Open access • Journal Article • DOI:10.1137/140983434

Fast and Backward Stable Computation of Roots of Polynomials — Source link

Jared L. Aurentz, Thomas Mach, Raf Vandebril, David S. Watkins

Institutions: University of Oxford, Katholieke Universiteit Leuven, Washington State University

Published on: 07 Jul 2015 - SIAM Journal on Matrix Analysis and Applications (Society for Industrial and Applied Mathematics)

Topics: QR algorithm, Cuthill–McKee algorithm, QR decomposition, Hessenberg matrix and Eigenvalue algorithm

Fast and backward stable computation of roots of polynomials

Jared L. Aurentz, Thomas Mach, Raf Vandebril, and David S. Watkins

Jared L. Aurentz
Mathematical Institute,
University of Oxford
aurentz@maths.ox.ac.uk

Thomas Mach
Dept. Computer Science, KU Leuven
thomas.mach@cs.kuleuven.be

Raf Vandebril
Dept. Computer Science, KU Leuven
raf.vandebril@cs.kuleuven.be

David S. Watkins
Dept. of Mathematics,
Washington State University
watkins@math.wsu.edu

Abstract

A stable algorithm to compute the roots of polynomials is presented. The roots are found by computing the eigenvalues of the associated companion matrix by Francis's implicitly-shifted QR algorithm. A companion matrix is an upper Hessenberg matrix that is unitary-plus-rank-one, that is, it is the sum of a unitary matrix and a rank-one matrix. These properties are preserved by iterations of Francis's algorithm, and it is these properties that are exploited here. The matrix is represented as a product of $3n-1$ Givens rotators plus the rank-one part, so only $O(n)$ storage space is required. In fact, the information about the rank-one part is also encoded in the rotators, so it is not necessary to store the rank-one part explicitly. Francis's algorithm implemented on this representation requires only $O(n)$ flops per iteration and thus $O(n^2)$ flops overall. The algorithm is described, normwise backward stability is proved, and an extensive set of numerical experiments is presented. The algorithm is shown to be about as accurate as the (slow) Francis QR algorithm applied to the companion matrix without exploiting the structure. It is faster than other fast methods that have been proposed, and its accuracy is comparable or better.

Article information

- Aurentz, Jared L.; Mach, Thomas; Vandebril, Raf; Watkins, David S. Watkins *Fast and backward stable computation of roots of polynomials*, SIAM Journal on Matrix Analysis and Applications, 36(3):942–973, 2015.
- This article equals the final publisher's version. A link is found below.
- Journal's homepage: http://epubs.siam.org/toc/sjmael/current
- Published version: http://epubs.siam.org/doi/abs/10.1137/140983434
- KU Leuven's repository url: https://lirias.kuleuven.be/handle/123456789/494876

SIAM J. MATRIX ANAL. APPL.
Vol. 36, No. 3, pp. 942–973

FAST AND BACKWARD STABLE COMPUTATION OF ROOTS OF POLYNOMIALS*

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

Abstract. A stable algorithm to compute the roots of polynomials is presented. The roots are found by computing the eigenvalues of the associated companion matrix by Francis's implicitly shifted QR algorithm. A companion matrix is an upper Hessenberg matrix that is unitary-plus-rank-one, that is, it is the sum of a unitary matrix and a rank-one matrix. These properties are preserved by iterations of Francis's algorithm, and it is these properties that are exploited here. The matrix is represented as a product of $3n-1$ Givens rotators plus the rank-one part, so only $O(n)$ storage space is required. In fact, the information about the rank-one part is also encoded in the rotators, so it is not necessary to store the rank-one part explicitly. Francis's algorithm implemented on this representation requires only $O(n)$ flops per iteration and thus $O(n^2)$ flops overall. The algorithm is described, normwise backward stability is proved, and an extensive set of numerical experiments is presented. The algorithm is shown to be about as accurate as the (slow) Francis QR algorithm applied to the companion matrix without exploiting the structure. It is faster than other fast methods that have been proposed, and its accuracy is comparable or better.

Key words. polynomial, root, rootfinding, companion matrix, eigenvalue, QR algorithm, rotators

AMS subject classifications. 65F15, 65H17, 15A18, 65H04

DOI. 10.1137/140983434

1. Introduction.

We describe a method for computing the *roots* of a monic polynomial $p(z)$ expressed in terms of the monomial basis, say,

$$p(z) = z^n + a_{n-1}z^{n-1} + \cdots + a_1 z + a_0.$$

We will assume throughout this paper that $a_0 \neq 0$, since otherwise we can factor out z and reduce the degree of the polynomial. Like many others before us, we

*Received by the editors August 21, 2014; accepted for publication (in revised form) by J. L. Barlow April 22, 2015; published electronically July 7, 2015. This research was partially supported by the Research Council KU Leuven, projects CREA-13-012, Can Unconventional Eigenvalue Algorithms Supersede the State of the Art, OT/11/055, Spectral Properties of Perturbed Normal Matrices and their Applications, and CoE EF/05/006, Optimization in Engineering (OPTEC), and fellowship F+/13/020, Exploiting Unconventional QR Algorithms for Fast and Accurate Computations of Roots of Polynomials; by the DFG research stipend MA 5852/1-1; by Fund for Scientific Research–Flanders (Belgium) project G034212N, Reestablishing Smoothness for Matrix Manifold Optimization via Resolution of Singularities; by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office, Belgian Network DYSCO (Dynamical Systems, Control, and Optimization); and by the European Research Council under the European Union's Seventh Framework Programme (FP7/20072013)/ERC grant agreement 291068. The views expressed in this article are not those of the ERC or the European Commission, and the European Union is not liable for any use that may be made of the information contained here.
http://www.siam.org/journals/simax/36-3/98343.html
†Mathematical Institute, University of Oxford, Woodstock Road, OX2 6GG Oxford, UK (jared.aurentz@maths.ox.ac.uk).
‡Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 3001 Leuven (Heverlee), Belgium (thomas.mach@cs.kuleuven.be, raf.vandebril@cs.kuleuven.be).
§Department of Mathematics, Washington State University, Pullman, WA 99164-3113 (watkins@math.wsu.edu).

solve this problem by using Francis's implicitly shifted QR algorithm[1] to compute the eigenvalues of the associated *companion* matrix

$$A = \begin{bmatrix} & & & -a_0 \\ 1 & & & -a_1 \\ & \ddots & & \vdots \\ & & 1 & -a_{n-1} \end{bmatrix}. \tag{1.1}$$

As the companion matrix is of Hessenberg form, one can directly apply Francis's algorithm [20, 21, 33] to retrieve the eigenvalues. This is what the `roots` command in MATLAB does. This method preserves the upper Hessenberg form but does not exploit other structures present in the companion matrix. Cleve Moler stated in 1991 [25], referring to this approach, "This method might not be the best possible because it uses n^2 storage and n^3 time. An algorithm designed specifically for polynomial roots might use order n storage and n^2 time."

In recent years several methods that use $O(n)$ storage and $O(n^2)$ time have been devised (see section 2), all of which exploit the fact that a companion matrix can be decomposed into the sum of a unitary and a rank-one matrix. If one then applies a unitary similarity transformation on this sum, one ends up again with a unitary-plus-rank-one matrix. More precisely, taking the similarity determined by the implicit-shifted QR algorithm always leads to a Hessenberg matrix equal to the sum of a unitary and a rank-one matrix. This is the main theoretical idea behind all the fast QR algorithms for companion matrices, differing, however, significantly in the way the matrix is represented and how the algorithm is implemented.

Of the fast methods proposed up to this point, none have been proved to be backward stable. The method that we propose here is backward stable and faster than the other fast QR-based methods that have been proposed. Our Fortran codes can be downloaded from http://people.cs.kuleuven.be/raf.vandebril.

The article is organized as follows. Section 2 discusses earlier work in this area. Section 3 introduces some terminology and notational conventions that will aid in the presentation of the new method. Section 4 presents our new memory-efficient representation of unitary-plus-rank-one matrices, the QR algorithm itself is discussed in section 5, and some implementation details are given in section 6. Section 7 presents the stability analysis. We finish with our numerical experiments in section 8.

2. Previous work. The research on fast companion algorithms was initiated by Bini, Daddi, and Gemignani [7] relying on the relation $A = A^{-*} + UV^*$, with A the iterates in the QR algorithm, and UV^* a rank two part. It is proved that the strictly upper triangular part of A stems from a rank three matrix. The authors rely solely on the low rank structure and present a memory efficient storage of Q and R needed to execute explicit QR steps, that is, explicitly computing Q, R, and forming their product RQ. Unfortunately the representation is not robust. Large discrepancies between the magnitudes of the vectors generating the low rank parts are observed, a problem which one is unable to fully solve. Moreover, the explicit version of the QR algorithm can be considered as a drawback as typically additional memory and computational effort is required compared to an implicit approach.

Bini et al. [8] develop an explicit QR algorithm operating directly on the Hessenberg matrix. They store the rank-one part with two vectors and the unitary matrix

[1] In this paper the terms Francis's algorithm and QR algorithm will be used interchangeably.

via quasi-separable generators. The quasi-separable representation is, however, not able to retain the unitarity. To overcome this problem the authors enforce the unitarity by taking out the tiny error and using it to update the generators of the rank-one part. This update is constructed in such a way that it does not destroy the Hessenberg structure.

Chandrasekaran et al. [14] were the first to perform implicit QR steps directly on the QR factorization of the Hessenberg matrix, where the Q matrix is decomposed in rotators and the low rank structure of the upper triangular part of R is stored via the sequentially semiseparable representation which essentially equals the quasi-separable representation in this case. Even though the rank of R should be bounded by 2, they admit 3 and require compression after each step to keep the rank numerically bounded by 3. Also in this paper focus is on retaining the low rank structure imposed by the unitary and rank-one matrix, but no effort is put into retaining the unitarity itself.

Delvaux, Frederix, and Van Barel [16] present an algorithm for block companion matrices. The approach resembles [14] as the QR factorization of the Hessenberg matrix is stored, but the R factor is now stored via a Givens-weight representation. Again the unitary-plus-low-rank structure deteriorates while running the QR algorithm. The authors propose a restoration technique based on a combination of the ones proposed in [14]: obtain the desired rank in R and [8]; restore the unitary structure. At the end both the unitary and the low rank part are combined to compute the eigenvalues.

Van Barel et al. [28] present a representation based on three sequences of rotators and a vector. Implicit QR steps are executed on the factorization directly and no compression steps are used to enforce any of the three structures. As a result one ends with a unitary-plus-rank-one matrix that approximates a Hessenberg matrix; numerical roundoff slightly perturbs the exact cancellation that should occur between the unitary and low rank part.

In [6] Bini et al. enhance their previous results from [8] and convert their explicit QR version to an implicit one. Two different representations are used in the implementation of the algorithm: in each iterate the unitary matrix is stored as a product of essentially 2×2 and 3×3 unitary matrices, and to update this matrix under a QR step the quasi-separable representation of the unitary matrix is computed and utilized. In [12] Boito et al. enhance and simplify their implicit version by doing all computations directly on the quasi-separable representation of the unitary matrix, and a compression technique is used to reduce the number of quasi-separable generators after a QR step to a minimum.

In [19] Eidelman, Gohberg, and Haimovici revisit the method from [14]. They also describe a factorization of the companion matrix using $3n - 3$ rotations similar to what we propose in this paper. However, they keep the low-rank part explicitly and go back to a semiseparable representation of R for the description of the QR algorithm. Furthermore, they do not provide numerical results.

For completeness we mention that besides the QR variants there is also an approach based on companion pencils [11] and there are fast nonunitary GR algorithms [2, 3, 35], but they are potentially unstable. Other, QR related approaches tackle root finding problems of polynomials expressed in other bases, for example, comrade or confederate matrices [18, 30]. Still other methods attack the polynomial problem directly, notably Bini's code [5], which uses the Ehrlich–Aberth method, and the more recent code MPSolve of Bini and Fiorentino [9], which uses multiprecision arithmetic to produce the zeros to any specified precision. In this paper we consider only codes that use fixed (double) precision arithmetic.

3. Core transformations. Our method makes heavy use of rotators. In the interest of flexibility we will introduce a more general concept. A *core transformation* G_i is a nonsingular matrix that is identical to the identity matrix except in the 2×2 submatrix in the $(i:i+1, i:i+1)$ diagonal block, which is called the *active part* of the core transformation. We will consistently use the subscript i on a core transformation G_i to indicate the position of the active part. It follows that the core transformations G_i and G_j commute whenever $|i-j| > 1$.

In some of our previous work [2,3] we have made use of core transformations that are not unitary, but in this paper we will use only unitary ones. Thus, in this paper, the term *core transformation* will mean *unitary* core transformation. Givens rotators, with active parts of the form $\left[\begin{smallmatrix} c & -s \\ s & \overline{c} \end{smallmatrix}\right]$ with $|c|^2 + s^2 = 1$, are core transformations, and so are Givens reflectors $\left[\begin{smallmatrix} c & s \\ s & -\overline{c} \end{smallmatrix}\right]$. We implemented our algorithms using rotators, but we could equally well have used reflectors. In our initial description we will refer to generic core transformations, which could be rotators, reflectors, or any other kind of unitary core transformations.

It is well known (and easy to prove) that every $n \times n$ unitary upper Hessenberg matrix can be factored into a descending sequence of $n-1$ core transformations:

$$Q = Q_1 Q_2 \dots Q_{n-1}. \tag{3.1}$$

To simplify the notation, to clarify some equations, and to increase the readability of the algorithms we will frequently depict a core transformation as $\vec{\ulcorner}$, where the tiny arrows indicate the position of the active part. For example, for $n = 9$, the factorization (3.1) can be depicted as

$$Q = Q_1 Q_2 \cdots Q_{n-1} = \ulcorner\ulcorner\ulcorner\ulcorner\ulcorner\ulcorner\ulcorner\ulcorner .$$

As another example, the equation

$$\vec{\ulcorner} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \tilde{x}_3 \\ 0 \end{bmatrix}$$

means that the vector x is multiplied by a core transformation G_3 on the left to produce a new vector that has a zero in the fourth position. With this notation established, we can now describe our new method.

4. Representation of the matrix. We will store each unitary-plus-rank-one upper Hessenberg matrix in QR decomposed form, as in [14]. Q is a unitary upper Hessenberg matrix, which can be stored compactly as a product of core transformations (3.1). R is an upper triangular unitary-plus-rank-one matrix. We decompose the latter into a unitary part and a rank-one part. All unitary matrices are stored as sequences of core transformations, as in [28]. We will prove that our representation of the unitary part also contains the information about the rank-one part encoded within the core transformations. Therefore there will be no need to keep track of or update the rank-one part in the course of the iterations. Everything will be updated automatically in the core transformations. Thus our algorithm will consist almost entirely

of unitary similarity transformations on unitary matrices represented as products of core transformations.

