Journal of Difference Equations and Applications,
1995, Vol. 1, pp. 187-207
Reprints available directly from the publisher
Photocopying permitted by license only

Numerical Implementation of a New Algorithm for Polynomials with Multiple Roots

LUIGI BRUGNANO*

Dipartimento di Energetica, Via C. Lombroso 6/17, 50134 Firenze (Italy), E-mail: na.brugnano@na-net.ornl.gov October 18, 1994

Recently, a new method for computing simultaneously the roots of a given polynomial and their respective multiplicities was presented [1]. The proposed algorithm is very straightforward, but its early numerical implementations presented serious numerical drawbacks. To better understand this fact, the method has been recast in matrix form. This reformulation allows us to find the sources of instability of the method. By choosing an appropriate scaling of the problem, we are able to stabilize the method itself. Moreover, additional considerations allow us to derive a more efficient numerical algorithm.

AMS Nos.: 12D10, 12O4, 15A23, 15A12, 65F35.
KEYWORDS: polynomial roots, multiple roots, Euclidean algorithm for polynomials, unsymmetric Lanczos algorithm, tridiagonal matrices, eigen-values of matrices, breakdown, look-ahead
(Received October 25, 1994; in final form November 14, 1994)

1. INTRODUCTION

In [1] a new method to find simultaneously all the roots of a complex polynomial and their multiplicities was presented. Unlike many other known methods, the accuracy of the method is comparable for both simple and multiple roots. Other relevant features of this method are:

- no need of starting point, and
- order of convergence equal to that of the QR method when applied to a tridiagonal matrix with simple eigenvalues.

In Section 2 we shall reformulate the method in a way that is more meaningful for its numerical implementation. This new formulation has the double advantage of both confirming the logical simplicity of the method itself, and allowing us to discuss the conditioning of the problem, pointing out the possible sources of instability. Indeed, problems of numerical instability were evident in the early implementations of the algorithm. Solutions to such problems are examined in Section 3. Other remarkable numerical aspects will be illustrated in Section 4. Finally, in Section 5 we report some numerical tests obtained with the Matlab function *btr* which implements the algorithm.

2. REFORMULATION OF THE METHOD

The method described in [1] is based on the Euclidean algorithm for polynomials. To discuss the numerical properties and the possible numerical drawbacks of this

*Work supported by CNR (contract # 93.00571.CT01) and MURST 40%.

method it is more convenient to recast it into matrix form, by using a known connection between the Euclidean algorithm and the Lanczos method [7, 8]. Since this matrix formulation, which allows us to obtain a more compact and efficient algorithm than that in [1], is essential to describe the structure of the presented method, we shall report it in details, for completeness. Obviously, the new formulation reflects the properties studied in [1] and for this reason a formal proof will be omitted.

Let

$$p(x) = \sum_{i=0}^{n} c_i x^{n-i}, \; c_0 = 1, \tag{1}$$

be the polynomial we are interested in and

$$C = \begin{pmatrix} -c_1 & \cdots & -c_{n-1} & -c_n \\ 1 & & & \\ & \ddots & & \\ & & 1 & \end{pmatrix} \tag{2}$$

be the associated companion matrix. One way to obtain the roots of $p(x)$ is to compute the eigenvalues of C by means of the classical QR method, as the Matlab function *roots* does. When the roots are all simple, this approach is very efficient and robust and furnishes accurate approximations in $O(n^3)$ flops. Nevertheless, this simple approach suffers from the drawbacks of QR, for example, when multiple roots are present. The new method, which is strictly related to the tridiagonalization of the matrix (2), will try to overcome these problems. In more details, let

$$p_1(x) = \frac{p'(x)}{n} = \sum_{i=0}^{n-1} c_i^{(1)} x^{n-1-i}, \; c_0^{(1)} = 1,$$

be the normalized derivative of $p(x)$, and the vectors $\mathbf{u}_0$ and $\mathbf{u}_1$ defined as follows:

$$\mathbf{u}_0 = \mathbf{0}, \quad \mathbf{u}_1 = \left(1 \; c_1^{(1)} \; \ldots \; c_{n-1}^{(1)}\right)^T. \tag{3}$$

We observe that the choice of $\mathbf{u}_1$ is of fundamental importance (see [1]). Then, we consider the following three term recurrence relation:

$$C^T \mathbf{u}_i = \mathbf{u}_{i-1} + \alpha_i \mathbf{u}_i + \beta_i \mathbf{u}_{i+1}, \quad i = 1,\ldots,n. \tag{4}$$

The scalars α_i and β_i in (4) are defined in order to satisfy the following set of conditions:

$$\mathbf{e}_i^T \mathbf{u}_{i+1} = 0, \qquad \mathbf{e}_{i+1}^T \mathbf{u}_{i+1} = 1. \tag{5}$$

As usual, $\mathbf{e}_i$ is the ith vector of the canonical base on $\mathbb{R}^n$. By imposing (5), one easily obtains the following relations:

$$\alpha_i = \mathbf{e}_i^T (C^T \mathbf{u}_i - \mathbf{u}_{i-1}),$$

$$\beta_i = \mathrm{e}_{i+1}^T (C^T \mathbf{u}_i - \alpha_i \mathbf{u}_i - \mathbf{u}_{i-1}).$$

If these conditions can be satisfied at each step, then the matrix

$$U = (\mathbf{u}_1 \ldots \mathbf{u}_n) \tag{6}$$

is lower triangular with unit diagonal. We observe that the first condition in (5) can be always satisfied, while the second one can be fulfilled only if the $(i + 1)$st entry of the vector

$$\mathbf{v}_{i+1} = C^T \mathbf{u}_i - \alpha_i \mathbf{u}_i - \mathbf{u}_{i-1} \tag{7}$$

is nonzero. In this case one easily obtains that $\mathbf{u}_{n+1} = \mathbf{0}$, and

$$C^T U = U T_n^T, \tag{8}$$

where, in general,

$$T_i = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{i-1} \\ & & 1 & \alpha_i \end{pmatrix}, \qquad i = 1, \ldots, n. \tag{9}$$

Moreover, we observe that the entries $u_{ii}, \ldots, u_{in}$ of $\mathbf{u}_i$ are the coefficients of the ith monic polynomial produced by the Euclidean algorithm described in [1, Section 3] (see also [8, Theorem 3]).

