SIAM J. SCI. COMPUT.
Vol. 35, No. 1, pp. A255–A269

FAST COMPUTATION OF THE ZEROS OF A POLYNOMIAL VIA FACTORIZATION OF THE COMPANION MATRIX*

JARED L. AURENTZ†, RAF VANDEBRIL‡, AND DAVID S. WATKINS†

Abstract. A new fast algorithm for computing the zeros of a polynomial in $O(n^2)$ time using $O(n)$ memory is developed. The eigenvalues of the Frobenius companion matrix are computed by applying a nonunitary analogue of Francis's implicitly shifted QR algorithm to a factored form of the matrix. The algorithm achieves high speed and low memory use by preserving the factored form. It also provides a residual and an error estimate for each root. Numerical tests confirm the high speed of the algorithm.

Key words. polynomial, root, companion matrix, QR algorithm

AMS subject classifications. 65F15, 15A18

DOI. 10.1137/120865392

1. Introduction.

The standard method for computing the zeros of a polynomial is to form the Frobenius companion matrix, balance it, and then compute its eigenvalues by Francis's implicitly shifted QR algorithm. This is the method used by the `roots` command of MATLAB. This works well for polynomials of low degree, but it becomes inefficient as the degree increases, as it requires $O(n^2)$ space and $O(n^3)$ time for a polynomial of degree n.

In recent years a variety of algorithms have been proposed [1, 2, 3] that solve the polynomial root problem in $O(n)$ space and $O(n^2)$ time by exploiting and preserving the low-rank structure of submatrices in the upper triangle of the companion matrix. In this paper we present a new algorithm that uses a novel method for preserving the low-rank structure. We start with the companion matrix in a factored form [4]. We then apply a bulge-chasing algorithm that preserves the factored form. In order to do this, we must use similarity transformations that are not unitary. Thus our algorithm is a nonunitary variant of Francis's algorithm. Each iterate is stored as a product of $n-1$ essentially 2×2 matrices, so the storage requirement is $4n$ asymptotically. The cost of each iteration is $O(n)$ flops, so the total flop count is $O(n^2)$.

In section 2 we introduce some terminology and prove Theorem 2.1, which is the result that makes our algorithm possible. Section 3 contains the description of the algorithm. In section 4 we present numerical results that show that our algorithm is competitive with other fast solvers and much faster than the `roots` command of MATLAB for polynomials of high degree.

*Submitted to the journal's Methods and Algorithms for Scientific Computing section February 9, 2012; accepted for publication (in revised form) November 30, 2012; published electronically January 10, 2013. This research was partially supported by the Research Council KU Leuven, project OT/11/055 (Spectral Properties of Perturbed Normal Matrices and their Applications), CoE EF/05/006 Optimization in Engineering (OPTEC), by the Fund for Scientific Research–Flanders (Belgium) project G034212N (Reestablishing smoothness for matrix manifold optimization via resolution of singularities), and by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office, Belgian Network DYSCO (Dynamical Systems, Control, and Optimization).

http://www.siam.org/journals/sisc/35-1/86539.html

†Department of Mathematics, Washington State University, Pullman, WA 99164-3113 (jaurentz@math.wsu.edu, watkins@math.wsu.edu).

‡Department of Computer Science, KU Leuven, 3001 Leuven (Heverlee), Belgium (raf.vandebril@cs.kuleuven.be).

2. Operations on core transformations. A *core transformation* is a nonsingular $n \times n$ matrix G_i that acts on two adjacent rows or columns. More precisely, G_i is identical to the identity matrix except for the two-by-two submatrix at the intersection of rows and columns i and $i+1$, which can be any nonsingular 2×2 matrix. Core transformations will normally be written, as shown here, with a subscript i that indicates that the transformation acts on rows i and $i+1$. Fiedler [4] showed (and see section 3) that any $n \times n$ Frobenius companion matrix C can be written as a product of $n-1$ core transformations:

$$C = C_1 C_2 \cdots C_{n-1}.$$

We are going to devise an algorithm that preserves this structure.

We will find it useful to use a symbolic double-headed arrow notation to denote core transformations. For example, if we modify a 4×4 matrix A by multiplying it on the left by a core transformation G_1 acting on rows one and two, we will write

$$G_1 A = \begin{array}{c} \vec{\hookrightarrow} \\ \end{array} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}.$$

The two-headed arrow denotes G_1. The arrowheads point to the rows of A on which it acts. If we subsequently multiply by two more core transformations, G_2 and G_3, we have

$$G_3 G_2 G_1 A = \begin{array}{c} \vec{\hookrightarrow} \\ \vec{\hookrightarrow} \\ \vec{\hookrightarrow} \end{array} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}.$$

Notice that two core transformations G_i and H_j will commute if $|i-j| > 1$, so we only need to consider interactions between core transformations with adjacent or equal indices. We will employ two types of operations on core transformations, one of which is trivial. If we have two core transformations G_i and H_i that act on the same two rows and columns, their product $G_i H_i$ is also a core transformation. The act of multiplying two such core transformations together to form a single core transformation is called *fusion*. The nontrivial operation is the *turnover*. Suppose we have three adjacent core transformations $F_i G_{i+1} H_i$. The turnover operation rewrites this product in the form $A_{i+1} B_i C_{i+1}$. Schematically

$$\begin{array}{c} \vec{\hookrightarrow} \\ \vec{\hookrightarrow} \\ \vec{\hookrightarrow} \end{array} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} = \begin{array}{c} \vec{\hookrightarrow} \\ \vec{\hookrightarrow} \\ \vec{\hookrightarrow} \end{array},$$

where we are depicting only rows and columns i, $i+1$, and $i+2$. The three core transformations on the left are F_i, G_{i+1}, and H_i. They can be multiplied together to form a matrix whose active part is 3×3, as depicted in the middle. This 3×3 matrix is then factored into a product of three core transformations A_{i+1}, B_i, and C_{i+1}, as shown on the right.

