An iterated eigenvalue algorithm for approximating roots of univariate polynomials

Steven Fortune
Bell Laboratories
sjf@research.bell-labs.com

Abstract

We present an iterative algorithm that approximates all roots of a univariate polynomial. The iteration uses floating-point eigenvalue computation of a generalized companion matrix. With some assumptions, we show that the algorithm approximates the roots within about $\log_{\rho/\epsilon} \chi(P)$ iterations, where ϵ is the relative error of floating-point arithmetic, ρ is the relative separation of the roots, and $\chi(P)$ is the condition number of the polynomial. Each iteration requires an $n \times n$ floating-point eigenvalue computation, n the polynomial degree, and evaluation of the polynomial to floating-point accuracy at up to n points.

We describe a careful implementation of the algorithm, including many techniques that contribute to the practical efficiency of the algorithm. On some hard examples of ill-conditioned polynomials, e.g. high-degree Wilkinson polynomials, the implementation is an order of magnitude faster than the Bini-Fiorentino implementation `mpsolve`.

1 Introduction

The algorithmic problem of approximating the roots of a univariate polynomial, presented by its coefficients, is classic in numerical analysis and symbolic algebra [20, 19]. Despite the extensive research history, there appears to be no algorithm for root approximation that is completely satisfactory both in theory and practice. Existing algorithms can be crudely and unexhaustively classified either as iterative algorithms or as theoretically complete algorithms. The theoretically complete algorithms include Weyl's algorithm and the Schönhage splitting-circle algorithm. The descendants of the splitting-circle algorithm (see [19] for references) have astonishingly fast asymptotic running times, at least in the real RAM model. The iterative algorithms include Newton's method, the Durand-Kerner method, and Aberth's method (see [3, 19] for references). The iterative algorithms usually have local convergence proofs, but are not known to converge globally and hence have no running time analysis.

In practice the iterative algorithms can perform very well, despite their lack of theoretical justification. Recently Bini and Fiorentino [3, 4] have developed `mpsolve`, a sophisticated and robust implementation of Aberth's method. They performed an extensive comparison of their implementation with the solvers in Maple™ and Mathematica™ and with Gourdon's implementation [15]

of the Schönhage splitting-circle algorithm. Their implementation is much the fastest, beating the other implementations by factors ranging from roughly 10 to 5000, depending upon instance.

1.1 Iteration based on floating-point eigenvalue computation

We present an iterative root-finding algorithm `eigensolve`. The algorithm bases its iteration on floating-point eigenvalue computation. (A similar algorithm was previously proposed by Malek and Vaillancourt [18].) At a high level, the algorithm is very simple (see figure 1). Let P be a polynomial and S the current estimate of the roots of P, represented as floating-point numbers. The iteration is to compute the eigenvalues, in hardware floating-point, of a generalized companion matrix $C(P, S)$; these computed eigenvalues then form the next estimate of the roots of P. The matrix $C(P, S)$ is an eigenvalue formulation of Lagrange interpolation (see section 2). This matrix is familiar; a standard way to find inclusion regions that are guaranteed to contain the roots of P is to apply Gershgorin's theorem to this matrix [3, 21]. Computation of $C(P, S)$ needs the value of P at the points of S to floating-point accuracy, requiring extended-precision arithmetic if evaluation of P is ill-conditioned.

It is not obvious that this iteration converges—indeed it might seem highly implausible, since the heart of the iteration is a floating-point eigenvalue computation on a matrix whose (eigenvalue) condition number may be vastly larger than $1/\epsilon$, ϵ the relative accuracy of hardware floating-point arithmetic. Figure 2 depicts the behavior of `eigensolve` on the Wilkinson polynomial of degree 200. (The condition number of this polynomial exceeds 10^{420}: its L_∞ norm is about 10^{380}, and a perturbation of the coefficient of x^{170} of size 10^{-40} creates complex roots.)

Why does the iteration converge? For a vector $S = (s_1, \ldots, s_n)$, $s_i \in \mathbb{C}$, let $Q_S(x) = \prod(x - s_i)$, and let $S_1,\ S_2,\ \ldots$, be the successive iterate vectors produced by `eigensolve`. For a polynomial P such as a Wilkinson polynomial, it can be observed experimentally that for each root λ of P,

$$|Q_{S_{t+1}}(\lambda)| \ \lesssim\ \epsilon \max\left(|Q_{S_t}(\lambda)|, |\lambda P'(\lambda)|\right) \tag{1}$$

(see figure 3). Here ϵ is the relative error of floating-point arithmetic ($\epsilon \approx 10^{-16}$ for IEEE double-precision floating-point). The condition $|Q_{S_k}(\lambda)| \approx |\epsilon \lambda P'(\lambda)|$, for all roots λ, is sufficient to guarantee that the iterate S_k approximates all the roots of P to relative error about ϵ (see section 3.2). It follows that the number of required iterations is bounded by about $\log_{(1/\epsilon)} \chi(P)$, where $\chi(P)$ is the condition number of the polynomial.

In section 3 below, we give a proof of an upper bound similar to (1), under some assumptions. The upper bound has an additional factor that depends upon the degree n and a measure of the relative root separation of P. The upper bound also depends upon a parameter α, which is a measure of the genericity of the iterate S_i. S_i is the vector of eigenvalues computed in floating-point arithmetic, so S_i is the vector of roots of $Q_{S_i} = \mathrm{char}(M + \Delta M) = P + \Delta P$, where $\mathrm{char}\, M = P$. The perturbation matrix ΔM, and hence the perturbation polynomial ΔP, are consequences of floating-point rounding during the eigenvalue computation. The genericity of S_i is determined from P and ΔP; roughly, α is large if ΔP is "exceptional" and $\alpha \approx 1$ if ΔP is "random" or "unexceptional". All experimental observations are that $\alpha \approx 1$. Section 3.4 below argues why it is plausible that α should be small, though it is far from clear how to prove a rigorous upper bound on α.

```
// P is a monic polynomial of degree n
// S = (s_1, ..., s_n) is a vector of length n of floating-point complex numbers
// C(P) is the (usual) companion matrix
// C(P,S) is a generalized companion matrix (section 2.2)
// fl is conversion to floating-point
// eig is floating-point eigenvalue calculation (section 2.3)
procedure eigensolve (P)
    S := eig(fl(C(P)))
    while ¬converged(S)
            S := eig(fl(C(P,S)))
    return S
```

Figure 1: The `eigensolve` algorithm.

1.2 The implementation

Even though the `eigensolve` iteration is simple, it is not trivial to develop a complete implementation. Section 4 describes an implementation of `eigensolve`. The implementation uses floating-point eigenvalue iteration as described above, appropriately shifted and scaled to match the precision and dynamic range of hardware floating-point arithmetic. In addition, the implementation has techniques to handle repeated or closely spaced roots, to detect convergence, and to approximate roots to arbitrary precision. Several different software arithmetic types are used, in order to provide arithmetic with large dynamic range, extended accuracy, and minimal computing cost. The paper by Bini and Fiorentino [3] is an additional source of information about the implementation of iterative polynomial root-finders.

Section 4 also compares the implementation of `eigensolve` with the Bini-Fiorentino implementation `mpsolve`. For ill-conditioned polynomials of high degree, `eigensolve` is much faster than `mpsolve`, by a factor of 20 or more (see figures 5 and 6). As discussed below, `eigensolve` is not uniformly faster than `mpsolve`. It is an interesting and apparently difficult engineering challenge to develop a practical root-finder that combines the best performance of `eigensolve` and `mpsolve`. The `eigensolve` implementation can be downloaded by following the directions on the eigensolve web page [13].