We observe that the algorithm (4) for producing the matrix T_n^T (see (9)) is equivalent to the unsymmetric Lanczos procedure, applied to the matrix C^T, starting from the two set of vectors given by (3) and $\mathbf{z}_0 = \mathbf{0}, \mathbf{z}_1 = \mathbf{e}_1$, the first vector of the canonical base on $\mathbb{R}^n$. However, this would result in a more expensive algorithm. Therefore it is preferable to use the set of conditions (5). In this case, the resulting basic procedure can be regarded as a particular case of the procedure described in [11, page 396] to reduce a lower Hessemberg matrix to tridiagonal form.

Therefore, T_n is a tridiagonal matrix similar to C. It can be formed if at each step the second condition in (5) can be satisfied. When this happens, we say that the procedure (4) + (5) *terminates regularly*. In this case all the roots of the polynomial (1) need to be simple (see [1]). Moreover, the QR method applied to either C or T_n gives a comparable (good) accuracy for all the roots in $O(n^3)$ flops. We observe that the overhead for the construction of the matrix T_n amounts to $O(n^2)$ flops, due to the particular structure of the matrix C, and therefore, the global procedure is not too expensive, if compared with the QR applied directly to C.

The case when the procedure (4) + (5) does not terminate regularly is more interesting. In this case we say that a *breakdown* has occurred. We use the term "breakdown" because of the connection with Lanczos and, moreover, because this situation is analogous to what happens when one tries to transform a general square matrix to tridiagonal form [5, 10, 11]. Two different kinds of breakdown may be distinguished, depending on whether the vector (7) is the null vector or not.

When $\mathbf{v}_{i+1} = \mathbf{0}$, we say that a *complete breakdown* has occurred at the ith step. In this case (see [1]) we have that

$$p(x) = d_i(x)p_i(x),$$

where

$$p_i(x) = \sum_{j=i}^{n} u_{ij}x^{n-j},$$

and $d_i(x)$ is the characteristic polynomial of the matrix T_i given by (9). Moreover, the roots of $d_i(x)$ are all simple and they are all the distinct roots of $p(x)$. It follows that the QR method will furnish good approximations for all the eigenvalues of T_i in $O(i^3)$ flops. The same procedure will now be applied to the polynomial $p_i(x)$. We observe that if a complete breakdown occurs at the ith step, then

$$C^T U_i = U_i T_i^T,$$

where U_i is the $n \times i$ matrix formed by the first i columns of U. This means that we have found an invariant subspace of C^T, which is the subspace spanned by all the right eigenvectors of C^T.

In the second case we cannot reduce the size of the problem, nor can we continue the procedure (4) + (5), since the vector $\mathbf{v}_{i+1}$ has a zero $(i+1)$st entry, but $\mathbf{v}_{i+1} \neq \mathbf{0}$: we say that a *partial breakdown* has occurred. To recover from this problem, we use a technique which is equivalent to the look-ahead used in connection with the Lanczos process [3, 4, 9]. Let $v_{i+1,i+k+1}$ be the first nonzero entry of the vector $\mathbf{v}_{i+1}$ (obviously, $k \geq 1$). Then, we set $\beta_i = v_{i+1,i+k+1}$; moreover we define the following set of *breakdown vectors*:

$$\begin{aligned} \mathbf{u}_{i+k+1} &= \beta_i^{-1}\mathbf{v}_{i+1}, \\ \mathbf{u}_{i+j} &= C^T\mathbf{u}_{i+j+1}, \quad j = k, k-1, \ldots, 1. \end{aligned} \tag{10}$$

We observe that the rectangular matrix

$$U_{i+k+1} = (\mathbf{u}_1 \ \ldots \ \mathbf{u}_{i+k+1})$$

is still lower triangular and with unit diagonal. It follows that we can define the following *breakdown step* for the procedure (4) + (5):

$$C^T\mathbf{u}_{i+1} = \mathbf{u}_i + \alpha_{i+1}\mathbf{u}_{i+1} + \sum_{j=1}^{k} \beta_{i+j}\mathbf{u}_{i+j+1} + \beta_{i+k+1}\mathbf{u}_{i+k+2}, \tag{11}$$

where the scalar α_{i+1} is defined in order to have

$$\mathbf{e}_{i+1}^T\mathbf{u}_{i+k+2} = 0,$$

while the scalars β_{i+j} are defined by imposing the following set of conditions:

$$\mathbf{e}_{i+j+1}^T\mathbf{u}_{i+k+2} = 0, \quad j = 1,\ldots,k,$$

$$\mathbf{e}_{i+k+2}^T\mathbf{u}_{i+k+2} = 1.$$

Then, the procedure (4) + (5) can be restarted from $\mathbf{u}_{i+k+1}$ and $\mathbf{u}_{i+k+2}$ in order to obtain $\mathbf{u}_{i+k+3}$, and so on. If no more breakdowns occur, then instead of (8) we obtain:

$$C^TU = U(T_n')^T,$$

where

$$T_n' = \begin{pmatrix}
\alpha_1 & \beta_1 & & & & & & & & & & \\
1 & \ddots & \ddots & & & & & & & & & \\
 & \ddots & \ddots & \beta_{i-1} & & & & & & & & \\
 & & 1 & \alpha_i & 0 & \cdots & 0 & \beta_i & & & & \\
 & & & 1 & \alpha_{i+1} & \beta_{i+1} & \cdots & \beta_{i+k} & \beta_{i+k+1} & & & \\
 & & & & 1 & 0 & & & & & & \\
 & & & & & \ddots & \ddots & & & & & \\
 & & & & & & 1 & 0 & & & & \\
 & & & & & & & 1 & \alpha_{i+k+2} & \beta_{i+k+2} & & \\
 & & & & & & & & 1 & \ddots & \ddots & \\
 & & & & & & & & & \ddots & \ddots & \beta_{n-1} \\
 & & & & & & & & & & 1 & \alpha_n
\end{pmatrix}. \tag{12}$$