Such a factorization is not always possible, but it is *almost* always possible. In the unitary case it is always possible. This property of unitary matrices is well known and has been exploited in many algorithms [3, 7, 8].

THEOREM 2.1. *Almost every nonsingular* 3×3 *matrix* M *can be factored into a product* $M = ABC$ *of the form*

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} 1 & & \\ & a_{22} & a_{23} \\ & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \\ b_{21} & b_{22} & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & c_{22} & c_{23} \\ & c_{32} & c_{33} \end{bmatrix}.$$

If M *is unitary, the factorization is always possible, and the factors* A, B, *and* C *can be taken to be unitary.*

Proof. Our proof will show how to construct A, B, and C, and it will also show exactly where the construction can fail. The set of matrices M on which the construction fails has Lebesgue measure zero. If M is unitary, our construction produces unitary A, B, and C.

The matrix A must be designed so that

$$A^{-1}M = BC = \begin{bmatrix} b_{11} & b_{12}c_{22} & b_{12}c_{23} \\ b_{21} & b_{22}c_{22} & b_{22}c_{23} \\ 0 & c_{32} & c_{33} \end{bmatrix}.$$

Thus the effect of A^{-1} on M must be to create a zero in the $(3,1)$ position and to make the $(1:2, 2:3)$ submatrix have rank one.

In the unitary case these two outcomes are linked: $m_{31} = 0$ if and only if $\left[\begin{smallmatrix} m_{12} & m_{13} \\ m_{22} & m_{23} \end{smallmatrix}\right]$ has rank one. This follows from the CS decomposition [10, Theorem 2.6.1] or more elementary arguments. Thus we can achieve our objective by choosing A to be a unitary matrix (rotator or reflector, say) such that $A^{-1}M$ has a zero in the $(3,1)$ position.

Now consider the general situation, in which M might or might not be unitary. In our numerical experiments we have considered two different approaches to the construction of the matrix A. One builds A all at once, so that the resulting $A^{-1}M$ has a zero in the desired location and a complementary rank-one matrix. The other builds A in two stages: $A = QG$, where the transformation $M \to Q^{-1}M$ creates a zero in the $(3,1)$ position, and the transformation $Q^{-1}M \to G^{-1}Q^{-1}M$ causes the $(1:2, 2:3)$ submatrix to have rank one without disturbing the zero in the $(3,1)$ position. Numerical tests have shown that the first construction yields slightly faster code. We will present the second construction here in the interest of clarity.

Let

$$Q = \begin{bmatrix} 1 & & \\ & q_{22} & q_{23} \\ & q_{32} & q_{33} \end{bmatrix}$$

be a unitary matrix such that

$$Q^{-1}M = \begin{bmatrix} \hat{m}_{11} & \hat{m}_{12} & \hat{m}_{13} \\ \hat{m}_{21} & \hat{m}_{22} & \hat{m}_{23} \\ 0 & \hat{m}_{32} & \hat{m}_{33} \end{bmatrix}.$$

If the $(1:2, 2:3)$ submatrix has rank one, take $A = Q$. If not, let

$$G = \begin{bmatrix} 1 & & \\ & 1 & g \\ & & 1 \end{bmatrix}$$

be a Gauss transform such that the $(1:2,2:3)$ submatrix of $G^{-1}Q^{-1}M$ has rank one. Notice that the effect of the left multiplication by G^{-1} is to subtract g times the third row from the second row, so the zero in the $(3,1)$ position is preserved. In order to satisfy the rank-one condition on the submatrix, g must be chosen so that

$$\begin{bmatrix} 1 & -g \end{bmatrix} \begin{bmatrix} \hat{m}_{22} & \hat{m}_{23} \\ \hat{m}_{32} & \hat{m}_{33} \end{bmatrix} = \alpha \begin{bmatrix} \hat{m}_{12} & \hat{m}_{13} \end{bmatrix}$$

for some scalar α (which might be zero). This is achieved by taking g to satisfy

$$\begin{bmatrix} 1 & -g \end{bmatrix} = \beta \begin{bmatrix} \hat{m}_{12} & \hat{m}_{13} \end{bmatrix} \begin{bmatrix} \hat{m}_{33} & -\hat{m}_{23} \\ -\hat{m}_{32} & m_{22} \end{bmatrix}$$

for some scalar β. An easy computation shows that the unique value for which this holds is

$$g = \frac{\hat{m}_{12}\hat{m}_{23} - \hat{m}_{13}\hat{m}_{22}}{\hat{m}_{12}\hat{m}_{33} - \hat{m}_{13}\hat{m}_{32}}. \tag{2.1}$$

Notice that the numerator in (2.1) is just the determinant of the $(1:2,2:3)$ submatrix of $\hat{M}$.

The construction breaks down only if $\hat{m}_{12}\hat{m}_{33} - \hat{m}_{13}\hat{m}_{32} = 0$, which means that the submatrix

$$\begin{bmatrix} \hat{m}_{12} & \hat{m}_{13} \\ \hat{m}_{32} & \hat{m}_{33} \end{bmatrix}$$

has linearly dependent rows. The set of matrices $\hat{M}$ satisfying this condition has Lebesgue measure zero and so does the corresponding set of matrices $M = Q\hat{M}$.