1.3 Relation to previous work

There is a huge literature on polynomial root-finding. Many references can be found in Pan's survey paper [19] and McNamee's bibliography [20]. The iterated eigenvalue algorithm discussed in this paper was apparently first proposed by Malek and Vaillancourt [18]. Malek and Vaillancourt gave no analysis and only tried their algorithm on low-degree polynomials, at most degree 40. This paper makes several contributions. First, we provide partial complexity analysis, which at least explains why the iteration converges. Second, we document the effectiveness of the algorithm on hard, high-degree polynomials. Third, we describe implementation details that contribute significantly to the

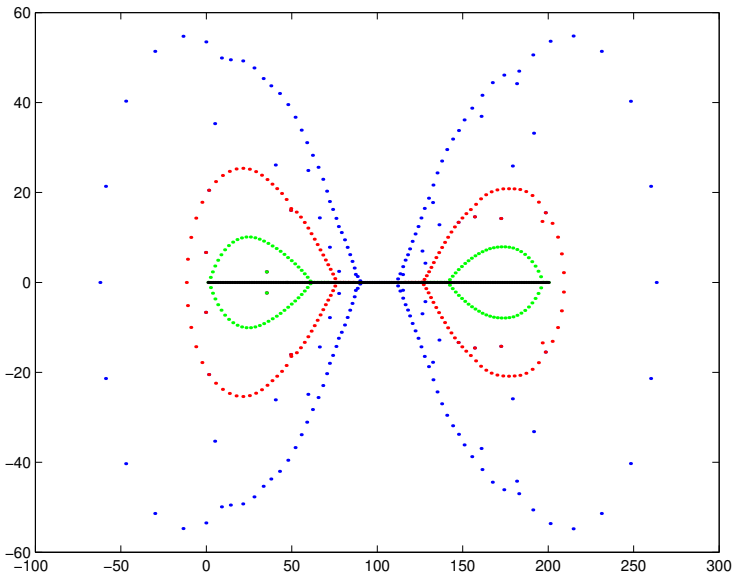

Figure 2: Iterates of the algorithm on the Wilkinson polynomial of degree 200. The roots are the integers 1, ..., 320.

practical efficiency of the algorithm.

This paper combines two previous conference papers, one that appeared in the 2001 International Symposium on Symbolic and Algebraic Computing [11] and another presented at the 2001 DIMACS workshop on real and algebraic geometry [12].

2 Background

Throughout $||\cdot||$ is the infinity norm. If λ is a simple root of polynomial P, then the *condition number of P at λ* is

$$\chi(P,\lambda) = \frac{||P||\max(1,|\lambda|^{n-1})}{|P'(\lambda)|}.$$

Using standard arguments [6] it is possible to show that $\min(|\Delta\lambda|, |\Delta\lambda/\lambda|) \leq \chi(P,\lambda)\ ||\Delta P||/||P||$, where $\Delta\lambda$ is the perturbation in λ resulting from a sufficiently small perturbation ΔP to P. If P has all simple roots, then $\chi(P)$, the *condition number of P*, is the maximum of $\chi(P,\lambda)$, λ varying over the roots of P.

Let P have degree n. If P has simple root λ then the *root separation* of P at λ is $\mathrm{sep}(P,\lambda) = \min_{\lambda'} |\lambda - \lambda'|$ where λ' varies over all roots distinct from λ. If P is simple, the (absolute) *root separation* of P is $\mathrm{sep}(P) = \min_\lambda \mathrm{sep}(P,\lambda)$, where λ varies over the roots of P. If λ is a simple root of P and $\alpha > 0$, s is *α-close* to λ if $|s-\lambda| \leq \alpha \min_{\lambda' \neq \lambda} |\lambda - \lambda'|$, where λ' varies over all roots of P distinct from λ.

Proposition 2.1 *If s is $1/n$-close to root λ of P, then $|P(s)| \leq e\,|s-\lambda|\ |P'(\lambda)|$.*

Proof: Suppose $P = \prod(x-\lambda_i)$ and s is α-close to λ_1. Then for $i \neq 1$, $|s-\lambda_i| = |s-\lambda_1 + \lambda_1-\lambda_i| \leq (1+\alpha)\,|\lambda_1-\lambda_i|$ and

$$|P(s)| \;=\; |(s-\lambda_1)| \prod_{i\neq 1} |s-\lambda_i| \;\leq\; |s-\lambda_1|\,(1+\alpha)^{n-1}|P'(\lambda_1)|.$$

The lemma follows for $\alpha = 1/n$ using $(1+1/n)^{n-1} \leq e$. □

Proposition 2.2 *Suppose λ is the root of P closest to s, and $|s-t| < |s-\lambda|/(n+1)$. Then $|P(t)|/e \leq |P(s)| \leq e\,|P(t)|$.*

Proof: Similar to the proof of proposition 2.1 □

For $S = (s_1, \ldots, s_k)$ a vector of complex numbers, define

$$Q_S(x) = \prod_{i=1}^{k}(x-s_i).$$

Notice that $Q'(s_j) = \prod_{i\neq j}(s_j-s_i)$.

For M a matrix, $\lambda(M)$ denotes the vector of eigenvalues of M, $\det M$ the determinant of M, and $\operatorname{char} M$ the characteristic polynomial of M. For monic $P = x^n + p_{n-1}x^{n-1} + \ldots + p_1 x + p_0$, the *companion matrix* is

$$C(P) = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ & \vdots & & \ddots & \\ -p_0 & -p_1 & -p_2 & \ldots & -p_{n-1} \end{bmatrix}.$$

Standard results are that $\lambda(C(P))$ are the roots of P and that $\lambda(M)$ are the roots of $\operatorname{char} M$.

2.1 Some matrix facts

Throughout this section, $D = \operatorname{diag}(d_1, \ldots, d_n)$ is a diagonal matrix and $R = (r_{ij})$ is a matrix of rank one.

Lemma 2.3 $\det(D+R) = \prod_{i=1}^{n} d_i + \sum_{i=1}^{n} r_{ii} \prod_{j\neq i} d_j.$

Proof: The proof is by induction on n; the case $n = 1$ is immediate. Write $M[i{:}j]$ for the matrix consisting of columns i through j of matrix M, and $M[i]$ for $M[i:i]$. An easy determinant identity is $\det M = \det M_0 + \det M_1$, where M_0 and M_1 are identical to M except in the kth column, and $M_0[k] + M_1[k] = M[k]$. Using this identity,

$$\det(D+R) = \det\Big[D[1] \;\; (D+R)\,[2:n]\Big] + \det\Big[R[1] \;\; (D+R)\,[2:n]\Big].$$

Using the induction hypothesis,

$$\det\Big[D[1]\ \ (D{+}R)\,[2:n]\Big] = d_1\Big(\prod_{i=2}^{n} d_i + \sum_{i=2}^{n} r_{ii} \prod_{j=2,\ j\neq i}^{n} d_j\Big) = \prod_{i=1}^{n} d_i + \sum_{i=2}^{n} r_{ii} \prod_{j=1,\ j\neq i}^{n} d_j.$$

Using the identity on the last column backwards to the second column,

$$\begin{aligned}\det\Big[R[1]\ \ (D{+}R)\,[2:n]\Big] &= \det\Big[R[1]\ \ D[2:n]\Big] + \sum_{j=2}^{n-1} \det\Big[R[1]\ \ (D{+}R)\,[2:j]\ \ R[j{+}1:n]\Big] \\ &= r_{11}\prod_{j=2}^{n} d_j\end{aligned}$$

since any matrix with at least two columns of the rank-one matrix R has zero determinant. □

Corollary 2.4 $(-1)^n \operatorname{char}(D{+}R) = \prod_{i=1}^{n}(x{-}d_i) + \sum_{i=1}^{n} r_{ii} \prod_{j\neq i}(x{-}d_i).$

Let $\det_{ij} M$ be the determinant of the matrix obtained from M by deleting the ith row and the jth column, and similarly for $\operatorname{char}_{ij} M$. The proof of the following lemma is similar to the proof of lemma 2.3.

Lemma 2.5

$$\det_{ij}(D{+}R) = \begin{cases} r_{ji} \displaystyle\prod_{k\neq i,j} d_k & i \neq j \\[2ex] \displaystyle\prod_{k\neq i} d_k + \sum_{l\neq i} r_{ll} \prod_{k\neq i,l} d_k & i = j. \end{cases}$$

Corollary 2.6

$$(-1)^{n-1}\operatorname{char}_{ij}(D{+}R) = \begin{cases} r_{ji} \displaystyle\prod_{k\neq i,j} (x{-}d_k) & i \neq j \\[2ex] \displaystyle\prod_{k\neq i} (x{-}d_k) + \sum_{l\neq i} r_{ll} \prod_{k\neq i,l} (x{-}d_k) & i = j. \end{cases}$$

2.2 Generalized companion matrices

Let $S = (s_1, \ldots, s_n)$ with s_i distinct complex numbers, and let P be a monic polynomial of degree n. The *Lagrange coefficients* $l_1, \ldots, l_n$ for P and S are defined by

$$l_i = \left| \frac{P(s_i)}{Q'_S(s_i)} \right|.$$

The *generalized companion matrix* $C(P,S)$ is the $n \times n$ matrix

$$C(P,S) \;=\; \begin{bmatrix} s_1 & 0 & \dots & 0 \\ 0 & s_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \vdots & s_n \end{bmatrix} - \begin{bmatrix} l_1 & l_2 & \dots & l_n \\ l_1 & l_2 & \dots & l_n \\ \vdots & \vdots & & \vdots \\ l_1 & l_2 & \dots & l_n \end{bmatrix}$$

By Corollary 2.4,

$$(-1)^n \operatorname{char} C(P,S) = \prod (x - s_i) \;+\; \sum_{i=1}^{n} l_i \prod_{j \neq i} (x - s_j).$$

The second term on the righthand side of this equation is the Lagrange interpolating polynomial [27] of degree $n-1$ that has value $P(s_i)$ at s_i. Since this polynomial is unique, and P is monic of degree of n, we have a result apparently first proved by Smith [21].