We observe that the $(k+1) \times (k+1)$ diagonal block of T_n' ($k+1$ is the number of the breakdown vectors) given by

$$\begin{pmatrix}
\alpha_{i+1} & \beta_{i+1} & \cdots & \beta_{i+k} \\
1 & & & \\
 & \ddots & & \\
 & & 1 & 0
\end{pmatrix}$$

has the structure of a companion matrix. This is due to the relations (10) and (11) used to recover the breakdown (see also [8, Theorem 3]). If more than one partial breakdown occurs, then a corresponding number of diagonal blocks of this kind

will be present in the matrix (12). Finally, we recall that if no complete breakdowns occur, then the eigenvalues of T_n' are simple (see [1]).

We remark the analogy between the above procedure to handle a partial breakdown and the look-ahead procedure for Lanczos: the vectors generated by (4) correspond to the "regular vectors" for the Lanczos algorithm, while the breakdown vectors defined in (10) correspond to the "inner vectors" produced by the look-ahead procedure. Nevertheless, from the practical point of view, there is a fundamental difference between the two things: in fact, when producing the breakdown vectors by means of (10), their number, k, is a priori known. This is not the case for the look-ahead procedure for Lanczos, where the size of the block corresponding to the current breakdown is dynamically determined (see [4, Section 4]). Moreover, we observe that the procedure (4) + (5) cannot exhibit a breakdown similar to an "incurable breakdown" (see [10]) for a Lanczos process. It follows that the above mentioned technique is always defined, unlike the general look-ahead for Lanczos in the case of an incurable breakdown (see [4, page 140]).

2.1. Recovering Information from a Breakdown

Let us examine the partial breakdown in more details. From the structure of the matrix (12) it follows that a "small" β_i implies that the i eigenvalues of T_i approximate, in some sense, all the eigenvalues of T_n'. In general, we have that

$$C^T U_i = U_i T_i + \Delta_i,$$

where

$$\Delta_i = \begin{pmatrix} 0 & \dots & 0 & 0 \\ \vdots & & \vdots & \vdots \\ \vdots & & \vdots & 0 \\ \vdots & & \vdots & v_{i+1,i+k+1} \\ \vdots & & \vdots & \vdots \\ 0 & \dots & 0 & v_{i+1,n} \end{pmatrix}_{n \times i},$$

and at least one of the $\{v_{i+1,j}\}$ is nonzero. This means that the columns of U_i span "approximately" an invariant subspace of C^T. If this subspace were an invariant subspace, then $p(x)$ would have exactly i (possibly) multiple roots, i.e., the i eigenvalues of T_i. Since the subspace is "nearly" invariant, then we may expect the roots of $p(x)$ to be grouped into i distinct clusters: indeed the i eigenvalues of T_i often correspond to the centers of these clusters.

Concerning the function *btr* used in the numerical tests, it is assumed that the i eigenvalues of T_i, say $\xi_1, \dots, \xi_i$, are centers of clusters when no other partial breakdowns have previously occurred and, moreover, at least one of the following conditions is satisfied:

- $|\beta_i|$ (see (12)) is smaller than a given tolerance;
- $i \leq [n/3]$ and the center $\eta = -c_1 n^{-1}$ (see (1)) of all the roots of the polynomial is inside the convex hull of the $\{\xi_j\}$.

3. STABILITY AND BALANCE OF THE PROBLEM

We now discuss the stability of the sequence (4) + (5) by considering, for sake of simplicity, only the case where no breakdowns occur. We observe that all the vectors $\mathbf{u}_i$ generated by the procedure belong to the Krylov subspace $K(\mathbf{u}_1, C^T)$. Let us consider now the Krylov matrix:

$$B = (\mathbf{u}_1 \; C^T\mathbf{u}_1 \; \ldots \; (C^T)^{n-1}\mathbf{u}_1).$$

From the relation (8) it follows that

$$\begin{aligned} B &= U\,(U^{-1}\mathbf{u}_1 \; T_n^T U^{-1}\mathbf{u}_1 \ldots (T_n^T)^{n-1}U^{-1}\mathbf{u}_1) \\ &= U\,(\mathbf{e}_1 \; T_n^T\mathbf{e}_1 \ldots \; (T_n^T)^{n-1}\mathbf{e}_1) \\ &= UW. \end{aligned} \tag{13}$$

It is easy to show that UW is the LU factorization of the matrix B, since W is upper triangular. We derive the conditioning of U by studying the conditioning of this factorization; that is how a perturbation on B propagates on U and W. One has:

$$(B + \delta B) = (U + \delta U)(W + \delta W). \tag{14}$$

By neglecting the term $\delta U \delta W$, from (13) and (14) it follows that

$$\delta U \doteq \delta B W^{-1} - U\delta W W^{-1}.$$

By passing to the norms, one obtains:

$$\begin{aligned} \frac{\|\delta U\|}{\|U\|} &\leq \|\delta B\| \, \frac{\|W^{-1}\|}{\|U\|} + \|\delta W\| \, \|W^{-1}\| \\ &\leq \|\delta B\| \, \|B^{-1}\| + \|\delta W\| \, \|W^{-1}\| \\ &\leq \kappa(B) \, \frac{\|\delta B\|}{\|B\|} + \kappa(W) \frac{\|\delta W\|}{\|W\|}. \end{aligned}$$