Letting $A = QG$, we have

$$A^{-1}M = \begin{bmatrix} \tilde{m}_{11} & \tilde{m}_{12} & \tilde{m}_{13} \\ \tilde{m}_{21} & \tilde{m}_{22} & \tilde{m}_{23} \\ 0 & \tilde{m}_{32} & \tilde{m}_{33} \end{bmatrix}$$

with the $(1:2,2:3)$ submatrix having rank one. Thus we can obtain

$$\begin{bmatrix} \tilde{m}_{12} & \tilde{m}_{13} \\ \tilde{m}_{22} & \tilde{m}_{23} \end{bmatrix} = bc^T = \begin{bmatrix} b_{12} \\ b_{22} \end{bmatrix} \begin{bmatrix} c_{22} & c_{23} \end{bmatrix}.$$

This factorization is unique up to an arbitrary constant. In our code we normalize b and c so that $\|b\|_2 = \|c\|_2$. Taking

$$\begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = \begin{bmatrix} \tilde{m}_{11} \\ \tilde{m}_{21} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} \tilde{m}_{32} & \tilde{m}_{33} \end{bmatrix},$$

we obtain B and C. In the unitary case we have $\|b\|_2 = \|c\|_2 = 1$, and B and C are automatically unitary. ∎

3. Computing the eigenvalues of a companion matrix. Let

$$p(x) = x^n - a_1x^{n-1} - a_2x^{n-2} \cdots - a_n.$$

Our task is to find the n values λ at which $p(\lambda) = 0$. We will assume that $a_n \neq 0$, since otherwise we can factor out a zero root and deflate the problem. The zeros of p are exactly the eigenvalues of the Frobenius companion matrix

$$C = \begin{bmatrix} a_1 & a_2 & \cdots & a_{n-1} & a_n \\ 1 & 0 & & & \\ & 1 & \ddots & & \\ & & \ddots & & \\ & & & 1 & 0 \end{bmatrix}.$$

Fiedler [4] showed that C can be factored into a product

$$C = C_1 C_2 \cdots C_{n-1} C_n,$$

where

$$C_j = \begin{bmatrix} I_{j-1} & & & \\ & a_j & 1 & \\ & 1 & 0 & \\ & & & I_{n-j-1} \end{bmatrix}, \quad j = 1, \ \ldots, \ n-1,$$

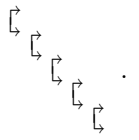
and $C_n = \operatorname{diag}\{I_{n-1}, a_n\}$. This decomposition can be obtained by applying Gaussian elimination to C with a row interchange at each step. For our purposes we want a factorization into a product of $n-1$ core transformations, so we will multiply C_{n-1} by C_n to get a new

$$C_{n-1} = \begin{bmatrix} I_{n-2} & & \\ & a_{n-1} & a_n \\ & 1 & 0 \end{bmatrix} \tag{3.1}$$

and a factorization

$$C = C_1 C_2 \cdots C_{n-1}. \tag{3.2}$$

We will operate on this factored form, which we denote schematically by



(3.3)

Here we are depicting five core transformations, so this is the case $n = 6$. A product of core transformations in this *descending* order is always upper Hessenberg.

We will develop a bulge-chasing algorithm that is a nonunitary variant of Francis's algorithm [5, 11, 12] that preserves the factored Hessenberg form (3.3). Each iteration begins with a matrix $A = C_1 \cdots C_{n-1}$ and terminates with a matrix $\hat{A} = \hat{C}_1 \cdots \hat{C}_{n-1}$. Multishift variants are easily implemented, but we will focus on the single-shift version here.

The iteration begins by choosing a shift ρ. We use the following standard shift choice: Compute the two eigenvalues of the lower-right 2×2 submatrix of A. Then

take ρ to be the one that is closer to a_{nn}.[1] Once we have ρ, we let G_1 be a unitary core transformation whose first column is proportional to

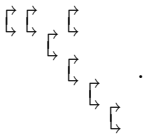
$$(A - \rho I)e_1 = \begin{bmatrix} a_{11} - \rho \\ a_{21} \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

and then we do a similarity transformation

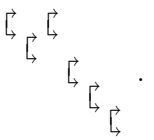
$$A \to G_1^{-1} A G_1 = G_1^{-1} C_1 \cdots C_{n-1} G_1.$$

This can be represented as

(3.4)

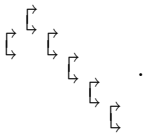


The core transformation G_1 commutes with $C_3, \ldots, C_{n-1}$, so it has been moved in next to C_2, which is the first core transformation with which it does not commute. The core transformations G_1^{-1} and C_1, which appear at the top left of (3.4), can be fused into a single core transformation:

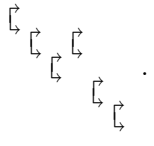


Then the three remaining transformations in the top-left corner can (almost always) be turned over via the construction in Theorem 2.1 to yield

(3.5)

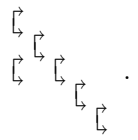


The core transformation at the far left in (3.5), which we will call G_2, is the bulge in the Hessenberg form. It needs to be removed, so multiply on the left by G_2^{-1} and on the right by G_2 to obtain

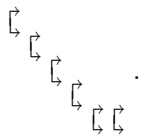


[1] This strategy is not globally convergent. Therefore we use a standard ad hoc remedy. If 15 iterations have passed without finding a root, we take one step with a random *exceptional* shift to break up any symmetries that may be preventing convergence.

Then do another turnover operation to get



The bulge has been moved down one position. The pattern of the iteration has now been established. To move the bulge down further, we move it from left to right by a similarity transformation and do another turnover operation. After $n-2$ such turnovers, the bulge has been moved to the bottom. Then we do one final similarity transformation to get to the form



Now the two core transformations in the lower-right corner can be fused, and the iteration is complete. The matrix has been returned to Hessenberg form.

