BIT Numer Math (2014) 54:7–30
DOI 10.1007/s10543-013-0449-x

BIT

Fast computation of eigenvalues of companion, comrade, and related matrices

Jared L. Aurentz · Raf Vandebril · David S. Watkins

Received: 6 December 2012 / Accepted: 24 September 2013 / Published online: 15 October 2013

Abstract The class of eigenvalue problems for upper Hessenberg matrices of banded-plus-spike form includes companion and comrade matrices as special cases. For this class of matrices a factored form is developed in which the matrix is represented as a product of essentially 2×2 matrices and a banded upper-triangular matrix. A non-unitary analogue of Francis's implicitly-shifted QR algorithm that preserves the factored form and consequently computes the eigenvalues in $O(n^2)$ time and $O(n)$ space is developed. Inexpensive *a posteriori* tests for stability and accuracy are performed as part of the algorithm. The results of numerical experiments are mixed but promising in certain areas. The single-shift version of the code applied to companion matrices is much faster than the nearest competitor.

Keywords Polynomial · Root · Companion matrix · Comrade matrix · LR algorithm

Mathematics Subject Classification 65F15 · 15A18

Communicated by Peter Benner.

The research was partially supported by the Research Council KU Leuven, projects 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).

J.L. Aurentz · D.S. Watkins (✉)
Department of Mathematics, Washington State University, Pullman, WA 99164-3113, USA
e-mail: watkins@math.wsu.edu

J.L. Aurentz
e-mail: jaurentz@math.wsu.edu

R. Vandebril
Department of Computer Science, KU Leuven, 3001 Leuven (Heverlee), Belgium
e-mail: raf.vandebril@cs.kuleuven.be

1 Introduction

This paper is about computing the zeros of polynomials. If a polynomial is expressed in terms of the monomial basis, say

$$p(z) = z^n + c_{n-1}z^{n-1} + \cdots + c_1 z + c_0,$$

its zeros can be found by computing the eigenvalues of the associated companion matrix

$$C = \begin{bmatrix} & & & -c_0 \\ 1 & & & -c_1 \\ & \ddots & & \vdots \\ & & 1 & -c_{n-1} \end{bmatrix}.$$

This is the method used by MATLAB's `roots` command.

Often polynomials are presented in terms of some other basis such as Chebyshev or Legendre polynomials. What happens then? If

$$p(z) = c_0 p_0(z) + c_1 p_1(z) + \cdots + c_n p_n(z),$$

where $p_0, \ldots, p_n$ is a polynomial basis with the property that each p_k has degree exactly k, then one can construct an upper Hessenberg *confederate matrix* [2] whose eigenvalues are the zeros of p. This procedure is cost effective if the (p_k) can be generated by a short recurrence such as the three-term recurrences that produce the classical Legendre, Hermite, and Chebyshev polynomials. In the case of a three-term recurrence, the confederate matrix takes a special form whose pattern of nonzeros is

$$\begin{bmatrix} \times & \times & & & & \times \\ \times & \times & \times & & & \times \\ & \times & \times & \times & & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \end{bmatrix},$$

i.e. tridiagonal plus a spike. This special type of confederate matrix is called a *comrade matrix*.

To find the zeros of p, we just need to compute the eigenvalues of this matrix. One way to do this is to apply the standard algorithm for upper Hessenberg matrices, Francis's implicitly-shifted QR algorithm [10, 17, 18]. This method preserves the upper-Hessenberg form, but it does not exploit the large number of zeros above the main diagonal of the matrix. Consequently the algorithm runs in $O(n^3)$ time and requires $O(n^2)$ storage. It is natural to look for more efficient ways to compute the eigenvalues of this matrix.

Several methods that do the computation in $O(n^2)$ time and $O(n)$ storage have been proposed. These include, for the companion case, Chandrasekaran et al. [6], Bini et al. [3, 4], and Boito et al. [5]. All of these methods are implementations of Francis's implicitly shifted QR algorithm that exploit the quasiseparable structure,

i.e. low-rank structure of the submatrices above the subdiagonal. For the comrade case we note the work of Eidelman, Gemignani, and Gohberg [7], Vandebril and Del Corso [14], and Zhlobich [19]. These are also implementations of Francis's algorithm that exploit the quasiseparable structure, except that the Zhlobich method uses the differential qd algorithm of Fernando and Parlett [8, 12, 16].

This paper is an outgrowth of earlier work [1] that was specific to the companion matrix. We did not make direct use of the quasiseparable structure. Instead we used a Fiedler factorization [9] of the companion matrix into a product of essentially 2×2 matrices. We then developed a non-unitary variant of Francis's algorithm that operates on the factored form and preserves it. The new algorithm requires $O(n)$ storage and runs in $O(n^2)$ time. In [1] it was shown to be three to four times faster than the nearest competition for large n.

In this work we consider the broader class of upper Hessenberg matrices having the banded-plus-spike structure, which includes both comrade and companion matrices as special cases. We will produce a factored form of the matrix into a product of essentially 2×2 matrices times a banded upper-triangular matrix that allows us to store it in $O(n)$ space. We will develop a non-unitary variant of Francis's algorithm that operates on the factored form and preserves it. Thus the total memory requirement is $O(n)$. More precisely, if the band width (recurrence length) is b, we need to store $(4 + b)n$ numbers. This is significantly less than is needed for a quasiseparable generator representation. Furthermore the cost of executing the algorithm will be shown to be $O(n)$ flops per iteration, $(12 + 4b)n$ for the single-shift case and $(28 + 8b)n$ for the double-shift case. Making the usual (reasonable) assumption that $O(1)$ iterations are needed per root, the total flop count is $O(n^2)$. In the companion case our new method is about three times faster than our previous method [1] and more than ten times faster than all other competitors.

The price we pay for extreme speed is that we sacrifice stability. The similarity transformations that we use are Gauss transforms, unit lower-triangular matrices, hence potentially unstable. Since the total similarity transformation for each iteration is a product of unit lower-triangular matrices, it is itself unit lower triangular. Thus our algorithm is an *LR* algorithm. More precisely, it is a structure-preserving implicitly-shifted *LR* algorithm.

The instability does not imply that our algorithm is not useful. Instead it means that once we have computed our roots, we must perform *a posteriori* tests to determine whether or not they can be trusted. In Sect. 6 two inexpensive tests are discussed. One is a residual test that measures backward stability, and the other is an estimate of the error in the computed root. The latter also furnishes us with enough information to do a step of Newton's method and thereby refine the root.

The numerical results in Sect. 7 give a preliminary indication of where our method might be useful. Our single-shift code shows promise. It is much faster than any competitors that we are aware of. The accuracy is satisfactory, especially after a Newton correction. The double-shift code was less successful. It too is very fast, but it is too unstable in its current form. We cannot recommend it for accurate computation of roots of high-degree polynomials. It could find use as an extremely fast method to get a rough estimate of the spectrum.

2 Confederate matrices

Results equivalent to the following appeared years ago in the book by Barnett [2]. We include brief sketches of proofs for the reader's convenience. Let $p(z)$ be any polynomial of degree n whose roots we would like to find.

Let $p_0(z), p_1(z), \ldots, p_n(z)$ be polynomials with complex coefficients such that each p_k has degree exactly k. These polynomials can otherwise be completely arbitrary, but we are mainly interested in ones that can be generated by a short recurrence, for example

$$\alpha_k p_k(z) = z p_{k-1}(z) - \beta_k p_{k-1}(z) - \gamma_k p_{k-2}(z). \tag{2.1}$$

Since the polynomial $p(z)$ whose roots we are looking for has degree exactly n, it can be written as a linear combination

$$p(z) = c_0 p_0(z) + \cdots + c_n p_n(z) \tag{2.2}$$

for unique coefficients $c_0, \ldots, c_n$ with $c_n \neq 0$. We need just a bit more notation. Let $\eta_0, \ldots, \eta_n$, and η be the unique nonzero constants such that $\eta_0 p_0, \ldots, \eta_n p_n$, and ηp are monic.