Corollary 2.7 *If P is monic, then $(-1)^n \operatorname{char}(C(P,S)) = P$; hence the eigenvalues of $C(P,S)$ are the roots of P.*

For complex s and real $r>0$, let $D(s,r) \subset \mathbb{C}$ be the closed disc of radius r centered at s. The following lemma follows easily from Gershgorin's theorem [14] and corollary 2.7 (see also [3, 9, 21]).

Lemma 2.8 *Let P be a monic polynomial of degree n, $S = (s_1, \dots, s_n)$ be distinct complex numbers and $l_1, \dots, l_n$ be the Lagrange coefficients. Then the union of discs $\bigcup_{i=1}^{n} D(s_i, n l_i)$ contains all the roots of P. Furthermore, each connected component made up of k discs contains k roots. In particular, if the discs are pairwise disjoint, each disc contains exactly one root.*

2.3 Floating-point eigenvalue computation

The computation of eigenvalues has been a central topic of numerical analysis for several decades. The QR algorithm[7, 14] is the standard for dense matrices. The conventional analysis is that the QR algorithm is backwards stable: for M an $n \times n$ matrix with floating-point entries, a call to the implementation `eig` of the QR algorithm

$$\Lambda := \texttt{eig}(M)$$

computes a vector of eigenvalues $\Lambda = (\lambda_1, \dots, \lambda_n)$ that are the true eigenvalues of a perturbed matrix $M + \Delta M$:

$$\begin{aligned} \Lambda &= \lambda(M + \Delta M) \\ ||\Delta M|| &\leq t(n)\, \epsilon\, ||M||, \end{aligned}$$

where ϵ is the relative error of floating-point arithmetic. The upper bound $t(n)$ is usually quoted as a slow-growing function of n, say $t(n) \leq 10n$ [1]. Apparently, there is no rigorous convergence proof of the QR algorithm. Assuming that the QR algorithm requires only $O(n)$ iterations, $t(n)$ can be rigorously bounded by $O(n^3)$ [24]. We assume that the QR algorithm is globally convergent and that $t(n)$ is bounded by a polynomial in n.

3 The algorithm and its analysis

For this section we assume that P is monic of degree n, with distinct nonzero roots $\lambda_1, \ldots, \lambda_n$. The `eigensolve` algorithm is given in figure 1. Section 4 gives an extensive discussion of implementation details.

3.1 Generic root estimates

Let $S = \{s_1, \ldots, s_n\}$, $s_i \in \mathbb{C}$. Let $\lambda_{\sigma(i)}$ be the root of P closest to s_i, and let $G_S = Q_S - P$. For real $\alpha > 0$, S is *α-generic* (with respect to P) if the following four conditions hold:

1. If $i \neq j$, then $s_i \neq s_j$.
2. $\max_i |s_i| \leq \alpha \max_i |\lambda_i|$.
3. $|l_i| \leq \alpha |s_i - \lambda_{\sigma(i)}|$, $i = 1, \ldots, n$, where l_i is the Lagrange coefficient as in section 2.2.
4. If s_i is $1/n$-close to $\lambda_{\sigma(i)}$, then $|G_S(\lambda_{\sigma(i)})| \leq \alpha |G_S(s_i)|$, $i = 1, \ldots, n$.

Our interest is in the case that S is the result of a floating-point eigenvalue computation on a matrix whose characteristic polynomial is P. In this case, $||G_S||$ should be small with respect to $||P||$; Section 3.4 below gives nonrigorous plausibility arguments that S should be α-generic for some small α.

Proposition 3.1 *Suppose S is α-generic and s_i is $1/n$-close to λ_k. Then $|Q_S(\lambda_k)| \leq \alpha |P(s_i)|$.*

Proof: Using condition (4), $|Q_S(\lambda_k)| = |G_S(\lambda_k)| < \alpha |G_S(s_i)| = \alpha |P(s_i)|$. □

3.2 Analysis of a single iteration

Let S_0 be the values computed by the initial call `eig(fl`$(C(P))$`)`, and for $t = 1, 2, \ldots$, let S_t be the values computed at the tth call to `eig` within the `while` loop. With

$$S_t = \{s_{t1}, \ldots, s_{tn}\}$$

define

$$Q_t(x) = Q_{S_t}(x) = \prod_{i=1}^{n} (x - s_{ti}).$$

Lemma 3.2 *Let $\lambda_{\max}$ be the root of P of largest magnitude. There is a polynomial q in n and α so that if S_t is α-generic, for every root λ of P,*

$$|Q_{t+1}(\lambda)| \leq \epsilon\, q(n, \alpha) \max\left(\frac{|\lambda_{\max}|}{\mathrm{sep}(P, \lambda)} |Q_t(\lambda)|,\ |\lambda_{\max} P'(\lambda)| \right),$$

as long as $\epsilon\, q(n, \alpha) < 1$.

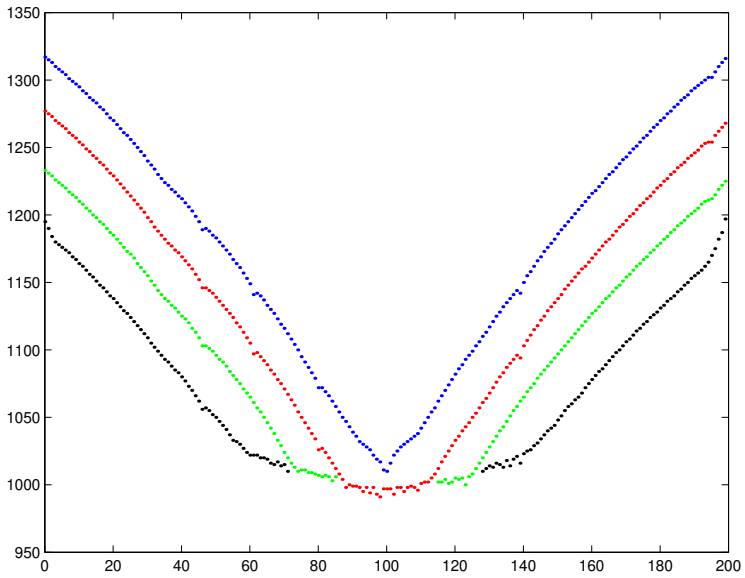

Figure 3: The x-axis are the roots $1, \ldots, 200$ of the Wilkinson polynomial W of degree 200; dots are $\log_2 |Q_i(\lambda)|$, i varying over the iteration, $\lambda = 1, \ldots, 200$. The lower envelope of the dots closely matches $\epsilon|\lambda W'(\lambda)|$.

The lower bound in lemma 3.2 can be motivated very simply. Let $R = (r_1, \ldots, r_n)$ where r_i is the best floating-point approximation to λ_i. Then for $k = 1, \ldots, n$, $|r_k - \lambda_k| \leq \epsilon|\lambda_k|$ and

$$|Q_R(\lambda_k)| = |(\lambda_k - r_k) \prod_{j \neq k} (\lambda_k - r_j)| \approx |\epsilon \lambda_k \prod_{j \neq k} (\lambda_k - \lambda_j)| = |\epsilon \lambda_k P'(\lambda_k)|.$$

The lower envelope of the dots in figure 3 closely matches $\epsilon \lambda P'(\lambda)$.

Proof of Lemma 3.2: Consider a root λ_k of P. For simplicity drop the double subscripts and write $S_t = (s_1, \ldots, s_n)$. Let $M = C(P, S_t)$. Recall that $M_{ij} = -l_j$ if $i \neq j$, and $M_{jj} = s_j - l_j$, where $l_j = P(s_j)/Q_t'(s_j)$ is the Lagrange coefficient.