The total iteration is a similarity transformation

$$\hat{A} = G^{-1}AG,$$

where G is a product of core transformations:

$$G = G_1 G_2 \cdots G_{n-1}.$$

Almost all the computational work is in the turnovers. Since there are about n turnovers at a cost of $O(1)$ per turnover, the total cost of an iteration is $O(n)$ flops.

The following theorem, which is a routine application of theory in [10], allows us to apply standard GR convergence theory to the new algorithm.

THEOREM 3.1. *The bulge-chasing iteration mapping A to $\hat{A}$ with shift ρ satisfies*

$$\hat{A} = G^{-1}AG, \quad \textit{where} \quad A - \rho I = GR,$$

for some upper-triangular matrix R.

Proof. Recall that G_1 was constructed so that its first column is proportional to $(A-\rho I)e_1$. Thus $(A-\rho I)e_1 = \alpha G_1 e_1$ for some nonzero α. Since the factors $G_2, \ldots, G_{n-1}$ do not touch the first component of any vector, $G_j e_1 = e_1$ for $j = 2, \ldots, n-1$. It follows that $Ge_1 = G_1 e_1$, so

$$(A-\rho I)e_1 = \alpha G e_1. \tag{3.6}$$

It then follows, as in [10, Theorem 4.5.5], that there exists an upper-triangular matrix R such that $A - \rho I = GR$. We sketch the argument: Let $\kappa(B, z)$ denote the $n \times n$ Krylov matrix

$$\kappa(B,z) = \begin{bmatrix} z & Bz & B^2 z & \cdots & B^{n-1}z \end{bmatrix}.$$

Then, using $G\hat{A} = AG$, (3.6), and $A(A-\rho I) = (A-\rho I)A$, we have

$$G\kappa(\hat{A}, e_1) = \kappa(A, Ge_1) = \alpha^{-1}\kappa(A, (A-\rho I)e_1) = \alpha^{-1}(A-\rho I)\kappa(A, e_1).$$

Therefore

$$A - \rho I = G(\alpha\kappa(\hat{A}, e_1)\kappa(A, e_1)^{-1}) = GR,$$

where $R = \alpha\kappa(\hat{A}, e_1)\kappa(A, e_1^{-1})$ is upper triangular. ∎

Theorem 3.1 implies that the bulge-chasing algorithm that we have outlined is an instance of a generic GR algorithm [10, Chapter 4] and is subject to the convergence theory of [10, Chapter 5], [9]. Repeated iterations should lead to quadratic convergence for each root. In our algorithm this is manifested in quadratic convergence of the bottom core transformation to upper-triangular form. Deflation or reduction of the problem is possible whenever one of the core transformations becomes (numerically) upper triangular.

Dealing with breakdowns. Since the turnover operation sometimes breaks down, the algorithm can fail occasionally. This can be remedied by changing the shift ρ, which is the one parameter we have at our disposal. Therefore, whenever there is a breakdown, simply repeat the iteration with a different shift. This happens rarely. In the course of running the numerical tests discussed in the following section, it never happened.

4. Numerical results.

Measures of stability and accuracy. Since the algorithm uses nonunitary transformations, there is no guarantee of stability. It is therefore important to provide a posteriori measures of the quality of our results. For each computed eigenvalue λ of a companion matrix

$$C = \begin{bmatrix} a_1 & a_2 & \cdots & a_{n-1} & a_n \\ 1 & 0 & & & \\ & 1 & \ddots & & \\ & & \ddots & & \\ & & & 1 & 0 \end{bmatrix}$$

associated with a polynomial

$$p(x) = x^n - a_1 x^{n-1} - a_2 x^{n-2} \cdots - a_n,$$

we get for free a computed eigenvector

$$v = \begin{bmatrix} \lambda^{n-1} & \lambda^{n-2} & \cdots & \lambda & 1 \end{bmatrix}^T.$$

Therefore we can compute the residual norm

$$r = \frac{\| (\lambda I - C)v \|_\infty}{\| C \|_\infty \| v \|_\infty} \tag{4.1}$$

as a test of backward stability. This is inexpensive because

$$(\lambda I - C)v = \begin{bmatrix} p(\lambda) & 0 & \cdots & 0 \end{bmatrix}^T.$$

If $|\lambda| > 1$ and n is very large, there is the danger of overflow in the computation of $p(\lambda)$ and $\| v \|$. Therefore, whenever $|\lambda| > 1$, we introduce $\mu = 1/\lambda$ and use the eigenvector

$$v = \begin{bmatrix} \mu & \mu^2 & \cdots & \mu^{n-1} & \mu^n \end{bmatrix}^T.$$

With this v we have

$$(\lambda I - C)v = \begin{bmatrix} \tilde{p}(\mu) & 0 & \cdots & 0 \end{bmatrix}^T,$$

where $\tilde{p}$ is the polynomial with reversed coefficients:

$$\tilde{p}(x) = -a_n x^n - a_{n-1}x^{n-1} \cdots - a_1 x + 1. \tag{4.2}$$

This eliminates the danger of overflow in the computation of the residual (4.1).

The quantity $|p'(\lambda)|^{-1}$ is an absolute condition number for the computed root λ. Thus the quantity

$$E = |p(\lambda)/p'(\lambda)|$$

is an estimate of the error. It is also the magnitude of one step of Newton's method applied to p. Therefore, if we are willing to pay the price to get the error estimate E, we can also refine our computed solution by taking one step of Newton's method at no additional cost. In our experiments we have done this routinely. In addition, once we have the refined value of the root, we do a second residual computation (4.1). In cases where $|\lambda| > 1$, we use the reversed polynomial $\tilde{p}$ (4.2) in place of p.