Now we get more specific. Starting from a companion matrix, we begin by embedding it in a larger matrix,

$$A = \left[\begin{array}{cccc|c} 0 & & & -a_0 & 1 \\ 1 & & & -a_1 & 0 \\ & 1 & & -a_2 & 0 \\ & & \ddots & & \vdots \\ & & 1 & -a_{n-1} & 0 \\ \hline & & 0 & 0 & 0 \end{array}\right], \tag{4.1}$$

with an extra row of zeros and a column that is nearly zero. The one in the $(1, n+1)$ position ensures that the unitary-plus-rank-one structure is preserved. The enlarged matrix clearly has one extra zero eigenvalue, which can be deflated out immediately. This curious beginning has at least two important consequences, as we shall see. It ensures that the information about the rank-one part is fully encoded in the core transformations, and it results in a simpler, cleaner algorithm.

If we take the QR factorization of the enlarged matrix, we obtain

$$Q = \left[\begin{array}{cccc|c} 0 & & & 1 & 0 \\ 1 & & & 0 & 0 \\ & 1 & & 0 & 0 \\ & & \ddots & \vdots & \vdots \\ & & 1 & 0 & 0 \\ \hline & & 0 & 0 & 1 \end{array}\right] \quad \text{and} \quad R = \left[\begin{array}{cccc|c} 1 & & & -a_1 & 0 \\ & 1 & & -a_2 & 0 \\ & & \ddots & \vdots & \vdots \\ & & 1 & -a_{n-1} & 0 \\ & & & -a_0 & 1 \\ \hline & & & 0 & 0 \end{array}\right].$$

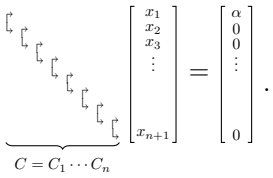
We store Q as a product of core transformations as in (3.1). In this specific case we have $Q = Q_1 \cdots Q_{n-1}$, where each Q_i has active part $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$. Here we are depicting the core transformations as reflectors for simplicity. In the actual code we used rotators $\left[\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right]$, which can be done if we insert a factor $(-1)^{n-1}$ in appropriate entries in Q and R.

Since Q is of dimension $n+1$, the factorization into core transformations should have n factors, but in this case there are only $n-1$. The last transformation is trivial ($Q_n = I$) because the bottom row of A is trivial. This is important.

The upper triangular matrix has unitary-plus-rank-one form: $R = Z_n + xe_n^T$, where

$$Z_n = \left[\begin{array}{cccc|c} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \vdots \\ & & & 1 & \\ \hline & & & 0 & 1 \\ & & & 1 & 0 \end{array}\right] \quad \text{and} \quad x = -\left[\begin{array}{c} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_0 \\ \hline 1 \end{array}\right]. \tag{4.2}$$

Let $C = C_1 C_2 \cdots C_n$ be a product of core transformations $C_1, \ldots, C_n$ such that $C_1 \cdots C_n x = \alpha e_1$, where $|\alpha| = \|x\|_2$. Pictorially, for $n = 8$, we have



Since C is a product of a descending sequence of core transformations it is a unitary upper Hessenberg matrix. Notice that since $|x_{n+1}| = 1 \neq 0$, the core transformation C_n is nontrivial (that is, nondiagonal) and thus it follows easily that all the C_i are nontrivial. Therefore C is a *proper* upper Hessenberg matrix, that is, its subdiagonal entries are all nonzero.

The information about the rank-one part is concentrated in the vector x. We form the C_i by *rolling up* x, transforming x to a multiple of e_1. In the process we are encoding the rank-one part in the core transformations C_i.

Letting $B = CZ_n$ and $y = \alpha e_n$, we have

$$R = C^*(B + e_1 y^T). \tag{4.3}$$

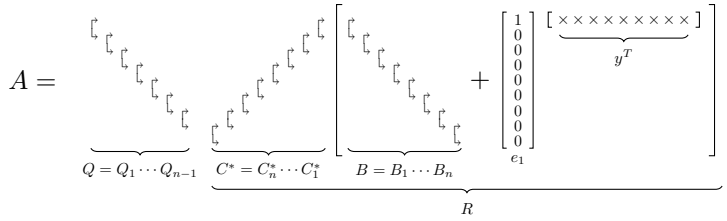
Notice that B is also a unitary upper Hessenberg matrix, so it can be factored into a descending sequence of core transformations: $B = B_1 \cdots B_n$. In fact it is obvious that we can take $B_i = C_i$, for $i = 1, \ldots, n-1$, and $B_n = C_n Z_n$. Expanding (4.3) we have

$$R = C_n^* \cdots C_1^*(B_1 \cdots B_n + e_1 y^T).$$

Now combining Q and R, we have

$$A = QR = QC^*(B + e_1 y^T) = Q_1 \cdots Q_{n-1} C_n^* \cdots C_1^*(B_1 \cdots B_n + e_1 y^T). \tag{4.4}$$

Pictorially, for $n = 8$, we have



Notice that the Q sequence of core transformations is shorter than the other sequences due to the fact that the last row of A is trivial.

4.1. Properties of the factorization. The factorized form (4.4) is the form in which we will store our matrix. Over the course of the iterations of Francis's algorithm, the contents of the Q_i, C_i, B_i, and y will evolve, but the form (4.4) will be preserved. Certain specific properties of the form will be preserved as well, as we now show.

Although we are not yet ready to describe the algorithm, we can make some general statements about it. Because the last row of A represents a zero eigenvalue that has been deflated from the problem, the iterations of our algorithm will be similarity transformations by matrices of the form $U = \left[\begin{smallmatrix} \tilde{U} & 0 \\ 0 & 1 \end{smallmatrix}\right]$, where $\tilde{U}$ is $n \times n$. Initially we have $A = QR$, where $Q = \left[\begin{smallmatrix} \tilde{Q} & 0 \\ 0 & 1 \end{smallmatrix}\right]$ and $R = \left[\begin{smallmatrix} \tilde{R} & \times \\ 0 & 0 \end{smallmatrix}\right]$, and these general forms

are preserved under such a similarity transformation, for (as we shall see in section 5) we have

$$\hat{A} = U^*AU = U^*QRU = U^*QVV^*RU = \hat{Q}\hat{R} \tag{4.5}$$

for some unitary matrix V of the form $\left[\begin{smallmatrix} \tilde{V} & 0 \\ 0 & 1 \end{smallmatrix}\right]$. We have $R = Z + xz^T$, where x is initially given by (4.2). In particular $x_{n+1} = -1$. Under the transformation $R \to \hat{R} = V^*RU$, x is transformed to $\hat{x} = V^*x$. Because of the form of V, the transformed x still satisfies $x_{n+1} = -1$ and will continue to do so forever. Similarly, the vector z is initially e_n, and, in particular, $z_{n+1} = 0$, a property that persists under the iterations. The vector y in (4.4) satisfies $y = \alpha z$, so we will have $y_{n+1} = 0$ forever.

Taking a closer look at the decomposition, we have $A = QC^*(B + e_1y^T)$ and

$$\hat{A} = \hat{Q}\hat{C}^*(\hat{B} + e_1\hat{y}^T) = (U^*QV)(V^*C^*W)(W^*BU + e_1y^TU), \tag{4.6}$$

where (see section 5) the unitary W has a different form from the other transforming matrices: $W = \left[\begin{smallmatrix} 1 & 0 \\ 0 & \tilde{W} \end{smallmatrix}\right]$. This is the part of the algorithm in which row and column $n+1$ get used. Notice that $We_1 = e_1 = W^*e_1$, which justifies leaving out the W^* that should have preceded the second e_1 in (4.6).

The equation $Cx = \alpha e_1$ holds initially, and we can now demonstrate that this relationship persists.

LEMMA 4.1. *Let C, x, and α be defined as above, with $Cx = \alpha e_1$. Take $\hat{C} = W^*CV$ to be the result of the unitary similarity transformation* (4.5) *defined in* (4.6). *Then $\hat{C}\hat{x} = \alpha e_1$.*

Proof. We have $\hat{C}\hat{x} = (W^*CV)(V^*x) = W^*Cx = \alpha W^*e_1 = \alpha e_1$. ∎

Nontriviality of the core transformations C_i. We noted above that the core transformations $C_1, \ldots, C_n$ in (4.4) are all nontrivial initially. Now we can show that they remain nontrivial forever. This is a consequence of the following result.

THEOREM 4.2. *For $i = 1, \ldots, n$, let $\left[\begin{smallmatrix} u_i & v_i \\ w_i & z_i \end{smallmatrix}\right]$ denote the active part of C_i in* (4.4), *and let $\gamma_i = |w_i| = |v_i|$. Then $\gamma_i > 0$, $i = 1, \ldots, n$. Moreover*

$$\gamma_1 \cdots \gamma_n = 1/\|x\|_2, \tag{4.7}$$

where x is given by (4.2).

Proof. The initial C_i satisfy $C_1 \cdots C_n x = \alpha e_1$, and we have noted above that this relationship persists as the C_i and x evolve in the course of the iterations. The condition $x_{n+1} = -1$ also persists, and obviously $\|x\|_2$ remains invariant. Since $x = \alpha C_n^* \cdots C_1^* e_1$, we find by direct computation that $x_{n+1} = \alpha \overline{v}_n \cdots \overline{v}_1$. Taking absolute values we have $1 = |\alpha| \gamma_n \cdots \gamma_1$ or $\gamma_1 \cdots \gamma_n = 1/|\alpha| = 1/\|x\|_2$. ∎

Preservation of triangular and Hessenberg forms. We want to know that the Hessenberg form is preserved by the iterations. To this end we must show that the triangular form of R is preserved.

THEOREM 4.3. *Suppose $R = C^*(B + e_1y^T) = C_n^* \cdots C_1^*(B_1 \cdots B_n + e_1y^T)$, where the core transformations $C_1, \ldots, C_n$ are all nontrivial. Then R is upper triangular.*

Proof. In the initial configuration we have $A = QR$, where

$$R = \begin{bmatrix} \tilde{R} & \times \\ 0 & 0 \end{bmatrix} \tag{4.8}$$

with $\tilde{R}$ of size $n \times n$, and $\times$ a vector. As we have noted above, this form of R persists during the iterations, so we just need to show that $\tilde{R}$ remains upper triangular. Since

$R = C^*(B + e_1 y^T)$, we have $H = CR$, where $H = B + e_1 y^T$. We rewrite this equation in the partitioned form

$$\begin{bmatrix} \times & \times \\ \tilde{H} & \times \end{bmatrix} = \begin{bmatrix} \times & \times \\ \tilde{C} & \times \end{bmatrix} \begin{bmatrix} \tilde{R} & \times \\ 0 & 0 \end{bmatrix}, \tag{4.9}$$

where $\tilde{H}$ and $\tilde{C}$ are both $n \times n$, and $\times$'s represent quantities that are not of immediate interest. The fact that the core transformations C_i are all nontrivial implies that C is a proper upper Hessenberg matrix ($c_{i+1,i} \neq 0$, $i = 1, \ldots, n$), which implies that $\tilde{C}$ is upper triangular and nonsingular. Similarly, $\tilde{H}$ is upper triangular. We note further that $\tilde{H} = \tilde{C}\tilde{R}$, which implies

$$\tilde{R} = \tilde{C}^{-1}\tilde{H}. \tag{4.10}$$

Since $\tilde{H}$ and $\tilde{C}^{-1}$ are upper triangular, $\tilde{R}$ must also be upper triangular. ∎

Remark 4.4. The matrices $\tilde{H}$ and $\tilde{C}$ are taken in part from row $n+1$ of H and C, respectively. These matrices have a row $n+1$ because we added a row artificially. Had we not done so, we would not have been able to prove this theorem.

The equation $A = QR$ can also be written as

$$\begin{bmatrix} \tilde{A} & \times \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} \tilde{Q} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{R} & \times \\ 0 & 0 \end{bmatrix}.$$

We have $\tilde{A} = \tilde{Q}\tilde{R}$, and it is on this submatrix that we principally operate. Because $a_0 \neq 0$, $\tilde{A}$ has no zero eigenvalues, it is nonsingular.

THEOREM 4.5. *If the core transformations C_i in* (4.4) *are all nontrivial, then $\tilde{A}$ is upper Hessenberg. If $\tilde{A}$ is nonsingular, then $\tilde{A}$ is properly upper Hessenberg if and only if $\tilde{Q}$ is properly upper Hessenberg. Thus, $\tilde{A}$ is properly upper Hessenberg if and only if $Q_1, \ldots, Q_{n-1}$ are all nontrivial.*

This tells us that a deflation will take place if and only if one of the Q_i becomes trivial.

Proof. Since $\tilde{R}$ is upper triangular and $\tilde{Q}$ is upper Hessenberg, $\tilde{A} = \tilde{Q}\tilde{R}$ must be upper Hessenberg. If $\tilde{A}$ is nonsingular, so is $\tilde{R}$, and thus $r_{ii} \neq 0$ for all i. Since $a_{i+1,i} = q_{i+1,i} r_{ii}$ for $i = 1, \ldots, n$, we see that $\tilde{A}$ is properly upper Hessenberg if and only if $\tilde{Q}$ is. ∎

Nontriviality of the core transformations B_i. Because of the assumption $a_0 \neq 0$ we can show that the core transformations B_i are also nontrivial.

THEOREM 4.6. *For $i = 1, \ldots, n$, let $\left[\begin{smallmatrix} u_i & v_i \\ w_i & z_i \end{smallmatrix}\right]$ denote the active part of B_i in* (4.4), *and let $\beta_i = |w_i| = |v_i|$. Then $\beta_i > 0$, $i = 1, \ldots, n$. Moreover*

$$\beta_1 \cdots \beta_n = |a_0| / \|x\|_2, \tag{4.11}$$

where x is given by (4.2).

