Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.18576/amis/110105

Forward Stable Computation of Roots of Real Polynomials with Real Simple Roots

Nevena Jakovčević Stor[*] *and Ivan Slapničar*

Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split, Rudjera Boškovića 32, 21000 Split, Croatia

Received: 12 Mar. 2016, Revised: 20 Oct. 2016, Accepted: 25 Oct. 2016
Published online: 1 Jan. 2017

Abstract: As showed in (Fiedler, 1990), any polynomial can be expressed as a characteristic polynomial of a complex symmetric arrowhead matrix. This expression is not unique. If the polynomial is real with only real distinct roots, the matrix can be chosen real. By using the accurate forward stable algorithm for computing eigenvalues of the real symmetric arrowhead matrices from (Jakovčević Stor, Slapničar, Barlow, 2015), we derive a new forward stable algorithm for computation of roots of such polynomials in $O(n^2)$ operations. The algorithm computes each root to almost full accuracy. In some cases, the algorithm invokes extended precision routines, but only in the non-iterative part. Our examples include numerically difficult problems, like the well-known Wilkinson's polynomials. Our algorithm compares favorably to other method for polynomial root-finding, like MPSolve or Newton's method.

Keywords: roots of polynomials, generalized companion matrix, eigenvalue decomposition, arrowhead matrix, high relative accuracy, forward stability

1 Introduction and Preliminaries

Polynomials appear in many areas of scientific computing and engineering. Developing fast algorithms and reliable implementations of polynomial solvers are of constant interest. The famous example by James H. Wilkinson in 1963 [18], usually referred to as *Wilkinson's polynomial*, is often used to illustrate difficulties when finding the roots of a polynomial. The polynomial of order n is defined by a simple formula:

$$W_n(x) = \prod_{i=1}^{n}(x-i) = (x-1)(x-2)\cdots(x-n).$$

For example, the location of the roots of W_{20} is very sensitive to perturbations in the coefficients [19]. Since then, many methods for finding roots of polynomials have been developed (see for example [3], [8] and [13]).

In [7], Miroslav Fiedler showed that any polynomial can be expressed as a characteristic polynomial of a complex symmetric arrowhead matrix. This expression is not unique. If the polynomial is real with only real distinct roots, the matrix can be chosen real. We have the following theorem:

Theorem 1.*[7, Theorem 3] Let $u(x)$ be a polynomial of degree n,*

$$u(x) = x^n + px^{n-1} + r(x), \tag{1}$$

Let

$$D = \operatorname{diag}(d_1, \ldots, d_{n-1}), \tag{2}$$

where d_j are all distinct and $u(d_j) \neq 0$. Let

$$v(x) = \prod_{j=1}^{n-1}(x-d_j),$$

$$\alpha = -p - \sum_{j=1}^{n-1} d_j, \tag{3}$$

$$z = \begin{bmatrix}\zeta_1 & \zeta_2 & \cdots & \zeta_{n-1}\end{bmatrix}^T,$$

where

$$\zeta_j^2 = \frac{-u(d_j)}{v'(d_j)} \equiv \frac{-u(d_j)}{\prod_{\substack{i=1\\ i\neq j}}^{n-1}(d_j - d_i)}. \tag{4}$$

Then the symmetric arrowhead matrix

$$A = \begin{bmatrix} D & z \\ z^T & \alpha \end{bmatrix}, \tag{5}$$

[*] Corresponding author e-mail: nevena@fesb.hr

has characteristic polynomial $(-1)^n u(x)$.

If $u(x)$ *has only real distinct roots and the* d_j*'s interlace them, then* A *is real.*

Fiedler concludes his paper by stating "One can hope to obtain, by some sophisticated special choice of the numbers d_j, stable or even universal algorithms for solving algebraic equations."[1]

The eigenvalues of the arrowhead matrix A from (5) are the zeros of the secular function [11],

$$\varphi_A(\lambda) = \alpha - \lambda - z^T (D - \lambda I)^{-1} z.$$

Finding roots of polynomials via solution of the corresponding secular equation is not a new idea, see, for example, [4] and the references therein.

In [11], the authors developed a forward stable algorithm for computing eigendecomposition of a real symmetric irreducible arrowhead matrix, which is exactly the matrix A given by Theorem 1 if the polynomial $u(x)$ has only distinct real roots, and the diagonal elements of the matrix D interlace those roots.

The arrowhead matrix A is *irreducible* if d_j are all distinct and $z_j \neq 0$, $j = 1,\ldots,n-1$ [11]. More precisely, the algorithm from [11] computes each eigenvalue and all individual components of the corresponding eigenvector of a given arrowhead matrix of floating-point numbers to almost full accuracy in $O(n)$ floating point-operations, a feature which no other method has.

In this case, we are interested only in the roots of u, that is, in the eigenvalues of A from (5), each of which is computed independently of the others in $O(n)$ operations. This, together with independent computation of elements of z, makes our algorithm suitable for parallel computing.

Let us define floating-point precisions: the *standard precision* or *machine precision* is $\varepsilon_M = 2^{-53} \approx 1.1102 \cdot 10^{-16}$ (see [9, Chapter 2] for details). The *double standard precision* denotes the precision $\varepsilon_M^2 = 2^{-106} \approx 1.2326 \cdot 10^{-32}$. The *polynomial precision* ε_P is the precision needed to store the coefficients of the polynomial to full accuracy, and the *double polynomial precision* is ε_P^2. We assume that $\varepsilon_P \leq \varepsilon_M$.