We conclude that $\|\delta U\| \, \|U\|^{-1}$ depends on the condition numbers of B and W. Concerning $\kappa(B)$, we have that

$$\mathrm{B} = (\mathbf{e}^T \otimes I) D_c^T (I \otimes \mathbf{u}_1),$$

where $\otimes$ denotes, as usual, the Kronecher product, $\mathbf{e} = (1 \; 1 \; \ldots 1)^T$, and

$$D_c = \begin{pmatrix} I & & & \\ & C & & \\ & & \ddots & \\ & & & C^{n-1} \end{pmatrix}.$$

It follows that

$$\kappa(B) \leq \kappa(\mathbf{e}^T \otimes I)\kappa(D_c)\kappa(I \otimes \mathbf{u}_1) = \kappa(D_c) = \kappa(C)^{n-1}.$$

Similarly, one has:

$$W = (\mathbf{e}^T \otimes I)Z_c^T(I \otimes \mathbf{e}_1),$$

where

$$Z_c = \begin{pmatrix} I & & & \\ & T_n & & \\ & & \ddots & \\ & & & T_n^{n-1} \end{pmatrix}.$$

It follows that

$$\kappa(W) \leq \kappa(\mathbf{e}^T \otimes I)\kappa(Z_c)\kappa(I \otimes \mathbf{u}_1) = \kappa(Z_c) = \kappa(T_n)^{n-1}.$$

We can conclude that it would be preferable to have both $\kappa(C)$ and $\kappa(T_n)$ small, in order to have a stable procedure. We observe that $\|\delta B\| \, \|B\|^{-1}$ and $\|\delta W\| \, \|W\|^{-1}$ are unaffected by a diagonal similarity transformation of C and T_n, while both $\kappa(B)$ and $\kappa(W)$ may change. Then, in order to stabilize the procedure we will perform such similarity transformations, by using suitable diagonal matrices D and D_1, according to the scheme

$$\begin{aligned} O &= (D^{-1}C^TD)(D^{-1}UD\,D_1) - (D^{-1}UD\,D_1)(D_1{}^{-1}D^{-1}T_n^TD\,D_1) \\ &=: \hat{C}^T\hat{U} - \hat{U}\hat{T}_n^T, \end{aligned} \tag{15}$$

in place of (8). The matrices D and D_1 will be chosen in order to have

$$\kappa(\hat{C}) \ll \kappa(\mathrm{C}) \text{ and } \kappa(\hat{T}_n) \ll \kappa(T_n).$$

3.1. Balancing the matrix *C*

Let us first suppose that all the coefficients c_i of the polynomial (1) are nonzero. Then the matrix

$$D = diag(1,c_1,\ldots,c_{n-1})$$

is nonsingular. One easily verifies that

$$\hat{C} = D\,C\,D^{-1} = \begin{pmatrix} -b_1 & \cdots & -b_{n-1} & -b_n \\ b_1 & & & \\ & \ddots & & \\ & & b_{n-1} & \end{pmatrix}, \tag{16}$$

where

$$b_i = c_i/c_{i-1}, \quad i = 1,\ldots,n. \tag{17}$$

We observe that a scaling of the variable x in (1) would result in a new matrix $\hat{C}$, obtained by multiplying (16) by the scaling factor. It follows that $\kappa(\hat{C})$ is independent of a scaling of x. To show the effectiveness of this scaling, let us now derive an estimate for $\kappa(C)$ and $\kappa(\hat{C})$. The inverses of the two matrices are, respectively:

$$C^{-1} = \begin{pmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ -c_0/c_n & -c_1/c_n & \cdots & -c_{n-1}/c_n \end{pmatrix},$$

and

$$\hat{C}^{-1} = \begin{pmatrix} & b_1^{-1} & & \\ & & \ddots & \\ & & & b_{n-1}^{-1} \\ -b_n^{-1} & -b_n^{-1} & \cdots & -b_n^{-1} \end{pmatrix}.$$

By considering the norm $\|\cdot\|_1$, one easily derives the following estimates:

$$\kappa(\hat{C}) \leq 4\,\frac{\max_i\{|b_i|\}}{\min_i\{|b_i|\}}, \tag{18}$$

$$\kappa(C) \leq 1 + \max_i\{|c_i|\}\left(1 + |c_n|^{-1}(1 + \max_i\{|c_i|\})\right). \tag{19}$$

To have an idea of the improvements, we shall discuss the following three cases:

1. all the roots of the polynomial are near a center ξ, $|\xi| \gg 1$;
2. all the roots of the polynomial are near a center ξ, $|\xi| \ll 1$;
3. r roots of the polynomial are near a center ξ_l, $|\xi_l| \gg 1$, and $n - r$ roots are near a center ξ_s, $|\xi_s| \ll 1$.

In the first case, it follows that

$$|c_i| \approx \binom{n}{i} |\xi|^i.$$

Since $|\xi| \gg 1$, then $\max_i\{|c_i|\} = |c_n| \approx |\xi|^n$, while $\max_i\{|b_i|\} \approx n|\xi|$, and $\min_i|\{b_i|\} \approx |\xi|/n$. It turns out that

$$\kappa(C) \approx 2|\xi|^n, \quad \kappa(\hat{C}) \approx 4n^2. \tag{20}$$

In the second case, by using similar arguments, one obtains the following estimates:

$$\kappa(C) \approx n|\xi|^{1-n}, \quad \kappa(\hat{C}) \approx 4n^2.$$

We observe that the estimate for $\kappa(\hat{C})$ is the same as before. This fact should be expected since, as we observed, $\kappa(\hat{C})$ is unaffected by a scaling of the variable x. As an example, let us consider the case of the polynomial

$$p(x) = (x + 20)^7 + 1.$$

Its companion matrix is given by (at most 3 significant digits are reported):

$$C = \begin{pmatrix} 1.4e+2 & -8.4e+3 & 2.8e+5 & -5.6e+6 & 6.72e+7 & -4.48e+8 & 1.28e+9 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

The corresponding balanced matrix is given by:

$$\hat{C} = \begin{pmatrix} 1.4e+2 & 6.e+1 & 3.33e+1 & 2.e+1 & 1.2e+1 & 6.67e+0 & 2.86e+0 \\ -1.4e+2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -6.e+1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -3.33e+1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2.e+1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1.2e+1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -6.67e+0 & 0 \end{pmatrix}.$$

One may verify that

$$\kappa(C) \approx 1.73e + 9,$$

and

$$\kappa(\hat{C}) \approx 1.40e + 2.$$

The relations (20) give $2.56e + 9$ and 196, respectively.

