,np

C Local Variables
      CHARACTER*20 filename
      CHARACTER*80 Line(200)
      INTEGER count,i
      REAL*16 p_real(0:MAXROOT)

      IF (polyno .EQ. 10) THEN
        filename = 'rob38'
      ELSE IF (polyno .EQ. 11) THEN
        filename = 'rob101'
      ELSE
        write(*,*) 'Wrong polynomial choice'
        RETURN
      END IF

      OPEN(2,FILE=filename)
      READ (2,'(A)') Line(1)

      DO WHILE (line(1) .NE. 'BEGIN')
        READ (2,'(A)') Line(1)
      END DO

C Count the number of coefficients terms in the polynomial
      count = 1
      READ (2,'(A)') Line(count)
      DO WHILE (Line(count) .NE. 'END')
        count = count+1
        READ (2,'(A)') Line(count)
      END DO
      count = count -1

C Read each coefficient and forms the polynomial
      DO i=1,count
        READ(Line(i),'(F40.0)') p_real(count-i)
        p(count-i) = QCMPLX(p_real(count-i))
      END DO
      np = count-1
```

```
C Form a monic polynomial
      write(*,*) 'Coefficients of monic polynomial are:\n'
      DO i=0,np
        p(i) = p(i)*(1.0Q0,0.0Q0)/p(np)
        write(*,*) p(i)
      END DO

C Close the input file
       CLOSE(2)

      END

C * ************************************************
C * Procedure Eigenv: This procedure finds all the
C *     eigenvalues of the matrix A
C *
C * A: complex symmetric matrix of order np.
C * np: order of the matrix which is also the degree of
C *     the polynomial
C * b: will contain the eigenvalues of A
C *
C * ************************************************
      SUBROUTINE eigenv(A,np,b,method)
      IMPLICIT REAL*8 (A-Z)
      INTEGER MAXROOT
      PARAMETER (MAXROOT=200)
C Parameters
      COMPLEX*32 A,b
      DIMENSION A(MAXROOT,MAXROOT)
      DIMENSION b(MAXROOT)
      INTEGER   np,method

C Local variables
      DIMENSION AR(MAXROOT,MAXROOT),AI(MAXROOT,MAXROOT)
      DIMENSION WR(MAXROOT),WI(MAXROOT),fv1(MAXROOT)
      DIMENSION iv1(MAXROOT)
      INTEGER iv1,is1,is2,ierr
      INTEGER i,j
      DIMENSION jvr(MAXROOT,MAXROOT),jvi(MAXROOT,MAXROOT)


C Separate the real and imaginary parts of A
      DO i=1,np
```

```
      DO j=1,np
        AR(i,j) = DREAL(DCMPLX(A(i,j)))
        AI(i,j) = DIMAG(DCMPLX(A(i,j)))
      END DO
    END DO

    IF (method .EQ. 1) THEN
C Call the Jacobi routine

      CALL DJACOMSUB(np,ar,ai,jvr,jvi)
      DO i=1,np
        wr(i) = ar(i,i)
        wi(i) = ai(i,i)
      END DO
    ELSE
C Call the Eispack routine to get the eigenvalues of the matrix

      CALL CBAL(MAXROOT,np,ar,ai,is1,is2,fv1)
      CALL COMHES(MAXROOT,np,is1,is2,ar,ai,iv1)
      CALL COMQR(MAXROOT,np,is1,is2,ar,ai,wr,wi,ierr)
      IF (ierr .NE. 0) THEN
        write(*,*) 'Error in getting the eigenvalues'
      END IF
    END IF

C Adjust the Eigenvalues
    DO i=1,np
      IF (DABS(wr(i)) .LT. 1.0D-14) THEN
        wr(i) = 0.0D0
      END IF
      IF (DABS(wi(i)) .LT. 1.0D-14) THEN
        wi(i) = 0.0D0
      END IF
    END DO

C Display the Eigenvalues
    write(*,*) '\nReal and Imaginary Eigenvalues:\n'
    DO i=1,np
      write(*,*) wr(i), '    ',wi(i)
    END DO

C Form the eigenvalues by combining type real and imaginary parts
    DO i=1,np
      b(i) = QCMPLX(wr(i))*(1.0Q0,0.0Q0) +
```