In this paper, we propose a new two-step algorithm: given a polynomial u with only distinct real roots of the form (1) whose coefficients are given floating-point numbers:

1. compute the generalized companion matrix A from (5), where the elements of z and α need to be computed in double polynomial precision, and then
2. compute the roots of u as the eigenvalues of A by using modified version of the forward stable algorithm *aheig* from [11, Algorithm 5].

[1] In a report by Corless and Litt [5], the matrix A from theorem 1 is referred to as *generalized companion matrix not expressed in monomial basis*. In this case, the basis is the barycentric basis.

To summarize, the proposed algorithm computes roots of given polynomials with floating-point coefficients to almost full standard floating-point accuracy in $O(n^2)$ operations, using the double polynomial precision only in the non-iterative part. The approach is original, since almost full accuracy and even forward stability is obtained using matrix algorithm and by limited use of higher precision.

The high-quality root finder *mpsolve* from the package MPSolve [3,4] computes the roots with the same accuracy, but internally uses higher precision than our algorithm, and does so in the iterative part. The proposed algorithm is clearly not as general as *mpsolve*, but compares favorably to *mpsolve* for the polynomials with distinct real roots.

The organization of the paper is the following. In Section 2, we describe our algorithm named *rootsah* (polynomial ROOTS via ArrowHead eigenvalues). In Section 3, we analyze the accuracy of the algorithm and give forward error bounds – in Section 3.1, we analyze the accuracy of the computed matrix A, and in Section 3.2, we analyze the accuracy of the computed inverse of the shifted matrix A. In Section 3.3, we discuss implementations of increased precision (double standard precision and double polynomial precision): extended precision routines from [6], Compensated Horner's method from [8, Algorithm 4] and Julia `BigFloat` floating-point type. Finally, in Section 4, we illustrate our algorithm with several numerically demanding examples and compare it to the methods from [3,4] and [8]. The implementation of *rootsah* and needed subroutines is publicly available in the Julia [12] package `Arrowhead.jl` [2] (file `src/arrowhead7.jl`).

2 The algorithm

Let $u(x)$ defined by (1) be a real polynomial with only real distinct roots, and let the matrix A be the real symmetric arrowhead matrix from (5), where the diagonal elements of the matrix D interlace the roots of $u(x)$.

The forward stable algorithm for solving EVP of arrowhead matrices [11] computes all eigenvalues to almost full accuracy. The algorithm is based on shift–and–invert strategy. Let d_i be the pole which is nearest to λ. Let A_i be the shifted matrix,

$$A_i = A - d_i I = \begin{bmatrix} D_1 & 0 & 0 & z_1 \\ 0 & 0 & 0 & \zeta_i \\ 0 & 0 & D_2 & z_2 \\ z_1^T & \zeta_i & z_2^T & a \end{bmatrix} \qquad (6)$$

where

$$\begin{aligned} D_1 &= \mathrm{diag}(d_1 - d_i, \ldots, d_{i-1} - d_i), \\ D_2 &= \mathrm{diag}(d_{i+1} - d_i, \ldots, d_{n-1} - d_i), \\ z_1 &= [\, \zeta_1 \ \zeta_2 \ \cdots \ \zeta_{i-1} \,]^T, \\ z_2 &= [\, \zeta_{i+1} \ \zeta_{i+2} \ \cdots \ \zeta_{n-1} \,]^T, \\ a &= \alpha - d_i. \end{aligned} \tag{7}$$

Then,

$$\lambda = \frac{1}{\nu} + d_i,$$

where ν is either the largest or the smallest eigenvalue of the matrix

$$A_i^{-1} \equiv (A - d_i I)^{-1} = \begin{bmatrix} D_1^{-1} & w_1 & 0 & 0 \\ w_1^T & b & w_2^T & 1/\zeta_i \\ 0 & w_2 & D_2^{-1} & 0 \\ 0 & 1/\zeta_i & 0 & 0 \end{bmatrix}, \tag{8}$$

where

$$\begin{aligned} w_1 &= -D_1^{-1} z_1 \frac{1}{\zeta_i}, \\ w_2 &= -D_2^{-1} z_2 \frac{1}{\zeta_i}, \\ b &= \frac{1}{\zeta_i^2}(-a + z_1^T D_1^{-1} z_1 + z_2^T D_2^{-1} z_2). \end{aligned} \tag{9}$$

Notice that all elements of the matrix A_i^{-1} are computed with high relative accuracy, except that in some cases the element b needs to be computed in double polynomial precision (for details see [11]). Also, the elements of z (Horner's method) and α (the trace preservation formula) of A need to be computed in double polynomial precision. Notice that our algorithm requires computation in higher precision only in the finite part, unlike algorithms from [3,8], which require usage of higher precision in the iterative part.

The described procedure is implemented in the algorithm *rootsah*.

The algorithm *aheig_mod* is a simple modification of the algorithm *aheig* from [11, Algorithm 5]. The algorithm *aheig* and its subroutines are analyzed in detail in [11]. The algorithm is essentially based on the assumption that all elements of the matrix A_i^{-1} from (8) can be computed with high relative accuracy, that is, $fl([A_i^{-1}]_{jl}) = [A_i^{-1}]_{jl}(1 + \kappa_{jl}\varepsilon_M)$, for some modest κ_{jl}. For all elements of A_i^{-1} but b, this accuracy is achieved by computing them in standard precision using the standard precision copies of z and α. If, according to the theory from [11], the element b needs to be evaluated in double standard precision, formula (9) is evaluated using z_{double} and α_{double} in order to obtain full possible accuracy. The detailed implementation of the algorithm *aheig* and needed subroutines is publicly available in the Julia [12] package `Arrowhead.jl` [2] (file `src/arrowhead3.jl`).