In the third case the following approximations can be derived:

$$|c_i| \approx \begin{cases} \binom{r}{i} |\xi_l|^i, & \text{if } i \leq r, \\ \binom{n-r}{i-r} |\xi_l|^r |\xi_s|^{i-r}, & \text{if } i > r. \end{cases}$$

From the equations (17), (18) and (19) it follows that

$$\kappa(C) \approx \frac{|\xi_l|^r}{|\xi_s|^{n-r}}, \quad \kappa(\hat{C}) \approx 4r(n-r)\frac{|\xi_l|}{} \leq n^2 \frac{|\xi_l|}{}.$$

Also in this case the estimate of $\kappa(\hat{C})$ is much smaller than the estimate of $\kappa(C)$ and moreover it is independent of a scaling of the variable x.

When some of the coefficients $c_i = 0$, we proceed as follows: let $c_j \neq 0$, $c_{j+k} \neq 0$, and $c_{j+r} = 0$, $r = 1,\ldots, k-1$. Then, by denoting with d_i the ith diagonal entry of D, we set $d_j = c_j$. The next diagonal entries of D are chosen as follows:

$$d_{i+1} = d_i\, m, \quad i = j,\ldots, j+k-1,$$

where $m = (c_{j+k}/c_j)^{1/k}$. This is done in order to have an optimal balance in the case where one single cluster of roots is present. As an example, the companion matrix associated to the polynomial

$$p(x) = x^{10} - 1024,$$

given by

$$C = \begin{pmatrix} 0 & \ldots & 0 & 1024 \\ 1 & & & \\ & \ddots & & \\ & & 1 & \end{pmatrix},$$

is transformed to

$$\hat{C} = D\,C\,D^{-1} = 2\begin{pmatrix} 0 & \ldots & 0 & 1 \\ 1 & & & \\ & \ddots & & \\ & & 1 & \end{pmatrix}.$$

We observe that $\kappa(C) = 1024$, while $\kappa(\hat{C}) = 1$.

As a remark, it can be said that in practice the check $c_i = 0$ is transformed to $|c_i| < tol$, where tol is a fixed tolerance. Moreover, some operations can be saved, when the polynomial $p(x)$ is complex, by considering the scaling by $|D|$, instead of D.

3.2. Balancing the matrix DT_nD^{-1}

After the balance of C, which produces the matrix $\hat{C}$, we want to balance the matrix DT_nD^{-1} in order to obtain the matrix (see (15)) $\hat{T}_n = D_1DT_nD^{-1}D_1^{-1}$. Again, we define D_1 in order to have a better conditioned matrix. For this purpose, let us denote by

$$DT_nD^{-1} = \begin{pmatrix} a_1 & c_1 & & \\ b_1 & \ddots & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ & & b_{n-1} & a_n \end{pmatrix} =: T_n^*.$$

When the matrix T_n^* is real and all the products b_ic_i are positive, the matrix D_1 which makes $\hat{T}_n$ symmetric obviously minimizes the condition number of $\hat{T}_n$. Conversely, we choose D_1 in order to have $|\hat{T}_n|$ symmetric. In this case in fact,

$$\|\hat{T}_n\|_F = \min_{D_1 \text{ diagonal}} \|D_1T_n^*D_1^{-1}\|_F$$

and then, $\hat{T}_n$ has also the minimum departure from normality (see [6, page 336]) for every diagonal similarity transformation of T_n^*.

In addition to this, it can be shown that the above mentioned choice for $\hat{T}_n$ minimizes a different upper bound for the condition number [2].

We end this section by stressing that the diagonal scalings here described have been fundamental, to get rid of the evident numerical instability observed in the early implementations of the method in [1].

4. CHECKING THE MULTIPLICITIES

From the previous arguments, it is evident that the proposed method is very simple, provided that a complete breakdown is clearly recognizable. This may not be an easy matter since, due to the propagation of round-off errors; the entries of the null vector (7), produced when the complete breakdown occurs, may be quite different from zero. In pathological cases, this happens even after the diagonal scalings previously described. Moreover, an incorrect detection of the exact number of the distinct roots frequently results in a very poor approximation of the roots themselves. It follows that a robust algorithm which implements this method must have the ability to check reliably whether a complete breakdown has occurred or not.

Fortunately, we observe that a very inexpensive test is able to guarantee a good accuracy. In fact, when the jth column of $\hat{U}$ is formed, its infinity norm, say μ_j, is kept. If it happens that

$$\mu_1,\ldots,\mu_i \gg \mu_{i+1},\ldots,\mu_n, \tag{21}$$

this means that a complete breakdown has occurred at the ith step. This implies that even if the complete breakdown is missed when the matrix $\hat{T}_n$ is formed, it can be recovered later. As an example, let us consider the polynomial

$$p(x) = \left(x - \frac{1}{8}\right)^4 \left(x - \frac{1}{2}\right)^3 (x - 3)\left(x - \frac{1}{3}\right).$$

The maxima $\{\mu_j\}$ versus the column index j in $\hat{U}$ are plotted in Figure 1: the "jump" between $j = 4$ and $j = 5$ is evident.