Proof. The subdiagonal entries of the upper Hessenberg matrix $B = B_1 \cdots B_n$ are exactly the elements w_i, the subdiagonal entries of the active parts of the B_i. These are also the subdiagonal entries of $H = B + e_1 y^T$ and the main diagonal entries of the upper triangular submatrix $\tilde{H}$ defined in the proof of Theorem 4.3. By similar reasoning the main diagonal entries of $\tilde{C}$ are exactly the subdiagonal entries of the active parts of the C_i. Since $\tilde{H} = \tilde{C}\tilde{R}$ we have $h_{i+1,i} = c_{i+1,i} r_{ii}$ for $i = 1$, $\ldots$, n. Taking absolute values we have $\beta_i = \gamma_i |r_{ii}|$, where the γ_i are as defined in Theorem 4.2. Now, taking a product, using Theorem 4.2, and noting that $|\det \tilde{R}| = |\det \tilde{A}| = |a_0|$, we have

$$\beta_1 \cdots \beta_n = \gamma_1 \cdots \gamma_n |\det \tilde{R}| = |a_0| / \|x\|_2. \qquad \blacksquare$$

Extracting y from the core transformations. We now demonstrate that the information about y is encoded in the core transformations by presenting a formula for computing y explicitly.

THEOREM 4.7. *$y^T = -\rho^{-1} e_{n+1}^T C^* B$, where $\rho = e_{n+1}^T C^* e_1$. ρ is the product of the subdiagonal entries of $C_1^*, \ldots, C_n^*$, and therefore $|\rho| = 1/\|x\|_2$.*

Proof. The entire last row of R is zero. Therefore

$$0 = e_{n+1}^T R = e_{n+1}^T C^*(B + e_1 y^T) = e_{n+1}^T C^* B + e_{n+1}^T C^* e_1 y^T.$$

This can be solved for y^T to yield the desired result. The scalar $\rho = e_{n+1}^T C^* e_1$ is easily shown to be equal to the product of the subdiagonal entries of the rotators C_i^* by direct computation. Thus, by Theorem 4.2, $|\rho| = 1/\|x\|_2$. ∎

This theorem justifies our claim that there is no need to store y, since we can compute it or any needed components at any time. However, it will turn out that our algorithm never needs any part of y anyway. We will make use of Theorem 4.7 in the backward error analysis.

Computing entries of A. At certain points in the algorithm we need to compute elements of A explicitly. More precisely, we need elements of $\tilde{A}$. We already have formulas, namely, $\tilde{R} = \tilde{C}^{-1}\tilde{H}$ and $\tilde{A} = \tilde{Q}\tilde{R}$. Now we need to show that the needed entries can be computed efficiently. The points at which we need elements of A are the shift computation, the computation of the transformation that starts each iteration, and the final eigenvalue computation. In the course of computing elements of $\tilde{A}$ we must compute elements of $\tilde{R}$. For example, for the transformation that starts an iteration in the double shift case, we need the submatix

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ 0 & a_{32} \end{bmatrix},$$

and for this we need

$$\begin{bmatrix} r_{11} & r_{12} \\ 0 & r_{22} \end{bmatrix}.$$

We will use conventional shift strategies that require the submatrix

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

For this we need

$$\begin{bmatrix} r_{n-2,n-1} & r_{n-2,n} \\ r_{n-1,n-1} & r_{n-1,n} \\ 0 & r_{n,n} \end{bmatrix}.$$

In each case we need just a few entries of R on or near the main diagonal. The same is true in the final eigenvalue computation. For example, if we just need to compute a single eigenvalue located at a_{ii}, we need only r_{ii}. We will show that each of these computations can be done easily in $\mathcal{O}(1)$ time.

Suppose, for example, we need r_{ii}. We already observed in the proof of Theorem 4.6 that $h_{i+1,i} = c_{i+1,i} r_{ii}$, so

$$r_{ii} = h_{i+1,i}/c_{i+1,i}. \tag{4.12}$$

$h_{i+1,i}$ and $c_{i+1,i}$ are the subdiagonal entries of the core transformations B_i and C_i, respectively, so we have these numbers in hand. This equation is a consequence of (4.10), which we now write as

$$\tilde{H} = \tilde{C}\tilde{R}. \tag{4.13}$$

(Recall that all three of these matrices are upper triangular, and the main diagonal entries of $\tilde{H}$ and $\tilde{C}$ have indices $(i+1, i)$.)

Now suppose we also need $r_{i-1,i}$. Picking an appropriate equation out of (4.13), exploiting triangularity of $\tilde{C}$ and $\tilde{R}$, we have

$$\begin{bmatrix} h_{i,i} \\ h_{i+1,i} \end{bmatrix} = \begin{bmatrix} c_{i,i-1} & c_{i,i} \\ 0 & c_{i+1,i} \end{bmatrix} \begin{bmatrix} r_{i-1,i} \\ r_{i,i} \end{bmatrix}, \tag{4.14}$$

which we can solve by back substitution. The first step yields r_{ii} by the formula already given above, and the second step gives $r_{i-1,i}$ with just a bit more work. The entries from H and C needed for this computation can be computed from B_{i-1}, B_i, C_{i-1}, and C_i with negligible effort.

If we also need $r_{i-2,i}$, we just need to carry the back solve one step further. To construct the additional required entries from H and C, we need to bring B_{i-2} and C_{i-2} into play.

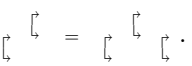
Once the required entries from $\tilde{R}$ have been generated, the entries of $\tilde{A}$ that we wish to generate can be obtained by applying $\mathcal{O}(1)$ core transformations from $\tilde{Q}$.

5. The algorithm. We now consider how to execute single and double steps of Francis's implicitly shifted QR algorithm [31] by directly operating on the factored form (4.4). In a standard QR step we disturb the Hessenberg structure by introducing a bulge at the top of the matrix, which is then chased by unitary similarity transformations to the bottom of the Hessenberg matrix until it slides off the matrix. For a detailed description see [32].

In our setting the bulge is represented by extra core transformations that are introduced and then chased through the factored form. First, we disturb the factorization by introducing the bulge (sections 5.2 and 5.5, for the single and the double shift, respectively), then we restore the factorization by chasing the bulge via unitary similarity transformations (sections 5.2 and 5.5) until it disappears (sections 5.4 and 5.7).

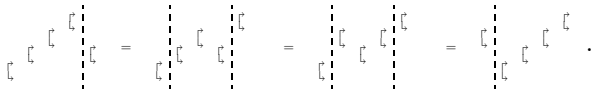
The algorithm utilizes two simple operations on core transformations called *fusion* and *turnover*. Two core transformations acting on the same rows can be *fused* into a single one, pictorially represented by

One can also change a factorization of core transformations between the following two forms:



This is the *turnover* operation, and it can be done in either direction. This is proved easily by thinking of computing the QR factorization of a 3×3 unitary matrix using, say, Givens rotations and R being the identity. It is also convenient to look at the turnover differently. Consider a core transformation on the right of an ascending

sequence. One can pass it through the sequence by a single turnover, and a new core transformation pops up on the left, positioned on the row above its original position (the vertical lines have no meaning except to group certain core transformations):



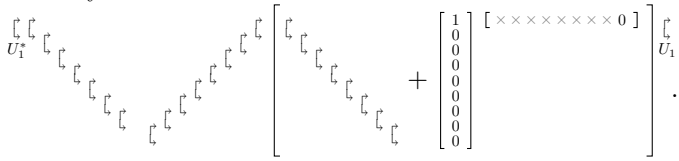
Similarly one can pass a core transformation from right to left through a descending sequence, thereby moving it down a single row.

5.1. Deflations and properness. The QR algorithm operates on proper Hessenberg matrices, that is, all subdiagonal elements are nonzero. Obviously, when zeros do appear on the subdiagonal, one can decouple the original problem into Hessenberg matrices of smaller size and compute their eigenvalues. This process is called deflation. A common criterion for checking whether a subdiagonal element numerically equals zero is to check whether the subdiagonal entries of the upper Hessenberg matrix are sufficiently small. However, when considering the matrix in the factored form (4.4), we see that a deflation is signaled by an almost diagonal core transformation Q_i (Theorem 4.5). Rather than explicitly computing all subdiagonal elements of the Hessenberg matrix, we will check for deflations by examining the Q_i: if the subdiagonal entry of interest is below the unit roundoff, we deflate. It is proved by Mach and Vandebril [24] that a deflation via core transformations of a single eigenvalue provides good relative backward error.

Once a deflation has occurred, we must operate on submatrices of the original matrix. This presents no difficulties, and we omit the details.

5.2. Single shift: Introducing the bulge. We begin by computing a shift μ by a standard shifting strategy (section 6). For this we need to construct the 2×2 submatrix in the lower-right-hand corner of A. We can do this in $O(1)$ flops as we have seen in section 4.1.

Then we must compute a vector $v = (A - \mu I)e_1$, for which we need a_{11} and a_{21}. We can compute these once we have r_{11}, which we obtain in one division as shown in section 4.1. Only the first two entries of v are nonzero. Let U_1 be a core transformation with first column proportional to v, that is, $U_1 e_1 = \gamma(A - \mu I)e_1$, and perform a similarity transformation with U_1. The resulting matrix $U_1^* A U_1$ can be described pictorially as

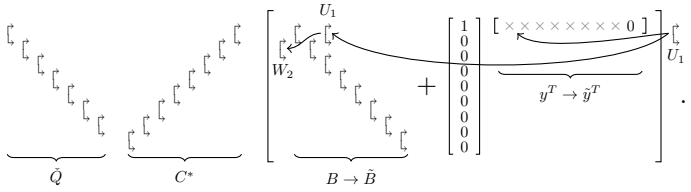


Clearly we can get rid of U_1^* by fusing it with Q_1. Then the representation is in the desired form, except for the core transformation U_1 on the right. This is the bulge. We must chase it through the matrix from right to left, from top to bottom, until it gets absorbed by performing a fusion.

5.3. Single shift: Chasing the bulge. We chase U_1 from right to left. First U_1 is applied on both terms in the factor $B+e_1y^T$; we get $(B+e_1y^T)U_1 = BU_1+e_1y^TU_1 = BU_1 + e_1\tilde{y}^T$. As B is a descending sequence of core transformations, we can pass U_1 through it by a single turnover, moving it down one position. We get $BU_1 = W_2\tilde{B}$. Pictorially we get (5.1), where each individual arrow corresponds to a specific action

to bring the core transformation U_1 more and more to the left. More precisely, we start with the two arrows leaving U_1 on the right. The top arrow expresses that U_1 is applied to y^T, thereby transforming y into $\tilde{y}$. The bottom arrow moves U_1 inside the brackets, to the right of B. The arrow starting from U_1 next to B indicates that the next step is to move U_1 to the left through the descending sequence by a turnover. As a result we get W_2, and B becomes $\tilde{B}$.

(5.1)



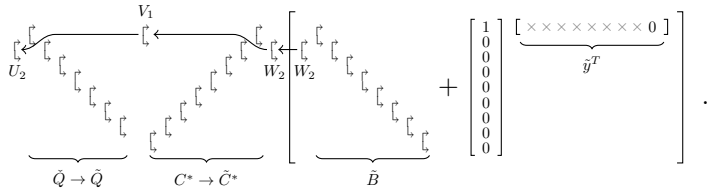
In our illustrations we are displaying the rank-one part for completeness, but it is important to remember that in our code we do not store or update y explicitly. The information about y is encoded in the B and C core transformations, as explained in Theorem 4.7. Whenever we need information about the rank-one part, we extract it from the core transformations as shown in Theorem 4.7.

The core transformation W_2 can be moved outside the brackets without affecting the rank one part as it does not touch the first row:

$$C^*(W_2\tilde{B} + e_1\tilde{y}^T) = C^*W_2(\tilde{B} + e_1\tilde{y}^T).$$

W_2 will become part of the matrix W as we mentioned in (4.6).

To bring the bulge W_2 completely to the left we pass it through the ascending sequence C^* by a turnover, moving it up a row and giving V_1. (V_1 becomes part of the matrix V in (4.6).) Then we pass it through the descending sequence Q, moving it back down a row, resulting in U_2.

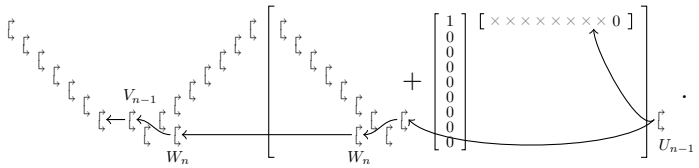


Now perform a similarity transformation by U_2. In the resulting matrix

$$U_2^*U_1^*AU_1U_2,$$

the bulge U_2 has disappeared from the left and has shown up on the right. We then pass U_2 through the matrix in exactly the same way as we did with U_1, resulting in a new, lower positioned, core transformation U_3 on the left. We then do a similarity transformation by U_3, moving it from the left to the right, pass it through the matrix again to obtain U_4, and so on. After $n-2$ steps we arrive at the bottom.

5.4. Single shift: Absorbing the bulge, end of the chase. The next figure illustrates the final pass through the matrix:



We pass U_{n-1} through the descending B sequence to produce W_n, which is then passed through the ascending C^* sequence to produce V_{n-1}. Because the descending Q sequence is shorter than the others, it is now possible to fuse V_{n-1} with Q_{n-1}. Once we do this, we have eliminated the bulge. We have returned the matrix to the form (4.4) so, by Theorem 4.5, the Hessenberg form has been restored.

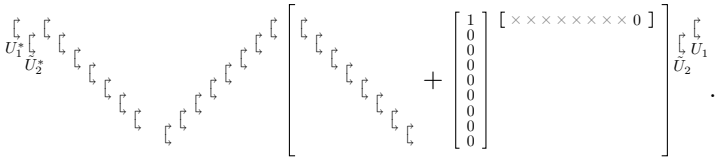
The entire similarity transformation is $\hat{A} = U^*AU$, where $U = U_1U_2\cdots U_{n-1}$. The core transformations $U_2, \ldots, U_{n-1}$ all satisfy $U_ie_1 = e_1$, so $Ue_1 = U_1e_1 = \gamma(A - \mu I)e_1$. We have effected a unitary similarity transformation from proper Hessenberg form to Hessenberg form with the "right" first column. Therefore we have executed an iteration of Francis's algorithm [31, Theorem 4.5.5], [32, Theorems 5.6.14, 6.3.12].

Notice that in the final step we created briefly a core transformation W_n that occupies rows/columns n and $n+1$. Row and column $n+1$ would not exist if we had not artificially adjoined them at the beginning. Because of the immediate deflation of the added zero eigenvalue, the Q sequence of core transformations is shorter than the others, allowing us to terminate the bulge chase with a fusion in Q_{n-1}.