Algorithm 1

$\lambda =$ **rootsah**(u, D)
% Computes the roots λ of the polynomial $u(x)$ from
% (1) of order n, with distinct real roots.
% Entries of D must interlace the roots of $u(x)$,
% see Section 4 for examples.

% Compute the values of $u(x)$ in the interpolating points
% d_j using double polynomial precision.
for $j = 1 : n-1$
  $s_{double}(j) = u(d(j))$
end
% Compute vector ν from Theorem 1 using double
% polynomial precision.
for $j = 1 : n-1$
  $\nu_{double}(j) = \prod(d(j) - d(1 : j-1, j+1 : n-1))$
end
% compute α from Theorem 1 using double polynomial
% precision.
  $\alpha_{double} = -p - \sum_{j=1}^{n-1} d_j$
% compute vector z from Theorem 1 using double
% polynomial precision.
for $j = 1 : n-1$
  $\zeta_{double}(j) = \sqrt{-s_{double}(j)/\nu_{double}(j)}$
end
% call modified algorithm *aheig*
for $k = 1 : n$
  $\lambda(k) =$ **aheig_mod**$(D, z_{double}, \alpha_{double}, k)$
end

The difference between algorithm *aheig* and its modification *aheig_mod* is that the latter algorithm and its needed subroutines take as input elements of the vector z and scalar α from (5) computed in double polynomial precision. This is necessary in order to compute the roots in the forward stable manner, as shown by the error analysis in the following section. The detailed implementation of the algorithms *rootsah* and *aheig_mod* is publicly available in the package `Arrowhead.jl` [2] (file `src/arrowhead7.jl`, functions *rootsah* and *eig*, respectively).

As already mentioned, algorithms *aheig* and *aheig_mod* compute eigenvalues of the arrowhead matrix by shifting the matrix A to the nearest pole and inverting, and computing the absolutely largest eigenvalue of the resulting arrowhead matrix (8) by bisection. This is always one of the extreme (outer) eigenvalues, so one point of the starting interval is either leftmost or the rightmost pole of the matrix from (8), and the other point is computed by using Gershgorin theorem. The implementation is given in the function *bisect* in the package `Arrowhead.jl` [2] (file `src/arrowhead3.jl`). The stopping criterion for bisection is relative, and, since we are computing absolutely largest eigenvalue, it takes about 50 steps until convergence to standard precision.

3 Accuracy of the algorithm

The error analysis of the algorithm *aheig* is given in [11, Sections 3 and 4]. This analysis assumes that A is the matrix of floating-point numbers in the standard precision ε_M. Here, however, A is computed by using formulas (1-5), which must be taken into account.

Let us first consider the errors in the polynomial evaluation. The classical method for evaluating polynomial $u(x)$ is Horner's method [9, Section 5.1]. Let

$$u(x) = \sum_{i=0}^{n} a_i x^i, \tag{10}$$

and let

$$cond(u,x) = \frac{\sum_{i=0}^{n} |a_i|\,|x|^i}{\left|\sum_{i=0}^{n} a_i x^i\right|} = \frac{\widetilde{u}(x)}{|u(x)|}. \tag{11}$$

Notice that $cond(u,x) \geq 1$.

Let $Horner(x,u)$ denote the value of $u(x)$ computed with precision ε by Horner's method. Depending on whether ε_M or ε_P is used to store the coefficients of $u(x)$, we set $\varepsilon = \varepsilon_M$ or $\varepsilon = \varepsilon_P$, respectively.

Then, the relative error in the computed value is bounded by [9, Section 5.1] [2]

$$\frac{|u(x) - Horner(u,x)|}{|u(x)|} \leq cond(u,x) \times 2n\varepsilon.$$

Thus, when $Horner(u,x)$ is evaluated in double precision ε, the relative error is bounded by

$$\frac{|u(x) - Horner_{double}(u,x)|}{|u(x)|} \leq cond(u,x) \times 2n\varepsilon^2.$$

Therefore,

$$Horner_{double}(u,x) = (1+\kappa_x\varepsilon^2)u(x), \tag{12}$$

where

$$|\kappa_x| \leq cond(u,x) \times 2n. \tag{13}$$

Notice that, if $cond(u,x)$ is uniformly bounded,

$$cond(u,x) \leq \frac{1}{\varepsilon}, \tag{14}$$

then

$$|\kappa_x| \leq 2n. \tag{15}$$

Two other possible ways to evaluate polynomial such that the bounds similar to (12,13) hold are: to evaluate all parts of the respective formulas by using extended precision routines from [6], or to use Compensated Horner's method from [8, Algorithm 4] (see Section 3.3 for details).

We now consider the accuracy of the computed matrices A, A_i and A_i^{-1} from (5), (6) and (8).

[2] In [8,9], the bounds are expressed in terms of quantities $\gamma_k = \frac{k\varepsilon}{1-k\varepsilon}$. For the sake of simplicity, we use standard first order approximations $\gamma_k \approx k\varepsilon$.