We consider two closely related problems. (a) Build a matrix A such that for $k = 1, \ldots, n$ the leading principal submatrix of A of order k has $\eta_k p_k$ as its characteristic polynomial. (b) Build a matrix B as in problem (a), but replace $\eta_n p_n$ by ηp. Of course there is no difference between these two problems in general, but if $p_0, \ldots, p_n$ are generated by a short recurrence like (2.1) while p is given by the long expression (2.2), there will be a difference in the structure of the matrices. We will produce upper Hessenberg matrices that solve problems (a) and (b).

For each k there are unique coefficients $a_{1k}, \ldots, a_{k+1,k}$, with $a_{k+1,k} \neq 0$, such that

$$z p_{k-1}(z) = \sum_{j=1}^{k+1} p_{j-1}(z)\, a_{jk}. \tag{2.3}$$

The subscripts on the coefficients are offset by one because the a_{jk} are about to become entries of a matrix.

Theorem 2.1 *Let A_n be the $n \times n$ upper Hessenberg matrix*

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & & a_{2n} \\ & a_{32} & a_{33} & & a_{3n} \\ & & \ddots & \ddots & \vdots \\ & & & a_{n,n-1} & a_{nn} \end{bmatrix}$$

generated from the coefficients of (2.3) for $k = 1, \ldots, n$. Then, for $k = 1, \ldots, n$, the characteristic polynomial of A_k, the $k \times k$ leading principal submatrix of A_n, is $\eta_k p_k(z)$, where η_k is a complex constant.

Proof We just sketch the proof. Rewrite (2.3) as a recurrence

$$p_k(z)a_{k+1,k} = zp_{k-1}(z) - \sum_{j=1}^{k} p_{j-1}(z)\, a_{jk}. \tag{2.4}$$

Then rewrite it as a recurrence for the monic polynomials $\eta_k p_k$. Next compute the determinant $\chi_k(z) = \det(zI - A_k)$ by expanding on the last column. This yields a recurrence that is identical to the monic version of (2.4). Therefore the monic polynomials $\eta_k p_k$ are the characteristic polynomials of A_k, $k = 1, \ldots, n$.

A second proof, which gives eigenvector information, can be obtained by writing (2.3) for $k = 1, \ldots, m$ as a single matrix equation. Let

$$w_m(z) = \begin{bmatrix} p_0(z) & \cdots & p_{m-1}(z) \end{bmatrix}. \tag{2.5}$$

Then, from (2.3)

$$zw_m(z) = w_m(z)A_m + p_m(z)a_{m+1,m}e_m^T,$$

an Arnoldi configuration. If we now take $z = z_i$ to be a zero of p_m, we get

$$z_i w_m(z_i) = w_m(z_i)A_m,$$

which shows that $w_m(z_i)$ is a left eigenvector of A_m with eigenvalue z_i. If p_m has m distinct roots $z_1, \ldots, z_m$, then these are the m eigenvalues of A_m. It follows that the characteristic polynomial of A_m is $\eta_m p_m$.

If A_m has multiple eigenvalues, a more elaborate argument is needed. We must take derivatives of (2.3) and build Jordan blocks. □

Theorem 2.1 solves problem (a). To solve problem (b) we just need to replace p_n by p. Starting from the case $k = n$ of (2.3), using (2.2) to eliminate p_n, and defining

$$\hat{c}_k = a_{n+1,n}c_k/c_n, \quad k = 0, \ldots, n-1,$$

we find that

$$zp_{n-1}(z) = \sum_{j=1}^{n} p_{j-1}(z)(a_{jn} - \hat{c}_{j-1}) + (a_{n+1,n}/c_n)p(z).$$

This is the equation we have to use in place of (2.3) to form the nth column. Thus

$$B = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} - \hat{c}_0 \\ a_{21} & a_{22} & a_{23} & & a_{2n} - \hat{c}_1 \\ & a_{32} & a_{33} & & a_{3n} - \hat{c}_2 \\ & & \ddots & \ddots & \vdots \\ & & & a_{n,n-1} & a_{nn} - \hat{c}_{n-1} \end{bmatrix} \tag{2.6}$$

is the solution of problem (b). Barnett [2] calls this a *confederate matrix*. Its importance to us is that its eigenvalues are the zeros of p.

Now consider the special case where the polynomials p_k are generated by the three-term recurrence (2.1). Now (2.3) takes the form

$$zp_{k-1}(z) = \gamma_k p_{k-2}(z) + \beta_k p_{k-1}(z) + \alpha_k p_k(z),$$

and A (A_n of Theorem 2.1) has the tridiagonal form

$$A = \begin{bmatrix} \beta_1 & \gamma_2 & 0 & \cdots & 0 \\ \alpha_1 & \beta_2 & \gamma_3 & & 0 \\ & \alpha_2 & \beta_3 & \ddots & \vdots \\ & & \ddots & \ddots & \gamma_n \\ & & & \alpha_{n-1} & \beta_n \end{bmatrix},$$

and B is tridiagonal plus a spike:

$$B = \begin{bmatrix} \beta_1 & \gamma_2 & 0 & \cdots & -\hat{c}_0 \\ \alpha_1 & \beta_2 & \gamma_3 & & -\hat{c}_1 \\ & \alpha_2 & \beta_3 & \ddots & \vdots \\ & & \ddots & \ddots & \gamma_n - \hat{c}_{n-2} \\ & & & \alpha_{n-1} & \beta_n - \hat{c}_{n-1} \end{bmatrix}.$$

This is a *comrade matrix* [2]. In the special case of a Newton basis:

$$p_k(z) = (z - \beta_k)p_{k-1}(z),$$

B is bidiagonal plus a spike. In the even more special case of the monomial basis $p_k(z) = z^k$, which is a Newton basis with all $\beta_k = 0$, B becomes a companion matrix.

3 The matrix decomposition

We find it convenient to work with matrices that have the spike in the first row instead of the last column. We consider the class of upper Hessenberg matrices that are upper banded except for a spike of nonzero entries in the first row. More precisely, given a nonnegative integer b, we consider all A for which $a_{ij} = 0$ when $i - j > 1$, or $i > 1$ and $j - i \geq b$. The matrices we considered in the previous section all fall into this category after reversing rows and columns and transposing. The cases $b = 0$, 1, and 2 correspond to companion, Newton, and comrade matrices, respectively.