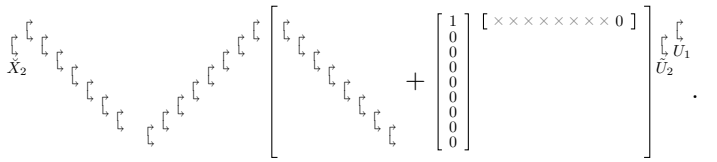
Remark 5.1. Early versions of our algorithm did not include the added row and column. As a consequence the bulge chase terminated prematurely with a fusion in the B sequence instead of the Q sequence. This meant that Hessenberg form had not been reached. To finish the operation we had to look into the R matrix and force one last entry to zero. For this purpose we needed to make use of the y vector, which we had to keep track of explicitly. This situation persisted until the first deflation was achieved, after which it became possible to complete the iteration without taking special action. We also noticed that from that point on we had no further need to keep track of the y vector. By adding an extra row and column with an artificial deflation, we were able to eliminate these complications.

5.5. Double shift: Introducing the bulge. In the single shift algorithm the bulge is represented by a single core transformation. In the double shift code three core transformations are needed to represent the bulge, but it turns out that we only need to pass two at a time through the matrix.

We begin by obtaining two shifts μ_1 and μ_2 by a standard shifting strategy. Then we compute the vector $v = (A-\mu_1 I)(A-\mu_2 I)e_1$, which has only the first three entries nonzero. Let $\tilde{U}_2$ and U_1 be two core transformations such that $U_1^*\tilde{U}_2^*v = \gamma^{-1}e_1$ for some γ, so that $\tilde{U}_2U_1e_1 = \gamma v$. Then carry out a similarity transformation to produce $U_1^*\tilde{U}_2^*A\tilde{U}_2U_1$. Pictorially, we have

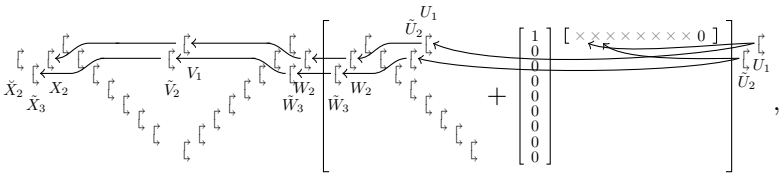


On the left side of the picture perform a turnover of $U_1^*\tilde{U}_2^*Q_1$ to produce $\breve{X}_2\hat{Q}_1\breve{Q}_2$. Then fuse $\breve{Q}_2$ with Q_2 to form $\hat{Q}_2 = \breve{Q}_2Q_2$. We now have



5.6. Double shift: Chasing the bulge. In each chasing step we will bring both core transformations on the outer right to the outer left and execute a similarity to swap them back to the right. After the similarity we end up again with two core transformations on the right and a single one on the left, all positioned one row lower than before.

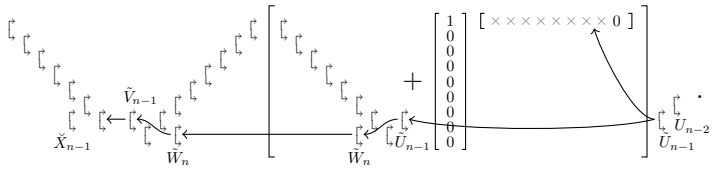
Let us illustrate the flow pictorially. Moving the two right core transformations to the left proceeds identically to the single shift case: apply them to the two terms, pass them through the B sequence, bring them outside the brackets, pass them through the C^* sequence, and finally go through the Q sequence to arrive on the left side. Because $\tilde{U}_2$ is positioned to the left of U_1, $\tilde{U}_2$ should go first. We get



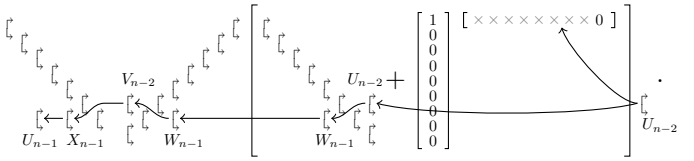
Now do a turnover of $\check{X}_2\tilde{X}_3X_2$ to produce $\tilde{U}_3U_2\check{X}_3$. Then do a similarity transformation with $\tilde{U}_3U_2$ to move those two core transformations from the left side to the right. This concludes the first chasing step. The next step is identical to the first, except that everything has been moved down by one.

We remind the reader that although we are showing the rank-one part in the pictures for completeness, we do not actually store or update y explicitly.

5.7. Double shift: Absorbing the bulge, end of the chase. After $n-3$ chasing steps the core transformations reach the bottom of the matrix, where they will be absorbed. We will pictorially depict what happens to core transformations $\tilde{U}_{n-1}$ and U_{n-2} individually. First $\tilde{U}_{n-1}$ goes through the B and C^* sequences and fuses with the bottom core transformation in the Q sequence:



Then bring core transformation U_{n-2} through the B, C^*, and Q sequences, where it can be fused with $\check{X}_{n-1}$, leaving a single core transformation U_{n-1} on the left.



Perform a similarity transformation with U_{n-1} to move it from the left side to the right. Then pass it through the matrix one more time and fuse it with the bottom core transformation in the Q sequence, exactly as in the single shift case. The bulge has been absorbed. We have returned the matrix to the form (4.4), and the iteration is complete.

The entire similarity transformation is $\hat{A} = U^* A U$, where

$$U = \tilde{U}_2 U_1 \tilde{U}_3 U_2 \cdots \tilde{U}_{n-1} U_{n-2} U_{n-1}.$$

Since $U_i e_1 = e_1$ and $\tilde{U}_i e_1 = e_1$ for $i > 1$, we have

$$U e_1 = \tilde{U}_2 U_1 e_1 = \gamma (A - \mu_1)(A - \mu_2) e_1.$$

We have effected a unitary similarity transformation with the "right" first column, transforming a properly upper Hessenberg matrix to an upper Hessenberg matrix. Therefore we have executed a Francis iteration of degree two [31, Theorem 4.5.5], [32, Theorems 5.6.14, 6.3.12].

6. Implementation. Two versions of the algorithm are implemented in Fortran 90. There are the *complex single shift* (CSS) code to retrieve the roots of complex polynomials and the *real double shift* (RDS) code to only deal with real arithmetic when dealing with real polynomials. In the next sections we briefly discuss the storage scheme, the basic operations, and the heuristics used.

6.1. Data storage. As core transformations we took rotators with real s,

$$\begin{bmatrix} c & -s \\ s & \overline{c} \end{bmatrix}, \tag{6.1}$$

where $|c|^2 + s^2 = 1$. Three sequences of these rotators, of which only c and s are required, need to be stored.

In the RDS code all core transformations are real and remain so during the iterations. Both c and s are real numbers, leading to a storage cost of roughly $6n$ reals.

In the CSS setting, if one wishes to keep the s entries real, more effort is required. This restriction demands that we start with an upper Hessenberg matrix whose subdiagonal entries are all real. This can always be arranged and is already fulfilled for companion matrices. In section 6.2 we will see that the turnover will not cause problems; only the fusion does. Fusing core transformations with real s results in a single core transformation for which the s value is typically not real. This problem can be remedied by including an extra unitary diagonal matrix D in the Q factor: $Q = Q_1 \cdots Q_{n-1} D$, where D contains some phase factors. Details are given below. The actual factored form utilized in the CSS code is therefore

$$A = QR = (Q_1 \cdots Q_{n-1}) D (C_n^* \cdots C_1^*)(B_1 \cdots B_n + \alpha e_1 y^T).$$

As a result we need to store a complex c, stored as two reals, and a real s for each core transformation. The diagonal D is complex, and each element takes up two reals. In total we get a storage cost of approximately $11n$ reals.

6.2. Operations. The next paragraphs describe how to execute a fusion and a turnover and pass a core transformation through a diagonal for both the CSS and RDS codes.

Turnover. Executing a turnover is equivalent to computing a QR factorization of

$$\begin{bmatrix} c_1 & -s_1 & \\ s_1 & \overline{c}_1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & c_2 & -s_2 \\ & s_2 & \overline{c}_2 \end{bmatrix} \begin{bmatrix} c_3 & -s_3 & \\ s_3 & \overline{c}_3 & \\ & & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_1 c_3 - s_1 c_2 s_3 & -c_1 s_3 - s_1 c_2 \overline{c}_3 & s_1 s_2 \\ s_1 c_3 + \overline{c}_1 c_2 s_3 & -s_1 s_3 + \overline{c}_1 c_2 c_3 & -\overline{c}_1 s_2 \\ s_2 s_3 & s_2 c_3 & \overline{c}_2 \end{bmatrix} = \quad .$$

The first two rotators are computed from the first column. After updating the last column we can compute the final rotator. As a result we can simply ignore the second column. Rotators creating zeros in entries can always be chosen such to have real s, and as a consequence the turnover keeps the s's real.

Passing a rotator through a diagonal. In the CSS code we also need to accomplish

$$\begin{bmatrix} d & 0 \\ 0 & e \end{bmatrix} \begin{bmatrix} c_1 & -s_1 \\ s_1 & \overline{c}_1 \end{bmatrix} = \begin{bmatrix} c_2 & -s_2 \\ s_2 & \overline{c}_2 \end{bmatrix} \begin{bmatrix} f & 0 \\ 0 & g \end{bmatrix}.$$

To this end take $f = e$, $g = d$, $s_2 = s_1$, and $c_2 = c_1 d\,\overline{e}$.

Fusion. We have

$$\begin{bmatrix} c_1 & -s_1 \\ s_1 & \overline{c}_1 \end{bmatrix} \begin{bmatrix} c_2 & -s_2 \\ s_2 & \overline{c}_2 \end{bmatrix} = \begin{bmatrix} c_1c_2 - s_1s_2 & -s_1c_2 - \overline{c}_1s_2 \\ s_1\overline{c}_2 + c_1s_2 & \overline{c}_1\overline{c}_2 - s_1s_2 \end{bmatrix} = \begin{bmatrix} c'_3 & -\overline{s'}_3 \\ s'_3 & \overline{c'}_3 \end{bmatrix},$$

where $s'_3 = s_1\overline{c}_2 + c_1s_2$ is not necessarily real when dealing with complex numbers (in the RDS case there are no issues). As a consequence the diagonal D is involved in a fusion in the CSS code. We compute

$$\begin{bmatrix} c'_3 & -\overline{s'}_3 \\ s'_3 & \overline{c'}_3 \end{bmatrix} = \begin{bmatrix} c_3 & -s_3 \\ s_3 & \overline{c}_3 \end{bmatrix} \begin{bmatrix} f & 0 \\ 0 & g \end{bmatrix},$$

which is realized by setting $s_3 = |s'_3|$, $\phi = s'_3/s_3$, $c_3 = c'_3\phi$, $f = \phi$, and $g = \overline{\phi}$. The values f and g are then incorporated into the diagonal D.

6.3. Heuristics and tunings.

Shift strategy. The *Wilkinson shift* [34] and the *Rayleigh quotient shift* are the most popular shift strategies for single and double shift QR algorithms. For the RDS code the eigenvalues of the trailing 2×2 submatrix under consideration are used as shifts, which are Rayleigh quotient shifts. This ensures that we can stick to real arithmetic during the entire QR algorithm. In the CSS case the Wilkinson shift, that is, the eigenvalue of the trailing 2×2 block closest to the last diagonal element of the matrix under consideration, is used.

Deflation. A rotator is assumed to signal a deflation if $|s| < \epsilon_m \approx 2.22 \cdot 10^{-16}$, where ϵ_m stands for the machine precision. In the CSS code the rotator is set explicitly to the identity and the unimodular factors are put in the diagonal matrix D. In the RDS the rotator is explicitly set to a diagonal matrix with ± 1 on its diagonal.

We search for deflations starting at the bottom of the matrix. After a deflation is detected the iterations are executed on the above positioned proper Hessenberg matrix larger than 2×2. The eigenvalues of 2×2 blocks are explicitly computed via the modified quadratic formula.

Square roots. During the turnover, the fusion, and the passing through a diagonal we have to ensure that the computed rotators continue to satisfy $|c|^2 + s^2 = 1$. If this condition is not enforced explicitly, the rotators will lose their unitarity due to roundoff errors over time, and the algorithm will fail. Thus each new triplet (c_{real}, c_{imag}, s) or pair (c, s) that we compute is renormalized: we compute $\eta = c_{real}^2 + c_{imag}^2 + s^2$ or $\eta = c^2 + s^2$, then compute $\sqrt{\eta}$ and divide through by this quantity. However, the square root is very expensive relative to the other operations, so we do the square root computation only if $|\eta - 1| > \epsilon_m$. Otherwise we just set $\sqrt{\eta} = 1$. This simple trick saves about 30% of the computing time.

Exploiting $C_i = B_i$ in early stages. In the first iteration $B_i = C_i$ for $i = 1, \ldots, n-1$. After each iteration the number of coinciding rotators decreases by one. Exploiting this by saving the number of turnovers explicitly executed saves approximately 10% of the computing time.

7. Backward stability. Our analysis will push the error back onto the companion matrix, not onto the coefficients of the polynomial. For the latter we refer the reader to [15, 17, 29].

We begin by noting that our initial factorization $A = QR = QC^*(B + e_1 y^T)$ is backward stable. That is, the exact matrix $QC^*(B + e_1 y^T)$ differs from A by an amount that is on the order of the unit roundoff multiplied by $\| A \|_2$. This follows from elementary considerations. In particular, operations by Givens rotators are backward stable [22].

Now it suffices to show that a single step of the algorithm is backward stable. In exact arithmetic we have $\hat{A} = U^* A U$, and we want to show that in floating point the computed $\hat{A}$ satisfies $\hat{A} = U^*(A + E)U$, where $\| E \|_2$ is tiny relative to $\| A \|_2$. Since A is built from Q, C, and B, we begin by obtaining backward error results for these pieces separately. Recall that in exact arithmetic we have

$$\hat{A} = \hat{Q}\hat{C}^*(\hat{B} + e_1 \hat{y}^T) = (U^* Q V)(V^* C^* W)(W^* B U + e_1 y^T U),$$

and in particular (in exact arithmetic)

$$\hat{Q} = U^* Q V, \qquad \hat{C} = W^* C V, \qquad \text{and} \qquad \hat{B} = W^* B U.$$

THEOREM 7.1. *After one step of the algorithm in floating-point arithmetic,*

$$\hat{Q} = U^*(Q + E_q)V, \qquad \hat{C} = W^*(C + E_c)V, \qquad \textit{and} \qquad \hat{B} = W^*(B + E_b)U,$$

where each of $\| E_q \|_2$, $\| E_c \|_2$, and $\| E_b \|_2$ is a modest multiple of the unit roundoff u.

Proof. First consider the matrix B. The transformation to $\hat{B}$ is effected entirely by turnovers that pass rotators through the descending sequence of B rotators. Writing the (exact arithmetic) equation as $W\hat{B} = BU$, we see that U is the product of all the rotators that were used to initiate turnovers in B, and W is the product of all the rotators that came out of such turnovers.