The total computational cost of these tests of stability and accuracy is Kn^2, but the constant K is small, so this adds almost nothing to the overall computing time.

Tests on high-degree polynomials. All our numerical experiments were done on a computer with an AMD Athlon dual core processor with 512 KB cache per core, running at 2.9 GHz. We coded[2] our method in Fortran 90 and compared it with two other methods: (1) The LAPACK routine ZHSEQR was used to compute the eigenvalues of the companion matrix by Francis's implicitly shifted QR algorithm. This is similar to the `roots` command of MATLAB. The special structure of the companion matrix is not exploited. (2) The single-shift code of Bini et al. [1] also computes the eigenvalues of the companion matrix by Francis's implicitly shifted QR algorithm, but it exploits the quasi-separable structure of the companion matrix and subsequent iterates to compute the roots in $O(n^2)$ time using $O(n)$ storage. It is one of the best of the structured codes that have been proposed so far.

Example 4.1. We computed the roots of polynomials with random complex coefficients, normally distributed with mean 0 and variance 1 in both the real and imaginary parts. Figure 4.1 shows the average execution time in seconds for our code (AVW), for the code of Bini et al. [1] (BBEGG), and for LAPACK. For each degree the time reported is an average over five random polynomials. The AVW times include the (small) time for the stability and accuracy tests described above. At all degrees AVW is significantly faster than BBEGG. From about degree 50 AVW is faster than LAPACK. At the highest degree that we tested (1133), AVW is four times as fast as BBEGG and more than fifteen times as fast as LAPACK. The lines in Figure 4.1 are least squares fits to the data points corresponding to the eight largest degrees. We excluded the lower degrees in order to get a closer fit to the data for larger degrees. The slope of the AVW line is 1.92, close to 2, indicating that the complexity of AVW is $O(n^2)$ in practice. The slopes of the other lines are 2.28 for BBEGG and 2.95 for LAPACK.

Table 4.1 shows the maximum (over all roots) of the residuals (4.1) for each polynomial. The column labeled AVW shows that our residuals increase as the degree

[2]The code is available at http://www.math.wsu.edu/students/jaurentz/publications/code.html.

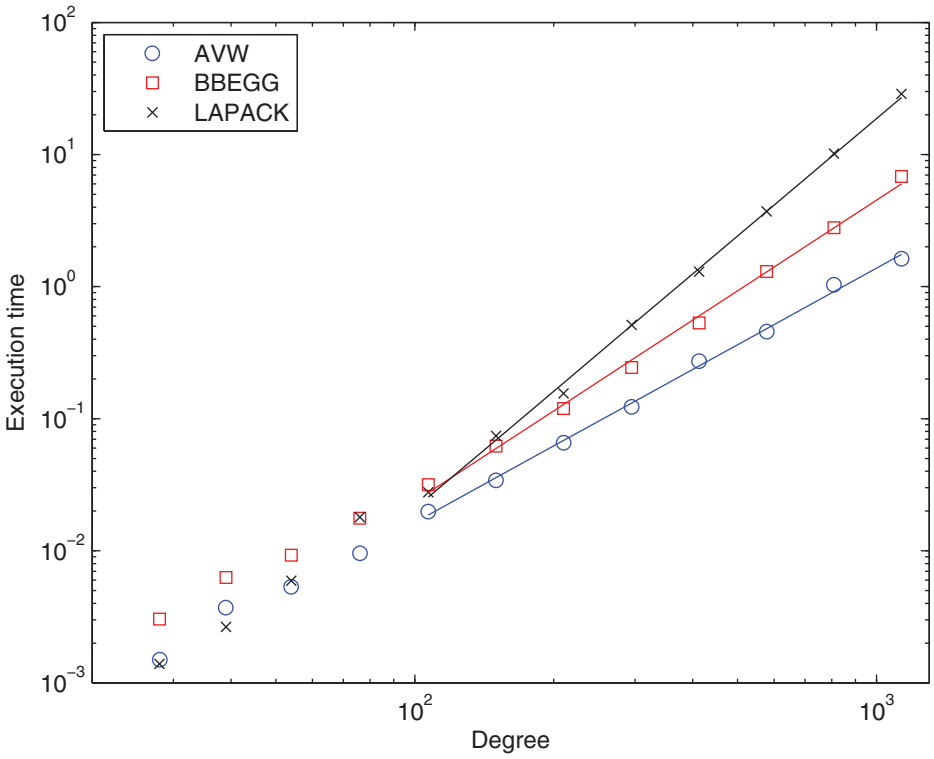


FIG. 4.1. *Execution times for random polynomials as a function of degree.*

TABLE 4.1
Maximum residual (random polynomials).

Degree	AVW	BBEGG	LAPACK
28	2.2×10^{-13}	1.4×10^{-14}	6.8×10^{-15}
39	4.6×10^{-13}	1.4×10^{-14}	2.9×10^{-15}
54	3.6×10^{-11}	2.5×10^{-14}	3.1×10^{-15}
76	3.0×10^{-12}	5.3×10^{-14}	8.2×10^{-15}
107	2.5×10^{-12}	7.5×10^{-14}	3.1×10^{-14}
150	1.4×10^{-11}	1.9×10^{-13}	3.1×10^{-14}
210	3.3×10^{-11}	2.0×10^{-13}	3.4×10^{-14}
295	4.6×10^{-11}	4.6×10^{-13}	4.5×10^{-14}
413	5.2×10^{-10}	1.1×10^{-12}	5.8×10^{-14}
578	2.2×10^{-09}	1.7×10^{-12}	5.7×10^{-14}
809	5.5×10^{-09}	1.2×10^{-11}	8.2×10^{-14}
1133	5.5×10^{-09}	1.2×10^{-11}	1.3×10^{-13}

is increased, the price of using nonunitary transformations. The residuals for BBEGG also increase, but not as rapidly. It is not known whether BBEGG is backward stable or not. Its accuracy is intermediate between AVW and LAPACK, which is backward stable.