3.1 Accuracy of A

Let $\hat{A}$ denote the matrix A computed according to Algorithm 1,

$$\hat{A} = \begin{bmatrix} D & \hat{z}^{(d)} \\ (\hat{z}^{(d)})^T & \hat{\alpha}^{(d)} \end{bmatrix}.$$

Here $\hat{z}^{(d)}$ and $\hat{\alpha}^{(d)}$ are computed in double polynomial precision which we denote by superscript (d). Let

$$\hat{z}^{(d)} = \begin{bmatrix} \hat{\zeta}_1^{(d)} & \hat{\zeta}_2^{(d)} & \cdots & \hat{\zeta}_{n-1}^{(d)} \end{bmatrix}^T.$$

By combining (4) and (12), the standard first order error analysis in double precision ε, gives

$$\hat{\zeta}_j^{(d)} = \sqrt{\frac{-u(d_j)(1+\kappa_{d_j}\varepsilon^2)}{\prod_{\substack{i=1 \\ i\neq j}}^{n-1}(d_j-d_i)(1+\varepsilon_1)(1+(n-3)\varepsilon_2)}}(1+\varepsilon_3)(1+\varepsilon_4), \tag{16}$$

where $|\varepsilon_{1,2,3,4}| \leq \varepsilon^2$. Therefore,

$$\hat{\zeta}_j^{(d)} = \zeta_j(1+\kappa_{\zeta_j}^{(d)}\varepsilon^2), \tag{17}$$

where, by using (13),

$$\left|\kappa_{\zeta_j}^{(d)}\right| \leq \frac{|\kappa_{d_j}|+(n-1)}{2} + 1 \leq n \cdot cond(u,d_j) + \frac{n+1}{2}.$$

Similarly, applying the standard first order error analysis in double precision ε to (3), gives

$$\hat{\alpha}^{(d)} = \alpha(1+\kappa_\alpha^{(d)}\varepsilon^2),$$

where

$$\left|\kappa_\alpha^{(d)}\right| \leq \frac{|a_1| + \sum_{j=1}^{n-1}|d_j|}{|\alpha|}(n-1) \equiv K_\alpha(n-1). \tag{18}$$

3.2 Accuracy of A_i^{-1}

Let $\hat{A}_i^{-1}$ denote the matrix A_i^{-1} computed according to Algorithm 1 from the matrix $\hat{A}$. All elements of A_i^{-1} but possibly b, are computed in the standard precision using the standard precision copies of $\hat{z}^{(d)}$ and $\hat{\alpha}^{(d)}$. Let $\hat{\zeta}_j$ and $\hat{\alpha}$ denote $\hat{\zeta}_j^{(d)}$ and $\hat{\alpha}^{(d)}$ rounded to the nearest standard precision number, respectively. Let $\varepsilon = \min\{\varepsilon_M, \varepsilon_P\}$. Then

$$\hat{\zeta}_j = \zeta_j\left(1+\kappa_{\zeta_j}\varepsilon_M\right), \quad j = 1,\ldots,n-1, \tag{19}$$

$$\hat{\alpha} = \alpha\left(1+\kappa_\alpha\varepsilon_M\right), \tag{20}$$

where, by using (17)–(18),

$$\left|\kappa_{\zeta_j}\right| \le \left(\frac{|\kappa_{d_j}|+(n-1)}{2}\right)\varepsilon_M+1, \quad j=1,\dots,n-1,$$
$$|\kappa_\alpha| \le K_\alpha(n-1)\varepsilon_M+1.$$

Further, according to (13)-(15), if

$$cond(u,d_j) \le \frac{1}{\varepsilon}, \quad j=1,\dots,n-1, \tag{21}$$

then (19) holds with

$$\left|\kappa_{\zeta_j}\right| \le n+2, \quad j=1,\dots,n-1, \tag{22}$$

and if

$$K_\alpha \le \frac{1}{\varepsilon}, \tag{23}$$

then (20) holds with

$$|\kappa_\alpha| \le n. \tag{24}$$

For $j \notin \{i,n\}$, similarly as in [11, Proof of Theorem 4], the standard first order error analysis gives

$$[\hat{A}_i^{-1}]_{jj} = fl\left(\frac{1}{d_j-d_i}\right) = \frac{1}{d_j-d_i}(1+\kappa_{jj}\varepsilon_M), \quad |\kappa_{jj}| \le 2.$$

Similarly, assuming that (21) and (22) hold, for $j \notin \{i,n\}$ we have

$$[\hat{A}_i^{-1}]_{ji} = fl([\hat{A}_i^{-1}]_{ij}) = fl\left(\frac{-\zeta_j(1+\kappa_{\zeta_j}\varepsilon_M)}{(d_j-d_i)\zeta_i(1+\kappa_{\zeta_i}\varepsilon_M)}\right)$$
$$= \frac{-\zeta_j}{(d_j-d_i)\zeta_i}(1+\kappa_{ji}\varepsilon_M), \quad |\kappa_{ji}| \le (2n+7).$$

Finally,

$$[\hat{A}_i^{-1}]_{ni} = fl([\hat{A}_i^{-1}]_{in}) = fl\left(\frac{1}{\zeta_i(1+\kappa_{\zeta_i}\varepsilon_M)}\right)$$
$$= \frac{1}{\zeta_i}(1+\kappa_{ni}\varepsilon_M), \quad |\kappa_{ni}| \le (n+3).$$