```
     &        QCMPLX(wi(i))*(0.0Q0,1.0Q0)
      END DO

      END
C  End of the eigenv routine

C ***************************************************************
C * Subroutine Scmsub: This subroutine finds all the roots of
C * a polynomial by finding the eigenvalues of a tridiagonal
C * complex symmetric matrix constructed by a modified
C * Euclidean algorithm.
C *
C * p: is an array containing the coefficients of the
C *    given polynomial
C * np: degree of the polynomial
C * t: matrix np*np computed by Shmeisser
C * stol: the tolerance of Schmeisser
C *
C ***************************************************************
      SUBROUTINE scmsub (p,np,t,stol)
        IMPLICIT COMPLEX*32  (A-Z)
        INTEGER MAXROOT
        PARAMETER (MAXROOT=200)

C Paramaters
        DIMENSION p(0:MAXROOT)
        INTEGER   np
        DIMENSION t(MAXROOT,MAXROOT)
        REAL*16   stol

C Local Variables
        DIMENSION dp(0:MAXROOT)  /201*0.0Q0/
        DIMENSION q(0:MAXROOT)   /201*0.0Q0/
        DIMENSION rem(0:MAXROOT) /201*0.0Q0/
        DIMENSION f1(0:MAXROOT)  /201*0.0Q0/
        DIMENSION f2(0:MAXROOT)  /201*0.0Q0/
        DIMENSION df2(0:MAXROOT) /201*0.0Q0/

        DIMENSION b(MAXROOT)     /200*0.0Q0/
        DIMENSION w5(MAXROOT)    /200*0.0Q0/
        DIMENSION diag(MAXROOT)  /200*0.0Q0/
        DIMENSION c(MAXROOT)     /200*0.0Q0/
```

```
      INTEGER i,j,nq,nrem,nf1,nf2,ndf2
      LOGICAL remzero /.false./,qzero /.false./


C Form f1
      CALL polycopy (p,np,f1,nf1,1.0Q0)

C Form f2
      CALL polyderiv(f1,nf1,f2,nf2)
      DO i=0,nf2
        f2(i) = f2(i)/nf1
      END DO


C Euclidean algorithm
      DO i=1,np-1
        write (*,*) 'Iteration no ',i,'\n'
        IF (IABS(nf1-nf2) .GT. 1) THEN
          WRITE (*,*) 'There are complex roots'
          CALL EXIT(1)
        END IF

        CALL polydiv(f1,nf1,f2,nf2,q,nq,rem,nrem)


C IMPORTANT  IMPORTANT
C This function polyadjust returns sometimes the wrong value
C use debugger to demonstrate that. So I have changed it into
C a subroutine instead,

        CALL polyadjust(rem,nrem,remzero,stol)
        CALL polyadjust(q,nq,qzero,stol)

        DO j=0,nrem
          rem(j) = -rem(j)
        END DO

        diag(i) = q(0)

C Do f1=f2
        CALL polycopy(f2,nf2,f1,nf1,1.0Q0)

C Check if the remainder is zero.
        IF (remzero) THEN
```

```
            CALL polyderiv(f2,nf2,df2,ndf2)
                w5(i) = df2(ndf2)
C Do f2=df2/w5(i)
            CALL polycopy(df2,ndf2,f2,nf2,1.0Q0/w5(i))
                c(i) = 0.0Q0
          ELSE
            c(i) = rem(nrem)
C Do f2=rem/c(i)
            CALL polycopy(rem,nrem,f2,nf2,1.0Q0/c(i))
          END IF
        END DO

        CALL polydiv(f1,nf1,f2,nf2,q,nq,rem,nrem)
        CALL polyadjust(rem,nrem,remzero,stol)
        CALL polyadjust(q,nq,qzero,stol)
        diag(np) = q(0)

C form the matrix t of Schmeisser

        DO i=1,np-1
          t(i,i) = -diag(i)
          t(i+1,i) = SQRT(c(i))
          t(i,i+1) = t(i+1,i)
        END DO

        t(np,np) = -diag(np)

C Display the matrix t of Schmeisser
        write(1,*) "\nSchmeisser's matrix"
        DO i=1,np
          DO j=1,np
            write(1,*) i,':',j,'  ',t(i,j)
          END DO
        END DO