In each case we computed error estimates $|p(\lambda)/p'(\lambda)|$. If we make a table of the maximum of the estimated errors for each run, we get numbers very similar to those in Table 4.1 because roots of random polynomials are (almost always) well conditioned. We have omitted the table for this reason.

TABLE 4.2
Maximum residual after Newton correction.

Degree	AVW	BBEGG	LAPACK
28	4.8×10^{-16}	4.8×10^{-16}	4.2×10^{-16}
39	1.7×10^{-16}	1.7×10^{-16}	1.7×10^{-16}
54	2.6×10^{-16}	2.6×10^{-16}	3.2×10^{-16}
76	3.0×10^{-16}	2.7×10^{-16}	3.0×10^{-16}
107	4.6×10^{-16}	4.6×10^{-16}	4.6×10^{-16}
150	5.3×10^{-16}	6.0×10^{-16}	6.0×10^{-16}
210	1.5×10^{-15}	1.5×10^{-15}	1.5×10^{-15}
295	1.3×10^{-15}	1.3×10^{-15}	1.3×10^{-15}
413	2.4×10^{-15}	2.4×10^{-15}	2.4×10^{-15}
578	1.6×10^{-15}	1.6×10^{-15}	1.6×10^{-15}
809	2.0×10^{-15}	2.0×10^{-15}	2.0×10^{-15}
1133	3.1×10^{-15}	2.4×10^{-15}	2.4×10^{-15}

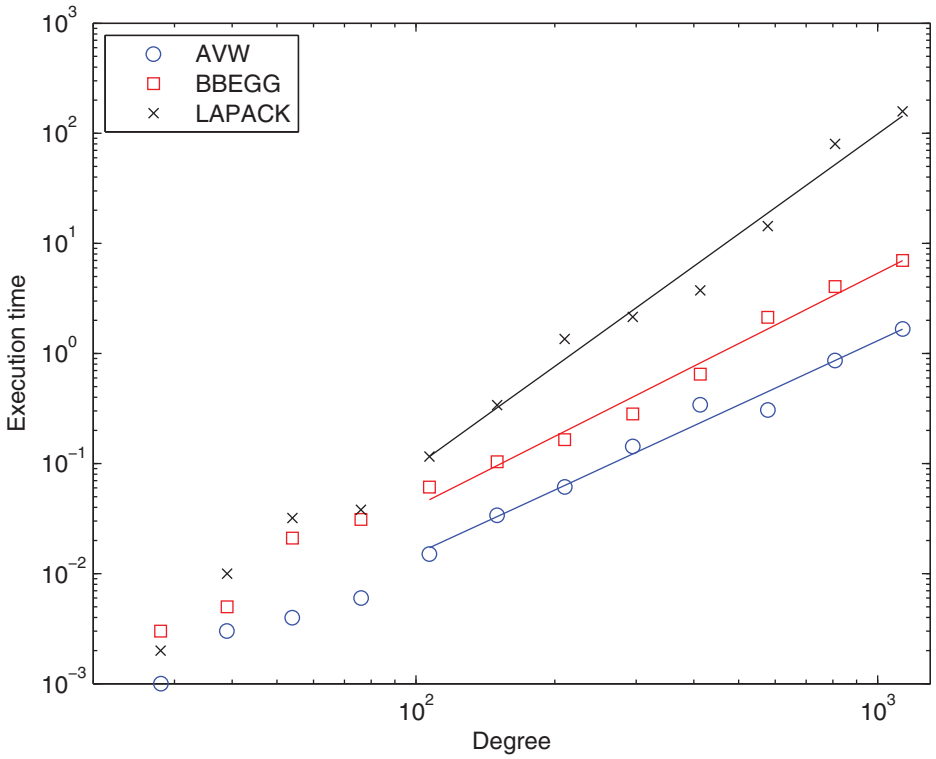


FIG. 4.2. *Execution times for polynomials* $p(x) = x^n - i$ *as a function of degree.*

Table 4.2 shows the maximum residual after one step of Newton's method. In every case the residual is on the level of the unit roundoff. We also computed the error estimates $|p(\lambda)/p'(\lambda)|$ for the corrected roots and got numbers on the order of 10^{-15}, as in Table 4.2.

Example 4.2. Figure 4.2 shows the times for computing the roots of $p(x) = x^n - i$. These results should not be considered typical; rather, they show AVW at its best. This is the rare case in which the companion matrix is unitary. Moreover each of the factors in (3.2) is unitary, and the factors remain unitary (in exact arithmetic) during the iterations. Thus AVW amounts to a fast unitary QR algorithm [6] in this case.

TABLE 4.3
Maximum residual and error for $p(x) = x^{20} + x^{19} + \cdots + 1$.

	AVW	BBEGG	LAPACK
residual	3.8×10^{-15}	9.4×10^{-15}	5.7×10^{-15}
error estimate	2.2×10^{-15}	4.3×10^{-15}	1.9×10^{-15}
actual error	2.1×10^{-15}	4.3×10^{-15}	1.8×10^{-15}

TABLE 4.4
Maximum residual and error for triple root of $p(x) = (x-1)^2(x^{21}-1)$.