Moreover, this feature is also used in the function *btr* to recognize the centers of clusters of roots in a complementary way than that described in Section 2.1. In fact, the presence of i clusters of roots originates a smaller "jump" in (21), than that produced by a complete breakdown.

5. NUMERICAL EXAMPLES

In this section we report some numerical tests obtained with the function *btr* by using Matlab (with IEEE arithmetic), which implements the algorithm previously described. The output of the function *btr* is compared with that of the function *roots*, which is the standard polynomial root-finder of Matlab.

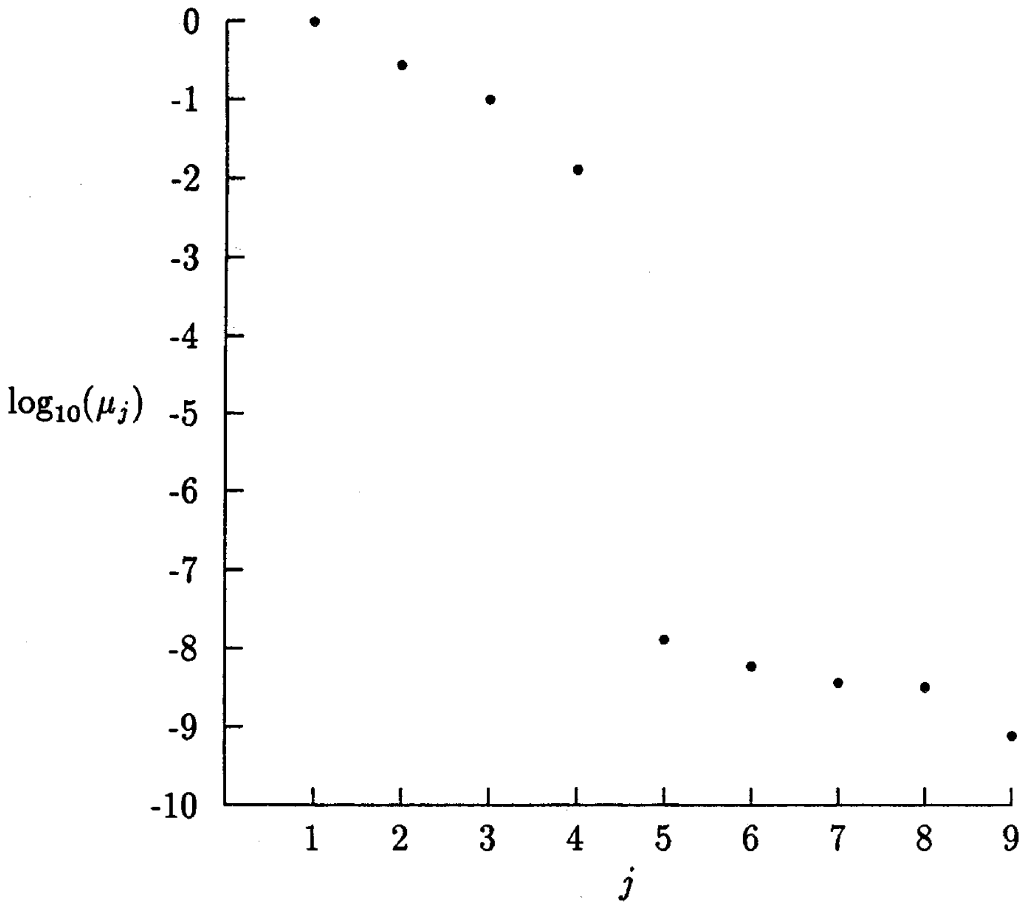

FIGURE 1. infinity norm of the columns of $\hat{U}$.

5.1. Description of the function

The calling sequence of the function *btr* is the following:

$$[r,m,b,info] = btr\ (p, flag)$$

The input parameters are:

p: vector which contains the coefficients of the polynomial, for decreasing powers of the variable;

flag: optional parameter set to:

0: (default) if all the roots and their respective multiplicities are requested;

1: if it is known that all the roots of the polynomial are simple; in this case m is not set, while r contains the output of the function *roots* of Matlab;

2: if only the distinct roots are needed (m is not set, in this case).

The output parameters are:

r: vector which contains the computed approximations of the roots;
m: vector which contains the computed multiplicities;
b: vector which contains the eventual centers of clusters of roots;
info: error flag set to:

0: if no problem occurred; in this case r contains only the distinct roots;

1: if at least one partial breakdown has occurred and the centers of probable clusters of roots (stored in b) have been detected;

2: if the distinct roots (contained in b) have been computed, but the respective multiplicities could not be computed completely. If μ is the maximum value in m, then all the values smaller than μ are multiplicities correctly computed, while all the entries equal to μ are lower bounds for the respective multiplicities. In this case, however, an eventual zero root, whose multiplicity is always correct, must be considered separately. Finally, r contains less accurate approximations of all the roots of the polynomial;

3: if there are probable multiple roots (contained in b), but the respective multiplicities could not be computed at all (m is not set); less accurate approximations of all the roots are contained in r.

When info = 2 or 3 r contains the output of the function *roots* of Matlab.

5.2. Examples

We now report some examples of application of the function *btr*. The first four examples are relative to cases in which info = 0 in output. Examples 5, 6 and 7 are relative to cases in which info = 1, 2 and 3, respectively. Finally, example 8 is the case of a polynomial having the coefficients given by small integers, exactly represented. In all the examples, the coefficients of the polynomial are computed by using the Matlab function *poly*.