      END
C  end of Schmeisser algorithm



C ***********************************************************
C * Procedure Polycopy: This procedure copies one polynomial
C * to another polynomial
C *
```

```
C ************************************************************
      SUBROUTINE polycopy(p1,np1,p2,np2,factor)
      IMPLICIT COMPLEX*32 (A-Z)
      INTEGER MAXROOT
      PARAMETER (MAXROOT=200)
      DIMENSION p1(0:MAXROOT),p2(0:MAXROOT)
      INTEGER np1,np2

C  Local Varibales
      INTEGER i

        DO i=0,MAXROOT
          p2(i) = 0.0Q0
        END DO

        DO i=0,np1
          p2(i) = factor * p1(i)
        END DO

        np2 = np1
      END


C ****************************************
C * Subroutine Polydiv: divides two polynomials u and v and yields
C * 2 polynomials q and rem where q is the quotient and rem is the
C * remainder
C *
C * u: poly. of degree nu;
C * v: poly of degree nv;
C * q: poly quotient u=v*q+rem of degree nq=nu-nv;
C * rem: poly remainder of degree nrem = nv-1;
C *
C * ************************************************
      SUBROUTINE polydiv(u,nu,v,nv,q,nq,rem,nrem)
      IMPLICIT COMPLEX*32 (A-Z)
      PARAMETER (MAXROOT=200)
      DIMENSION u(0:MAXROOT),v(0:MAXROOT),q(0:MAXROOT)
      DIMENSION rem(0:MAXROOT)
      INTEGER nu,nv,nq,nrem

C Local variables
      INTEGER k,j
```

```
      DO j=0,nu
        rem(j) = u(j)
        q(j) = 0.0Q0
      END DO

      DO k=nu-nv,0, -1
        q(k) = rem(nv+k)/v(nv)
        DO j=nv+k-1,k, -1
          rem(j) = rem(j) - q(k) * v(j-k)
        END DO
      END DO

      DO j=nv,nu
        rem(j) = 0.0Q0
      END DO

      nq = nu-nv
      nrem = nv-1
      IF (nrem .LT. 0) THEN
        nrem = 0
      END IF

      END


C ***************************************************
C * Function Polyderiv: this procedure finds derivative of a
C *    polynomial u and returns it in du.
C *
C * u: poly. of degree nu;
C * du: derivative polynomial of degree ndu=nu-1;
C *
C ***************************************************
      SUBROUTINE polyderiv(u,nu,du,ndu)
      IMPLICIT COMPLEX*32 (A-Z)
      PARAMETER (MAXROOT=200)
      DIMENSION u(0:MAXROOT),du(0:MAXROOT)
      INTEGER nu,ndu

C  Local variables
      INTEGER i

        DO i=1,nu
          du(i-1) = i*u(i)
```

```
      END DO
      nu = nu-1
    END


C ***************************************************
C * Procedure Polyadjust: adjusts the degree of the polynomial
C * by eliminating the zero coefficients.
C *
C * p: is the polynomial to be adjusted
C * np: degree of polynomial p
C * stol: Schmeisser Tolerance
C *
C * function returns False if polynomial is not zero
C * and              true if polynomial is zero
C *
C * ***************************************************
    SUBROUTINE polyadjust(u,nu,polzero,stol)
    IMPLICIT COMPLEX*32 (A-Z)
    PARAMETER (MAXROOT=200)
    DIMENSION u(0:MAXROOT)
    INTEGER nu
    LOGICAL polzero
    REAL*16  stol

C  Local variables
    INTEGER i

      i = nu
      DO WHILE (CQABS(QCMPLX(u(i))) .LT. CQABS(QCMPLX(stol))
   &           .AND. (i .GE. 0))
        u(i) = QCMPLX(0.0Q0)
        nu = nu-1
        i=i-1
      END DO

      IF (i .LT. 0) THEN
        nu=0
        polzero =.true.
      ELSE
        polzero =.false.
      END IF
      RETURN
    END
```