It suffices to consider a single turnover $W_{i+1}\tilde{B}_i\tilde{B}_{i+1} = B_i B_{i+1} U_i$. In practice the turnover is executed by multiplying together the incoming matrices, then performing a QR decomposition by rotators on the product. Since these operations are backward stable [22], we deduce that in floating-point arithmetic

$$W_{i+1}\tilde{B}_i\tilde{B}_{i+1} = B_i B_{i+1} U_i + E_t,$$

where $\| E_t \|_2$ is on the order of u. There is no complicating factor like $\| A \|_2$ because all the participating matrices are unitary and have norm 1. In terms of the big matrices we have

$$W_{i+1}\tilde{B} = BU_i + E_t = (B + E_{t'})U_i,$$

where $E_{t'} = E_t U_i^*$ has the same norm as E_t. This then gives

$$\tilde{B} = W_{i+1}^*(B + E_{t'})U_i,$$

which shows that the turnover results in a tiny backward error in B. Since the whole transformation from B to $\hat{B}$ is just a sequence of such operations, we deduce that $\hat{B} = W^*(B + E_b)U$, where $\| E_b \|_2$ is a modest multiple of the unit roundoff u.

The exact same argument applies to C. It applies to Q as well with some slight modifications. In our complex code the matrix Q includes a unitary diagonal factor D, and rotators need to be passed through D. These operations are trivial and easily seen to have tiny backward errors. The operations on Q also include a couple of fusions, one each at the beginning and the end of the iteration. These are just matrix multiplications, and they are backward stable. ∎

THEOREM 7.2. *Let $\hat{A}$ be the result of one step of the algorithm in floating-point arithmetic, starting from A. Then $\hat{A} = U^*(A + E)U$, where $\| E \|_2 / \| A \|_2$ is a modest multiple of the unit roundoff u.*

Proof. To avoid notational congestion we consider the unitary and rank-one parts separately. $A = A_u + A_r$, where $A_u = QC^*B$ and $A_r = QC^*e_1y^T$. For the unitary part we have, by Theorem 7.1,

$$\hat{A}_u = \hat{Q}\hat{C}^*\hat{B} = U^*(Q + E_q)(C + E_c)^*(B + E_b)U.$$

Thus

$$\hat{A}_u = U^*(A_u + E_u)U,$$

where

$$\| E_u \|_2 \leq \| E_q \|_2 + \| E_c \|_2 + \| E_b \|_2 + \cdots .$$

The dots denote higher-order terms.

Now consider the rank-one part $A_r = QC^*e_1y^T$. From Theorem 4.7 we know that $y^T = \beta \| x \|_2 e_{n+1}^T C^*B$, where β is a scalar satisfying $|\beta| = 1$, so

$$A_r = \beta \| x \|_2 QC^*e_1e_{n+1}^T C^*B.$$

Again applying Theorem 7.1 we have

$$\begin{aligned}
\hat{A}_r &= \beta \| x \|_2 \hat{Q}\hat{C}^*e_1e_{n+1}^T\hat{C}^*\hat{B} \\
&\qquad\qquad \text{(using } We_1 = e_1 \text{ and } e_{n+1}^T V^* = e_{n+1}^T\text{)} \\
&= \beta \| x \|_2 U^*(Q + E_q)(C + E_c)^*e_1e_{n+1}^T(C + E_c)^*(B + E_b)U \\
&= U^*(\beta \| x \|_2 QC^*e_1e_{n+1}^T C^*B + E_r)U \\
&= U^*(A_r + E_r)U,
\end{aligned}$$

where

$$\| E_r \|_2 \leq \| x \|_2 \left(\| E_q \|_2 + 2\| E_c \|_2 + \| E_b \|_2 + \cdots \right).$$

Noting that $1 \leq \| x \|_2 \leq \| A \|_2$, and in fact $\| x \|_2 \approx \| A \|_2$; we deduce that $\| E_r \|_2$ is a modest multiple of $u \| A \|_2$.

Combining the unitary and rank-one parts we have $\hat{A} = U^*(A + E)U$, where $E = E_u + E_r$, and $\| E \|_2$ is a modest multiple of $u \| A \|_2$. ∎

Applying Theorem 7.2 repeatedly, we find that the whole process is backward stable: The matrix we have at the end of the iterations is unitarily similar to a matrix that is very close to the matrix we started with. We now show that the final eigenvalue

TABLE 1
Number of runs for each polynomial degree considered.

Deg.	No. runs	Deg.	No. runs	Deg.	No. runs	Deg.	No. runs
6	16,384	14	2048	128	128	2,048	8
8	16,384	16	1024	256	64	4,096	4
10	8,192	32	512	512	32	8,192	2
12	4,096	64	256	1024	16	16,384	1

extraction process is also backward stable. In the complex case this is trivial. The final matrix is upper triangular, and its eigenvalues are the main diagonal entries. These are of the form $d_i r_{ii}$, where d_i is the ith main diagonal entry of the diagonal matrix D. In section 4 we saw that r_{ii} can be obtained by a single division (4.12). Thus the computation of each eigenvalue requires one division and one multiplication. These are backward stable operations.

The real case is not much harder. In this case the final matrix is quasi-triangular with 2×2 blocks representing complex conjugate eigenvalues. We need to generate these 2×2 blocks. The process was explained in section 4. In particular a 2×2 upper triangular system (4.14) has to be solved by back substitution. Before this can be solved, it has to be set up via two matrix multiplications by core transformations. After it is solved, one additional matrix multiplication by a core transformation Q_i is required. Thus all the operations involved in this process are either matrix multiplications or back substitutions, and these operations are backward stable [22].

8. Numerical experiments. Speed and accuracy of both the single and double shift implementation (AMVW) are examined and compared with other companion QR algorithms. We compared against LAPACK's Hessenberg eigenvalue solver (xHSEQR); LAPACK's eigensolver for general matrices (xGEEV);[2] the method of Boito et al. [12] (BEGG); and the algorithm of Chandrasekaran et al. [14] (CGXZ). We note that BEGG has only a single shift implementation and CGXZ is available only as a double shifted version.

We also experimented with the codes of Bini et al. [6], available in both single shift and double shift versions. We found that their single shift code was slightly slower than BEGG, and their double shift code was marginally faster than CGXZ. However, these codes were often much less accurate than the others. Therefore, we did not included the results from these codes.

The computations were executed on an Intel Core i5-3570 CPU running at 3.40 GHz with 8 GB of memory. GFortran 4.6.3 was used to compile the Fortran codes. For the comparison LAPACK version 3.5 was used.

We have data from four categories of experiments: polynomials with random coefficients, polynomials with roots of unity of the form $z^n - 1$, special polynomials used for testing polynomial solvers [9, 14, 23, 27], and polynomials designed to test the stability of the code. In the first two experiments (see sections 8.1 and 8.2), the computing time was examined. The depicted runtime is averaged over a decreasing number of runs, as shown in Table 1. The accuracy is investigated in all experiments, with the error measure adapted to the type of experiment and described further on.

8.1. Polynomials with random coefficients. Polynomials with random coefficients in the monomial basis are known to be well-conditioned as their eigenvalues

[2] xHSEQR and xGEEV are either the complex implementations ZHSEQR and ZGEEV or the real double precision ones DHSEQR and DGEEV. xGEEV balances the problem first.

are typically located around the unit circle. The coefficients are normally distributed with mean 0 and variance 1. For testing the single shift code, complex coefficients were used, while real polynomials were piped to the double shift code.

The measure of accuracy for a single problem size is the maximum of all relative residuals of all computed eigenpairs over 10 runs, where the relative residual for a particular eigenpair (λ, v) equals

$$\frac{\|Av - \lambda v\|_\infty}{\|A\|_\infty \|v\|_\infty}. \tag{8.1}$$

For avoiding over- and underflow and for easily computing the eigenvectors we refer to [2].

Figures 1 and 2 illustrate that the accuracy of our proposed algorithm AMVW is comparable to that of LAPACK, significantly better than BEGG, and slightly better than CGXZ. Considering speed, we note that our algorithm is more than three times faster than BEGG in the single shift case and becomes faster than LAPACK for polynomials of degree 12 or greater. In the double shift case the speedup is less pronounced, but the AMVW time is still less than half that of CGXZ. The crossover with LAPACK now takes place at degree 16.

Jenkins and Traub [23] state that polynomials with normally distributed random coefficients are a "poor choice as the randomness 'averages out' in the coefficients and the polynomials differ but little from each other". Therefore we also used the test set (iv) from [23], polynomials with random coefficients $a_j = m_j \cdot 10^{e_j}$, having m_j uniformly distributed in $(-1, 1)$ and e_j uniformly distributed in $(-\mu, \mu)$, with $\mu = 5, 10, 15, 20, 25$. The results were very similar to those for normally distributed coefficients, so we have not displayed them.

8.2. Polynomials $z^n - 1$. The polynomials with roots of unity $z^n - 1$ also have well-conditioned roots as the companion matrix itself is unitary. Furthermore, the roots lie on the unit circle and are known exactly: $z_j = \cos(\frac{2j}{n}\pi) + i \cdot \sin(\frac{2j}{n}\pi)$, $j = 0, \ldots, n-1$, where i is the imaginary unit. The accuracy here is the maximum absolute difference between the computed and the exact roots, which also equals the maximum relative difference.

The results shown in Figures 3 and 4 are along the same lines as for the random case. In the single shift case the algorithm's accuracy is comparable to that of LAPACK and better than BEGG. For the double shift case, it performs slightly better than the other approaches. In the single shift case, it is more than three times faster than the fastest currently available approach, in the double shift case, twice as fast.

8.3. Special real polynomials. In this section we compute roots of difficult polynomials and report the forward and backward errors of the various methods. All the test polynomials have real coefficients and we used them to test both the real and the complex codes. In cases where the roots are known either exactly or to high accuracy, we report the maximum relative forward error. To get a correct measure of backward error, we take the computed roots and use extended precision arithmetic to compute the coefficients $\hat{a}_i$ of a polynomial having those roots. We then compare these to the original a_i. To compute the $\hat{a}_i$ we used the multiprecision arithmetic MPFUN [26] and a function from the CGXZ code [14]. The backward error measure is the maximal error on the coefficients relative to the norm of the coefficients

$$\max_i \frac{|a_i - \hat{a}_i|}{\|x\|_2} \tag{8.2}$$

with $x = -[a_1, \ldots, a_{n-1}, a_0, 1]$.

TABLE 2
Special polynomials tested by Chandrasekaran et al. in [14].

No.	Description	Deg.	Roots
1	Wilkinson polynomial	10	$1, \ldots, 10$
2	Wilkinson polynomial	15	$1, \ldots, 15$
3	Wilkinson polynomial	20	$1, \ldots, 20$
4	scaled and shifted Wilkinson poly.	20	$-2.1, -1.9, \ldots, 1.7$
5	reverse Wilkinson polynomial	10	$1, 1/2, \ldots, 1/10$
6	reverse Wilkinson polynomial	15	$1, 1/2, \ldots, 1/15$
7	reverse Wilkinson polynomial	20	$1, 1/2, \ldots, 1/20$
8	prescribed roots of varying scale	20	$2^{-10}, 2^{-9}, \ldots, 2^{9}$
9	prescribed roots of varying scale -3	20	$(2^{-10} - 3), \ldots, (2^{9} - 3)$
10	Chebyshev polynomial	20	$\cos(\frac{2j-1}{40}\pi)$
11	$z^{20} + z^{19} + \cdots + z + 1$	20	$\cos(\frac{2j}{21}\pi)$

TABLE 3
Special polynomials used to test MPSolve in [9].

No.	Description	Deg.	Roots
12	trv_m, C. Traverso	24	known
13	mand31 Mandelbrot example ($k = 5$)	31	known
14	mand63 Mandelbrot example ($k = 6$)	63	known

Balancing. The LAPACK codes that we have used in these tests are xHSEQR, which does not balance, and xGEEV, which has a balancing step. As the tables below show, balancing is occasionally very helpful, but there are also examples where balancing is detrimental.

Since our algorithm relies on the unitary-plus-rank-one structure of the companion matrix, we do not have as much freedom to balance the matrix as one has in the general case. In particular, the diagonal balancing strategy used by xGEEV cannot be used. However, we do have the freedom to replace $p(z)$ by $\alpha^{-n}p(\alpha z)$, where α is a positive scalar chosen to even out the coefficients of the polynomial. Currently, an efficient and reliable strategy for choosing α is not known; this is a question that requires further study. We implemented one very simple strategy here: choose α so that the magnitude of the constant term is 1. All solvers were tested on both the original polynomials and on polynomials balanced in this way. The results for the balanced polynomials were reported only in those cases where there was a significant difference between results for the balanced and unbalanced polynomials. In some cases balancing was beneficial, but in other cases it was harmful.

The test set. The first 11 polynomials, Table 2, are polynomials also tested in [14], for which the roots are known: Wilkinson polynomials and polynomials with particular distributions of the roots. We added polynomial 9 because it triggers a special behavior of the CGXZ code compared with polynomial 8.

Some polynomials taken from MPSolve [9] are given in Table 3. We assume the roots computed by MPSolve, using variable precision arithmetic, to be exact. The MPSolve collection contains many polynomials deliberately designed to be too ill-conditioned to solve using double precision arithmetic; we did not report on those. In [9] it is stated that the polynomial trv_m, provided by Carlo Traverso, arises from the symbolic processing of a system of polynomial equations and has multiple roots. The Mandelbrot polynomials are defined iteratively as follows: $p_0(z) = 1$, $p_i(z) = zp_{i-1}(z)^2 + 1$, for $i = 1, 2, \ldots, k$, with $n = 2^k - 1$.

TABLE 4
Special polynomials provided by V. Noferini.

No.	Description	Deg.	Roots
15	polynomial from V. Noferini	12	almost random
16	polynomial from V. Noferini	35	almost random

TABLE 5
Polynomials to test root finding algorithms from [23].