Using $|l_j| \leq \alpha\, |s_j - \lambda_{\sigma(j)}|$, $\lambda_{\sigma(j)}$ the root closest to s_j, it is easy to get

$$||M|| \leq 2n\alpha^2\ |\lambda_{\max}|. \tag{2}$$

where $||\cdot||$ (here and elsewhere) is the ∞-norm. By the assumptions on `eig`, there is a perturbation matrix Δ so that

$$\begin{aligned} |\,\Delta_{ij}\,| &\leq \epsilon\, n^{O(1)}\, ||M|| \\ S_{t+1} &= \lambda(M + \Delta) \\ Q_{t+1}(x) &= \mathrm{char}(M + \Delta). \end{aligned} \tag{3}$$

By expanding the determinant of $\mathrm{char}(M + \Delta)$ and letting $O(\Delta^2)$ denote the sum of all terms with at least two factors from Δ, we get

$$Q_{t+1}(x) \;=\; \mathrm{char}\, M \;+\; \sum_{i,j} (-1)^{i+j} \Delta_{ij}\, \mathrm{char}_{ij}\, M \;+\; O(\Delta^2);$$

of course $\operatorname{char} M = P$. We henceforth ignore the $O(\Delta^2)$ terms; this can be justified rigorously by a somewhat tedious argument similar to the argument below, using $\epsilon\, q(n,\alpha) < 1$. Substituting λ_k for x, we get

$$|Q_{t+1}(\lambda_k)| \;\le\; \sum_{i,j} |\Delta_{ij}\ (\operatorname{char}_{ij} M)(\lambda_k)|. \tag{4}$$

We first consider the case $i \neq j$. By corollary 2.6 and using $M_{ij} = -l_j$,

$$\begin{aligned} \operatorname{char}_{ij} M &= (-1)^n\, l_j \prod_{p\neq i,j} (x - s_p) \\ \Big|\, (\operatorname{char}_{ij} M)(\lambda_k) \,\Big| &= \Big|\, \frac{l_j}{\lambda_k - s_j} \prod_{p\neq i} (\lambda_k - s_p) \,\Big|. \end{aligned} \tag{5}$$

Since S_t is α-generic, $|l_j/(\lambda_k - s_j)| \le \alpha$. There are now two subcases. First, if s_i is $1/n$-close to λ_k, then using propositions 3.1 and 2.1 followed by (2), (3) and (5)

$$\Big|\prod_{p\neq i} (\lambda_k - s_p)\Big| \;=\; \left|\frac{Q_S(\lambda_k)}{\lambda_k - s_i}\right| \;\le\; \alpha \left|\frac{P(S)}{\lambda_k - s_i}\right| \;\le\; \alpha e |P'(\lambda_k)|$$

$$|\Delta_{ij}\ (\operatorname{char}_{ij} M)(\lambda_k)| \;\le\; \epsilon\ \alpha^3\, n^{O(1)}\ |\lambda_{\max}\ P'(\lambda_k)| \tag{6}$$

Second, if s_i is not $1/n$-close to λ_k, then easily $|\lambda_{\max}| \le n|s_i - \lambda_k|/\operatorname{sep}(P, \lambda_k)$. Again using (2), (3), and (5)

$$\begin{aligned} |\Delta_{ij}| &\le \epsilon\ n^{O(1)}\ \left|\frac{s_i - \lambda_k}{\operatorname{sep}(P,\lambda_k)}\right| \\ |(\Delta_{ij}\ \operatorname{char}_{ij} M)(\lambda_k)| &\le \epsilon\, \alpha^2\, n^{O(1)}\ \left|\frac{Q_t(\lambda_k)}{\operatorname{sep}(P,\lambda_k)}\right|. \end{aligned} \tag{7}$$

Now suppose $i = j$. By lemma 2.6

$$\begin{aligned} (-1)^{n-1} \operatorname{char}_{ii} M &= \prod_{j\neq i} (x - s_j) \;+ \sum_{p\neq i} l_p \prod_{j\neq i,p} (x - s_j) \\ &= \Big(1 + \sum_{p\neq i} \frac{l_p}{x - s_p}\Big) \prod_{j\neq i} (x - s_j). \end{aligned}$$

The analysis in this case is essentially identical to the analysis following equation (5) above. Lemma 3.2 now follows from (4), (6), (7), and the corresponding bounds in the case $i = j$. □

The proof of lemma 3.2 gives an upper bound on q of roughly $O(t(n)\, n^4\, \alpha^3)$, where $t(n)$ is the backward error bound on floating-point eigenvalue computation from section 2.3. In practice, for polynomials with degree between 100 and 1000, we have observed $q \lesssim 2^{10}$, with no obvious dependence on degree.

Lemma 3.2 depends upon the factor $|\lambda_{\max}|/\operatorname{sep}(P,\lambda)$, implying that convergence is slow if roots are closely spaced relative to the largest root $\lambda_{\max}$. This slow convergence can be observed empirically in a straightforward implementation of the algorithm. However, convergence is greatly improved by deflation and shifting, as discussed in section 4.

3.3 Bounding the number of iterations

Theorem 3.3 *Suppose every iterate S_i of* `eigensolve` *is α-generic. Then with q as in Lemma 3.2 and*

$$\begin{aligned} \beta &= \epsilon\, q(n,\alpha)\, \frac{|\lambda_{\max}|}{\mathrm{sep}(P)} \\ f &= \lceil \log_\beta \chi(P) \rceil \end{aligned}$$

we have that S_f approximates the roots of P almost to floating-point precision. That is, there is a permutation σ on $\{1,\ldots,n\}$ so that for $i = 1,\ldots,n$,

$$|\lambda_i - s_{f\sigma(i)}| \le n^{O(1)}\, \epsilon \max(1, |\lambda_i|).$$

Lemma 3.4 *If S_0 is computed using the QZ-algorithm [26] on the companion matrix $C(P)$, then for each root λ of P, $|Q_0(\lambda)| \le n^{O(1)}\, \epsilon ||P|| \max(1, |\lambda|^n)$.*

Proof: By the assumptions on the behavior of `eig`, $S_0 = \lambda(\tilde{M})$ where $\tilde{M} = C(P) + \Delta M$, $\Delta M \le n^{O(1)}\, \epsilon ||C(P)||$. By a result of Van Dooren and Dewilde [26] (see also [8]), $\lambda(\tilde{M}) = \lambda(C(\tilde{P}))$ where $\tilde{P} = P + \Delta P$, $||\Delta P|| \le n^{O(1)}\, \epsilon ||P||$. But $\tilde{P} = Q_0$, since they are both monic polynomials with the same roots. Hence $|Q_0(\lambda)| = |P(\lambda) + \Delta P(\lambda)| = |\Delta(P)(\lambda)|$. The lemma follows using $|\Delta P(\lambda)| \le n ||\Delta P|| \max(1, |\lambda|^n)$. □

As discussed in section 4.2.4, in practice it is sufficient to use the ordinary QR algorithm on the initial companion matrix $C(P)$.

Lemma 3.5 *Suppose at the tth iteration, each root λ of P satisfies*

$$|Q_t(\lambda)| \le n^{O(1)}\, \epsilon \max(1, |\lambda|) |P'(\lambda)|.$$

Then S_t approximates the roots of P almost to floating-point precision, as in theorem 3.3.

Proof: Apply lemma 2.8 with the role of P taken by Q_t and Q by P. Then S_t is contained in the union $\cup D(\lambda, n^{O(1)}\, \epsilon |\lambda|)$, λ varying over the roots of P. Furthermore each connected component of the union consisting of k discs contains exactly k elements of S_t. The lemma follows easily, since any component of k discs is contained in a single disc of radius about k times bigger than any individual disc. □

Proof of Theorem 3.3: The ratio of the bounds in lemmas 3.4 and 3.5 satisfies

$$\frac{n^{O(1)}\, \epsilon ||P|| \max(1, |\lambda|^n)}{n^{O(1)}\, \epsilon \max(1, |\lambda|) |P'(\lambda)|} \le n^{O(1)} \chi(P).$$

The bound on f follows from lemma 3.2. □

3.4 The genericity assumption

Each iterate $S = \{s_1, \ldots, s_n\}$ is the result of a floating-point eigenvalue computation on a matrix M so that char $M = P$. Hence $Q_S = \text{char}(M + \Delta M)$ for some perturbation matrix ΔM determined by the rounding during the floating point computation. Let $G_S = Q_S - P$. It is plausible that G_S should be both "generic," in the sense that it should have no unusual properties, and "tiny" in norm relative to the norm of P. We give plausibility arguments why each iterate S is α-generic, for some small $\alpha > 0$, given that G_S is "generic" and "tiny". We remark that it is far from obvious how to make these plausibility arguments rigorous. Each case below corresponds to a case in the definition in section 3.1.

