2002

TR-2002020: Inverse Power and Durand-Kerner Iterations for Univariate Polynomial Root-Finding

D. A. Bini

L. Gemignani

V. Y. Pan

Inverse Power and Durand-Kerner Iterations for Univariate Polynomial Root-Finding

D. A. Bini *
Dipartimento di Matematica, Università di Pisa
Via Buonarroti 2, 56127 Pisa, Italy
bini@dm.unipi.it

L.Gemignani †
Dipartimento di Matematica, Università di Pisa
Via Buonarroti 2, 56127 Pisa, Italy
gemignan@dm.unipi.it

V. Y. Pan ‡
Department of Mathematics and Computer Science
Lehman College of CUNY, Bronx, NY 10468, USA
vpan@lehman.cuny.edu

December 4, 2002

Abstract

Univariate polynomial root-finding is an oldest classical problem of mathematics and computational mathematics, which is still an important research topic, due to its impact on computational algebra and geometry. The Weierstrass (Durand-Kerner) approach and its variations as well as matrix methods based on the QR algorithm are among the most popular practical choices for simultaneous approximation of all roots of a polynomial. We propose an alternative application of the inverse power iteration to the class of generalized companion matrices for polynomial root-finding, demonstrate its effectiveness, and relate its study to unifying the derivation of the Weierstrass (Durand-Kerner) algorithm (having quadratic convergence) and its extensions having convergence rates

*Supported by GNCS and by MIUR
†Supported by GNCS and by MIUR
‡Supported by NSF Grant CCR 9732206 and PSC CUNY Awards 66383-0032 and 64406-0033

$4, 6, 8, \ldots$. Our experiments show substantial improvement versus the latter algorithm, even though the inverse power iteration is most effective for the more limited tasks of approximating a single root or a few selected roots.

Key words: polynomial root-finding, Weierstrass (Durand-Kerner) algorithm, higher order root-finders, generalized companion matrices, matrix methods for root-finding, inverse power iteration.

2000 Math. Subject Classification: 65H05, 30C15, 65B99, 65E05

1 Introduction

Univariate polynomial root-finding is the four millenia old problem of mathematics and computational mathematics whose study throughout many centuries had been the central subject in these fields and has made greatest impact on them (see [McN93], [McN97], [P97] for the bibliography and a survey). The problem still remains an important research topic, particularly due to its applications to algebraic and geometric computations. From the asymptotic computational complexity point of view, the problem was solved in [P95],[P01],[P02], where optimal (up to polylog factors) computational time bounds were reached under both arithmetic and Boolean and both sequential and parallel computational models. The algorithms in [P95],[P01],[P02], however, are quite involved, which complicates their numerical analysis and practical implementation. The users presently prefer some easily implementable iterative algorithms. As a rule, rapid convergence of these algorithms is proved only locally, that is, where the roots are isolated from each other and are already approximated closely, but under the customary choices of initial approximations, rapid global convergence has been confirmed by statistics of extensive application of the algorithms in computational practice.

For simultaneous approximation of all n roots of a given polynomial, most popular practical choices are the Weierstrass (Durand-Kerner) iteration (hereafter we refer to it as the *W(D-K) iteration*) and its variations such as Aberth's (Ehrlich / Börsch-Supan's), Farmer-Loizou's and Werner's (see [PHI98] on derivation of such algorithms). Their convergence rate is quite high (it varies from 2 to 4), but the matrix methods based on the QR algorithms are still considered competitive. The latter methods (see [F90], [TT94], [EM95], [MV95a], [MV95b], [F01], and the bibliography therein) use the order of n^3 flops for all zeros (versus $O(n^2)$ per iteration in the W(D-K) iteration and its modifications) performed numerically with single precision.