We now analyze the accuracy of the computed element b. Let

$$K_b = \frac{|\alpha|+|d_i|+|z_1^TD_1^{-1}z_1|+|z_2^TD_2^{-1}z_2|}{|-a+z_1^TD_1^{-1}z_1+z_2^TD_2^{-1}z_2|}, \tag{25}$$

where D_1, D_2, z_1, z_2 and a are defined by (7). We have two cases. First, if

$$K_b \not\gg 1,$$

then b is computed in standard precision using $\hat{\zeta}_j$ and $\hat{\alpha}$. Let $\hat{b}$ denote the computed b. The standard first order error analysis of (9) gives

$$\hat{b} = fl\left(\frac{1}{\zeta_i^2(1+\kappa_{\zeta_i}\varepsilon_M)^2}\left(\alpha(1+\kappa_\alpha\varepsilon_M)-d_i + \sum_{\substack{j=1\\ j\neq i}}^{n-1}\frac{\zeta_j^2(1+\kappa_{\zeta_j}\varepsilon_M)^2}{d_j-d_i}\right)\right)$$
$$= b(1+\kappa_b\varepsilon_M),$$

where

$$|\kappa_b| \le (n+2+\max\{2\max_{j\neq i}|\kappa_{\zeta_j}|,|\kappa_\alpha|\})\cdot K_b+2|\kappa_{\zeta_i}|+3.$$

Additionally, if (21) and (23) hold, then (22) and (24) hold, as well, and

$$|\kappa_b| \le (3n+6)\cdot K_b+2n+7.$$

Second, if

$$K_b \gg 1,$$

then, according to the theory from [11], the element b needs to be computed in double precision ε using $\hat{\zeta}_j^{(d)}$ and $\hat{\alpha}^{(d)}$ in order to obtain full possible accuracy. The standard first order error analysis of (9) in double precision ε gives

$$\hat{b}^{(d)} = fl\left(\frac{1}{\zeta_i^2(1+\kappa_{\zeta_i}^{(d)}\varepsilon^2)^2}\left(\alpha(1+\kappa_\alpha^{(d)}\varepsilon^2)-d_i + \sum_{\substack{j=1\\ j\neq i}}^{n-1}\frac{\zeta_j^2(1+\kappa_{\zeta_j}^{(d)}\varepsilon^2)^2}{d_j-d_i}\right)\right)$$
$$= b(1+\kappa_b^{(d)}\varepsilon^2),$$

where

$$|\kappa_b^{(d)}| \le (n+2+\max\{2\max_{j\neq i}|\kappa_{\zeta_j}^{(d)}|,|\kappa_\alpha^{(d)}|\})\cdot K_b+2|\kappa_{\zeta_i}^{(d)}|+3.$$

Finally, let

$$\kappa_{\hat{A}_i}^{(d)} = \max\{2\max_{j\neq i}|\kappa_{\zeta_j}^{(d)}|,|\kappa_\alpha^{(d)}|\}. \tag{26}$$

If, in addition to (21) and (23),

$$K_b \le \frac{1}{\varepsilon}, \tag{27}$$

and

$$\kappa_{\hat{A}_i}^{(d)}\cdot K_b \le \frac{1}{\varepsilon}, \tag{28}$$

then

$$\hat{b} = fl(\hat{b}^{(d)}) = b(1+\check{\kappa}_b\varepsilon_M),$$

where

$$|\check{\kappa}_b| \le n+5.$$

The above results are summarized in the following lemma:

Lemma 1. *Let (21) and (23) hold, and let K_b be defined by (25). For all non-zero elements of the matrix A_i^{-1} from (8) computed according to Algorithm 1 and Remark 1, except for the element $[A_i^{-1}]_{ii}$, we have*

$$[\hat{A}_i^{-1}]_{kl} = [A_i^{-1}]_{kl}(1+\kappa_{kl}\varepsilon_M), \quad |\kappa_{kl}| \leq (2n+7).$$

For the computed element $b = [A_i^{-1}]_{ii}$ we have the following: if $K_b \not\gg 1$, then

$$\hat{b} = b(1+\kappa_b\varepsilon_M), \quad |\kappa_b| \leq (3n+6)\cdot K_b + 2n + 7.$$

If $K_b \gg 1$ and if (27) and (28) hold, then

$$\hat{b} = b(1+\check{\kappa}_b\varepsilon_M), \quad |\check{\kappa}_b| \leq n+5.$$

The forward error of the computed roots is bounded as follows:

Theorem 2. *Let (21) and (23) hold, and let K_b be defined by (25). Let*

$$\hat{\lambda} = \lambda(1+\kappa_\lambda\varepsilon_M)$$

be the root of $u(x)$ computed according to Algorithm 1 and Remark 1. If $K_b \not\gg 1$, then

$$|\kappa_\lambda| \leq 3\sqrt{n}[(3n+6)\cdot K_b + 2n + 7] + 3.18n\left(\sqrt{n}+1\right) + 4,$$

and if $K_b \gg 1$ and (27) and (28) hold, then

$$|\kappa_\lambda| \leq (6n+21)\sqrt{n} + 3.18n\left(\sqrt{n}+1\right) + 4.$$

Proof. Using the same notation as in [11, §3], the first summand in the above bound for κ_λ follows from [11, Theorems 5 and 6], while the second summand is the error bound for bisection from [15, §3.1]. □

3.3 Implementation of increased precision

If the polynomial coefficients are stored as floating-point numbers in the standard precision ε_M to full accuracy, then double standard precision can be implemented as follows:

- in general, one can evaluate the respective formulas by using extended precision routines `add2`, `sub2`, `mul2`, `div2`, and `sqrt2` from [6] – this is $O(10)$ times slower. In these routines, double standard precision is simulated by keeping each number as a pair consisting of higher and lower part of mantissa. For example, let

  $$[z, zz] = add2(x, xx, y, yy)$$

  where all quantities are floating-point numbers with t binary-digits mantissa. Then

  $$|z+zz-[(x+xx)+(y+yy)]| \leq (|x+xx|+|y+yy|)2^{-2(t-1)}.$$

  If $xx = 0$ and $yy = 0$, then (exactly) $z+zz = x+y$. We see that this is nearly equivalent to using double standard precision (the precision is $\frac{1}{2}\varepsilon_M^2$ instead of ε_M^2). Julia implementation of these routines is part of the package `DoubleDouble.jl` [2].
- in Intel `ifort` FORTRAN compiler [10], convert all quantities from standard 64 bit `REAL(8)` to 128 bit `REAL(16)` and then evaluate the respective formulas – this is only 3 times slower,
- in Matlab, convert all quantities to variable precision command `sym` with parameter `'f'`, and then evaluate the respective formulas – this is 300 to 1000 times slower than standard precision.