(1) The manifold of nonsimple polynomials has codimension 1 in the space of polynomials. Hence for almost all choices of G_S, $P+G_S$ is simple.

(2) By standard results on containment discs for roots of polynomials [16, section 6.4], for any polynomial R of degree $n > 0$ there is a real $r(R) > 0$ so that the annulus $A(R) = \{x \in \mathbb{C} : r(R) \geq |x| \geq r(R)/(ne)\}$ contains the root of largest magnitude of R. The value $r(R)$ is given by a continuous and well-conditioned function of the coefficients of R. As long as G_S is actually a coefficientwise small perturbation of P (i.e. the ith coefficient of G_S is small relative to the ith coefficient of P), then the annuli $A(P)$ and $A(P+G_S)$ are similar, and $\max_i |s_i| \lesssim ne \max_i |\lambda_i|$. Unfortunately, error analysis only guarantees that $||\Delta M||$ is small relative to $||M||$. There is some theoretical work relating $||\Delta M||$ and $||G_S||$ [8, 26]. Furthermore, experimental work by Toh and Trefethen [25] suggests that eigenvalue computation of the ordinary companion matrix $C(P)$ is indeed coefficientwise backwards stable, at least if the companion matrix is balanced [14] before the floating-point eigenvalue computation.

(3) The Lagrange coefficient l_i satisfies $l_i = P(s_i)/Q'_S(s_i) = P(s_i)/(P'(s_i)+G'_S(s_i))$. If G_S is "generic", then so is G'_S. Hence $P'(s_i)$ and $G'_S(s_i)$ are uncorrelated, and $|l_i| \lesssim |P(s_i)/P'(s_i)|$. Let λ_k be the root of P closest to s_i (the argument works for any root). Let t_0=0, t_1, $\ldots$, t_n be the coefficients of the Taylor expansion of P about λ_k. For $\Delta = s_i - \lambda_k$, a little manipulation yields

$$\frac{P(s_i)}{P'(s_i)} = \Delta \frac{t_1 + t_2\Delta + \ldots + t_n\Delta^{n-1}}{t_1 + 2t_2\Delta + \ldots + (n-1)t_n\Delta^{n-1}} = \Delta \frac{T_1(\Delta)}{T_2(\Delta)}.$$

Since G_S is "generic" and s_i is a root of $P+G_S$, Δ is "generic," and $|T_2(\Delta)| \approx |T_2|(|\Delta|)$, where $|T_2|$ is the polynomial obtained from T_2 by replacing each coefficient with its absolute value. Hence $|T_1(\Delta)/T_2(\Delta)| \lesssim 1$, and $|l_i| \lesssim |\Delta|$.

(4) Let λ_k and γ be the roots of P and G_S, respectively, closest to s_i. As long as γ is not too close to s_i, in particular as long as $(n+1)|\lambda_k - s_i| < |\gamma - s_k|$, Proposition 2.2 guarantees $|G_S(\lambda_k)| < e\,|G_S(s_i)|$. (The condition that s_i is $1/n$-close to λ_k is not needed, except in the informal sense that $|\lambda_k - s_i|$ is small.)

4 Implementation

The algorithm has been implemented in C++ using EISPACK [22] (or LAPACK [1]) for floating-point eigenvalue computation and GMP [17] for extended-precision arithmetic. The source code,

exclusive of libraries, is about 3300 lines of C++. The implementation accepts polynomials of arbitrary degree with either real or complex coefficients, with real and imaginary parts specified either as an arbitrary-precision integers or arbitrary-precision rationals. The implementation can approximate roots to any requested relative error. The interface—polynomial file format and available options—is similar to the interface of Bini and Fiorentino's `mpsolve`, though not as extensive. The implementation is available by following the directions on the `eigensolve` web site [13].

The remainder of this section describes the implementation. It can be viewed as having three parts: the initial approximation, the eigenvalue iteration itself, and a higher-level driver. The driver contains methods to test convergence, to compute root approximations to arbitrary precision, and to deal with repeated or very closely spaced roots. The driver is largely independent of the eigenvalue iteration. Section 4.1 below describes the driver, section 4.2 describes the initial approximation and the eigenvalue iteration, and section 4.3 reports on the overall performance of the implementation.

4.1 The driver

Lemma 2.8 gives computable bounds on the error of a set of root estimates. In principle, the driver could rely just on these error bounds for detecting convergence and isolation of root estimates. The implementation uses a somewhat more complicated scheme, with multiple kinds of error estimates, because this scheme can often detect root convergence more quickly.

Let $S = \{s_1, \ldots, s_n\}$ be root estimates and $D_1, \ldots, D_n$ be discs with $s_i \in D_i$. A *cluster* is a maximal subset of S so that the union of the corresponding discs is connected. The *Gershgorin disc* for s_i is the disc $D(s_i, |nl_i|)$, whre l_i is the Lagrange coefficient. The *Newton disc* for s_i is the disc $D(s_i, n|P(s_i)/P'(s_i)|)$; by standard results [16], the Newton disc always contains a root of P.

Let ρ be the requested relative error on root approximations. A disc $D(c, r)$ is *small* if $|r| \leq \rho|c|$. Suppose $T = \{t_1, \ldots, t_k\} \subseteq S$. A *small containment disc set* for T is a set of small discs $D_1, \ldots, D_k$ so that (1) $s_i \in D_i$; (2) for $i \neq j$ either $D_i = D_j$ or $D_i \cap D_j = \emptyset$; (3) if there are t root estimates in some disc D_i, then the disc contains at least t roots of polynomial P.

The driver partitions the current root estimates into two subsets, the *tentatively converged* root estimates for which a small containment disc set is known, and the remainder, the *active* root estimates. All root estimates are active after the initial computation.

The driver, slightly simplified, is given as pseudocode in figure 4. At each iteration, the driver computes the Gershgorin discs for the active root estimates (see section 4.2.2 for a description of the computation of Lagrange coefficients). These discs are combined with the small containment discs for the tentatively converged root estimates. The clusters are computed in a straightforward fashion with an algorithm that checks every pair of discs for intersection.

Consider a cluster that contains more than one root estimate. If the corresponding actual roots are repeated or very closely spaced, then the eigenvalue iteration converges slowly. Section 4.1.1 describes a cluster-shrinking heuristic [3] that attempts to replace the root estimates in the cluster with another set of root estimates that are known to lie in a smaller set of discs.

The extraction of tentatively converged root estimates is a bit involved. There are three kinds of clusters: clusters consisting solely of root estimates that were tentatively converged at the previous

```
compute initial root estimates
while true
        compute Gershgorin discs for active root estimates
        compute clusters
        shrink clusters, if possible
        extract tentatively converged root estimates
        if no root estimates are active then quit endif
        apply eigenvalue iteration to active root estimates
        end
```

Figure 4: Simplified driver.

iteration (clearly these estimates can remain tentatively converged); clusters consisting solely of active estimates; and clusters consisting of both active and tentatively converged estimates (this last is theoretically possible but rare; all root estimates become active).

Consider a cluster in the second category, with $k \geq 1$ root estimates. Suppose the union of the corresponding Gershgorin discs is contained in a disc D. By lemma 2.8, D contains k roots, so if D is small, all the root estimates in the cluster can become tentatively converged. If $k > 1$ and D is not small, the root estimates remain active. Finally, if $k = 1$ and D is not small, Newton iteration can be attempted to update the root estimate to the requested relative error and to discover a small Newton disc that is guaranteed to contain a root.

In practice it happens reasonably frequently that some Lagrange coefficient l_i is small, but the disc $D(s_i, nl_i)$ intersects some other larger Gershgorin disc. Hence Lemma 2.8 does not guarantee that there is a root in the disc $D(s_i, nl_i)$; however, often there is a root in the disc. The following strategy attempts to discover when the disc in fact contains a root, while minimizing the evaluation of the derivative P' (which is computationally expensive).

An active root estimate s_i is a *candidate* if its Gershgorin disc $D(s_i, nl_i)$ is locally small, in particular $nl_i \leq \sqrt{\epsilon} \min_{j \neq i} |s_j - s_i|$, and the only intersection of disc $D(s_i, nl_i)$ with other active discs is with active discs of much larger radius. If s_i is a candidate, its Newton disc $D(s_i, |nP(s_i)/P'(s_i)|)$ is computed; if its radius is not relatively small, s_i is no longer considered a candidate. Otherwise the Newton disc is compared with the discs of all other tentatively converged root estimates. If there is no intersection, s_i is updated to requested relative error with Newton iteration and becomes tentatively converged. If there is an intersection (which is rare), all root estimates that are involved in the intersection are marked active.