In the present paper, our first innovation (in Section 2 and Corollary 4.6) is a simple unified derivation of the W(D-K) iteration and its extensions having convergence rates $4, 6, 8, \ldots$. This is a purely technical innovation, not leading to any acceleration of the W(D-K) iteration, but showing once again the relevance of matrix methods. We briefly recall some effective policies for choosing

initial approximations to the roots in Section 3. Then in Sections 4–8, we revisit the inverse power iteration and apply it to the general class of generalized companion matrices associated with the input polynomial. This class includes Frobenius-plus-diagonal matrices as well as rank-one-plus-diagonal matrices and was much studied in the literature (see [B81], [C91], [PHI98] and the bibliography therein), but never with the use of the inverse power iteration. The latter quite natural but yet unexplored approach, which can be immediately applied to approximating a single zero, uses $O(n)$ flops per iteration step and exhibits fast convergence in our tests. We supply some details of application of the method to the generalized companion matrices; in particular, we show explicit expressions for the eigenvectors of a generalized companion matrix in terms of its eigenvalues and their approximations. The derivation of the expressions turns out to be closely related to our derivation of the W(D-K) iteration and its cited extensions to iterations with higher convergence rates.

The main part of our work (Sections 4–8) is the elaboration upon the inverse power iteration for the diagonal-plus-rank-one matrices including the deflation and extension of the approximation from a single root to several or all roots. In Section 8 we report the results of our numerical experiments which clearly demonstrate significant improvement over the W(D-K) iteration, even for computing all roots, which seems to be the hardest task for the inverse power versus W(D-K) iteration. In Section 9 we summarize our study and comment on its further directions, in particular on associating Frobenius or Frobenius-plus-diagonal matrices with the given polynomial.

Our analysis of the inverse power iteration can be partly extended to the eigencomputation for an important class of semisimple + diagonal matrices.

Acknowledgments: Victor Pan thanks S. Fortune, B. Mourrain, and E.E. Tyrtyshnikov for preliminary discussions on matrix methods for polynomial root-finding and X. Wang for providing help to simplify the original proof of equation (2.6).

2 The Weierstrass (Durand-Kerner) iteration and its higher order extensions

Given n distinct values $s_1, \dots, s_n$ approximating the unknown distinct roots (zeros) $z_1, \dots, z_n$ of a polynomial

$$p(x) = \prod_{i=1}^{n} (x - z_i), \tag{2.1}$$

the W(D-K) iteration consists in recursive computation of improved approximations

$$t_i = s_i - d_i, \quad i = 1, \dots, n, \tag{2.2}$$

where

$$q(x) = \prod_{i=1}^{n}(x - s_i), \quad q_i(s_i) = q'(s_i), \quad i = 1, \ldots, n, \tag{2.3}$$

$$d_i = p(s_i)/q_i(s_i), \quad q_i(x) = q(x)/(x - s_i) = \prod_{j \neq i}(x - s_j). \tag{2.4}$$

The W(D-K) iteration has a local quadratic convergence and only requires computation of the values $p(s_i)$ and $q'(s_i)$, $i = 1, \ldots, n$. Hereafter we refer to d_i's as to the W(D-K) corrections.

The iteration has been derived by applying Newton's method to the Viéte system of polynomial equations, relating the coefficients of $p(x)$ to the symmetric functions in its zeros $z_1, \ldots, z_n$. Moreover, the iteration can be written as

$$z_i^{new} = z_i^{old} - p(z_i^{old})/q'(z_i^{old}), i = 1, \ldots, n,$$

which is Newton's iteration

$$z_i^{new} = z_i^{old} - p(z_i^{old})/p'(z_i^{old}), i = 1, \ldots, n,$$

with $p'(x)$ replaced by its approximation $q'(x)$. Our next alternative derivation via the Lagrange interpolation formula,

$$p(x) = q(x) + \sum_{i=1}^{n} d_i q_i(x), \tag{2.5}$$

produces also iterations with the convergence rates $4, 6, 8, \ldots$. We present this unified derivation because it is technically simple and provides new insights, even though it leads to no acceleration of the original W(D-K) algorithm (see table 1 and the end of this section).

Theorem 2.1. *Let $z_i \neq s_j$ for all $j \neq i$. Then we have*

$$s_i - z_i = d_i/(1 + \sum_{j \neq i} d_j/(z_i - s_j)), i = 1, \ldots, n. \tag{2.6}$$

Proof. Substitute $x = z_i$ into (2.5) and obtain that $q(z_i) + \sum_{j=1}^{n} d_j q_j(z_i) = 0$. If $z_i = s_i$, then $d_i = 0$, and (2.6) trivially holds. Otherwise, divide by $q(z_i)$, substitute (2.3) and (2.4), and obtain that

$$1 + \sum_{j=1}^{n} d_j/(z_i - s_j) = 0, \quad d_i/(s_i - z_i) = 1 + \sum_{j \neq i} d_j/(z_i - s_j).$$