The matrix decomposition that we will use is obtained by Gaussian elimination with a row interchange at each step. The first step explains the whole algorithm. For illustration we consider the case $n = 6$ and $b = 2$, in which the matrix has the form

$$B = \begin{bmatrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \\ a_1 & d_1 & b_1 & & & \\ & a_2 & d_2 & b_2 & & \\ & & a_3 & d_3 & b_3 & \\ & & & a_4 & d_4 & b_4 \\ & & & & a_5 & d_5 \end{bmatrix}. \tag{3.1}$$

At step 1 we multiply A on the left by a matrix C_1^{-1} that has the effect of interchanging the first two rows, then subtracting c_1/a_1 times the new first row from the second. The result is

$$C_1^{-1}A = \begin{bmatrix} a_1 & d_1 & b_1 & & & \\ 0 & \hat{c}_2 & \check{c}_3 & c_4 & c_5 & c_6 \\ & a_2 & d_2 & b_2 & & \\ & & a_3 & d_3 & b_3 & \\ & & & a_4 & d_4 & b_4 \\ & & & & a_5 & d_5 \end{bmatrix}.$$

$C_1 = \mathrm{diag}\{\tilde{C}_1, I_4\}$, where

$$\tilde{C}_1 = \begin{bmatrix} c_1/a_1 & 1 \\ 1 & 0 \end{bmatrix}.$$

The second step is just like the first, resulting in

$$C_2^{-1}C_1^{-1}A = \begin{bmatrix} a_1 & d_1 & b_1 & & & \\ & a_2 & d_2 & b_2 & & \\ & 0 & \hat{c}_3 & \check{c}_4 & c_5 & c_6 \\ & & a_3 & d_3 & b_3 & \\ & & & a_4 & d_4 & b_4 \\ & & & & a_5 & d_5 \end{bmatrix}.$$

$C_2 = \mathrm{diag}\{1, \tilde{C}_2, I_3\}$, where

$$\tilde{C}_2 = \begin{bmatrix} \hat{c}_2/a_2 & 1 \\ 1 & 0 \end{bmatrix}.$$

After five steps our decomposition is finished.

$$A = C_1 \cdots C_5 \begin{bmatrix} a_1 & d_1 & b_1 & & & \\ & a_2 & d_2 & b_2 & & \\ & & a_3 & d_3 & b_3 & \\ & & & a_4 & d_4 & b_4 \\ & & & & a_5 & d_5 \\ & & & & & \hat{c}_6 \end{bmatrix}. \tag{3.2}$$

In the general $n \times n$ case the factorization has the form

$$A = C_1 C_2 \cdots C_{n-1} R, \tag{3.3}$$

where R is a banded upper-triangular matrix with band width $b+1$. Each C_i is a core transformation (defined immediately below) acting on rows i and $i+1$ with active part of the form (let $\hat{c}_1 = c_1$)

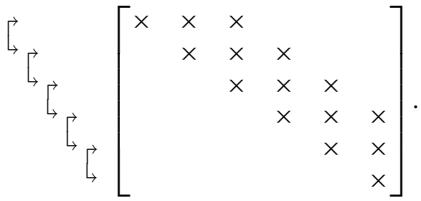
$$\tilde{C}_i = \begin{bmatrix} \hat{c}_i/a_i & 1 \\ 1 & 0 \end{bmatrix}.$$

The total flop count for the reduction is approximately $2nb$.

4 Operations on core transformations

A *core transformation* is a nonsingular $n \times n$ matrix C_i that "acts on two adjacent rows or columns". More precisely, C_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. It is called the *active part* of the core transformation. Core transformations will normally be written, as shown here, with a subscript i that indicates that the transformation acts on rows i and $i+1$.

We find it convenient to introduce a compact notation for core transformations. For example, we will depict the factorization (3.2) as



Each double arrow represents a core transformation C_i, with the arrows pointing to rows i and $i+1$, the rows upon which the core transformation acts.

A core transformation that is unit lower triangular will be called a *Gauss transform*. The active part of a Gauss transform has the form

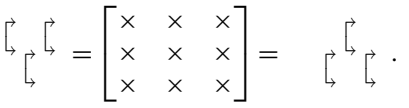
$$\begin{bmatrix} 1 & 0 \\ m & 1 \end{bmatrix}.$$

Multiplying a matrix A by a Gauss transform G_i on the left has the effect of adding m times the ith row of A to the $(i+1)$st row.

Notice that two core transformations C_i and B_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 three types of operations on core transformations, called *fusion*, *turnover*, and *passing through*.

Fusion If we have two core transformations C_i and B_i that act on the same two rows and columns, their product $C_i B_i$ is also a core transformation. The act of multiplying two such core transformations together to form a single core transformation is a *fusion*.

Turnover Suppose we have three adjacent core transformations $A_i B_{i+1} G_i$. The *turnover* operation rewrites this product in the form $\hat{G}_{i+1} \hat{A}_i \hat{B}_{i+1}$. Schematically

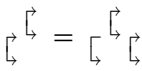


Here we are depicting rows (and columns) i through $i+2$, which is where all the action is. The three core transformations on the left are A_i, B_{i+1}, and G_i. They can be multiplied together to form a matrix whose active part is 3×3, as depicted in the

middle. This 3×3 matrix is then factored into a product of three core transformations $\hat{G}_{i+1}$, $\hat{A}_i$, and $\hat{B}_{i+1}$, as shown on the right.

Such a factorization is always possible if the core transformations are unitary. This property of unitary matrices is well known and has been exploited in many algorithms [6, 13, 15]. In the nonunitary case it is *almost* always possible. For a discussion of turnover operations in general see [1].

In this paper we will only use special types of turnover operations, those for which the resulting $\hat{G}_{i+1}$ is a Gauss transform. We call these *turnover operations that produce a Gauss transform on the left*. Such operations can be denoted schematically by



Since Gauss transforms only alter one row, we depict the Gauss transform here with a single arrowhead.

To figure out when this is possible, suppose the active part of the product $A_i B_{i+1} G_i$ is

$$\begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{bmatrix}.$$

If we wish to factor this into a product with a Gauss transform

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & \hat{m} & 1 \end{bmatrix}$$

on the left, we must have

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & -\hat{m} & 1 \end{bmatrix} \begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{bmatrix} = \begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ 0 & \hat{s}_{32} & \hat{s}_{33} \end{bmatrix}.$$

If $s_{31} = 0$, this can be realized by taking $\hat{m} = 0$. Otherwise we must take $\hat{m} = s_{31}/s_{21}$. This fails if $s_{21} = 0$, and in this case we cannot do the turnover. Now suppose $s_{21} \neq 0$. Then we can complete the turnover if (and only if) we are able to compute a factorization

$$\begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ 0 & \hat{s}_{32} & \hat{s}_{33} \end{bmatrix} = \begin{bmatrix} \hat{a}_{11} & \hat{a}_{12} & \\ \hat{a}_{21} & \hat{a}_{22} & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & \hat{b}_{22} & \hat{b}_{23} \\ & \hat{b}_{32} & \hat{b}_{33} \end{bmatrix}.$$

One easily checks that such a factorization is possible if and only if the submatrix

$$\begin{bmatrix} s_{12} & s_{13} \\ s_{22} & s_{23} \end{bmatrix}$$

has rank one. To get an idea when this condition will hold, consider the product of three core transformations $A_i B_{i+1} G_i$, whose 3×3 active part has the form

$$\begin{bmatrix} a_{11}g_{11} + a_{12}b_{22}g_{21} & a_{11}g_{12} + a_{12}b_{22}g_{22} & a_{12}b_{23} \\ a_{21}g_{11} + a_{22}b_{22}g_{21} & a_{21}g_{12} + a_{22}b_{22}g_{22} & a_{22}b_{23} \\ b_{32}g_{21} & b_{32}g_{22} & b_{33} \end{bmatrix}.$$

The relevant 2×2 submatrix is

$$\begin{bmatrix} a_{11}g_{12} + a_{12}b_{22}g_{22} & a_{12}b_{23} \\ a_{21}g_{12} + a_{22}b_{22}g_{22} & a_{22}b_{23} \end{bmatrix},$$

which has determinant $\det(A_i)g_{12}b_{23}$. We conclude that it has rank one if and only if either $g_{12} = 0$ or $b_{23} = 0$. From this we conclude that we will be able to do a turnover producing a Gauss transform on the left if either B_{i+1} or G_i is a Gauss transform.