5.2.1. Example 1

$$p(x) = (x - 29.68 + .753i)^2(x - .0942 - .5987i)^2(x + 1.42 + .9218i)^3.$$

The output of the function *btr* is

r	m
29.68000000000000 - 0.753000000000001i	2
-1.42000000000023 - 0.92180000000079i	3
0.09419999999982 + 0.598699999998581	2

b = [], info = 0,

while the function *roots* gives

```
29.68000014817261 - 0.753000013955281
29.67999985182740 - 0.752999986044721
-1.41998645972678 - 0.921798908357101
-1.42000582476997 - 0.921812272096011
-1.42000771550325 - 0.921788819546891
 0.09419998782261 + 0.598700002405951
 0.09420001217739 + 0.598699997594051
```

5.2.2. Example 2

$$p(x) = (x - 3.36 + .3258i)^9(x + 12.41 + .9141i)^9.$$

The output of the function *btr* is

r	m
-12.40999999999970 - 0.914099999999961	9
3.36000000000021 - 0.325799999999971	9

b = [], info = 0,

while that of the function *roots* is

```
-12.77423469222427 - 0.788147669315631
-12.60395472671754 - 0.584511724835521
-12.76878653181290 - 1.055809017942461
-12.34702784017470 - 0.541144127430351
-12.58955494234346 - 1.252784809693961
-12.12618666309136 - 0.669307617092171
-12.32961708763153 - 1.284760907695341
-12.03639853300218 - 0.905012074523461
-12.11423898300204 - 1.145422051471111
  3.44428574907961 - 0.365703858488311
  3.39838891107880 - 0.410406436773981
  3.33506408699160 - 0.414742859356331
  3.45007675819657 - 0.301658871828271
  3.28427407460334 - 0.377847114866421
  3.26868973303240 - 0.317528105111401
  3.41293189646263 - 0.249451213101341
  3.29486622130227 - 0.261014519191781
  3.35142256925276 - 0.233847021282171
```

5.2.3. Example 3

$$p(x) = (x + 5.23 + .9196i)^{20}.$$

The output of the function *btr* is

r	m
-5.23000000000000 - 0.919600000000001i	20

```
b = [], info = 0,
```

while that of the function *roots* is

```
-7.06646890586786 - 1.563627214268411
-6.67971405788494 - 2.163085247304991
-7.17077717529115 - 0.848555759469931
-6.97693978447484 - 0.171771389105291
-6.11185853460875 - 2.535040938514891
-6.55797097197921 + 0.342779912505621
-5.49540838923491 - 2.648767582640221
-6.02309317925395 + 0.642499047056781
-4.93954624698426 - 2.542656742216891
-5.46840869678273 + 0.732319413301081
-4.50646290630606 - 2.289305293001721
-4.21126848511275 - 1.967392082635801
-4.96244973971443 + 0.647466870306761
-4.02187182249706 - 1.636310216118421
-4.54453992390096 + 0.436250673910731
-3.89665861674688 - 1.294600239699991
-3.84596051334508 - 0.928563774303441
-4.22776619932901 + 0.144785326372601
-3.88328833453693 - 0.555529027434461
-4.00954751614838 - 0.192895736739111
```

5.2.4. Example 4

$$p(x) = (x - 3.5i)^7\Big((x - 2 - i)^3 + 9\Big).$$

The output of the function *btr* is

r	m
3.04004191152840 - 0.801405432758481	1
3.04004191150082 + 2.801405432749371	1
0.00000000000287 + 3.500000000004351	7
-0.08008382304408 + 0.999999999963891	1

```
b = [], info = 0,
```

while that of the function *roots* is

```
3.04004191152596 - 0.801405432764011
3.04004191152564 + 2.801405432764211
0.03567145707096 + 3.507963047722011
0.01592315003620 + 3.533058164272891
0.02829695011724 + 3.477221531249091
-0.01609433202827 + 3.532925240697661
0.00001277865008 + 3.463805117755561
-0.02820274634022 + 3.477221826279121
-0.03560725750567 + 3.507805072023471
-0.08008382305192 + 1.000000000000001
```

5.2.5. Example 5

$$p(x) = (x - .5 - .5i)^{12}\Big((x - 1 - i)^{13} + 7\Big).$$

We observe that the coefficients of this polynomial are not exactly represented with the precision used (eps = 2.22e-16). The output of the function *btr* is

r	m
1.86937186330328 + 1.77019628961130i	1
1.41186248528784 + 2.08599218101293i	1
2.12771862108672 + 1.27795767670212i	1
0.86000040474407 + 2.15300035887785i	1
2.12771860867197 + 0.72204232046301i	1
0.34021053387091 + 1.95587006652174i	1
1.86937186359015 + 0.22980373046989i	1
-0.02842953100919 + 1.53976145571115i	1
1.41186250665853 - 0.08599217807615i	1
-0.16146878254039 + 1.00000000026227i	1
0.86000040850695 - 0.15300036768500i	1
-0.02842953218662 + 0.46023854308441i	1
0.54153068275233 + 0.62955721884181i	1
0.61044412968105 + 0.58019046465754i	1
0.46284578982323 + 0.62264253181178i	1
0.62810317537419 + 0.50155333857464i	1
0.41138061899990 + 0.57772333300095i	1
0.39346630246839 + 0.52230866788170i	1
0.59852077608661 + 0.43969886440222i	1
0.40150685032027 + 0.47199509619107i	1
0.55241713033045 + 0.40985455978736i	1
0.42736225388047 + 0.43324661905397i	1
0.46479129755549 + 0.40932611018731i	1
0.50763101257235 + 0.40190318747118i	1
0.34021053017107 + 0.04412993118294i	1

b
1.00000000000058 + 1.00000000000058i
0.50000000000055 + 0.50000000000055i

r is also the output of the function *roots*.