No.	Description	Deg.	Roots
17	$p_1(z)$ with $a =$1 e−8	3	1 e−8, -1 e−8, 1
18	$p_1(z)$ with $a =$1 e−15	3	1 e−15, -1 e−15, 1
19	$p_1(z)$ with $a =$1 e+8	3	1 e+8, -1 e+8, 1
20	$p_1(z)$ with $a =$1 e+15	3	1 e+15, -1 e+15, 1
21	$p_3(z)$ underflow test	10	1 e−1, . . . ,1 e−10
22	$p_3(z)$ underflow test	20	1 e−1, . . . ,1 e−20
23	$p_{10}(z)$ deflation test $a =$ 10 e+3	3	1, 1 e+3, 1 e−3
24	$p_{10}(z)$ deflation test $a =$ 10 e+6	3	1, 1 e+6, 1 e−6
25	$p_{10}(z)$ deflation test $a =$ 10 e+9	3	1, 1 e+9, 1 e−9
26	$p_{11}(z)$ deflation test $m = 15$	60	$\exp(\frac{ik\pi}{2m})$, $0.9\exp(\frac{ik\pi}{2m})$

TABLE 6
Bernoulli and truncated exponential.

No.	Description	Deg.	Roots
27	Bernoulli polynomial ($k = 20$)	20	—
28	truncated exponential ($k = 20$)	20	—

The examples in Table 4, were created for us by Vanni Noferini. The polynomials have normally distributed real roots with mean 0 and standard deviation 1, where one random root is scaled by 1 e+12 and another one by 1 e+9. These polynomials are difficult to balance.

The polynomials in Table 5 are from Jenkins and Traub [23] and were designed to test particular properties of rootfinding methods. The cubic polynomial $p_1(z) = (z-a)(z+a)(z-1)$, used in test cases 17−20, examines the termination criteria, i.e., convergence difficulties; $p_3(z) = \prod_{i=1}^{n}(z-10^i)$ checks the occurence of underflow; and $p_{10}(z) = (z-a)(z-1)(z-a^{-1})$ and $p_{11}(z) = \prod_{k=1-m}^{m-1}(z-\exp(\frac{ik\pi}{2m}))\prod_{k=m}^{3m}(z-0.9\exp(\frac{ik\pi}{2m}))$ having zeros on two half-circles examine the deflation strategy.

Table 6 reports the Bernoulli polynomial $\sum_{k=0}^{n}\binom{n}{k}b_{n-k}z^k$, with the Bernoulli numbers of the first kind b_{n-k} and the truncated exponential $k!\sum_{k=0}^{n}\frac{z^k}{k!}$.

The polynomials in Table 7 were used in [4] and originate from [1, 10, 13]:

$$p_1(z) = 1 + \left(\frac{m}{m+1} + \frac{m+1}{m}\right) z^m + z^{2m},$$

$$p_2(z) = \frac{1}{m}\left(\sum_{j=0}^{m-1}(m+j)z^j + (m+1)z^m + \sum_{j=0}^{m-1}(m+j)z^{2m-j}\right),$$

$$p_3(z) = (1-\lambda)z^{m+1} - (\lambda+1)z^m + (\lambda+1)z - (1-\lambda).$$

These polynomials exhibit particular symmetries, with $p_1(z)$ and $p_2(z)$ palindromic and $p_3(z)$ antipalindromic.

TABLE 7
Polynomials tested in [4]*, coming from* [1, 10, 13].

No.	Description	Deg.	Roots
29–33	$p_1(z)$ with $m = 10, 20, 30, 256, 512$	$2m$	—
34–38	$p_2(z)$ with $m = 10, 20, 30, 256, 512$	$2m$	—
39–43	$p_3(z)$ with $m+1 = 20, \ldots, 1024$, $\lambda = 0.9$	$m+1$	—
44–48	$p_3(z)$ with $m+1 = 20, \ldots, 1024$, $\lambda = 0.999$	$m+1$	—

Discussion. For the single and double shift codes, forward and backward errors of both balanced and unbalanced polynomials are reported. Note, however, that the results of the unbalanced polynomials are shown only for those cases where balancing had a significant impact. To make a fair comparison with an unstructured QR algorithm on the companion matrix, we added to each table a column containing the results linked to xGEEV, allowing for a more advanced balancing strategy. This also implies that the results of xGEEV in Tables 9, 11, 13, and 15 originate from a balanced polynomial, put in a companion matrix balanced once more before its roots were computed.

Tables 8 and 9 depict the results for the single shifted code, without and with balancing, respectively. Comparing first LAPACK's balanced (ZGEEV) and unbalanced (ZHSEQR) solvers we see that in general the additional scaling does improve the forward error, as seen in polynomials 5, 6, 8, 12, and 17–22. Only the polynomials 15 and 16 seem to suffer from the balancing. We can deduce that the forward error of AMVW is almost always in close proximity to that of LAPACK's ZHSEQR, except for polynomials 3, 12, 17, 18, and 21. Balancing those polynomials brings the forward error up to the same level as ZHSEQR and sometimes even better, as illustrated by polynomials 17 and 18. However, for polynomial 8 we seem to be unable to achieve the accuracy of the ZGEEV method with full balancing. Moreover, one should be careful with balancing, for example, for polynomial 20 AMVW loses five decimal places w.r.t. the unbalanced version, whereas BEGG gains 8; furthermore, for polynomials 15 and 16 the balancing is disastrous.

Tables 10 and 11 report the backward errors for the single shifted code, without and with balancing, respectively. Overall, we can state by looking at Table 10 that the backward error is almost always at the level of machine precision and is comparable to the error of LAPACK's ZHSEQR and ZGEEV. Moreover, we have an excellent backward error for polynomials 15 and 16, whereas ZGEEV loses many digits. We perform well for polynomials 19 and 20, whereas ZHSEQR fails completely. Only for polynomial 28 are we two or three digits behind. Polynomial 28 is the only case where balancing seems to help AMVW. For all other cases the balancing has no effect or a negative effect on the backward error; for example, we record a dramatic loss for polynomial 16. Also for the backward error, the effect of the balancing strongly depends on the method, for example, ZHSEQR behaves dissimilarly in case 19 compared to cases 15 and 16.

Tables 12 and 13 report the data of the double shift code with and without balancing. The forward errors of all three methods, excluding DGEEV, are typically comparable; except for polynomials 8, 12, and 18 our algorithm is worse than CXGZ, and for 12 is also much worse than DHSEQR. Except for polynomials 15 and 16, DGEEV provides the best forward error. After balancing, we achieve an error comparable to the one of CXGZ for 12 and 18, but not for polynomial 8. Polynomial 8 is an interesting case. The representation used by CXGZ and the fact that it operates

TABLE 8
Unbalanced single shift version: relative forward errors.

No.	AMVW	ZHSEQR	ZGEEV	BEGG
1	2.1050 e−10	2.2132 e−11	1.8227 e−10	2.6921 e−11
2	5.0840 e−06	1.9949 e−06	2.9499 e−06	6.2327 e−08
3	6.4836 e+00	4.7536 e−03	3.0706 e−03	9.9687 e−01
4	3.4537 e−12	9.9365 e−13	9.6145 e−13	8.2891 e−13
5	2.9382 e−06	3.8860 e−06	8.2346 e−10	4.3172 e−06
6	4.3553 e−01	4.7760 e−01	1.0849 e−04	5.1168 e−01
7	2.1438 e+00	2.4780 e+00	2.7442 e−01	2.2530 e+00
8	3.3777 e−01	1.5862 e+00	2.8288 e−13	1.0000 e+00
9	6.6578 e−02	3.5919 e−02	4.7317 e−02	8.8937 e−01
10	1.2156 e−10	3.6616 e−11	5.3622 e−12	2.6118 e−11
11	1.5779 e−15	3.0227 e−15	2.2861 e−15	1.3545 e−14
12	1.1470 e+02	2.3742 e−04	7.0187 e−08	2.0689 e−04
13	1.2361 e−06	5.9051 e−06	1.0481 e−06	2.9379 e−07
14	2.3801 e−01	2.3893 e−01	2.1137 e−01	1.8421 e−01
15	6.2579 e−13	4.1005 e−13	6.4643 e−09	4.5861 e+00
16	3.1063 e−04	1.5007 e−04	7.6375 e−02	9.5143 e+00
17	5.3671 e−02	6.4531 e−09	2.2830 e−14	1.0000 e+00
18	5.2684 e+06	4.5303 e−02	1.5777 e−15	1.0000 e+00
19	5.0000 e−09	4.5648 e−01	4.4703 e−16	1.2872 e−01
20	7.5000 e−16	7.0539 e+06	1.2500 e−16	2.5385 e+13
21	1.8865 e+07	1.0000 e+00	3.2526 e−15	9.1219 e+07
22	2.8557 e+16	9.7471 e+15	1.2585 e−13	1.5085 e+17
23	4.0658 e−16	2.2204 e−16	1.3323 e−15	2.1103 e−12
24	5.2940 e−16	3.7253 e−16	4.4409 e−16	5.2636 e−10
25	1.2925 e−16	1.1102 e−15	7.7716 e−16	1.7481 e−06
26	3.9438 e−08	6.8424 e−08	1.5569 e−08	3.3993 e−08

TABLE 9
Balanced single shift version: relative forward errors.

No.	AMVW	ZHSEQR	ZGEEV	BEGG
3	1.0936 e−02	1.6164 e−02	1.3548 e−02	3.0069 e−03
5	1.3543 e−09	3.3472 e−10	1.9049 e−10	1.7691 e−10
6	2.4207 e−06	4.6529 e−05	6.5809 e−07	4.8093 e−07
7	7.3168 e−02	7.8505 e−02	3.5468 e−02	9.3801 e−04
8	2.6127 e−03	1.0478 e−01	1.0003 e−13	4.2559 e+00
12	9.4884 e−08	6.5912 e−08	6.5913 e−08	6.5914 e−08
15	1.7688 e−08	1.6919 e−01	3.7136 e−08	1.4816 e+01
16	4.0845 e+00	2.3640 e+00	2.8642 e−01	5.5635 e+00
17	3.3307 e−16	8.0898 e−14	8.0898 e−14	1.9533 e−12
18	2.2204 e−16	1.3331 e−11	1.3331 e−11	7.8722 e−08
19	5.8531 e−13	2.4438 e−14	7.4506 e−16	6.7711 e−13
20	5.0957 e−11	3.7500 e−16	2.5000 e−16	2.0993 e−08
21	1.7896 e−08	6.7911 e−09	4.1200 e−15	1.8439 e−07

on a row companion matrix[3] seem to make CXGZ particularly suited to compute roots of this polynomial with excellent forward error. However, the backward error of all methods is excellent as shown in Table 14. But, if one shifts the roots by −3 as done deliberately by us in polynomial 9, CXGZ loses its advantages.

Tables 14 and 15 depict the backward errors for the double shift code. Some interesting polynomials, when comparing AMVW with LAPACK, are 12, 15, 16, and

[3]This triggers initial zero shifts, and as a consequence the smallest roots are computed first up to high accuracy.

TABLE 10
Unbalanced single shift version: backward error measure (8.2).

No.	AMVW	ZHSEQR	ZGEEV	BEGG
1	5.1196 e−15	4.1115 e−16	5.8997 e−16	1.5451 e−15
2	3.9604 e−15	1.3259 e−15	8.2223 e−15	3.4513 e−15
3	1.0444 e−14	4.4686 e−15	1.0897 e−14	3.3521 e−01
4	2.5540 e−15	1.4871 e−15	4.7586 e−15	1.4956 e−14
5	8.2139 e−16	1.0871 e−15	1.3384 e−15	2.1723 e−15
6	2.2860 e−15	2.5367 e−15	1.0886 e−15	1.2435 e−14
7	2.2331 e−15	1.2368 e−15	1.2356 e−15	2.9261 e−14
8	2.7015 e−15	4.4440 e−16	4.4440 e−15	4.3155 e−14
9	2.7186 e−14	4.3771 e−15	1.7130 e−15	5.4169 e−01
10	1.5407 e−15	2.5169 e−15	2.8764 e−15	1.7898 e−14
11	2.0003 e−15	2.8510 e−15	3.5121 e−15	2.6659 e−14
12	4.7822 e−15	9.4479 e−13	4.7233 e−15	4.7723 e−06
13	4.9819 e−15	4.3757 e−15	2.7359 e−15	2.2766 e−14
14	1.8597 e−15	3.8582 e−15	6.5575 e−15	5.5561 e−14
15	1.3389 e−15	2.1200 e−15	2.7315 e−12	7.5294 e−01
16	3.3954 e−15	1.2577 e−15	1.7363 e−10	4.8497 e−01
17	7.0711 e−17	2.3551 e−16	3.1702 e−22	8.6299 e−17
18	1.9626 e−17	4.7103 e−16	1.2551 e−30	1.5701 e−16
19	2.8284 e−16	7.9289 e−01	5.2684 e−24	1.7032 e−01
20	9.9516 e−17	3.5184 e+13	9.9516 e−17	7.0711 e−01
21	6.8952 e−16	8.8275 e−16	1.2931 e−18	7.6787 e−15
22	4.2578 e−16	1.2713 e−15	4.1379 e−17	1.0589 e−13
23	2.5722 e−16	3.8583 e−16	1.4147 e−15	5.1444 e−16
24	1.3171 e−16	2.6342 e−16	5.2684 e−16	6.8489 e−15
25	1.5701 e−26	8.0922 e−16	5.3948 e−16	7.0711 e−11
26	7.6615 e−15	8.8366 e−15	1.0720 e−14	2.0853 e−13
27	2.1675 e−15	4.5595 e−15	1.7593 e−15	1.7038 e−14
28	3.1876 e−12	5.0699 e−14	3.2455 e−15	7.3074 e−03
29	4.9669 e−15	5.1055 e−15	3.7699 e−15	2.8623 e−14
30	9.5901 e−15	8.2199 e−15	1.1074 e−14	1.8039 e−13
31	9.9974 e−15	1.0205 e−14	1.5443 e−14	4.3526 e−13
32	2.1807 e−13	1.3192 e−13	1.7909 e−13	3.7859 e−11
33	8.0583 e−13	4.4225 e−13	2.3127 e−13	1.6207 e−10
34	2.9405 e−15	4.8468 e−15	2.5078 e−15	1.9583 e−14
35	4.9934 e−15	6.1108 e−15	8.8445 e−15	3.9206 e−14
36	6.4280 e−15	1.1107 e−14	9.6821 e−15	9.2465 e−14
37	7.1336 e−14	7.0023 e−14	2.9302 e−14	3.3525 e−12
38	8.0993 e−14	2.5129 e−13	7.2148 e−14	1.2479 e−11
39	3.0150 e−15	8.8480 e−15	5.3923 e−15	7.8171 e−14
40	4.8915 e−15	1.0332 e−14	1.0953 e−14	2.4283 e−13
41	8.9851 e−15	1.6620 e−14	2.8066 e−14	5.7554 e−13
42	2.5584 e−13	1.2546 e−13	3.1821 e−13	7.4880 e−11
43	3.4998 e−13	5.1111 e−13	3.6126 e−13	4.1188 e−10
44	2.6053 e−15	4.2770 e−14	1.4583 e−14	1.3910 e−13
45	5.8822 e−15	4.2237 e−14	1.7358 e−14	2.3743 e−13
46	8.0065 e−15	4.8979 e−14	1.7963 e−14	5.7847 e−13
47	4.2851 e−13	2.9020 e−13	4.7464 e−13	7.4323 e−11
48	1.9136 e−12	1.2078 e−12	9.1693 e−13	4.1296 e−10

18–20; otherwise we are in each other's proximity. AMVW appears robust for the balancing; we gain accuracy in cases 12, 19, and 20 and arrive at the same level as LAPACK, but we lose accuracy for polynomials 3, 9, 15, and 16. CXGZ, on the other hand, seems to benefit often from the balancing, for example, of cases 1, 2, and 3.