Turnover, type 1 The type of turnover that is most used by our algorithms is one for which G_i is a Gauss transform and the $\hat{G}_{i+1}$ that is produced is a Gauss transform. Thus we start with

$$\begin{bmatrix} a_{11} & a_{12} & \\ a_{21} & a_{22} & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & b_{22} & b_{23} \\ & b_{32} & b_{33} \end{bmatrix} \begin{bmatrix} 1 & & \\ g_{21} & 1 & \\ & & 1 \end{bmatrix}$$

and end with

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & \hat{g}_{32} & 1 \end{bmatrix} \begin{bmatrix} \hat{a}_{11} & \hat{a}_{12} & \\ \hat{a}_{21} & \hat{a}_{22} & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & \hat{b}_{22} & \hat{b}_{23} \\ & \hat{b}_{32} & \hat{b}_{33} \end{bmatrix}.$$

Equating the two products we have

$$\begin{bmatrix} a_{11} + a_{12}b_{22}g_{21} & a_{12}b_{22} & a_{12}b_{23} \\ a_{21} + a_{22}b_{22}g_{21} & a_{22}b_{22} & a_{22}b_{23} \\ b_{32}g_{21} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} \hat{a}_{11} & \hat{a}_{12}\hat{b}_{22} & \hat{a}_{12}\hat{b}_{23} \\ \hat{a}_{21} & \hat{a}_{22}\hat{b}_{22} & \hat{a}_{22}\hat{b}_{23} \\ \hat{g}_{32}\hat{a}_{21} & \hat{b}_{32} + \hat{g}_{32}\hat{a}_{22}\hat{b}_{22} & \hat{b}_{33} + \hat{g}_{32}\hat{a}_{22}\hat{b}_{23} \end{bmatrix},$$

showing that we can compute the turnover by the operations

$$\begin{aligned} \hat{a}_{11} &= a_{11} + a_{12}(b_{22}g_{21}), & \hat{a}_{12} &= a_{12}, \\ \hat{a}_{21} &= a_{21} + a_{22}(b_{22}g_{21}), & \hat{a}_{22} &= a_{22}, \\ \hat{g}_{32} &= b_{32}g_{21}/\hat{a}_{21}, & & \\ \hat{b}_{22} &= b_{22}, & \hat{b}_{32} &= b_{32} - (\hat{g}_{32}a_{22})b_{22}, \\ \hat{b}_{23} &= b_{23}, & \hat{b}_{33} &= b_{33} - (\hat{g}_{32}a_{22})b_{23}. \end{aligned}$$

These computations require 12 flops. This type of turnover is used in both the single- and double-shift algorithms.

Turnover, type 2 In the double-shift algorithm, two other types of turnover will be used. The type-2 turnover starts with A_i and B_{i+1} Gauss transforms and ends with $\hat{G}_{i+1}$ and $\hat{B}_{i+1}$ Gauss transforms. Thus we start with

$$\begin{bmatrix} 1 & & \\ a_{21} & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & b_{32} & 1 \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} & \\ g_{21} & g_{22} & \\ & & 1 \end{bmatrix}$$

and end with

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & \hat{g}_{32} & 1 \end{bmatrix} \begin{bmatrix} \hat{a}_{11} & \hat{a}_{12} & \\ \hat{a}_{21} & \hat{a}_{22} & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & \hat{b}_{32} & 1 \end{bmatrix}.$$

Equating the two products we have

$$\begin{bmatrix} g_{11} & g_{12} & \\ g_{21} + a_{21} g_{11} & g_{22} + a_{21} g_{12} & \\ b_{32} g_{21} & b_{32} g_{22} & 1 \end{bmatrix} = \begin{bmatrix} \hat{a}_{11} & \hat{a}_{12} & \\ \hat{a}_{21} & \hat{a}_{22} & \\ \hat{g}_{32}\hat{a}_{21} & \hat{b}_{32} + \hat{g}_{32}\hat{a}_{22} & 1 \end{bmatrix},$$

showing that we can compute the turnover by the operations

$$\begin{bmatrix} \hat{a}_{11} & \hat{a}_{12} \\ \hat{a}_{21} & \hat{a}_{22} \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} + a_{21} g_{11} & g_{22} + a_{21} g_{12} \end{bmatrix},$$

$$\hat{g}_{32} = b_{32} g_{21} / \hat{a}_{21}, \quad \text{and} \quad \hat{b}_{32} = b_{32} g_{22} - \hat{g}_{32} \hat{a}_{22}.$$

This costs 9 flops.

Turnover, type 3 The one other kind of turnover that we will need starts with three Gauss transforms and ends with three Gauss transforms. We begin with

$$\begin{bmatrix} 1 & & \\ a_{21} & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & b_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ g_{21} & 1 & \\ & & 1 \end{bmatrix},$$

and wish to end with

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & \hat{g}_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ \hat{a}_{21} & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & \hat{b}_{32} & 1 \end{bmatrix}.$$

For this we require

$$\begin{bmatrix} 1 & & \\ a_{21} + g_{21} & 1 & \\ b_{32} g_{21} & b_{32} & 1 \end{bmatrix} = \begin{bmatrix} 1 & & \\ \hat{a}_{21} & 1 & \\ \hat{g}_{32}\hat{a}_{21} & \hat{g}_{32} + \hat{b}_{32} & 1 \end{bmatrix},$$

which is achieved by taking

$$\hat{a}_{21} = a_{21} + g_{21}, \qquad \hat{g}_{32} = b_{32} g_{21} / \hat{a}_{21}, \quad \text{and} \quad \hat{b}_{32} = b_{32} - \hat{g}_{32}.$$

This costs 4 flops.

Passing through The one other ingredient that we will need is the *passing through* operation, in which a core transformation is "passed through" an upper-triangular matrix. We start with a product RG_i and end with $\hat{G}_i \hat{R}$, where R and $\hat{R}$ are upper triangular and G_i and $\hat{G}_i$ are Gauss transforms. The multiplication of R by G_i on the right amounts to subtracting m times the $(i+1)$st column of R to the ith. This disturbs the triangular form, creating a nonzero element $m r_{i+1,i+1}$ in position $(i+1, i)$. This element can be eliminated by a Gauss transform $\hat{G}_i^{-1}$ on the left, provided that $r_{ii} \neq 0$. The appropriate multiplier (defining $\hat{G}_i$) is $\hat{m} = m r_{i+1,i+1} / r_{ii}$. The transformation $RG_i \rightarrow \hat{G}_i^{-1} RG_i = \hat{T}$ restores the triangular form by subtracting $\hat{m}$ times row i from row $i+1$.

In our algorithm the matrix R will not just be upper triangular; it will also be upper banded as in (3.3). It is clear that the passing through operation does not increase the bandwidth. This key result depends on the fact that the core transformation is a Gauss transform. If the bandwidth of R is b, the passing through operation costs about $4b$ flops.

5 Bulge chasing on the condensed form

The matrix A in (3.1) is upper Hessenberg, so its eigenvalues can be found by a bulge-chasing algorithm such as Francis's implicitly-shifted QR algorithm or one of its nonunitary analogs. We will achieve economy by operating on the factored form (3.3) of A.