```
C End polyadjust


C ************************************************************
C * Subroutine Fied2sub: This procedure finds all the roots of
C * a polynomial by finding the eigenvalues of a
C * Complex symmetric matrix constructed by Fiedler's method
C *
C * p: is an array containing the coefficients of the
C *    given polynomial
C * np: degree of the polynomial p
C * b: array containing initial values for the Fiedler's method
C * a: matrix np*np obtained by Fiedler theroem 2.2
C * d: array containing auxilliary values
C * pb: value of polynomial evaluated at x=b(i)
C * dv1,dv2 are local variables
C * ro: constant of value -1 or 1 (see fiedler paper)
C * ftol: Fiedler's Tolerance
C *
C ************************************************************
      SUBROUTINE fied2sub (p,np,b,a,ftol)
        IMPLICIT COMPLEX*32  (A-Z)
        INTEGER MAXROOT
        PARAMETER (MAXROOT=200)

C Parameters
        DIMENSION p(0:MAXROOT)
        INTEGER   np
        DIMENSION b(MAXROOT)
        DIMENSION a(MAXROOT,MAXROOT)
        REAL*16   ftol

C Local variables
        DIMENSION d(MAXROOT)    /200*0.0Q0/
        DIMENSION dv(MAXROOT)   /200*0.0Q0/
        COMPLEX*32 ro /-1.0Q0/,dv1,dv2,pb

        INTEGER i,j


C Form the derivative v'(x) at each initial value bk
C v(x) = (x-b1)(x-b2)..(x-bn)

        i=1
```

```
      DO WHILE (i .LE. np)
        dv1 = 1.0Q0
        DO j=1,i-1
          dv1 = dv1 * (b(i) - b(j))
        END DO
        dv2 = 1.0Q0
        DO j=i+1,np
          dv2 = dv2 * (b(i) - b(j))
        END DO
        dv(i) = dv1*dv2
        i = i+1
      END DO

C Form the auxiliary values dk

      Do i=1,np
        IF (ABS(ro*dv(i)) .LT. ftol) THEN
          d(i) = 0.0Q0
        ELSE
          CALL polyeval(p,np,b(i),pb)
          d(i) = CQSQRT(pb/(ro*dv(i)))
        END IF
        write(*,*) i,':',d(i),'di:',pb,'pb:',dv(i),'dvi:'
      END DO


C Form the matrix A
      DO i=1,np
        DO j=i+1,np
          a(i,j) = -1.0Q0*ro*d(i)*d(j)
          a(j,i) = a(i,j)
        END DO
      END DO

      DO i=1,np
        a(i,i) = b(i) - ro*d(i)*d(i)
      END DO

    END
C End of Fiedler's algorithm


C * ************************************************
```

```
C * Procedure Polyeval: This procedure evaluates a polynomial
C *     at a certain value v and returns the value in pv
C *
C * u: poly. of degree nu;
C * v: value at which the poly. is to be evaluated;
C * pv: value of the polynomial at the value v;
C *
C * ************************************************
      SUBROUTINE polyeval(u,nu,v,pv)
      IMPLICIT COMPLEX*32 (A-Z)
      PARAMETER (MAXROOT=200)
      DIMENSION u(0:MAXROOT)
      INTEGER nu

C  Local variables
      INTEGER i
      COMPLEX*32 p

        p=u(nu)
        DO i=nu-1,0,-1
          p = p*v + u(i)
        END DO
        pv = p
      END

C End polyeval


C ************************************************************
C * Subroutine Fied3sub: This procedure finds all the roots of
C * a polynomial by finding the eigenvalues of a
C * complex symmetric matrix constructed by Fiedler3's method
C *
C * p: is an array containing the coefficients of the
C *    given polynomial
C * np: degree of the polynomial p
C * b: array containing initial values for the Fiedler's method
C * a: matrix np*np computed by Fiedler3 thoerem
C * d: array containing auxiliary values
C * pb: value of polynomial evaluated at x=b(i)
C * dv1,dv2 are local variables
C * ro: constant of value -1 or 1 (see Fiedler)
C*  ftol: Fiedler's Tolerance
C *
```

```
C ***************************************************************
      SUBROUTINE fied3sub (p,np,b,a,ftol)
        IMPLICIT COMPLEX*32  (A-Z)
        INTEGER MAXROOT
        PARAMETER (MAXROOT=200)