4.1.1 Cluster shrinking

Suppose $s_1, \ldots, s_k$, $1 < k < n$, form a cluster of root estimates of radius R. The following strategy attempts to replace this cluster with another cluster $s'_1, \ldots, s'_k$ of radius $r \ll R$; the strategy is very similar to the cluster analysis proposed by Bini and Fiorentino [3]. Let c be the center of mass of $s_1, \ldots, s_k$. Starting from c, use Newton iteration to find a zero z of $P^{(k-1)}$ near c. Use the Van

Vleck bound [16, section 6.4] to compute a radius r about c that contains k roots. If $r \ll R$, choose $s'_1, \ldots, s'_k$ uniformly spaced on the circle of radius r about R; otherwise do not replace the cluster $s_1, \ldots, s_k$.

This shrinking strategy is potentially applied to every cluster of size greater than one. However, shrinking is expensive; in particular, computation of the Van Vleck bound requires the evaluation of the first k derivatives of P at z. Various filtering steps are used to decide if the shrinking strategy will be worthwhile. The most nontrivial is to evaluate $\hat{r} = (P(c)/P'(c))^{1/k}$ and to continue with shrinking only if $\hat{r} \ll R$. It is not hard to see that $\hat{r}$ is a lower bound to the Van Vleck radius at c, and usually c and z are not far apart.

4.2 The eigenvalue iteration

The input to the eigenvalue iteration is a set of active root estimates

$$\tilde{S} = s_1, \ldots, s_k, \quad 1 < k \leq n;$$

the remaining root estimates $s_{k+1}, \ldots, s_n$ have tentatively converged to roots $\lambda_{k+1}, \ldots, \lambda_n$, respectively. Let

$$\tilde{P} = \frac{P}{\prod_{j=k+1}^{n} (x - \lambda_j)}.$$

Section 4.2.1 below describes the arithmetic types used in the implementation. Section 4.2.2 illustrates the use of these arithmetic types in the Lagrange coefficient calculation. Section 4.2.3 discusses the formulation of the matrix $C(\tilde{P}, \tilde{S})$—actually a shifted and scaled approximation of $C(\tilde{P}, \tilde{S})$. Finally, the initial companion matrix solve is discussed in section 4.2.4.

4.2.1 Arithmetic types

Assuming IEEE arithmetic, a native C++ `double` value is at most 2^{1024} and the relative error of any arithmetic operation on `double`s is at most $\epsilon = 2^{-53} \approx 10^{-16}$. Arithmetic on `double` values is extremely fast; unfortunately, neither the dynamic range nor the precision is adequate. The implementation uses several alternate arithmetic types to represent real numbers. There are complex analogs of each of these classes as well.

`dpe` ('double plus exponent'): This class [2, 3] contains a `double` d, with $d = 0$ or $.5 \leq |d| \leq 1$, and an `int` i, representing the number $d \cdot 2^i$. It allows representation of real numbers with the precision of a `double` but with very large dynamic range. C++ operator overloading provides convenient syntax for the usual arithmetic operations.

`LongDouble`: This class provides extended-precision floating-point arithmetic with error bound accumulation. The maximum bit-length of any result is specifiable on a per-operation basis. It is built using GMP integers (type `mpz_t`) for arithmetic operations.

Again C++ operator overloading is used. However only assignment and update operators (`+=`,`*=`, etc.) are provided; this simplifies the implementation of the class and minimizes GMP memory allocation, though slightly complicating use of the class.

`OffsetLongDouble`: This class consists of a reference to a `LongDouble` l and a `dpe` value d, representing the real number $l + d$, and a `dpe` approximation to the sum. The only arithmetic operation is to compute the difference of two `OffsetLongDouble` instances, returning an approximation represented as a `dpe`; it is described below.

Root estimates are represented using the complex version of the `OffsetLongDouble` class. Usually a collection of closely spaced roots is represented by a set of class instances with a common `LongDouble` reference but with differing `dpe` values.

4.2.2 Lagrange coefficients

The computation of $C(\tilde{S}, \tilde{P})$ seems to require the deflated Lagrange coefficient

$$\tilde{l}_i = \frac{\tilde{P}(s_i)}{\prod_{j=1, j\neq i}^{k} (s_i - s_j)}, \quad i = 1, \ldots, k,$$

for each active root estimate s_i. However, it suffices to use the approximation $\tilde{l}_i \approx l_i$, as long as $|s_j - \lambda_j| \leq \epsilon \, |s_i - s_j|$ for any tentatively converged root estimate s_j. This requirement is quite mild, and in any event can be guaranteed by appropriate Newton iteration. To see that $\tilde{l}_i \approx l_i$, note that

$$\frac{l_i}{\tilde{l}_i} = \prod_{j=k+1}^{n} \frac{s_i - \lambda_j}{s_i - s_j} = \prod_{j=k+1}^{n} \frac{(s_i - s_j) + (s_j - \lambda_j)}{s_i - s_j} = \prod_{j=k+1}^{n} (1 + \delta_j)$$

where $|\delta_j| \leq \epsilon$.

The computation of the Lagrange coefficients $l_i = P(s_i) / \prod_{j\neq i}(s_j - s_i)$ illustrates the use of the various arithmetic types. Each Lagrange coefficient is represented using the `dpe` class, since it may be very large or small but will only be used to construct a floating-point matrix, so is needed only to floating-point precision.

The computation of all Lagrange coefficients requires quadratically many differences between pairs of roots. Hence it is worthwhile to optimize the `OffsetLongDouble` difference operation. If s_i and s_j have the same `LongDouble` reference, the difference operation is just difference on the `dpe` components; if s_i and s_j have different `LongDouble` references and are well-separated, the difference can be computed from the corresponding `dpe` approximations. If s_i and s_j are close together but with different `LongDouble` references, then they are each converted to `LongDouble`, subtracted, and the result rounded to `dpe`. The last case is relatively infrequent, so overall the `OffsetLongDouble` class usually provides the difference operation in about the time taken by a `dpe` difference. (Indeed, this performance goal motivates the design of the `OffsetLongDouble` class in the first place.)

The evaluation of P at s_i is performed using Horner's rule. The value is needed only to floating-point accuracy, and hence can be represented with the `dpe` class. However, the computation may

involve massive cancellation; so the `LongDouble` class is required for intermediate values. Since the required arithmetic precision is not known *a priori*, the value $P(s_i)$ is computed using a standard doubling strategy, that is, using b, $2b$, $4b$, ... bits of precision, until the relative error in the result is at most ϵ. The initial value of b is determined by *precision caching*, i.e. by using the precision required for evaluation of $P(t_i)$, where t_i is the root estimate of the preceding iteration corresponding to s_i. The correspondence of root estimates between iterations is determined using a simple heuristic—the index of the root estimates in a lexicographic sort.

The products of differences and the final division in the evaluation of l_i can be computed using the `dpe` class.

4.2.3 The generalized companion matrix

The root estimates $s_1, \ldots, s_k$ and the Lagrange coefficients $l_1, \ldots, l_k$ are available as instances of the `OffsetLongDouble` and `dpe` class, respectively. Recall

$$C(\tilde{P}, \tilde{S}) = \mathrm{diag}(s_1, \ \ldots, \ s_k) - L$$

where $L_{ij} = \tilde{l_j}$. It is not sensible to construct the matrix $C(\tilde{P}, \tilde{S})$ directly, since for example its entries may not lie in the dynamic range allowed by hardware floating-point.

Instead we compute the center of mass c of $s_1, \ldots, s_k$ (using the `LongDouble` class) and compute the following matrix rounded to hardware floating point

$$\sigma\Big(\mathrm{diag}(s_1{-}c, \ \ldots, \ s_k{-}c) - L \Big),$$

where the scaling factor σ is chosen so that the maximum entry has magnitude about one. The eigenvalues $\mu_1, \ldots, \mu_k$ of this matrix are then computed using a library eigenvalue subroutine. The new root estimates are $c + \mu_i/\sigma$, $i{=}1, \ldots, k$, which are represented using the `OffsetLongDouble` class as the pair $(c, \mu_i/\sigma)$. In this way, hardware floating-point eigenvalue calculation can be used even if the actual root estimates are very large, very small, or very closely spaced.