	AVW	BBEGG	LAPACK
residual	1.2×10^{-15}	1.6×10^{-16}	2.8×10^{-16}
error estimate	2.5×10^{-06}	1.4×10^{-06}	1.5×10^{-06}
actual error	7.4×10^{-06}	4.3×10^{-06}	4.6×10^{-06}

AVW is faster than BBEGG and LAPACK at all degrees. At degree 1133 AVW is four times as fast as BBEGG and almost 100 times as fast as LAPACK. The slopes of the three least-squares lines are 1.94, 2.12, and 3.02.

On this class of problems, all the codes produced accurate results. Since we know the exact values of the roots, we can compute the actual errors. The maximum error observed over all degrees occurred, not surprisingly, at the highest degree (1133). For AVW, BBEGG, and LAPACK, respectively, the maximum errors were 1.4×10^{-13}, 8.6×10^{-13}, and 1.9×10^{-14}. After one Newton correction, the maximum errors were under 10^{-15} for all three methods.

Tests on low-degree polynomials. We tried out our code on several polynomials of low degree.

Example 4.3. Table 4.3 shows the maximum of residuals (4.1) when we computed the roots of

$$p(x) = x^{20} + x^{19} + \cdots + x + 1.$$

The first row lists the residuals (4.1), and the second row lists the error estimates $|p(\lambda)/p'(\lambda)|$. The exact roots are known to be the 21st roots of unity excluding 1, so we are able to compute the errors in this case. These are shown in the third row of the table. These numbers are all excellent. After a Newton correction, the numbers are even better. All of them are below 10^{-15}.

Example 4.4. The polynomial $p(x) = (x-1)^2(x^{21}-1)$ has a triple root 1, which none of the methods are able to compute accurately. Each of them computes three roots near 1 that are in error by approximately 10^{-6}. More precise values are given in Table 4.4. Note that the error estimate $|p(\lambda)/p'(\lambda)|$ gives a good order-of-magnitude estimate of the error. The residuals (4.1) are all excellent. A Newton correction does not improve these numbers.

The results for the other roots of this polynomial are given in Table 4.5. The numbers for AVW are not as good as we would like. One Newton correction suffices to fix them, giving a maximum residual of 1.4×10^{-15} and maximum error of 8×10^{-16}.

Example 4.5. Table 4.6 shows the maximum of residuals (4.1) and errors when we computed the roots of the 20th degree polynomial

$$p(x) = (x + 2.1)(x + 1.9) \cdots (x - 1.5)(x - 1.7).$$

TABLE 4.5
Maximum residual and error roots of $p(x) = (x-1)^2(x^{21}-1)$*, excluding triple root.*

	AVW	BBEGG	LAPACK
residual	3.7×10^{-10}	1.5×10^{-13}	2.1×10^{-14}
error estimate	3.6×10^{-11}	1.3×10^{-14}	2.5×10^{-15}
actual error	3.6×10^{-11}	1.3×10^{-14}	2.5×10^{-15}

TABLE 4.6
Maximum residual and error for $p(x) = (x+2.1)(x+1.9)\cdots(x-1.5)(x-1.7)$.

	AVW	BBEGG	LAPACK
residual	1.1×10^{-15}	7.0×10^{-17}	4.1×10^{-17}
error estimate	4.2×10^{-11}	3.4×10^{-12}	8.5×10^{-13}
actual error	4.0×10^{-10}	4.1×10^{-10}	4.1×10^{-10}

TABLE 4.7
Residuals and errors for roots of $p(x) = (x+2.1)(x+1.9)\cdots(x-1.5)(x-1.7)$ *computed by AVW.*

Root	Condition	Error	Error estimate	Residual
1.7	1.7×10^{3}	8.0×10^{-13}	7.9×10^{-13}	1.8×10^{-16}
1.5	3.6×10^{3}	9.6×10^{-13}	9.5×10^{-13}	1.1×10^{-16}
1.3	2.8×10^{3}	1.4×10^{-12}	1.8×10^{-12}	2.4×10^{-16}
1.1	1.1×10^{3}	5.6×10^{-13}	5.8×10^{-13}	3.2×10^{-16}
0.9	7.6×10^{2}	2.5×10^{-13}	2.1×10^{-13}	1.7×10^{-16}
0.7	1.5×10^{3}	7.4×10^{-14}	5.2×10^{-14}	1.5×10^{-17}
0.5	3.2×10^{3}	2.8×10^{-14}	1.3×10^{-14}	1.5×10^{-18}
0.3	5.5×10^{3}	5.0×10^{-15}	4.9×10^{-16}	3.1×10^{-20}
0.1	8.0×10^{3}	2.4×10^{-14}	6.7×10^{-17}	2.9×10^{-21}
−0.1	9.6×10^{3}	2.2×10^{-14}	8.3×10^{-16}	2.9×10^{-20}
−0.3	9.5×10^{3}	5.1×10^{-14}	1.8×10^{-14}	6.2×10^{-19}
−0.5	8.0×10^{3}	6.9×10^{-13}	4.3×10^{-14}	1.9×10^{-18}
−0.7	5.9×10^{3}	6.0×10^{-12}	7.6×10^{-13}	4.9×10^{-17}
−0.9	4.7×10^{3}	3.7×10^{-11}	6.2×10^{-12}	7.4×10^{-16}
−1.1	1.2×10^{4}	1.4×10^{-10}	2.5×10^{-11}	1.1×10^{-15}
−1.3	5.6×10^{4}	3.0×10^{-10}	5.4×10^{-11}	3.1×10^{-16}
−1.5	1.6×10^{5}	4.0×10^{-10}	6.3×10^{-11}	9.5×10^{-17}
−1.7	2.6×10^{5}	3.2×10^{-10}	4.0×10^{-11}	3.2×10^{-17}
−1.9	2.1×10^{5}	1.4×10^{-10}	2.9×10^{-11}	2.5×10^{-17}
−2.1	6.6×10^{4}	2.6×10^{-11}	1.1×10^{-11}	2.8×10^{-17}

The maximum errors are consistent with the fact that some of the eigenvalues are ill conditioned with condition numbers approaching 10^6.