C Parameters
        DIMENSION p(0:MAXROOT)
        INTEGER   np
        DIMENSION b(MAXROOT)
        DIMENSION a(MAXROOT,MAXROOT)
        REAL*16   ftol

C Local variables
        DIMENSION c(MAXROOT)    /200*0.0Q0/
        DIMENSION dv(MAXROOT)   /200*0.0Q0/

        COMPLEX*32  bt
        COMPLEX*32 ro /-1.0Q0/,dv1,dv2,pb

        INTEGER i,j


C Form the derivative v'(x) at each initial value bk
C v(x) = (x-b1)(x-b2)..(x-bn-1)

        i=1
        DO WHILE (i .LE. np-1)
          dv1 = 1.0Q0
          DO j=1,i-1
            dv1 = dv1 * (b(i) - b(j))
          END DO
          dv2 = 1.0Q0
          DO j=i+1,np-1
            dv2 = dv2 * (b(i) - b(j))
          END DO
          dv(i) = dv1*dv2
          i = i+1
        END DO


C Form the auxilliary values ck

        Do i=1,np-1
```

```
        IF (ABS(ro*dv(i)) .LT. ftol) THEN
          c(i) = 0.0Q0
        ELSE
          CALL polyeval(p,np,b(i),pb)
          c(i) = CQSQRT(pb*ro/dv(i))
        END IF
      END DO


C Form the matrix A

      DO i=1,np-1
        a(i,np) = c(i)
        a(np,i) = c(i)
      END DO

      DO i=1,np-1
        a(i,i) = b(i)
      END DO

C Add the np-1 bk
      bt = 0.0Q0
      DO i=1,np-1
        bt = bt+b(i)
      END DO
      a(np,np) = -p(np-1) - bt

C Display the matrix A

    END

C End of Fiedler3's algorithm


C ***************************************************************
C * Subroutine Compasub: This procedure finds all the roots of
C * a polynomial by finding the eigenvalues of a
C * the companion matrix of the polynomial.
C *
C * p: is an array containing the coefficients of the
C *    given polynomial
C * np: degree of the polynomial p
C * c: matrix np*np obtained by the companion method
C *
```

```
C ************************************************************
      SUBROUTINE compasub (p,np,c)
        IMPLICIT COMPLEX*32  (A-Z)
        INTEGER MAXROOT
        PARAMETER (MAXROOT=200)

C Parameters
        DIMENSION p(0:MAXROOT)
        INTEGER   np
        DIMENSION c(MAXROOT,MAXROOT)

C Local Variables
        INTEGER i


C Form the companion matrix

        DO i=1,np
          c(1,i) =-1.0Q0*p(np-i)
        END DO

        DO i=1,np
          c(i,i-1) = 1.0Q0
        END DO

C Find the eigenvalues of C
C         CALL c_eigenv(c,np,b)

      END

C End of companion's algorithm


C * ************************************************************
C * Procedure Compalg: This procedure implements the three stages
C *  of the composite algorithm discussed.  For a polynomial
C *  with multiple zeros, it forms a reduced polynomial having
C *  simple zeros. Then, in the second stage, it computes
C *  the simple zeros by one of the algorithms I, II, or III.
C *  Finally, the multiplicities of zeros are estimated by the modified
C *  Lagouanelle's formula.
C *
C *  p: array containing the coefficients of the input polynomial
C *  np: degree of polynomial p
```

```
C *  scmtol: Schmeisser's tolerance required to determine when a
C *       remainder is to be set to zero
C *  fiedtol: Fiedler's tolerance required to accelarate convergence
C *    to multiple roots
C *
C *  b: array containig the simple zeros of the polynomial p
C *  m: array containing the multiplicity of each simple zero
C * ************************************************************
      SUBROUTINE compalg(p,np,scmtol,fiedtol)

        IMPLICIT COMPLEX*32 (A-Z)
        INTEGER MAXROOT
        PARAMETER (MAXROOT=200)

C Parameters
        DIMENSION p(0:MAXROOT)
        INTEGER np
        REAL*16 scmtol, fiedtol