It sometimes happens that some Lagrange coefficients have much smaller magnitude than others; the following matrix simplification heuristic can then be employed. If a Lagrange coefficient l_i satisfies $|l_i| \leq \kappa\epsilon \max_j |l_j|$, then the ith row and column is eliminated from the matrix. Since the eigenvalue computation time is cubic in the matrix size, the computation time can be substantially reduced if even a small fraction of the rows and columns are eliminated. This elimination can be justified as a perturbation of the matrix of relative magnitude $\kappa\epsilon$, or equivalently as increasing the relative error in arithmetic to $\kappa\epsilon$. The number of iterations increases with κ but the time per iteration decreases. We use $\kappa \approx 100$.

4.2.4 The initial companion matrix

The initial root estimates are obtained from an eigenvalue computation on a matrix obtained from the polynomial coefficients. As in the general case, it is important to shift and scale.

Let $P(x) = x^n + p_{n-1}x^{n-1} + \ldots + p_0$. We compute the coefficients of $Q(y) = P(a(x{+}c))$, where c is the center of mass of the roots, $c = (\lambda_1 + \ldots + \lambda_n)/n = -p_{n-1}/n$, and the scaling factor a is

chosen so that the coefficients of Q have magnitude smaller than the maximum allowed `double`. The coefficients of Q are used to construct the ordinary companion matrix $C(Q)$. Its eigenvalues are computed, then rescaled and shifted, as in the general case.

If also possible, the scaling factor a is chosen so that the low-order coefficient is 1. The first step of the floating-point eigenvalue solver is to balance [14] the companion matrix. We have discovered experimentally that balancing is most successful, i.e., the magnitude of matrix entries is reduced the most, if the low order coefficient is close to 1. This leads to more accurate initial root estimates, in the sense of requiring the fewest subsequent iterations.

4.3 Performance

This section compares `eigensolve` with the Bini-Fiorentino distribution of `mpsolve` (version 2.1, January 2000). The Bini-Fiorentino distribution contains a library of many test polynomials of widely varying degrees and difficulty. The `eigensolve` web site [13] gives complete timing information for most of these polynomials under a variety of scenarios. The timings given here are on a 250 Mhz SGI running IRIX, with programs compiled using the IRIX C/C++ compiler with optimization flag `-O2`. Both programs were directed to approximate all roots to 15 decimal digits (default behaviour for `eigensolve`, flags `-Ga -o15` for mpsolve).

Figure 5 compares the execution times of `eigensolve` and `mpsolve` on some ill-conditioned examples from the distribution. On these examples, `eigensolve` does very well, with the eigenvalue iteration clearly converging much faster than the Aberth iteration used in `mpsolve`. For example, for the Wilkinson polynomial of degree 80, with initial root estimates chosen as described in section 4.2.4, Aberth iteration take 45 iterations to converge, while `eigensolve` takes just one (figure 7).

The `eigensolve` implementation is not uniformly faster than `mpsolve`, see figure 6. For example, `mpsolve` can take advantage of sparsity, and hence is much faster than `eigensolve` for polynomials like x^d-1 for large d. (It is also the case that the initial guess of `mpsolve` lies on a circle, which is particularly appropriate for a polynomial in this family. If the polynomial is shifted, e.g. $(x-1)^d-1$, `mpsolve` does badly.) Similarly, `mpsolve` does well on Mignotte-like polynomials. Even though these polynomials have some closely spaced roots, they are sparse and most roots are well-conditioned. There is no obvious way for `eigensolve` to exploit sparsity, since the generalized companion matrix $C(P, S)$ is dense even if P is sparse.

The `mpsolve` implementation also accepts polynomials presented as a straight-line program. In this representation the Mandelbrot polynomials are very compact. For the Mandelbrot polynomial of degree 511, `mpsolve` requires about 1/20th the time in the straight-line representation compared with the coefficient representation, and hence would be about three times faster than `eigensolve` in the coefficient representation.

The 'geometric' polynomials, $\prod_i^d (x - 4^i)$, form an interesting set of examples. Both implementations appear to find the roots in order from largest to smallest; both appear to take an iteration or two to find each root. However, an eigenvalue iteration is slower than the corresponding Aberth iteration. Usually the improved convergence rate more than makes up for the slower iteration, but evidently not in this case.

Polynomial	deg	mpsolve	eigen	ratio
Wilkinson	40	1.1	.2	5.5
	80	12.3	1.9	6.5
	160	144.0	12.1	11.9
	320	2138.0	87.6	24.4
	640		1448.0	
	1000		5938.0	
Laguerre	80	5.1	1.6	3.2
	160	59.0	9.9	6.0
	320	806.0	86.4	9.3
	1000		5110.0	
Chebyshev	160	39.0	10.0	3.9
	320	450.0	86	5.2
Chromatic a	342	1150.0	67.8	17.0
Chromatic ad	340	1848.0	81.5	22.7
Chromatic cd	171	123.1	11.5	10.7
	683	>15750.0	610.5	>25.8
Hermite	160	20.4	8.1	2.5
	320	230.0	78.8	2.9
Mandelbrot	127	11.4	4.3	2.6
	255	134.0	30.9	4.3
	511	1874.0	246.6	7.6
	1023		3048.0	
Sendra	160	291.0	10.9	26.7
	320	5200	78.4	66.3
Exponential	50	.3	.5	.6
	100	3.8	2.5	1.5
	200	43.0	17.2	2.5
	400	484.7	125.4	3.9
Toeplitz 1	128	5.8	4.2	1.4
	256	60.2	31.6	1.9
Toeplitz 2	128	16.3	6.2	2.6
	256	190.4	49.25	3.8

Figure 5: Comparison of `eigensolve` and `mpsolve` (version 2.1, January 2000) on ill-conditioned polynomials; times are seconds on a 250MHz SGI R10000. Roots are computed to 15 decimal digits. All the polynomials except Wilkinson 1000 and Laguerre 1000 are from the `mpsolve` distribution.

Polynomial	deg	mpsolve	eigen	ratio
$x^d - 1$	400	3.5	23.8	0.15
	800	12.0	134.0	0.09
	1600	52.2	835.6	0.06
$(x-1)^d - 1$	400	1824.3	42.5	42.92
Mignotte	200	0.9	11.2	0.08
	500	9.7	108.4	0.09
Geometric	20	0.04	0.23	0.17
	40	0.15	3.08	0.05
	80	1.24	44.63	0.03
Shifted Wilk.	100	508.0	5.3	95.80

Figure 6: Comparison of `eigensolve` and `mpsolve`, continued. The Mignotte and Geometric polynomials are from the `mpsolve` distribution. The "Shifted Wilkinson" polynomial has integer roots $10^4+1, \ldots, 10^4+100$.

4.3.1 Optimizations

The running time of `eigensolve` is roughly balanced between floating-point eigenvalue computation and polynomial evaluation, with the other costs of the algorithm negligible. For the polynomials in figure 5, the number of iterations is roughly linear in the degree d, typically varying between $d/25$ and $d/100$. The optimizations discussed above contribute significantly to the speed of `eigensolve`. For example, for the Wilkinson polynomial of degree 320, solve time increased roughly by a factor 1.9 without shifting the initial companion matrix, 1.8 without precision caching, 1.25 without matrix simplification, and 1.3 without deflation. Without any of these optimizations, solve time increased by a factor of 5.4. Shifting the initial companion matrix is very beneficial in some cases: for the contrived "Shifted Wilkinson" polynomial in figure 5, shifting reduces the number of iterations from 22 to 2.

5 Conclusion

The conditioning of numerical problems, e.g. the computation of eigenvalues and the solution of linear systems, is a central topic of numerical analysis. For many such problems backwards-stable algorithms are known; such algorithms are the best possible if numeric input is only known approximately. In some cases, an input instance may be known exactly. If the instance is ill-conditioned, the backwards-stable algorithm does not provide an accurate solution using standard (hardware) floating-point precision. In principle the backwards-stable algorithm would suffice if arithmetic were sufficiently accurate, but in general this requires expensive software extended-precision arithmetic.