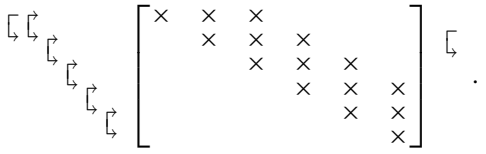
Single-shift case Given a single shift ρ, we set the iteration in motion by a similarity transformation

$$A \rightarrow L_1^{-1} A L_1 = A_1,$$

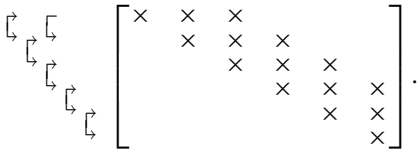
where L_1 is a Gauss transform whose first column is proportional to $(A - \rho I)e_1$, which is

$$\begin{bmatrix} a_{11} - \rho & a_{21} & 0 & \cdots & 0 \end{bmatrix}^T.$$

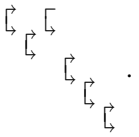
See [16, § 4.5]. This succeeds as long as we choose ρ so that $\rho \neq a_{11}$. The entries a_{11} and a_{21} are readily obtained by multiplying the first column of the core transformation C_1 by the $(1,1)$ entry of R. The situation immediately after the initial similarity transformation can be depicted by



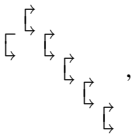
The transformation L_1^{-1} on the left can be fused with its neighbor C_1 to form a single core transformation. The transformation L_1 on the right can be passed through the upper-triangular matrix to bring it into contact with the other core transformations, resulting in



The algorithm will proceed by a sequence of similarity transformations by Gauss transforms. After each of these, the Gauss transform that ends up on the right will immediately be passed through the triangular matrix to bring it into contact with the other core transformations. Bearing this in mind, our depictions will leave out the triangular matrix from this point on for simplicity. Thus we will write



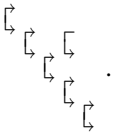
The next step is to do a type-1 turnover of the three transformations on the left to obtain



with a Gauss transform, which we will call L_2, on the left. Next a similarity transformation

$$A_1 \to L_2^{-1} A_1 L_2 = A_2$$

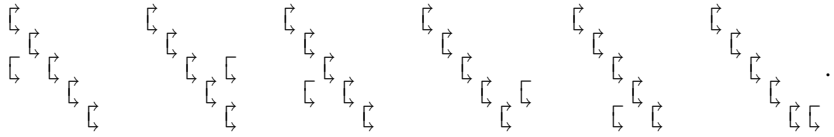
removes L_2 from the left and makes it appear on the right. We pass L_2 through the upper-triangular matrix at a cost of $4b$ flops to obtain



Now another turnover causes a new Gauss transform L_3 to appear on the left, and this is then moved to the right by another similarity transformation

$$A_2 \to L_3^{-1} A_2 L_3 = A_3.$$

The pattern of the iteration is now clear:



Once the Gauss transform has been chased to the bottom right corner, it can be fused with the adjacent core transformation, completing the iteration. The result is an upper Hessenberg matrix

$$\hat{A} = L_{n-1}^{-1} \cdots L_1^{-1} A L_1 \cdots L_{n-1} = L^{-1} A L$$

in the factored form (3.3). The first column of L is the same as the first column of L_1, which is proportional to $(A - \rho I)e_1$. By Theorem 4.5.5 of [16], this bulge chase effects an iteration of the generic GR algorithm. Since each of the transforming matrices L_i is unit lower triangular, their product L is also unit lower triangular Thus this is specifically an LR iteration. Its merit is that it perserves the structure (3.3).

To get a flop count for the iteration, note first that almost all of the work is in the turnover and pass-through operations. There are $n-2$ turnovers at 12 flops each, and there are $n-1$ passing-through operations at approximately $4b$ flops each. Thus the total flop count for one iteration is about $(12+4b)n$.

Double-shift case To start a double step on A in the form (3.3), given two shifts ρ_1 and ρ_2, we begin by computing

$$x = (A - \rho_1 I)(A - \rho_2 I)e_1 = A^2 e_1 - (\rho_1 + \rho_2)Ae_1 + \rho_1 \rho_2 e_1,$$

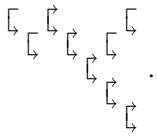
which has nonzero entries only in the first three positions. The nonzero part is

$$\begin{bmatrix} a_{11}^2 + a_{12}a_{21} - (\rho_1 + \rho_2)a_{11} + \rho_1\rho_2 \\ a_{21}a_{11} + a_{22}a_{21} - (\rho_1 + \rho_2)a_{21} \\ a_{32}a_{21} \end{bmatrix}.$$

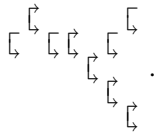
This costs $O(1)$ flops. All of the information needed for this computation is in the first two core transformations C_1 and C_2, and the upper-left-hand 2×2 submatrix of R in (3.3). If ρ_1 and ρ_2 are chosen so that the first two entries of x are nonzero, we can continue. We build Gauss transforms L_2 and L_1 such that $L_1^{-1}L_2^{-1}x = \alpha e_1$, i.e. $L_2L_1e_1 = \alpha^{-1}x$. This means that the first column of L_2L_1 is proportional to x. Then we do the similarity transform

$$A \to L_1^{-1}L_2^{-1}AL_2L_1,$$

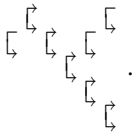
which can be pictured as



Here we continue to suppress the upper triangular matrix. Do a type-2 turnover of the three leftmost transformations to obtain

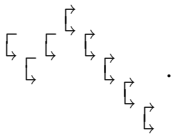


Then do a fusion:

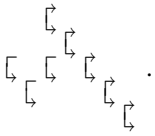


The three core transformations that are disturbing the upper Hessenberg form are all Gauss transforms. We need to chase them to the bottom of the matrix to get rid of them.

The transformations on the right can be moved to the left by two type-1 turnover operations:



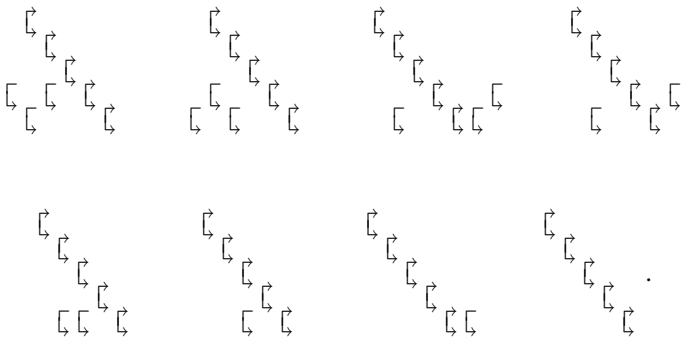
Then there are (at least) two options. (1) The more obvious approach is to do a similarity transformation that removes all three Gauss transforms from the left and makes them appear on the right. Then they can all be passed through the triangular part and then through the spine of core transformations by three turnover operations. (2) A less obvious approach is to do a preliminary type-3 turnover of the three Gauss transforms. This costs only four flops and results in three new Gauss transforms. Then we do a similarity transformation that removes only the two leftmost Gauss transforms and makes them appear on the right. Then we pass these through the upper-triangular part and use two turnovers of type 1 to pass them through the spine of core transformations. We choose the second method because it is cheaper at $28 + 8b$ flops, versus $36 + 12b$ flops for the first method. Whichever method we choose, we end up with the configuration



The three Gauss transforms have been moved down by one position.

Repeating this procedure some $n - 4$ times, we bring the Gauss transforms to the bottom. We then eliminate each Gauss transform by fusion at the first opportunity.

The last few steps have the form



The steps are, in order, type-3 turnover, similarity, fusion, type-1 turnover, fusion, similarity, fusion. The matrix has been returned to the condensed form (3.3), and the iteration is complete.

The complete bulge-chasing step effects a similarity transformation

$$\hat{A} = L^{-1}AL,$$

where L is a product of a large number of Gauss core transformations. More precisely, $L = L_2L_1\tilde{L}$, where L_2 and L_1 are the Gauss transforms that initiated the step, and $\tilde{L}$ is the product of all of the similarity transformations in the bulge chase. Since $\tilde{L}e_1 = e_1$, we see that $Le_1 = L_2L_1e_1 = \alpha^{-1}x$. That is, the first column of L is proportional to the first column of $(A-\rho_1 I)(A-\rho_2 I)$. By Theorem 4.5.5 of [16], this bulge chase effects an implicit double step of the *LR* algorithm.

If A is real, and the shifts ρ_1 and ρ_2 are either real or a complex conjugate pair, then the vector $x = (A-\rho_1 I)(A-\rho_2 I)e_1$ is real, and the entire iteration can be done in real arithmetic.