The evaluation of the polynomial $u(x)$ can also be successfully performed by Compensated Horner's method from [8, Algorithm 4], where both quantities h and c from this algorithm must be preserved for subsequent computations by extended precision routines.

If polynomial coefficients cannot be stored as floating-point numbers in the standard precision ε_M to full accuracy, we have following options:

- in Matlab, convert all quantities to variable precision command `sym` with parameter `'f'`, and then evaluate the respective formulas,
- in Julia, we store the coefficients as 256 bit mantissa `BigFloat` numbers, with $\varepsilon_P = 2^{-255} \approx 1.727\cdot 10^{-77}$. The double precision ε_p is obtained by command `set_bigfloat_precision(512)`. This solution is slower than using extended precision routines, but timings compare favorably to the ones for MPSolve, see Example 3.

4 Numerical Examples

In this section we shall compare the following algorithms:

- *rootsah* - the Julia [12] implementation of Algorithm 1, publicly available in the Julia package `Arrowhead.jl` [2], with double standard precision implemented using Julia package `DoubleDouble.jl` [2], and the polynomial precision and double polynomial precision implemented as `BigFloat` numbers,
- *roots* - Matlab or Julia standard command which computes roots of polynomials as eigenvalues of the companion matrix. Both, Matlab and Julia versions give the same results.
- *Math* - Mathematica [20] *Roots* routine with 100 digits of precision rounded to 16 decimal digits.
- *mpsolve* -multiprecision polynomial root finder from the package MPSolve[3] [3,4].
- *NewtonCHS* - Newton method with Compensated Horner's method from [8, Algorithm 6].

We illustrate our algorithm with three examples. Some timings are given in Example 3.

[3] We have used version 3.1.4 with the following command line parameters `mpsolve -as -Ga -o15`.

*Example 1.*The coefficients $a_0,\ldots,a_{18}$ of Wilkinson's polynomial W_{18} are, row-wise,[4]

6402373705728000	−22376988058521600
34012249593822720	−30321254007719424
17950712280921504	−7551527592063024
2353125040549984	−557921681547048
102417740732658	−14710753408923
1661573386473	−147560703732
10246937272	−549789282
22323822	−662796
13566	−171
1	

Coefficients of W_{18} are stored to full accuracy in the standard precision ε_M, so here $\varepsilon=\varepsilon_M=\varepsilon_P$.

Finding values of d_j which interpolate roots is not an easy task.[5] In this example, the interpolating points d_j were computed as roots of $u'(x)$ with the `roots` command from Matlab or Julia. Those methods cannot be used directly to accurately compute the roots of W_{18}, but the computed interpolating points are sufficiently accurate.

We have

$$\max K_b = 214.5 \not\gg 1,$$
$$\max_j\{cond(u,d_j)\} = 7.29\cdot 10^{13},$$
$$K_\alpha = 35,$$

so by Theorem 2, the roots of W_{18} are computed by *rootsah* to (almost) full accuracy, in a forward stable manner.

The roots computed by *roots*, *rootsah*, *mpsolve* and *Math* are, respectively:

$\lambda^{(roots)}$	$\lambda^{(rootsah,mpsolve,Math)}$
18.00001193040660	18
16.99987506992020	17
16.00057853967064	16
14.99841877954789	15
14.00282666587300	14
12.99649084561071	13
12.00308090986650	12
10.99809154207482	11
10.00081885564820	10
8.999776556759201	9
8.000029075840132	8
7.000002735870642	7
5.999998227088450	6
5.000000283698958	5
3.999999981972712	4
3.000000000132610	3
2.000000000018936	2
0.999999999999808	1

[4] We use W_{18} since all its coefficients are exactly stored as 64-bit floating-point numbers.

[5] For example, the abstract of [16] states, "The efficiency of computing an initial approximation resists formal study, and the users rely on empirical data."

Since for every root, the corresponding quantity $K_b \not\gg 1$, the algorithm *rootsah* computes fully accurate roots, using the double standard precision (approximately equivalent to 32 decimal digits) to compute the entries of the matrix A and only the standard precision to compute the corresponding matrix $\hat{A}_i^{-1}$ and its absolutely largest eigenvalue.

The *mpsolve* performs computations using 231 decimal digits (and 462 decimal digits for one root) to guarantee and obtain 15 accurate digits. However, if the requirement for accurate digits is only slightly relaxed to requiring 14 accurate digits, *mpsolve* performs computations using only 19 decimal digits.

The *NewtonCHS* also computes the roots of W_{18} to full accuracy as described in [8, Theorem 6]. However, the starting points x_0 which satisfy the conditions of [8, Theorem 6], must be chosen with greater care and must be relatively close to the desired root (for example, $x_0=17.1$ to obtain $\lambda_2=17$, or $x_0=1.1$ to obtain $\lambda_{18}=1$. Since the Accurate Newton's method takes on average 6 steps to convergence for each root, it needs approximately $12n^2$ effective extended precision computations, while our algorithm needs in this case $5n^2$ extended precision computations to compute the matrix $\hat{A}$.