TABLE 11
Balanced single shift version: backward error measure (8.2).

No.	AMVW	ZHSEQR	ZGEEV	BEGG
3	3.6523 e−12	9.3948 e−12	3.6005 e−15	4.0993 e−14
8	1.3945 e−15	4.3632 e−15	3.5552 e−15	8.0150 e−02
9	3.4155 e−11	1.8478 e−11	5.3243 e−15	3.3980 e−14
12	2.9828 e−14	7.8137 e−13	8.0198 e−15	5.1514 e−14
15	3.2535 e−12	3.5266 e−06	7.7020 e−11	1.8265 e+06
16	4.6922 e−04	9.1242 e−05	3.3598 e−09	4.4007 e+01
19	4.1352 e−13	1.4142 e−16	5.6569 e−16	9.3791 e−13
20	3.6032 e−11	1.9903 e−16	2.9855 e−16	2.9680 e−08
28	2.1364 e−14	1.3348 e−14	4.7639 e−15	4.6575 e−14

TABLE 12
Unbalanced double shift version: relative forward errors.

No.	AMVW	DHSEQR	DGEEV	CGXZ
1	1.6210 e−10	4.9825 e−11	2.3028 e−10	1.5539 e−09
2	1.3451 e−05	2.0112 e−06	2.8902 e−06	4.0939 e−02
3	1.1105 e−01	4.7901 e−03	1.9342 e−03	1.1366 e+00
4	1.0592 e−11	9.7908 e−13	9.6026 e−13	1.3891 e−12
5	2.9139 e−06	7.3219 e−07	4.6375 e−09	5.2079 e−07
6	3.1037 e−01	4.7761 e−01	1.0670 e−04	2.7667 e−01
7	2.1373 e+00	2.4780 e+00	2.7437 e−01	1.7181 e+00
8	1.3517 e−02	1.5862 e+00	2.9221 e−13	2.3874 e−12
9	6.9255 e−02	3.6109 e−02	4.7155 e−02	3.6172 e−02
10	9.1321 e−11	3.5446 e−11	2.7766 e−12	8.4860 e−12
11	2.1897 e−15	1.7772 e−15	1.7902 e−15	1.5029 e−15
12	3.8450 e+09	1.5396 e−07	6.5915 e−08	9.9998 e−01
13	1.6734 e−06	5.8771 e−06	1.4481 e−06	1.2551 e−07
14	1.9244 e−01	2.3890 e−01	2.0926 e−01	1.8676 e−01
15	3.5791 e−13	2.8406 e−13	1.6215 e−07	6.5713 e−09
16	5.0829 e−02	6.0853 e−02	3.4274 e−01	3.1499 e−02
17	1.5574 e−01	1.5467 e−01	1.6544 e−16	4.9644 e−01
18	2.3283 e+05	1.1886 e+07	0.0000 e+00	1.0000 e+00
19	3.4217 e−02	0.0000 e+00	2.2204 e−16	2.9289 e−01
20	1.4074 e+14	1.1102 e−16	2.5000 e−16	8.9582 e+13
21	8.2155 e+06	8.3282 e+05	3.3881 e−15	2.2859 e+04
22	1.3689 e+17	1.2372 e+16	2.1125 e−13	3.9170 e+16
23	2.2204 e−16	2.7105 e−16	5.4570 e−16	1.1018 e−13
24	6.6310 e−14	1.3136 e−11	4.6322 e−15	4.9989 e−08
25	3.8147 e−16	1.8274 e−09	2.4169 e−14	5.0000 e−01
26	2.0278 e−08	6.9016 e−08	2.0442 e−08	2.0860 e−08

TABLE 13
Balanced double shift version: relative forward errors.

No.	AMVW	DHSEQR	DGEEV	CGXZ
2	2.5408 e−07	6.4221 e−07	1.4197 e−06	3.7187 e−08
3	4.3991 e−03	1.6125 e−02	1.3044 e−02	9.9774 e−05
5	1.1167 e−09	9.9303 e−10	7.7720 e−10	1.9518 e−10
6	9.5742 e−06	4.7094 e−05	4.6864 e−07	3.8280 e−07
7	9.1226 e−02	7.8035 e−02	3.6341 e−02	3.7364 e−03
12	1.2993 e−07	6.5912 e−08	6.5913 e−08	6.5913 e−08
15	2.8718 e−08	1.4421 e+00	1.3549 e−06	3.0468 e−11
16	9.3849 e+00	1.0386 e+00	3.0570 e−01	1.0000 e+00
17	1.4724 e−14	2.8245 e−12	2.8245 e−12	4.4409 e−16
18	2.3219 e−07	1.9722 e−16	1.9722 e−16	5.0000 e−06
19	5.9605 e−16	2.2204 e−16	2.2204 e−16	6.6613 e−16
20	1.1102 e−16	0.0000 e+00	2.2204 e−16	0.0000 e+00
21	4.6878 e−08	1.0450 e−04	4.3944 e−15	2.6235 e−11

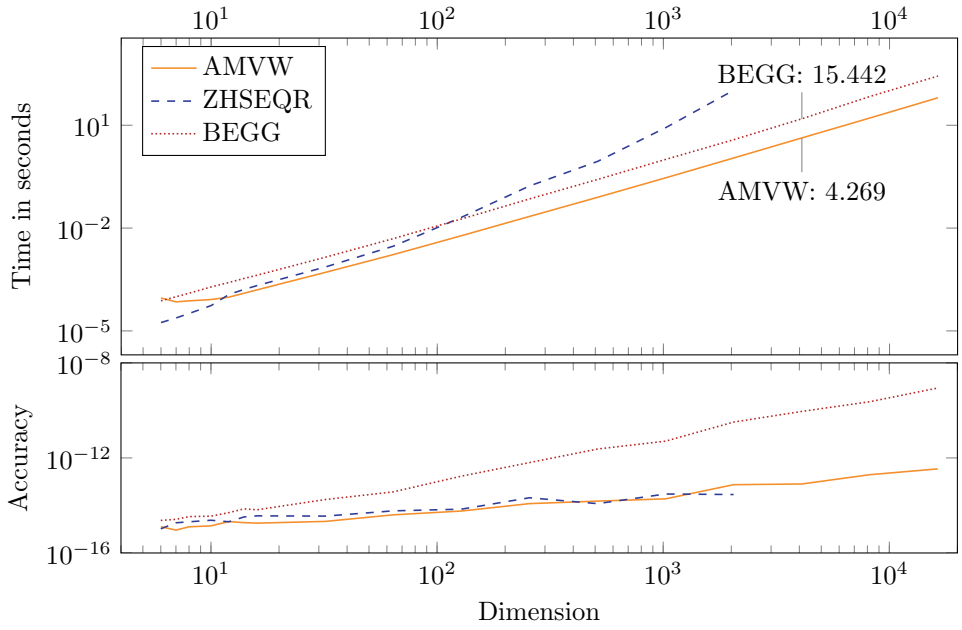


FIG. 1. *Runtime and accuracy for the single shift code for random coefficients.*

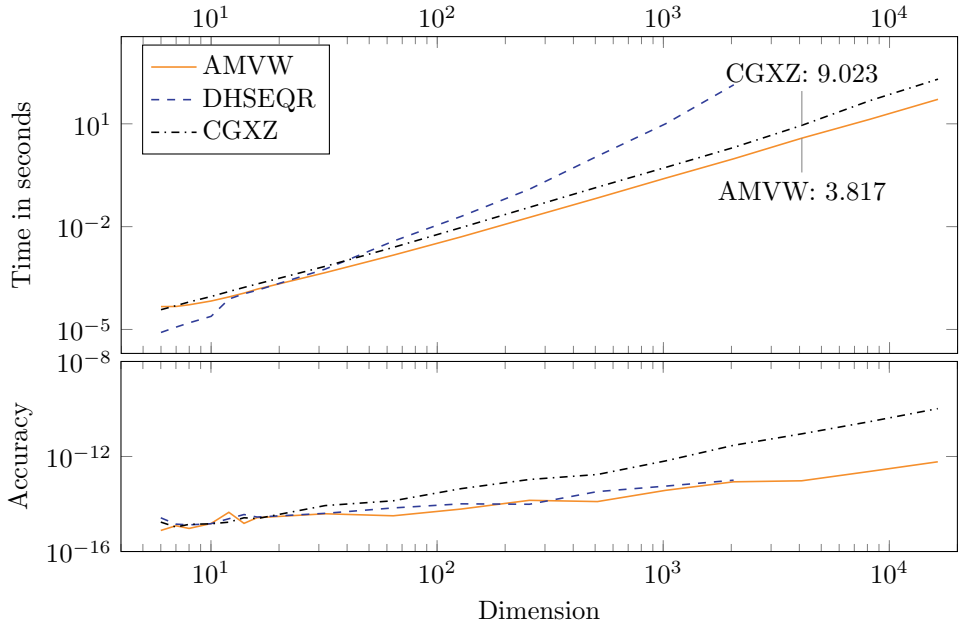


FIG. 2. *Runtime and accuracy for the double shift code for random coefficients.*

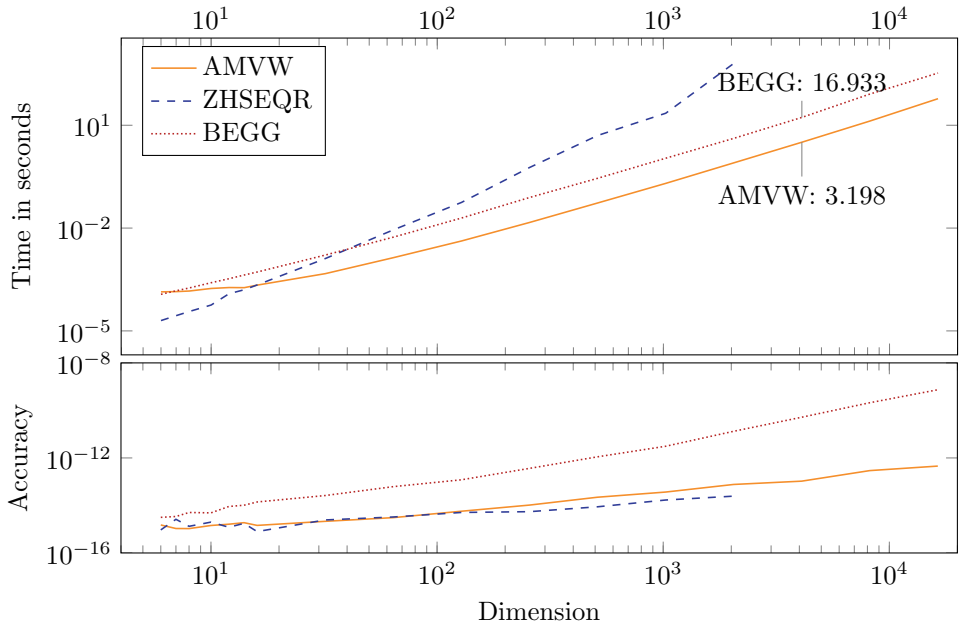


FIG. 3. *Runtime and accuracy for the single shift code for roots of unity.*

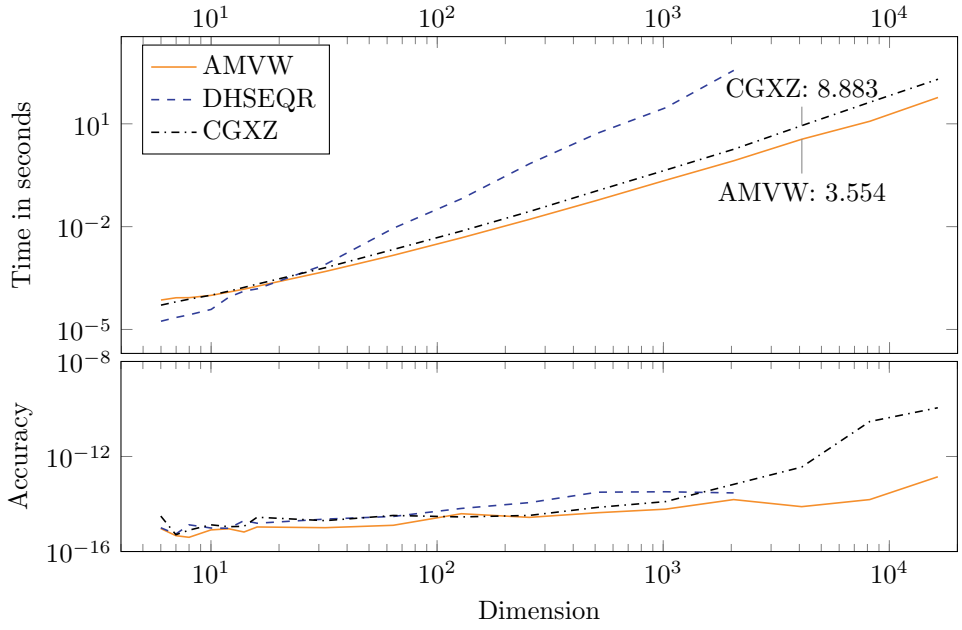


FIG. 4. *Runtime and accuracy for the double shift code for roots of unity.*

TABLE 14
Unbalanced double shift version: backward error measure (8.2).