The total flop count for an iteration is approximately $(28+8b)n$ flops. This is more than twice the $(12+4b)n$ flop count for a single step. However, in the real case the double step is more economical because the arithmetic is real, not complex.

Shift strategies and deflation criteria Although our matrices are stored in a compressed form, it is nevertheless easy to assemble the parts needed in order to apply standard shift strategies and deflation criteria.

For both single- and double-shift codes, we compute the eigenvalues of the lower-right-hand 2×2 submatrix. In the double-shift code we take those two numbers as shifts for the double step. In the single-shift code, we take the one that is closer to $a_{n,n-1}$ as the shift. In both codes we interject occasional exceptional shifts if too many iterations have passed without a deflation.

The deflation criterion is standard. We set $a_{j+1,j}$ to zero if

$$|a_{j+1,j}| < u\big(|a_{jj}| + |a_{j+1,j+1}|\big),$$

where u is the unit roundoff.

6 Tests of quality of the computed roots

At each turnover there is the possibility of a breakdown, so the iterations can sometimes fail. Our experience is that this almost never happens, but a remedy is to start the iteration over with a different set of shifts. A more serious threat is that a near breakdown occurs, leading to inaccurate results. It is difficult to predict such events. Therefore there is a need for *a posteriori* tests to measure the quality of the computed results. Fortunately some inexpensive tests are available.

For each computed root λ, the corresponding right eigenvector is

$$v = \begin{bmatrix} p_{n-1}(\lambda) \\ p_{n-2}(\lambda) \\ \vdots \\ p_1(\lambda) \\ p_0(\lambda) \end{bmatrix}.$$

To see this, compare with (2.5), and take into account that the matrix has been flipped over and transposed. These can be computed in $O(nb)$ flops using the short recurrence. Then we can compute the residual $\|(\lambda I - A)v\|$. This is an inexpensive computation because of the structure, in fact

$$(\lambda I - A)v = \begin{bmatrix} \alpha_n p(\lambda)/c_n \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

We can compute $p(\lambda)$ in $2n$ flops using (2.2). In an actual residual computation, the zero components would be nonzero due to the roundoff errors made in computing the numbers $p_k(\lambda)$ using the short recurrence. We ignore these small numbers, which are of the order of the unit roundoff with respect to the numbers $p_0(\lambda), \ldots, p_{n-1}(\lambda)$. To get a dimensionless number, we compute

$$\big\|(\lambda I - A)v\big\|_\infty \big/ \big(\|A\|_\infty \|v\|_\infty\big) = \big|\alpha_n p(\lambda)/c_n\big| \big/ \big(\|A\|_\infty \|v\|_\infty\big).$$

This residual is a measure of backward stability of the computation of λ. We always compute this quantity for each λ.

Differentiating the short recurrence, we can get the quantities $p_k'(\lambda)$ as well. For example, if the recurrence is

$$\alpha_k p_k(z) = z p_{k-1}(z) - \beta_k p_{k-1}(z) - \gamma_k p_{k-2}(z),$$

we have

$$\alpha_k p_k'(z) = p_{k-1}(z) + (z - \beta_k) p_{k-1}'(z) - \gamma_k p_{k-2}'(z),$$

which allows us to compute $p_1'(\lambda), \ldots, p_{n-1}'(\lambda)$ in $O(nb)$ flops. We can then compute $p'(\lambda)$ in $2n$ flops using the derivative of (2.2). The quantity $1/|p'(\lambda)|$ is a measure of

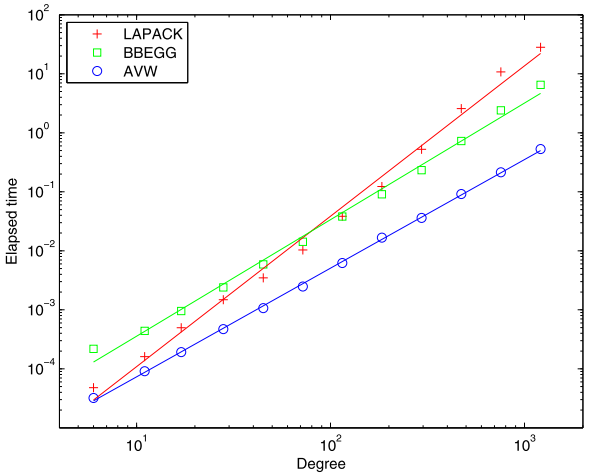


Fig. 1 Timing comparison, companion case, single shift

the sensitivity of the computed root λ, and $|p(\lambda)/p'(\lambda)|$ is an estimate of the error. We always compute this quantity.

Once we have $p(\lambda)/p'(\lambda)$, we can do a Newton correction $\lambda \leftarrow \lambda - p(\lambda)/p'(\lambda)$ for free. This often results in a significant improvement in the root.

7 Numerical experiments

All of our numerical experiments were done on a computer with an AMD Athlon dual core processor with 512 KB cache per core, running at 2.3 GHz. All of the codes compared here were written in Fortran. Our codes are available online.[1]

Companion matrices, single-shift code Our first experiments are on companion matrices. The coefficients of the polynomials are random numbers distributed normally. We compared our code, labeled AVW in Fig. 1, against the LAPACK code ZHSEQR and a structured companion code, labeled BBEGG in Fig. 1. The LAPACK code uses Francis's implicitly-shifted QR algorithm, requiring $O(n^2)$ memory and $O(n^3)$ flops. The BBEGG code is from is from Bini et al. [4]. It is an implicitly-shifted (single shift, complex) QR algorithm that uses a quasiseparable generator representation of the matrix. It requires $O(n)$ memory and $O(n^2)$ flops. On the log-log plot in Fig. 1, the least-squares fit lines for BBEGG and AVW are approximately parallel, indicating that they have the same computational complexity. In fact the slopes are 1.97 for BBEGG and 1.84 for AVW, indicating that both have about $O(n^2)$ complexity in practice. We note that BBEGG is faster than LAPACK for polynomials of large degree, while AVW is faster than both codes at all degrees. The AVW times include the (small) time to do the *a posteriori* tests described in the previous section. The highest degree polynomial considered in this test was 1210. At that degree LAPACK

[1] http://www.math.wsu.edu/students/jaurentz/publications/code.html.

Table 1 Maximum of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$, companion case, single shift

Degree	17	72	295	1210
LAPACK	1.1×10^{-14}	3.1×10^{-14}	6.8×10^{-14}	2.3×10^{-13}
BBEGG	1.5×10^{-14}	1.1×10^{-13}	4.1×10^{-12}	2.4×10^{-11}
AVW	7.4×10^{-12}	5.0×10^{-11}	3.7×10^{-10}	1.4×10^{-09}

Table 2 Maximum of error estimates $|p(\lambda)/p'(\lambda)|$, companion case, single shift

Degree	17	72	295	1210
LAPACK	4.3×10^{-15}	8.5×10^{-15}	1.4×10^{-14}	2.4×10^{-14}
BBEGG	2.4×10^{-14}	1.8×10^{-13}	1.3×10^{-12}	1.2×10^{-11}
AVW	4.7×10^{-12}	7.5×10^{-11}	5.4×10^{-11}	1.7×10^{-10}

Table 3 Maximum of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$ after Newton correction, companion case, single shift

Degree	17	72	295	1210
LAPACK	8.1×10^{-16}	1.8×10^{-15}	2.6×10^{-15}	5.2×10^{-15}
BBEGG	7.5×10^{-16}	1.5×10^{-15}	3.5×10^{-15}	5.5×10^{-15}
AVW	8.6×10^{-16}	1.4×10^{-15}	2.6×10^{-15}	5.3×10^{-15}

Table 4 Maximum of error estimates $|p(\lambda)/p'(\lambda)|$ after Newton correction, companion case, single shift

Degree	17	72	295	1210
LAPACK	3.2×10^{-16}	3.4×10^{-16}	3.2×10^{-16}	3.1×10^{-16}
BBEGG	5.2×10^{-16}	3.1×10^{-16}	2.9×10^{-16}	3.8×10^{-16}
AVW	3.3×10^{-16}	3.1×10^{-16}	3.1×10^{-16}	3.1×10^{-16}

took 28.2 seconds to find all of the roots, BBEGG took 6.5 seconds, and AVW took 0.5 seconds, including the time to do the tests.