C Local variables
        DIMENSION dp(0:MAXROOT)   /201*0.0Q0/
        DIMENSION g(0:MAXROOT)   /201*0.0Q0/
        DIMENSION f(0:MAXROOT)   /201*0.0Q0/
        DIMENSION f1(0:MAXROOT)   /201*0.0Q0/
        DIMENSION f2(0:MAXROOT)   /201*0.0Q0/
        DIMENSION rem(0:MAXROOT)   /201*0.0Q0/
        DIMENSION m(0:MAXROOT)   /201*0.0Q0/

        DIMENSION b(MAXROOT)     /200*0.0Q0/
        DIMENSION a(MAXROOT,MAXROOT) /40000*0.0Q0/

        INTEGER i,choice, iter, method, ndp,ng,nf1,nf2,nrem,nf
        LOGICAL breakdown


C First stage of the composite algorithm
C Get the derivative p' of p and make it monic
        CALL polyderiv(p,np,dp,ndp)
        DO i=0,ndp
          dp(i)=dp(i)/dp(ndp)
        END DO

C Find the greatest common divisor of p and p'
        CALL gcd(p,np,dp,ndp,g,ng,a,breakdown,scmtol)
```

```
C Form the reduced polynomials f=p/g
      CALL polydiv(p,np,g,ng,f,nf,rem,nrem)

C Second stage of the composite algorithm
      IF (breakdown .EQ. .FALSE.) THEN
C Get distinct roots from first block of Schmeisser
        CALL eigenv(a,nf,b, method)
      ELSE
C Use Fiedler 2 or 3 iteratively
C Initial values are either equidistant on a circle or randomly

        write(*,*) '\n\nHere are the following procedures'
        write(*,*) '4. Fiedler2 with initial values on a circle'
        write(*,*) '5. Fiedler2 with initial values in a square\n'
        write(*,*) '6. Fiedler3 with initial values on a circle'
        write(*,*) '7. Fiedler3 with initial values in a square\n'

        write(*,10)
10       FORMAT (' \nSelect procedure (4-7): ',$)
        read(*,*) choice

        write(*,20)
20       FORMAT (' \nEnter number of iterations for Fiedler: ',$)
        read(*,*) iter

C Choose Fiedler 2
        IF (choice .EQ. 4 .OR. choice .EQ. 5) THEN
          CALL setinit(choice,b,nf)
          DO i=1, iter
            CALL fied2sub(f,nf,b,a,fiedtol)
            CALL eigenv(a,nf,b,method)
          END DO
C Choose Fiedler 3
        ELSE IF (choice .EQ. 6 .OR. choice .EQ. 7) THEN
          CALL setinit(choice,b,nf)
          DO i=1, iter
            CALL fied3sub(f,nf,b,a,fiedtol)
            CALL eigenv(a,nf,b,method)
          END DO

        ELSE
          write(*,*) '\nwrong choice\n'
        END IF
```

```
      END IF
C End of 2nd stage

C Third stage of the composite algorithm
      CALL polyderiv(f,nf,f1,nf1)
      CALL polyderiv(p,np,dp,ndp)
      CALL polydiv(dp,ndp,g,ng,f2,nf2,rem,nrem)
      CALL multiplic(f1,nf1,f2,nf2,b,m)
    END

C End of compalg procedure


C * ************************************************
C * Procedure Multiplic: This procedure is based on
C *    Lagouanelle's limiting formula, and estimates the
C *    multiplicities of roots of a polynomial.
C *
C * u: array containing the coefficients of the reduced
C *   polynomial u=p/g
C * nu: degree of u
C * v: array containing the coefficients of the reduced
C *   differentiated polynomial v=dp/g
C * nv: degree of v
C * b: array containing the distinct zeros of the polynomial p
C * m: array containing the estimated multiplicity of each
C * distinct zero in b
C * ************************************************
    SUBROUTINE multiplic(u,nu,v,nv,b,m)
    IMPLICIT COMPLEX*32 (A-Z)
    INTEGER MAXROOT
    PARAMETER (MAXROOT=200)

C Parameters
    DIMENSION u(0:MAXROOT), v(0:MAXROOT)
    DIMENSION b(MAXROOT)
    INTEGER   nu,nv,i,m


C Local variables
    DIMENSION  m(MAXROOT)
    COMPLEX*32 ub,vb
    REAL*8 ubr,vbr
```