Table 4.7 gives a more detailed view, listing data for the individual eigenvalues. We have included only the numbers produced by our code AVW. The column labeled *Condition* gives the condition number of the root, considered as an eigenvalue of the companion matrix. We observe that the actual errors are consistent with the eigenvalue condition numbers, and the error estimates $|p(\lambda)/p'(\lambda)|$ give good order-of-magnitude estimates of the true errors. The Newton correction does not significantly improve these ill-conditioned roots. However, it does improve the residuals slightly. After the correction, all residuals are below 10^{-17}.

Shifting the monomial basis. One potential problem that should not be overlooked arises when a_n, the polynomial's constant coefficient, is near zero. Recall that

TABLE 4.8
Maximum AVW residual for polynomial with tiny constant term, with and without shifting.

Degree	No shift	Shift	Shift+Newton
60	8.0×10^{-2}	1.6×10^{-12}	3.2×10^{-16}
120	9.3×10^{-2}	7.2×10^{-12}	3.8×10^{-16}
240	1.2×10^{-1}	2.5×10^{-10}	5.5×10^{-16}
480	1.1×10^{-1}	5.3×10^{-10}	9.4×10^{-16}
960	2.2×10^{-2}	4.1×10^{-09}	1.5×10^{-15}

if a_n is exactly zero, we can deflate out a zero root. If it is small, but not small enough for a deflation, we have a problem. In the initial factorization (3.2) the last factor (3.1) contains the core transformation

$$\begin{bmatrix} a_{n-1} & a_n \\ 1 & 0 \end{bmatrix}.$$

If a_n is tiny, this matrix is ill conditioned, so we are saddled with ill conditioning right from the start.

This problem can be circumvented by shifting the monomial basis. Since for any shift b

$$p(x) = \sum_{k=0}^{n} \frac{p^{(k)}(b)}{k!}(x-b)^k,$$

we can obtain the coefficients of the shifted polynomial by evaluating the derivatives $p^{(k)}(b)$, $k = 0, \ldots, n$, in about n^2 flops.

Example 4.6. We tested one simple strategy for shifting the basis. If $|a_n|$ is too small, we shift by $b = .001/p'(0) = .001/a_{n-1}$. This strategy is clearly not adequate for all situations, but it suffices to illustrate the potential of the shifting technique. We built random complex polynomials of various degrees n with normally distributed coefficents with mean zero and variance 1 in the real and imaginary parts. We then multiplied the coefficient a_n by 10^{-14} to make the coefficient tiny. Table 4.8 shows residuals (4.1) for the unshifted polynomials versus polynomials shifted by b. The residuals are poor for the unshifted polynomials and good for the shifted ones. We have also shown the residuals after a Newton correction is applied to the results of the shifted case. These are excellent. We have not shown the result of a Newton correction in the unshifted case, as that correction actually makes the residuals worse.

5. Conclusions. We have introduced a new fast method for computing the zeros of a polynomial by a nonunitary analogue of Francis's implicitly shifted QR algorithm that acts on a factored form of the Frobenius companion matrix. For polynomials of high degree our method is very fast. However, speed is obtained at the expense of guarantees of backward stability. For this reason, our method includes an a posteriori residual computation and provides an estimate of the error of each computed root.

Acknowledgment. We thank the editor and two referees for their careful reading of the manuscript and for their questions, criticisms, and suggestions, which lead to significant improvements in the paper.

REFERENCES

[1] D. A. Bini, P. Boito, Y. Eidelman, L. Gemignani, and I. Gohberg, *A fast implicit QR algorithm for companion matrices*, Linear Algebra Appl., 432 (2010), pp. 2006–2031.

[2] D. A. Bini, Y. Eidelman, L. Gemignani, and I. Gohberg, *Fast QR eigenvalue algorithms for Hessenberg matrices which are rank-one perturbations of unitary matrices*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 566–585.

[3] S. Chandrasekaran, M. Gu, J. Xia, and J. Zhu, *A fast QR algorithm for companion matrices*, Oper. Theory Adv. Appl., 179 (2007), pp. 111–143.

[4] M. Fiedler, *A note on companion matrices*, Linear Algebra Appl., 372 (2003), pp. 325–331.

[5] J. G. F. Francis, *The QR transformation, part* II, Computer J., 4 (1961), pp. 332–345.

[6] W. B. Gragg, *The QR algorithm for unitary Hessenberg matrices*, J. Comput. Appl. Math., 16 (1986), pp. 1–8.

[7] R. Vandebril, M. Van Barel, and N. Mastronardi, *Matrix Computations and Semiseparable Matrices, Volume* II: *Eigenvalue and Singular Value Methods*, Johns Hopkins University Press, Baltimore, MD, 2008.

[8] M. Van Barel, R. Vandebril, P. Van Dooren, and K. Frederix, *Implicit double shift QR-algorithm for companion matrices*, Numer. Math., 116 (2010), pp. 177–212.

[9] D. S. Watkins and L. Elsner, *Convergence of algorithms of decomposition type for the eigenvalue problem*, Linear Algebra Appl., 143 (1991), pp. 19–47.

[10] D. S. Watkins, *The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods*, SIAM, Philadelphia, 2007.

[11] D. S. Watkins, *Fundamentals of Matrix Computations*, 3rd ed., John Wiley and Sons, New York, 2010.

[12] D. S. Watkins, *Francis's algorithm*, Amer. Math. Monthly, 118 (2011), pp. 387–403.