Although this test does not make a direct comparison with our earlier method [1], it is not hard to draw a conclusion. Comparing with the similar experiment in [1], we find that our new code is faster by a factor of about three.

Table 1 shows the residual norm $\|(\lambda I - A)v\|/\|A\|\|v\|$ for a variety of degrees, from small to large. We note that the LAPACK residuals are best, followed by BBEGG and AVW. Table 2 shows a similar picture for the error estimates $|p(\lambda)/p'(\lambda)|$.

The error estimate computation allows us to do a step of Newton's method to correct each of the roots. The results after the correction are shown in Tables 3 and 4 for the residual and the error estimate. We see that for all three methods the results are equally good. All roots are accurate to machine precision.

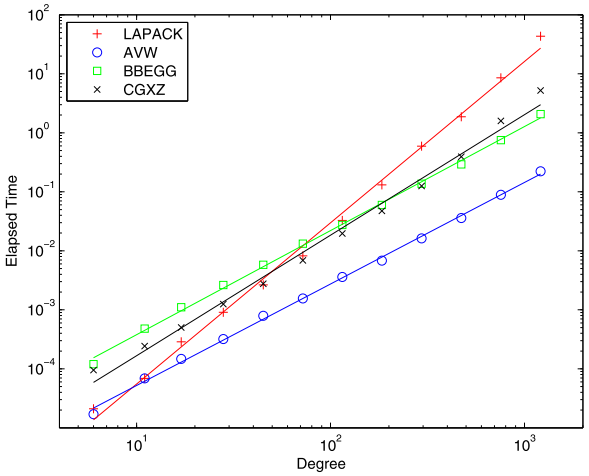


Fig. 2 Timing comparison, companion case, double shift

Table 5 Maximum of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$, companion case, double shift

Degree	17	72	295	1210
LAPACK	1.3×10^{-14}	2.9×10^{-14}	5.8×10^{-14}	2.5×10^{-13}
AVW	1.7×10^{-07}	9.1×10^{-07}	2.8×10^{-06}	3.3×10^{-02}

Table 6 Maximum of error estimates $|p(\lambda)/p'(\lambda)|$, companion case, double shift

Degree	17	72	295	1210
LAPACK	1.2×10^{-14}	8.2×10^{-15}	1.9×10^{-14}	5.6×10^{-13}
AVW	1.4×10^{-07}	1.5×10^{-06}	3.8×10^{-07}	9.1×10^{-03}

Companion matrices, double-shift code We did similar experiments with real polynomials and double-shift code. Here we compared against three other codes, which are labeled LAPACK, BBEGG, and CGXZ in Fig. 2. The LAPACK code is the real double-shift code DSHEQR. BBEGG is the real, double-shift code from Bini et. al. [4]. CGXZ is the real, double-shift code from Chandrasekaran et. al. [6]. The timing results, shown in Fig. 2, are similar to those of the single-shift case. Once again AVW was much faster than the other methods.

Unfortunately the accuracy results were not so favorable. Tables 5 and 6 show that both the residuals and the error estimates are unacceptably high, especially at degree 1210. The Newton correction (not shown here) improves the results, but only in the low-degree cases.

Colleague matrices, single shift We also did experiments using the Chebyshev polynomial basis. The polynomials are random linear combinations of Chebyshev polynomials with the complex coefficients distributed randomly. Chebyshev polynomials are generated by a three-term recurrence, so the matrices are comrade matrices. For

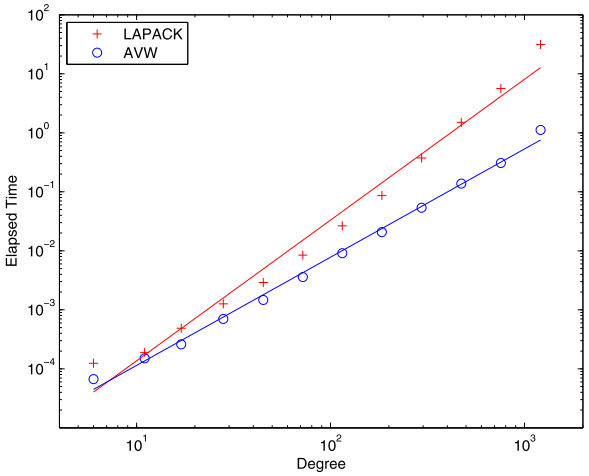


Fig. 3 Timing comparison, colleague case, single shift

Table 7 Maximum of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$, colleague case, single shift

Degree	17	72	295	1210
LAPACK	1.7×10^{-13}	1.1×10^{-12}	6.1×10^{-12}	2.8×10^{-11}
AVW	7.7×10^{-12}	2.8×10^{-10}	1.3×10^{-08}	3.3×10^{-07}

Table 8 Maximum of error estimates $|p(\lambda)/p'(\lambda)|$, colleague case, single shift

Degree	17	72	295	1210
LAPACK	1.9×10^{-14}	2.7×10^{-14}	3.1×10^{-14}	5.1×10^{-14}
AVW	3.8×10^{-13}	7.9×10^{-12}	3.9×10^{-11}	1.2×10^{-10}

the special cases of Chebyshev polynomials, Good [11] named these *colleague matrices*. As Fig. 3 shows, our code AVW is faster than LAPACK at all degrees and much faster at high degrees. At degree 1210, LAPACK took 31.4 seconds to compute all of the roots, while AVW took 1.1 seconds.

Tables 7 and 8 show the maxima of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$ and error estimates $|p(\lambda)/p'(\lambda)|$, respectively. We observe that LAPACK is more accurate than AVW and the AVW figures deteriorate gradually with increasing degree. After the Newton correction, the AVW results are as good as those of LAPACK, as Tables 9 and 10 show.

Colleague matrices, double shift We tested double-shift codes on real colleague matrices. In these tests we also compared against the code of Eidelman, Gemignani, and Gohberg [7], which we call EGG for short. This fast method is applicable to upper Hessenberg symmetric-plus-rank-one matrices, so it can be applied to the colleague case. It has been shown to be backward stable [7].