The results for W_{20} are similar.

*Example 2.*Consider the polynomial u of degree 5 with the coefficients

−6.189700196426900e+26
4.181389724724491e+42
−6.277101735386680e+57
7.136238463529799e+44
−2.028240960365167e+31
1.000000000000000e+00

or, in Julia's `BigInt` format,

−6189700196426900000010608640
418138972472449060109790789074129288324710 4
−627710173538668006693750196912569324311…
…1159424202737451008
7136238463529799405291429847247475681913733 12
−202824096036516704239472512860 16
1

The coefficients of $u(x)$ are stored to full accuracy in the standard precision ε_M, so here also $\varepsilon=\varepsilon_M=\varepsilon_P$. The interpolating points d_j were computed as the roots of $u'(x)$ using *roots* command additionally corrected by one step of the Weierstrass–Durand–Kerner method [1] . The values d_j and $cond(u,d_j)$ from (11) are given in Table 1. For the decreasingly ordered roots of u, λ_k, $k=1,2,3,4,5$, the corresponding quantities K_b from (25), $\kappa^{(d)}_{\hat{A}_i}$ from (26) and their respective products from (28), all rounded up, are given in Table 2. We see that the condition (27) is always fulfilled. Also, $K_\alpha=1$ from (18), so (23) is fulfilled. The condition (28) does not hold

Table 1: Interpolating points d_j and $cond(u,d_j)$.

j	d_j	$cond(u,d_j)$
1	5.277655813324802e+13	4
2	1.759218604441599e+13	$3.58\cdot 10^{16}$
3	6.253878705847983e-16	12.4
4	2.627905491153268e-16	46.4

Table 2: Values K_b, $\kappa^{(d)}_{\hat{A}_i}$ and $\kappa^{(d)}_{\hat{A}_i}\cdot K_b$.

k	K_b	$\kappa^{(d)}_{\hat{A}_i}$	$\kappa^{(d)}_{\hat{A}_i}\cdot K_b$
1	1	$3.6\cdot 10^{17}$	$3.6\cdot 10^{17}$
2	$3.01\cdot 10^{15}$	$4.7\cdot 10^{2}$	$1.42\cdot 10^{18}$
3	$3.01\cdot 10^{15}$	$4.7\cdot 10^{2}$	$1.42\cdot 10^{18}$
4	12.6	$3.58\cdot 10^{17}$	$4.48\cdot 10^{18}$
5	12.6	$3.58\cdot 10^{17}$	$4.48\cdot 10^{18}$.

literally. However, we have $\kappa^{(d)}_{\hat{A}_i}\cdot K_b \not\gg \frac{1}{\varepsilon_M}$, which is sufficient to obtain almost full accuracy.

The roots computed by *roots*, *rootsah*, *mpsolve* and *Math* are, respectively:

$\lambda^{(roots)}$	$\lambda^{(rootsah,mpsolve,Math)}$
$2.028240960365167e+31$	$2.028240960365167e+31$
$1.759218622977980e+13$	$1.759218623050247e+13$
$1.759218585905220e+13$	$1.759218585832953e+13$
0	$4.440892098500623e-16$
0	$2.220446049250314e-16$

We see that the roots computed *rootsah*, *mpsolve* and *Math* fully coincide. In *rootsah*, in addition to the elements $\hat{z}^{(d)}$ and $\hat{\alpha}^{(d)}$ of the matrix $\hat{A}$, the element b of $\hat{A}_2^{-1}$ was computed in double standard precision. Again, *mpsolve* uses 231 decimal digits to guarantee 15 accurate digits, and uses 19 decimal digits to guarantee and obtain relative accuracy of 10^{-14}. The *NewtonCHS* also computes the roots to full accuracy, provided the respective starting points are chosen with greater care. However, the conditions of [8, Theorem 6] cannot be used - for example, for the largest root λ_1, there is no starting point x_0 which satisfies the conditions, except λ_1 itself. For λ_2, the starting point x_0 which satisfies the conditions can differ from λ_2 in just last digit.

*Example 3.*In this example we consider Chebishev polynomials, $T_n(x)$, and Legendre polynomials, $L_n(x)$, defined by the three-term recurrences

$$T_0(x)=1,\quad T_1(x)=x,$$
$$T_{n+1}(x)=2xT_n(x)-T_{n-1}(x),\quad n\geq 1,$$

and

$$L_0(x)=1,\quad L_1(x)=x,$$
$$L_{n+1}(x)=\frac{2n+1}{n+1}xL_n(x)-\frac{n}{n+1}L_{n-1}(x),\quad n\geq 1,$$

Table 3: Execution time (in seconds), maximal relative error and maximal precision used for *rootsah* and *mpsolve*.

Polynomial	Method	Time	Rel. error	Precision
$T_{100}(x)$	*rootsah*	0.0182	$\approx 10^{-16}$	154
	mpsolve	0.184	$\approx 10^{-15}$	462
$T_{375}(x)$	*rootsah*	0.27	$\approx 10^{-16}$	154
	mpsolve	1.08	$\approx 10^{-14}$	924
$L_{160}(x)$	*rootsah*	0.07	$\approx 10^{-16}$	154
	mpsolve	0.17	$\approx 10^{-14}$	924
$L_{320}(x)$	*rootsah*	0.23	$\approx 10^{-16}$	154
	mpsolve	0.74	$\approx 10^{-14}$	924

respectively.