The algorithm in this paper uses an iterative conditioning technique to obtain a forward-stable algorithm. Given an instance that is known exactly, but is perhaps ill-conditioned, the problem is first solved using a standard backwards-stable floating-point algorithm. The result of the floating-point computation then is used to compute a new instance which is better conditioned but has the

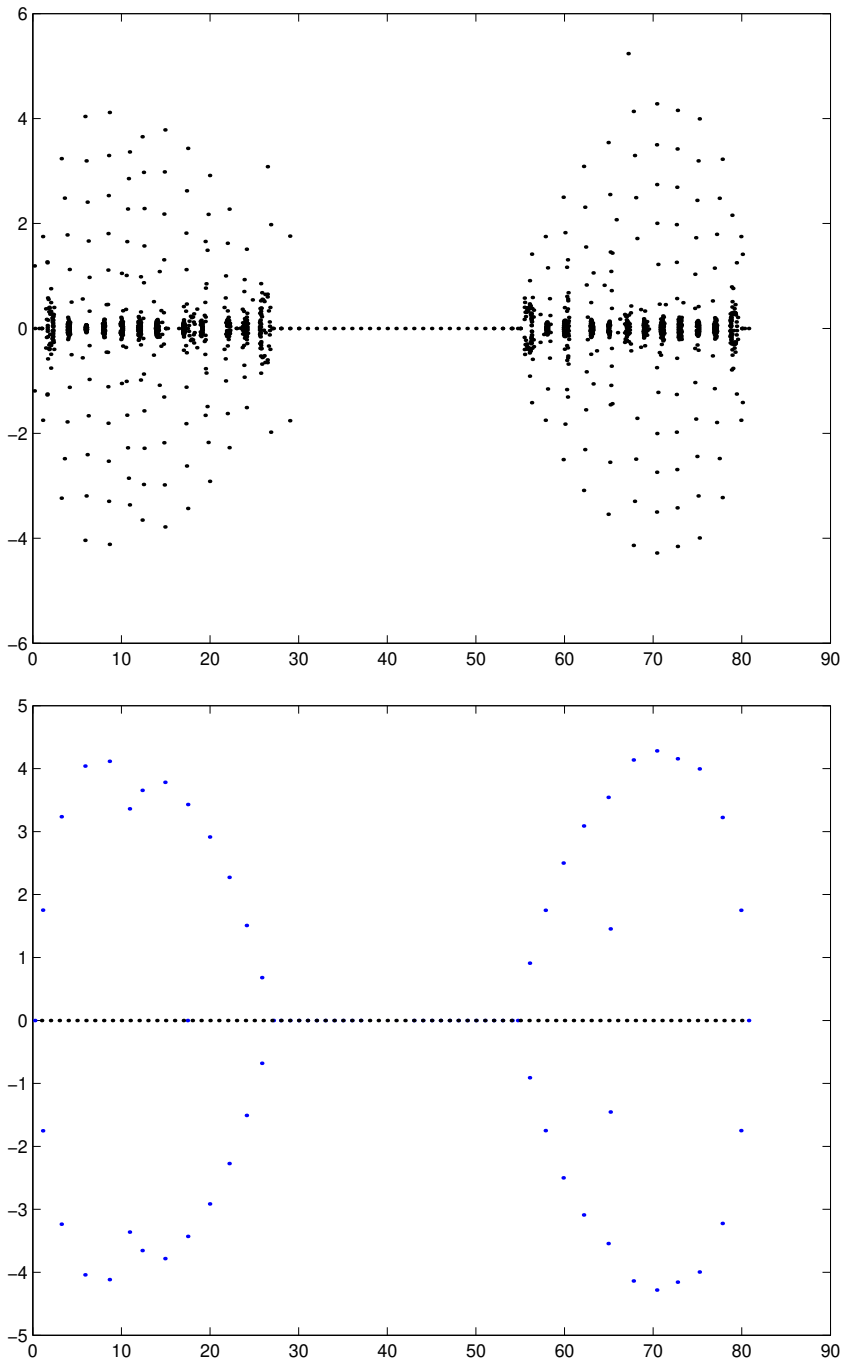

Figure 7: Aberth iterates (top) and `eigensolve` iterates (bottom) on the Wilkinson polynomial of degree 80. In both cases the initial root estimates are computed using the method in section 4.2.4. The top figure was produced by extending `eigensolve` to include Aberth iteration.

same solution as the original instance. This iteration is repeated until a well-conditioned instance is obtained, and the solution to it can be returned.

This iterative conditioning technique appears very promising. In addition to the algorithm in this paper, the technique has been applied to the sign-evaluation of the determinant of integer matrices [5] and to the computation of the inertia of symmetric integer matrices [10]. Roughly, the technique replaces a single computation which requires an n-fold increase in arithmetic precision with n computations, each of which requires standard arithmetic. Application of the technique is not trivial, since considerable cleverness may be required to use the result of a floating-point computation to improve the conditioning of an instance. It is worth exploring whether this technique applies to other numerical problems.

Acknowledgements. Thanks to Dario Bini and Giuseppe Fiorentino for providing a public web site with `mpsolve` and many test polynomials.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorenson, *LAPACK User's Guide*, Second Edition, SIAM, 1995.

[2] D.H. Bailey, A portable high performance multiprecision package, RNR Technical report RNR-90-022, NAS Applied Research Branch, NASA Ames Research Center.

[3] D.A. Bini, G. Fiorentino, Design, analysis, and implementation of a multiprecision polynomial rootfinder, *Numerical Algorithms*, **23:**127–173, 2000.

[4] D.A. Bini, G. Fiorentino, Numerical computation of polynomial roots using MP-Solve, manuscript, 1999. Software and paper available for download from the address `ftp://ftp.dm.unipi.it/pub/mpsolve/`.

[5] K. Clarkson, Safe and effective determinant evaluation, *Proc. 33rd Annual IEEE Symp. on Found. of Computer Science*, pp. 387–395, 1992.

[6] J. Demmel, On condition numbers and the distance to the nearest ill-posed problem, *Numerische Mathematik*, **51**:251–289, 1987.

[7] J. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997.

[8] A. Edelman, H. Murakami, Polynomial roots from companion matrix eigenvalues, 1994. *Mathematics of Computation*, **64**:763–776, 1995.

[9] L. Elsner, A remark on simultaneous inclusions of the zeros of a polynomial by Gershgorin's theorem, *Numer. Math.*, **21**:425–427, 1973.

[10] S. Fortune, Exact computation of the inertia of a symmetric integer matrix , *STOC 2000*, pp. 556–564.

[11] S. Fortune, Polynomial root finding using iterated eigenvalue computation, *ISSAC 2001*, pp. 121–128.

[12] S. Fortune, Convergence analysis of an iterated-eigenvalue polynomial root-finding algorithm, submitted to workshop proceedings.

[13] S. Fortune, Eigensolve web page, `http://cm.bell-labs.com/who/sjf/eigensolve.html`.

[14] G.H. Golub, C.F. Van Loan, *Matrix Computations*, 2nd edition, The Johns Hopkins University Press, 1990.

[15] X. Gourdon, *Combinatoire, Algorithmique et Géométrie des Polynômes*, Ph.D. thesis, L'École Polytechnique, 1996, available at `pauillac.inria.fr/algo/gourdon/`.

[16] P. Henrici, *Applied and computational complex analysis: Vol. 1*, John Wiley and sons, 1974.

[17] T. Granlund, GNU MP: the GNU multiple precision arithmetic library, Edition 2.0, 1996. Code available for download from `www.swox.com/gmp/`.

[18] F. Malek, R. Vaillancourt, A composite polynomial zerofinding matrix algorithm, *Computers Math. Applic.* **30**(2):37–47, 1995.

[19] V. Pan, Solving a polynomial equation: some history and recent progress, *SIAM Review* **39**:2, pp. 187–220, 1997.

[20] MJ. McNamee, A bibliography on roots of polynomials, *J. Comput. Applied Mathematics*, **47** pp. 391–394, 1993. An updated bibliography is available at the address `http://www.atkinson.yorku.ca/ mcnamee/BIBLIOG4.html`.

[21] B.T. Smith, Error bounds for the zeroes of a polynomial based upon Gershgorin's theorem, *Journal of the Assoc. Comput. Mach.*, **17**:661-674, 1970.

[22] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, C.B. Moler, Matrix Eigensystem Routines - EISPACK Guide, volume 6 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1976. Source code is available from netlib at `www.netlib.org`.

[23] P. Tilli, Convergence conditions of some methods for the simultaneous computation of polynomial zeroes, *Calcolo* **35**, pp. 3–15, 1998.

[24] F. Tisseur, *Backward stability of the QR algorithm*, Technical Report No 239, UMR 5585, Lyon Saint-Etienne, October 1996.

[25] K.C. Toh, L. N. Trefethen, Pseudozeroes of polynomials and pseudospectra of companion matrices, *Numerische Mathematik*, **68**:403–425, 1994.

[26] P. Van Dooren, P. Dewilde, The eigenstructure of an arbitrary polynomial matrix: computational aspects, *Lin. Alg. Appl*, **50**:545–579, 1983.

[27] C.K. Yap, *Fundamental problems of algorithmic algebra*, Oxford University Press, Oxford, 2000.