Table 9 Maximum of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$ after Newton correction, colleague case, single shift

Degree	17	72	295	1210
LAPACK	7.7×10^{-15}	2.7×10^{-14}	1.7×10^{-13}	9.0×10^{-13}
AVW	7.4×10^{-15}	2.7×10^{-14}	1.4×10^{-13}	9.9×10^{-13}

Table 10 Maximum of error estimates $|p(\lambda)/p'(\lambda)|$ after Newton correction, colleague case, single shift

Degree	17	72	295	1210
LAPACK	9.8×10^{-16}	6.6×10^{-16}	4.9×10^{-16}	2.1×10^{-16}
AVW	1.7×10^{-15}	3.2×10^{-16}	2.1×10^{-16}	3.4×10^{-16}

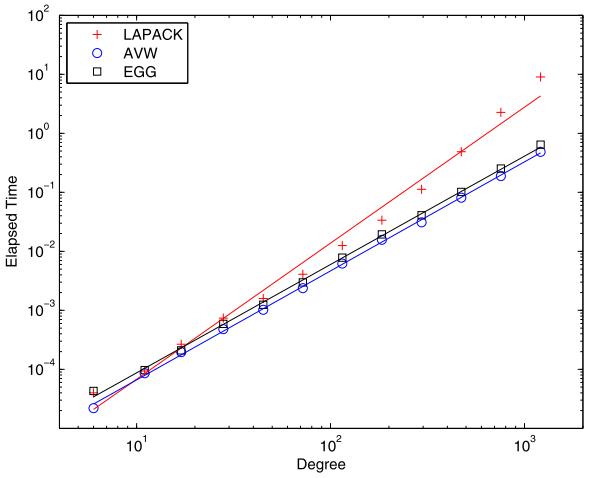


Fig. 4 Timing comparison, colleague case, double shift

The timing comparison is shown in Fig. 4. For high degrees both EGG and AVW are much faster than the LAPACK code DHSEQR. At degree 1210 LAPACK took 9 seconds, EGG took 0.64 seconds, and AVW took 0.49 seconds. AVW is only about thirty percent faster then EGG.

Tables 11 and 12 provide further numerical support for the backward stability of EGG, which is seen to be about as accurate as LAPACK. Our code AVW is much less accurate. The situation is not as bad as for the companion matrix (double-shift) experiment, but still unsatisfactory. Tables 13 and 14 show that a Newton correction improves the accuracy of the AVW results, but does not bring it up to the level of the other two methods. A second Newton correction (not shown here) brings the AVW results up to machine precision. The clear winner of this experiment is EGG, which is about as accurate as LAPACK and only marginally slower than AVW.

Table 11 Maximum of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$, colleague case, double shift

Degree	17	72	295	1210
LAPACK	1.7×10^{-13}	1.7×10^{-12}	6.3×10^{-12}	5.4×10^{-11}
EGG	1.9×10^{-13}	2.1×10^{-12}	7.4×10^{-12}	1.1×10^{-10}
AVW	4.3×10^{-09}	1.2×10^{-06}	4.6×10^{-07}	2.4×10^{-03}

Table 12 Maximum of error estimates $|p(\lambda)/p'(\lambda)|$, colleague case, double shift

Degree	17	72	295	1210
LAPACK	7.7×10^{-14}	1.4×10^{-14}	2.6×10^{-14}	2.8×10^{-13}
EGG	2.9×10^{-14}	2.7×10^{-14}	1.0×10^{-13}	9.2×10^{-14}
AVW	1.4×10^{-10}	9.8×10^{-09}	1.9×10^{-09}	1.7×10^{-06}

Table 13 Maximum of residuals $\|(\lambda I - A)v\|/\|A\|\|v\|$ after Newton correction, colleague case, double shift

Degree	17	72	295	1210
LAPACK	8.2×10^{-15}	3.5×10^{-14}	1.0×10^{-13}	1.6×10^{-12}
EGG	6.5×10^{-15}	3.3×10^{-14}	1.3×10^{-13}	2.1×10^{-12}
AVW	5.8×10^{-15}	5.8×10^{-13}	5.0×10^{-12}	4.3×10^{-05}

Table 14 Maximum of error estimates $|p(\lambda)/p'(\lambda)|$ after Newton correction, colleague case, double shift

Degree	17	72	295	1210
LAPACK	8.1×10^{-16}	3.4×10^{-16}	2.3×10^{-16}	7.3×10^{-16}
EGG	1.6×10^{-15}	2.3×10^{-16}	4.6×10^{-16}	2.0×10^{-16}
AVW	8.5×10^{-16}	4.7×10^{-15}	1.1×10^{-14}	2.0×10^{-08}

8 Conclusions

We considered the problem of computing the eigenvalues of upper Hessenberg matrices of banded-plus-spike form, a class that includes companion and comrade matrices as special cases. We developed a factored form in which the matrix is represented as a product of core transformations and a banded upper-triangular matrix. A non-unitary variant of Francis's implicitly-shifted *QR* algorithm that preserves the factored form is used to compute the eigenvalues in $O(n^2)$ time and $O(n)$ space. Inexpensive *a posteriori* tests for stability and accuracy are performed as part of the algorithm. The output of these tests allows a Newton correction to be taken for free. Single-step and double-step versions of the code have been developed. Numerical tests show promise for the single-step code. It is reasonably accurate on random matrices of order up to 1000 or more, especially after the Newton correction. In the case of companion matrices it is three times faster than our earlier code [1] and more than ten times faster

than any other code we are aware of. The double-shift code was disappointing. It is very fast but too unstable.

References

1. Aurentz, J.L., Vandebril, R., Watkins, D.S.: Fast computation of the zeros of a polynomial via factorization of the companion matrix. SIAM J. Sci. Comput. **35**, A255–A269 (2013)
2. Barnett, S.: Polynomials and Linear Control Systems. Dekker, New York (1983)
3. Bini, D.A., Eidelman, Y., Gemignani, L., Gohberg, I.: Fast QR eigenvalue algorithms for Hessenberg matrices which are rank-one perturbations of unitary matrices. SIAM J. Matrix Anal. Appl. **29**, 566–585 (2007)
4. Bini, D.A., Boito, P., Eidelman, Y., Gemignani, L., Gohberg, I.: A fast implicit *QR* algorithm for companion matrices. Linear Algebra Appl. **432**, 2006–2031 (2010)
5. Boito, P., Eidelman, Y., Gemignani, L., Gohberg, I.: Implicit *QR* with compression. Indag. Math. **23**, 733–761 (2012)
6. Chandrasekaran, S., Gu, M., Xia, J., Zhu, J.: A fast QR algorithm for companion matrices. Oper. Theory, Adv. Appl. **179**, 111–143 (2007)
7. Eidelman, Y., Gemignani, L., Gohberg, I.: Efficient eigenvalue computation for quasiseparable Hermitian matrices under low rank perturbation. Numer. Algorithms **47**, 253–273 (2008)
8. Fernando, K.V., Parlett, B.N.: Accurate singular values and differential qd algorithms. Numer. Math. **67**, 191–229 (1994)
9. Fiedler, M.: A note on companion matrices. Linear Algebra Appl. **372**, 325–331 (2003)
10. Francis, J.G.F.: The QR transformation, part II. Comput. J. **4**, 332–345 (1962)
11. Good, I.J.: The colleague matrix, a Chebyshev analogue of the companion matrix. Q. J. Math. **12**, 61–68 (1961)
12. Parlett, B.N.: The new qd algorithms. Acta Numer. **4**, 459–491 (1995)
13. Van Barel, M., Vandebril, R., Van Dooren, P., Frederix, K.: Implicit double shift *QR*-algorithm for companion matrices. Numer. Math. **116**, 177–212 (2010)
14. Vandebril, R., Del Corso, G.M.: An implicit multishift *QR*-algorithm for Hermitian plus low rank matrices. SIAM J. Sci. Comput. **32**, 2190–2212 (2010)
15. Vandebril, R., Van Barel, M., Mastronardi, N.: Matrix Computations and Semiseparable Matrices, Vol. II: Eigenvalue and Singular Value Methods. Johns Hopkins University Press, Baltimore (2008)
16. Watkins, D.S.: The Matrix Eigenvalue Problem: *GR* and Krylov Subspace Methods. SIAM, Philadelphia (2007)
17. Watkins, D.S.: Fundamentals of Matrix Computations, 3rd edn. Wiley, New York (2010)
18. Watkins, D.S.: Francis's algorithm. Am. Math. Mon. **118**(5), 387–403 (2011)
19. Zhlobich, P.: Differential qd algorithm with shifts for rank-structured matrices. SIAM J. Matrix Anal. Appl. **33**, 1153–1171 (2012)