5.2.6. Example 6

$$p(x) = (x - \frac{3}{11})^{12}(x - \frac{11}{3})^{2}(x - \frac{2}{7}i)^{4}(x - \frac{5}{2} - \frac{i}{4})^{2}(x - \frac{1}{4}).$$

The output of the function *btr* is

r
3.66666649103512 - 0.000000471473501
3.66666684229855 + 0.000000471473181
2.49999966848470 + 0.249999452317361
2.50000033151500 + 0.250000547682891
0.00010424404417 + 0.285655743003691
0.00005853309731 + 0.285818606882191
-0.00005846575645 + 0.285610051249211
-0.00010431138503 + 0.285772741722361
0.30081003803887 - 0.004243843562521
0.29936349251867 + 0.010927955894671
0.29442365433683 - 0.017518062617761
0.28267816279563 - 0.025541866387741
0.28988146468057 + 0.023220091640521
0.26889641274602 - 0.026783817863301
0.27538517950855 + 0.028459153197401
0.25645240800147 - 0.021358581635121
0.26075389837410 + 0.025371778458121
0.24814228810254 - 0.010775644443931
0.24578768616312 + 0.002552498697471
0.25015258745710 + 0.015690338621891
0.12500000000375 + 0.000000000000051

b	m
3.66666666666752 + 0.000000000000561	2
2.50000000000169 + 0.250000000000801	2
0.00000000000962 + 0.285714285725771	3
0.27272727271518 - 0.000000000030641	3
0.12500000017482 - 0.000000000470601	1

info = 2.

r is also the output of the function *roots*. We observe that the smallest multiplicities are correct.

5.2.7. Example 7

$$p(x) = (x + 2.1)^8\Big((x + 3 - 3i)^7 + 8\Big).$$

The output of the function *btr* is

```
                    r
-3.83915504449396 + 4.05226714271923i
-2.70050903215436 + 4.31215566471126i
-4.34590019252492 + 2.99999999927567i
-1.78738582717114 + 3.58396420801876i
-3.83915504087592 + 1.94773285795804i
-1.78738582698312 + 2.41603579207927i
-2.70050903679804 + 1.68784432860191i
-2.17796596201854 + 0.011399577878951i
-2.14513394136878 + 0.064886416082851i
-2.08454212543294 + 0.075734425962171i
-2.16076887437706 - 0.047060808279421i
-2.03750107415917 + 0.041342010708791i
-2.10971088496114 - 0.074028755966541i
-2.02777685613080 - 0.013158210034761i
-2.05660028055007 - 0.059114649716201i
                    b
-2.70050875619969 + 4.31215629188734i
-3.83915545681845 + 4.05226771733856i
-1.78738471978077 + 3.58396423302981i
-4.34590138629509 + 2.99999973862940i
-1.78738586282653 + 2.41603267952237i
-3.83915377615715 + 1.94772947858773i
-2.70051080046758 + 1.68785201582409i
-2.10000000009861 + 0.00000000096017i

m = [], info = 3.
```

r is also the output of the function *roots*. The vector b contains quite good approximations of all the distinct roots.

5.2.8. Example 8

$$p(x) = (x + i)^6(x - i)^6(x + 1)^6(x - 1)^6.$$

The output of the function *btr* is

```
                    r                    | m
1.00000000000000                         | 6
                  + 1.000000000000000i   | 6
-1.00000000000000                        | 6
                  - 1.000000000000000i   | 6

b = [], info = 0,
```

while the function *roots* gives less than three correct digits. The function *btr* can be requested at the e-mail address of the author.

ACKNOWLEDGEMENTS

The author is very indebted to Professor D. Trigiante for his precious help in the preparation of the paper.

References

[1] L. Brugnano, D. Trigiante, Polynomial roots: the ultimate answer? *Lin. Alg. and its Appl.* (to appear).
[2] L. Brugnano, D. Trigiante, *Sulla minimizzazione del numero di condizione di una matrice tridiagonale*, internal report, Dipartimento di Energetica, Universitá di Firenze, 1993.
[3] R. W. Freund, *The look-ahead Lanczos process for large nonsymmetric matrices and related algorithms*, Linear Algebra for Large Scale and Real-Time Applications, Kluwer, The Netherlands, 1993.
[4] R. W. Freund, M. H. Gutknecht, and N. Nachtingal, An implementation of the look-ahead Lanczos algorithm for non-hermitian matrices, *SIAM J. Sci. Comput.* **14** (1993), 137-158.
[5] G. A. Geist, Reduction of a general matrix to tridiagonal form, *SIAM J. Matrix Anal. Appl.* **12** (1991), 362-373.
[6] G. H. Golub and C. F. Van Loan, *Matrix computations*, 2nd ed., Johns Hopkins University Press, Baltimore, 1989.
[7] W. B. Gragg, Matrix interpretations and applications of the continued fraction algorithm, *Rocky Mount. J. Math.* **4** (1974), 213-225.
[8] W. B. Gragg and A. Lindquist, On the partial realization problem, *Lin. Alg. and its Appl.* **50** (1983), 277-319.
[9] M. H. Gutknecht, A completed theory of the unsymmetric Lanczos process and related algorithms, Part I, *SIAM J. Matrix Anal. Appl.* **13** (1992), 594-639.
[10] B. N. Parlett, Reduction to tridiagonal form and minimal realizations, *SIAM J. Matrix Anal. Appl.* **13** (1992), 567-593.
[11] J. H. Wilkinson, The algebraic eigenvalue problem, Oxford University Press, Oxford, 1965.

Journal of Difference Equations and Applications,
1996, Vol. 2, pp.101
Reprints available directly from the publisher
Photocopying permitted by license only

Errata

Paper: Numerical implementation of a new algorithm for polynomials with multiple roots, by L. Brugnano, Volume 1, Number 2 pp. 187–207.

page 197, second formula (row 3)

errata: $4\tau(n-\tau)\dfrac{|\xi l|}{} \leq n^2 \dfrac{|\xi l|}{}$.

corrige: $4\tau(n-\tau)\dfrac{|\xi l|}{|\xi s|} \leq n^2 \dfrac{|\xi l|}{|\xi s|}$.