The roots of $T_n(x)$ for $n\geq 1$ are

$$x_k=\cos\left(\frac{2k-1}{n}\frac{\pi}{2}\right),\quad k=1,\ldots,n. \tag{29}$$

For $n\leq 52$, the coefficients of $T_n(x)$ are accurately stored in the standard precision ε_M. The coefficients of $T_n(x)$ for $53\leq n\leq 375$ can be accurately stored in the polynomial precision $\varepsilon_P=1.727\cdot 10^{-77}$ (for example, using Julia's `BigFloat` numbers).

The roots of $L_n(x)$ for $n\geq 1$ are all real and simple and lie in the interval $(-1,1)$. For $n\leq 28$, the coefficients of $L_n(x)$ are accurately stored in the standard precision ε_M. The coefficients of $L_n(x)$ for $29\leq n\leq 320$ can be accurately stored in the polynomial precision $\varepsilon_P=1.727\cdot 10^{-77}$.

The interpolating points d_j were obtained using the fact the roots of $T_n(x)$ interlace the roots of $T_{n+1}(x)$ and the roots of $L_n(x)$ interlace the roots of $L_{n+1}(x)$ for $n\geq 1$.

Comparison between *rootsah* and *mpsolve* in terms of speed[6], relative error in the computed roots and the used maximal precision (number of decimal digits) is given in Table 3. For Chebishev polynomials, relative errors were computed comparing the computed roots with (29). Also, we have $\max_i cond(T_{375},d_i)\approx 10^{143}$ and $K_b\not\gg 1$ for all roots, so, by Theorem 2, *rootsah* computed all roots to almost full standard precision accuracy. For Legendre polynomials, we have $\max_i(cond(L_{160},d_i)\approx 10^{62}$, $\max_i(cond(L_{320},d_i)\approx 10^{123}$ and $K_b\not\gg 1$ for all roots, so, by Theorem 2 *rootsah* computed all roots to almost full standard precision accuracy. The relative error for *mpsolve* is the one given by the program itself.

To conclude, we see that *rootsah* computes the roots with the same (or slightly better) accuracy, and is several times faster and uses several times less decimal digits in the computation than *mpsolve*.

[6] Test were performed on the computer with `Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz` processor with four cores running Linux operating system.

Acknowledgment

This work was supported by the Croatian Science Foundation, OptPDMechSys, IP-2014-09-9540.
The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

[1] O. Aberth, Mathematics of Comput. **27**, 122, 339344 (1973).
[2] Arrowhead.jl, https://github.com/ivanslapnicar/Arrowhead.jl.
[3] D. A. Bini and G. Fiorentino, Numerical Alg. **23** 127-173 (2000).
[4] D.A. Bini and L. Robol, J. Comp. Appl. Math. **272** 276-292 (2014).
[5] R. M. Corless, G. Litt, Generalized companion matrices for polynomials not expressed in monomial bases, unpublished.
[6] T. J. Dekker, Numer. Math., **18** 224-242 (1971).
[7] M. Fiedler, Lin. Alg. Appl., **141** 265-270 (1990).
[8] S. Graillat, Comput. Math. Appl., **56** 1114-1120 (2008).
[9] N. Higham, Accuracy and Stability of Numerical Algorithms, Second Edition, SIAM, Philadelphia, 2002.
[10] Intel Fortran Compiler, http://software.intel.com/en-us/fortran-compilers
[11] N. Jakovčević Stor, I. Slapničar and J. L. Barlow, Lin. Alg. Appl., **464** 62-89 (2015).
[12] The Julia Language, http://julialang.org/
[13] G. Malajovich and J. P. Zubelli, Numer. Math., **89** 749-782 (2001).
[14] MATLAB. The MathWorks, Inc., Natick, Massachusetts, USA, http://www.mathworks.com.
[15] D. P. O'Leary and G.W. Stewart, J. Comput. Phys. **90**, 2, 497-505 (1990).
[16] V. Y. Pan, Root-refining for a polynomial equation, In: V. P. Gerdt, W. Koepf, E. W. Mayr, and E. V. Vorozhtsov (Eds.), Computer Algebra in Scientific Computing, Heidelberg, Springer, 2012.
[17] F. Tisseur, SIAM J. Matrix Anal. Appl., **22** 1038-1057 (2001).
[18] J. H. Wilkinson, The Algebraic Eigenvalue Problem, Clarendon Press, Oxford, (1965).
[19] J. H. Wilkinson, The perfidious polynomial, In: Gene H. Golub (Ed.), Studies in Numerical Analysis, Mathematical Association of America, 24, 1984.
[20] Wolfram Mathematica, Documentation Center, http://reference.wolfram.com/mathematica/guide/Mathematica.html

Nevena Jakovčević Stor received the PhD degree in Mathematics from the University of Zagreb. Her research interests are in the areas of numerical linear algebra and applied mathematics including numerical methods for eigenvalue and singular value problems. She has published research articles in reputed international journals of mathematical sciences.

Ivan Slapničar is Professor of Mathematics University of Split. He was Visiting professor at the Utah State University in Logan (USA), FP7 Marie Curie IEF Fellow at the Technical University of Berlin (Germany) and the Fulbright-Schuman International Educator/Lecturer at the Massachusetts Institute of Technology (USA). He received PhD degree in Mathematics from Fernuniversität Hagen, Germany. He is referee of several international journals in the frame of pure and applied mathematics. His main research interests are: linear algebra, numerical analysis, numerical linear algebra and applications.