```
      DO i=1,nu
        CALL polyeval(u,nu,b(i),ub)
        CALL polyeval(v,nv,b(i),vb)
        ubr = DREAL(ub)
        vbr = DREAL(vb)
        m(i) = IDNINT(vbr/ubr)
      END DO
C Display the multiplicities of the roots
      DO i=1,nu
        write(*,*) m(i)
      END DO
      END
C End of multiplic procedure


C ***************************************************************
C * Subroutine Gcd: This subroutine finds the greatest common
C * divisor of two polynomials.
C *
C * p: is an array containing the coefficients of the
C *    given polynomial
C * np: degree of the polynomial
C * t: matrix np*np computed by Schmeisser
C * stol: The Tolerance of Schmeisser
C *
C ***************************************************************
      SUBROUTINE gcd (p1,np1,p2,np2,g,ng,t,breakdown,stol)
        IMPLICIT COMPLEX*32  (A-Z)
        INTEGER MAXROOT
        PARAMETER (MAXROOT=200)

C Paramaters
        DIMENSION p1(0:MAXROOT),p2(0:MAXROOT),g(0:MAXROOT)
        INTEGER   np1,np2,ng
        DIMENSION t(MAXROOT,MAXROOT)
        LOGICAL breakdown
        REAL*16   stol

C Local Variables
        DIMENSION f1(0:MAXROOT)  /201*0.0Q0/
        DIMENSION f2(0:MAXROOT)  /201*0.0Q0/

        DIMENSION q(0:MAXROOT)   /201*0.0Q0/
        DIMENSION rem(0:MAXROOT) /201*0.0Q0/
```

```
      DIMENSION b(MAXROOT)    /200*0.0Q0/
      DIMENSION diag(MAXROOT) /200*0.0Q0/
      DIMENSION c(MAXROOT)    /200*0.0Q0/


      INTEGER i,j,nq,nrem,nf1,nf2,rc,it,nt
      LOGICAL remzero /.false./,qzero /.false./


C Form polynomials f1 and f2
      CALL polycopy (p1,np1,f1,nf1,1.0Q0)
      CALL polycopy (p2,np2,f2,nf2,1.0Q0)

C Euclidean algorithm for finding the greatest common divisor
      breakdown=.FALSE.
      rc = 1
      it = 0
      DO WHILE (rc .NE. 0)
        it=it+1
        write (*,*) 'Iteration no ',it,'\n'
        IF (IABS(nf1-nf2) .GT. 1) THEN
          WRITE (*,*) 'Euclidean algorithm breakdown'
          breakdown = .TRUE.
        END IF
C Divide the two polynomials
        CALL polydiv(f1,nf1,f2,nf2,q,nq,rem,nrem)

C IMPORTANT  IMPORTANT
C This function polyadjust returns sometimes the wrong value
C use debugger to demonstrate that. So it has been changed into
C a subroutine instead.

        CALL polyadjust(rem,nrem,remzero,stol)
        CALL polyadjust(q,nq,qzero,stol)

        DO j=0,nrem
          rem(j) = -rem(j)
        END DO

        diag(it) = q(0)

C Check if remainder is sufficiently near zero.
        IF (remzero) THEN
          rc = 0
```

```
          CALL polycopy(f2,nf2,g,ng,1.0Q0)
          c(it)=0.0Q0
        ELSE
          CALL polycopy(f2,nf2,f1,nf1,1.0Q0)
          c(it) = rem(nrem)
C Do f2=rem/c(it)
          CALL polycopy(rem,nrem,f2,nf2,1.0Q0/c(it))
        END IF
      END DO

C If no breakdown occurs, form first block matrix
      IF (breakdown .EQ. .FALSE.) THEN
        nt=it
C Form the first block of Schmeisser's matrix
        DO i=1,nt
          t(i,i) = -diag(i)
          t(i+1,i) = SQRT(c(i))
          t(i,i+1) = t(i+1,i)
        END DO

C Display the first block matrix t of Schmeisser
        write(1,*) "\nSchmeisser's first block matrix"
        DO i=1,nt
          DO j=1,nt
            write(1,*) i,':',j,'  ',t(i,j)
          END DO
        END DO
      END IF
    END
C  End of gcd procedure
```