No.	AMVW	DHSEQR	DGEEV	CGXZ
1	1.2554 e−15	3.5731 e−15	1.4486 e−15	1.5982 e−11
2	4.6694 e−15	2.4362 e−15	1.5226 e−15	1.8855 e−06
3	2.4299 e−14	4.9498 e−15	4.8598 e−15	1.6969 e−01
4	2.0819 e−15	4.7586 e−15	2.0394 e−15	2.6427 e−14
5	3.2376 e−16	4.1775 e−16	5.8485 e−16	4.5953 e−15
6	1.2676 e−15	1.2676 e−15	5.4327 e−16	1.6540 e−14
7	1.3531 e−15	1.2344 e−15	1.4243 e−15	1.8041 e−15
8	2.0200 e−15	4.1208 e−15	3.8784 e−15	2.3755 e−14
9	1.2532 e−15	3.0366 e−15	4.9163 e−15	1.7128 e−04
10	2.8916 e−15	3.2139 e−15	1.2145 e−15	2.1735 e−14
11	2.5681 e−15	4.1186 e−15	3.0526 e−15	1.9382 e−15
12	1.4556 e−06	4.7803 e−12	8.1089 e−15	9.9727 e−01
13	3.3535 e−15	2.0554 e−15	2.7044 e−16	1.0223 e−14
14	9.9463 e−15	5.3784 e−15	5.1942 e−15	2.7746 e−13
15	1.1921 e−15	1.8423 e−15	1.5479 e−10	5.5105 e−10
16	6.5917 e−15	2.6367 e−15	1.4353 e−10	2.3723 e−05
17	7.8505 e−17	1.8694 e−16	1.1698 e−24	2.3551 e−16
18	4.8682 e−18	9.9894 e−17	0.0000 e+00	3.1402 e−16
19	4.9218 e−02	0.0000 e+00	1.4142 e−16	3.5355 e−01
20	7.0711 e−01	9.9516 e−17	2.9855 e−16	7.1278 e−01
21	9.6550 e−17	8.8275 e−16	1.3793 e−17	1.1988 e−05
22	9.9309 e−16	8.0260 e−16	4.1379 e−17	9.8621 e−01
23	2.5722 e−16	1.2861 e−16	6.4305 e−16	3.8583 e−16
24	6.7435 e−14	2.6342 e−16	1.0537 e−15	3.5347 e−15
25	2.6974 e−16	1.3487 e−16	1.7036 e−24	3.5355 e−11
26	4.7127 e−15	1.0052 e−14	6.5066 e−15	1.9228 e−14
27	1.0238 e−15	1.9382 e−15	8.6142 e−15	6.6957 e−14
28	4.4046 e−14	4.3488 e−14	5.4360 e−15	6.8382 e−01
29	2.5594 e−15	2.8052 e−15	5.3901 e−15	5.4346 e−15
30	3.7532 e−15	1.2390 e−14	1.7220 e−14	2.9183 e−14
31	7.7228 e−15	1.2892 e−14	1.5818 e−14	1.9156 e−14
32	6.4705 e−14	9.7108 e−14	1.2664 e−13	6.1607 e−11
33	1.3277 e−13	2.6016 e−13	4.8189 e−13	3.9090 e−09
34	3.8754 e−15	3.7173 e−15	5.6945 e−15	2.5309 e−15
35	9.1937 e−15	5.3933 e−15	5.9242 e−15	3.8843 e−15
36	7.0017 e−15	1.1381 e−14	5.1544 e−15	7.0245 e−15
37	6.2601 e−14	7.5814 e−14	2.2697 e−14	3.0492 e−12
38	5.1983 e−14	2.5335 e−13	4.9573 e−14	2.8747 e+07
39	2.7223 e−15	1.1619 e−14	7.4048 e−15	2.8782 e−15
40	9.2429 e−15	1.0906 e−14	1.1375 e−14	1.1950 e−14
41	9.9618 e−15	5.3342 e−14	3.9347 e−14	1.1091 e−14
42	6.1524 e−14	5.2484 e−13	5.6221 e−13	6.0132 e−13
43	1.6310 e−13	2.0665 e−12	2.1802 e−12	6.0115 e−12
44	6.7513 e−15	4.6688 e−14	1.7129 e−14	9.3708 e−14
45	5.4463 e−15	3.8494 e−14	2.0670 e−14	1.1650 e−13
46	1.3952 e−14	4.2139 e−14	2.7587 e−14	1.2530 e−13
47	9.2145 e−14	2.4008 e−13	8.0347 e−13	1.2574 e−12
48	1.3591 e−13	2.2192 e−12	1.0772 e−12	5.2213 e−12

TABLE 15
Balanced double shift version: backward error measure (8.2).

No.	AMVW	DHSEQR	DGEEV	CGXZ
1	1.7383 e−15	1.2337 e−14	1.5451 e−15	2.4046 e−14
2	1.2141 e−13	8.7755 e−14	4.2634 e−15	2.8220 e−14
3	4.1017 e−12	9.4995 e−12	5.5798 e−15	6.9387 e−14
8	1.0908 e−15	3.3128 e−15	2.8280 e−15	1.8422 e−14
9	1.4707 e−11	1.8488 e−11	2.4582 e−15	4.2897 e−14
12	2.3391 e−15	7.7783 e−13	4.1584 e−16	1.0812 e−14
15	4.6325 e−11	2.2070 e−03	3.4985 e−10	4.1042 e−12
16	7.6627 e−06	4.0016 e−07	1.2261 e−09	2.4671 e−04
18	4.7103 e−16	1.3945 e−31	1.3945 e−31	7.0711 e−21
19	8.4853 e−16	2.8284 e−16	4.2426 e−16	8.4853 e−16
20	9.9516 e−17	0.0000 e+00	2.9855 e−16	0.0000 e+00
21	5.5585 e−15	9.8398 e−13	2.7586 e−17	2.7586 e−17
22	4.3034 e−15	4.3103 e−19	2.7586 e−17	3.1413 e−11
28	3.9028 e−15	1.7981 e−14	2.9271 e−15	1.1011 e−14

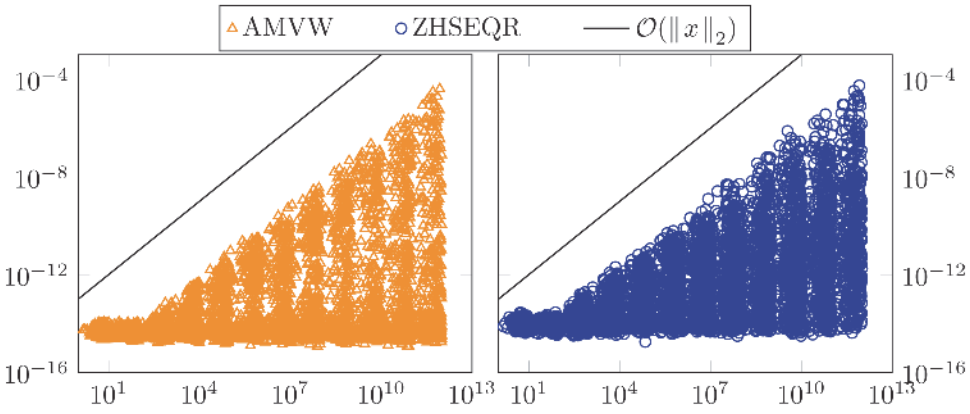


FIG. 5. *Relative backward error of the coefficients of the polynomial over* $\|x\|_2$.

8.4. Tightness of the backward error bound. This experiment was suggested to us by Froilán Dopico to test the backward error bound. We use polynomials of degree 20 as in test set (iv) from Jenkins and Traub [23]. The polynomials have random coefficients $a_j = m_j \rho_j \cdot 10^{e_j}$, m_j uniformly distributed in $(-1, 1)$, e_j uniformly distributed in $(-\mu, \mu)$, with $\mu = 0, 1, 2, \ldots, 12$, and ρ_j uniformly distributed on the unit circle. The measure of backward error used is (8.2). For each ρ 500 polynomials are sampled. Each polynomial is represented by a geometrical symbol in Figure 5. The x-value is $\|x\|_2$, and the y-value represents the backward error. To increase the readibility Figure 5 is split into two parts, one for our algorithm and one for ZHSEQR. The results depicted are only for the CSS code; the results for the double shift code look similar and are omitted. The algorithms BEGG and CGXZ exhibit similar behavior.

The backward analysis shows that the normwise backward error on the companion matrix depends on $\|x\|_2$. The normwise backward error on the coefficients of the polynomial has an additional factor of $\max|a_i|$, which is bounded by $\|x\|_2$. The plot shows the relative normwise backward error on the coefficients, where relative means

that we divided the error by $\|x\|_2$. We observe that the maximum achieved errors have the same slope as the curve representing $\|x\|_2$, implying that our error bound is tight. Moreover, exactly the same behavior is observed for ZHSEQR.

9. Conclusions. We have presented a fast and backward stable algorithm to compute the roots of a polynomial presented in monomial basis form. The algorithm is Francis's implicitly shifted QR algorithm applied to a special representation of the companion matrix consisting of $3n-1$ rotators. Thus the memory requirement is $O(n)$. The flop count is $O(n)$ per iteration or $O(n^2)$ overall. Extensive tests indicate that the new algorithm is about as accurate as the (slow) Francis algorithm applied directly to the companion matrix. It is faster than other fast algorithms that have been devised for this problem, and its accuracy is comparable or better.

Acknowledgments. The authors thank Vanni Noferini (University of Manchester) for providing polynomials 15 and 16, which proved to be quite challenging for some of the methods. We also thank Froilán Dopico (Universidad Carlos III de Madrid) for suggesting the experiment in subsection 8.4. We also greatly appreciate Piers Lawrence's comments on an earlier version of the manuscript and the referees' detailed remarks.

REFERENCES

[1] T. Aktosun, D. Gintides, and V. G. Papanicolaou, *The uniqueness in the inverse problem for transmission eigenvalues for the spherically symmetric variable-speed wave equation*, Inverse Problems, 27 (2011), 115004.

[2] J. L. Aurentz, R. Vandebril, and D. S. Watkins, *Fast computation of the zeros of a polynomial via factorization of the companion matrix*, SIAM J. Sci. Comput., 35 (2013), pp. A255–A269.

[3] J. L. Aurentz, R. Vandebril, and D. S. Watkins, *Fast computation of eigenvalues of companion, comrade, and related matrices*, BIT, 54 (2014), pp. 7–30.

[4] R. Bevilacqua, G. M. Del Corso, and L. Gemignani, *A CMV-Based Eigensolver for Companion Matrices*, preprint, arXiv:1406.2820 [math.NA], 2014.

[5] D. A. Bini, *Numerical computation of polynomial zeros by means of Aberth's algorithm*, Numer. Algorithms, 13 (1996), pp. 179–200.

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

[7] D. A. Bini, F. Daddi, and L. Gemignani, *On the shifted QR iteration applied to companion matrices*, Electron. Trans. Numer. Anal., 18 (2004), pp. 137–152.

[8] 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.

[9] D. A. Bini and G. Fiorentino, *Design, analysis, and implementation of a multiprecision polynomial rootfinder*, Numer. Algorithms, 23 (2000), pp. 127–173.

[10] D. A. Bini, G. Fiorentino, L. Gemignani, and B. Meini, *Effective fast algorithms for polynomial spectral factorization*, Numer. Algorithms, 34 (2003), pp. 217–227.

[11] P. Boito, Y. Eidelman, and L. Gemignani, *Implicit QR for rank-structured matrix pencils*, BIT, 54 (2013), pp. 85–111.

[12] P. Boito, Y. Eidelman, L. Gemignani, and I. Gohberg, *Implicit QR with compression*, Indag. Math., 23 (2012), pp. 733–761.

[13] A. Böttcher and M. Halwass, *Wiener–Hopf and spectral factorization of real polynomials by Newton's method*, Linear Algebra Appl., 438 (2013), pp. 4760–4805.

[14] S. Chandrasekaran, M. Gu, J. Xia, and J. Zhu, *A fast QR algorithm for companion matrices, in Recent Advances in Matrix and Operator Theory*, Oper. Theory Adv. Appl., 179, Springer, New York, 2007, pp. 111–143.

[15] F. De Terán and F. M. Dopico, *Low rank perturbation of regular matrix polynomials*, Linear Algebra Appl., 430 (2009), pp. 579–586.

[16] S. Delvaux, K. Frederix, and M. Van Barel, *An algorithm for computing the eigenvalues of block companion matrices*, Numer. Algorithms, 62 (2013).
[17] A. Edelman and H. Murakami, *Polynomial roots from companion matrix eigenvalues*, Math. Comp., 64 (1995), pp. 763–776.
[18] Y. Eidelman, L. Gemignani, and I. C. Gohberg, *Efficient eigenvalue computation for quasiseparable Hermitian matrices under low rank perturbation*, Numer. Algorithms, 47 (2008), pp. 253–273.
[19] Y. Eidelman, I. Gohberg, and I. Haimovici, *Separable Type Representations of Matrices and Fast Algorithms—Volume* 2: *Eigenvalue Method*, Oper. Theory Adv. Appl., 235, Springer, New York, 2013.
[20] J. G. F. Francis, *The QR. transformation. A unitary analogue to the LR transformation—Part* 1, Computer J., 4 (1961), pp. 265–271.
[21] J. G. F. Francis, *The QR Transformation—Part* 2, Computer J., 4 (1962), pp. 332–345.
[22] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.
[23] M. A. Jenkins and J. F. Traub, *Principles for testing polynomial zerofinding programs*, ACM Trans. Math. Software, 1 (1975), pp. 26–34.
[24] T. Mach and R. Vandebril, *On deflations in extended QR algorithms*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 559–579.
[25] C. B. Moler, *Cleve's corner: Roots—of polynomials, that is*, Mathworks Newsletter, 5 (1991), pp. 8–9.
[26] *MPFUN Multiprecision Software*. http://www.netlib.org/mpfun (2005).
[27] K.-C. Toh and L. N. Trefethen, *Pseudozeros of polynomials and pseudospectra of companion matrices*, Numer. Math., 68 (1994), pp. 403–425.
[28] 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.
[29] P. Van Dooren and P. Dewilde, *The eigenstructure of an arbitrary polynomial matrix: Computational aspects*, Linear Algebra Appl., 50 (1983), pp. 545–579.
[30] R. Vandebril and G. M. Del Corso, *An implicit multishift QR-algorithm for Hermitian plus low rank matrices*, SIAM J. Sci. Comput., 32 (2010), pp. 2190–2212.
[31] D. S. Watkins, *The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods*, SIAM, Philadelphia, 2007.
[32] D. S. Watkins, *Fundamentals of Matrix Computations*, 3rd ed., Pure Appl. Math., John Wiley, New York, 2010.
[33] D. S. Watkins, *Francis's algorithm*, Amer. Math. Monthly, 118 (2011), pp. 387–403.
[34] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Numer. Math. Sci. Comput., Oxford University Press, New York, 1988.
[35] P. Zhlobich, *Differential qd algorithm with shifts for rank-structured matrices*, SIAM J. Matrix Anal. Appl., 33 (2012).