Multiply both sides by $(s_i - z_i)/(1 + \sum_{j \neq i} d_j/(z_i - s_j))$ to obtain (2.6). □

Let us write

$$\Delta = \max_i \{|d_i| + |z_i - s_i|\}. \tag{2.7}$$

We immediately deduce from (2.6) that

$$\begin{aligned} z_i &= t_i + O(\Delta^2), \\ t_i &= s_i - d_i, \end{aligned} \qquad i = 1, \ldots, n \tag{2.8}$$

which shows quadratic local convergence of the W(D-K) iteration (2.2).

Substitute (2.8) on the right hand side of (2.6) and obtain a known iteration of the fourth order (cf. [PHI98]):

$$\begin{aligned} z_i &= t_i + O(\Delta^4), \\ t_i &= s_i - d_i/(1+u_i), \qquad i = 1, \ldots, n \\ u_i &= \textstyle\sum_{j \neq i} d_j/(s_i - d_i - s_j), \end{aligned} \tag{2.9}$$

Similarly substitute (2.9) into (2.6) and obtain an iteration of the sixth order:

$$\begin{aligned} z_i &= t_i + O(\Delta^6), \\ t_i &= s_i - d_i/(1+v_i), \qquad i = 1, \ldots, n. \\ v_i &= \textstyle\sum_{j \neq i} d_j/(s_i - d_i/(1+u_i) - s_j), \end{aligned} \tag{2.10}$$

Continue this pattern, substitute (2.10) into (2.6), obtain an iteration of the 8th order:

$$\begin{aligned} z_i &= t_i + O(\Delta^8), \\ t_i &= s_i - d_i/(1+w_i), \qquad i = 1, \ldots, n, \\ w_i &= \textstyle\sum_{j \neq i} d_j/(s_i - d_i/(1+v_i) - s_j), \end{aligned} \tag{2.11}$$

and so on. In this process, each increase of the convergence rate by two requires additional computation of the values

$$y_i = 1 + \sum_{j \neq i} d_j/(h_i - s_j), i = 1, \ldots, n, \tag{2.12}$$

where h_i are readily computable.

Observe that the cost of the W(D-K) iteration is just $4n^2 + O(n)$ arithmetic operations (ops), whereas the cost of each single higher order extension is $3n^2$ ops. If we assume that all arithmetic operations have the same cost, then the arithmetic cost of the k-th higher order extension is $c(k) = (4 + 3k)n^2$ ops, $k = 0, 1, \ldots$. We may evaluate the performance of each iteration in terms of the weighted cost $c(k)\log_2(2k+2)$, where $(2k+2)$ is the order of convergence. This expression represents the ratio of the estimates for the arithmetic cost of reaching a bound ϵ on the approximation error by the selected method, and by an ideal quadratically convergent method having unit cost per iteration where $\epsilon \to 0$. Table 1 shows that if we assume that all arithmetic operations have

Method	DK2	DK4	DK6	DK8
weighted cost over $\mathbb{R}$	$4n^2$	$3.5n^2$	$3.8n^2$	$4.3n^2$
weighted cost over $\mathbb{C}$	$12n^2$	$13.5n^2$	$16.2n^2$	$19n^2$

Table 1: Weighted cost of the higher order extensions.

the same cost, then acceleration of the 4-th order is the best choice. On the other hand if we work with complex numbers, each complex addition costs two real ops, each complex multiplication costs 6 ops and complex division 11 ops. Therefore the approximation of the complex roots of a polynomial with real coefficients costs $12n^2 + O(n)$ ops for each step of the W(D-K) iteration, and the additional cost of $15n^2$ ops for each single higher order extension. In this case we deduce that the most efficient method still remains the W(D-K) iteration.

3 The choice of initial approximations

It is well-known from extensive numerical tests that as a rule the W(D-K) algorithm as well as its various extensions converge rapidly starting with a quite random set of initial approximations $s_1, \ldots, s_n$. A customary choice is $s_i = a\omega^{i-1}$, $i = 1, \ldots, n$, where $\omega = \exp(2\pi\sqrt{-1}/n)$ is a primitive n^{th} root of 1, $a/\max_i |z_i|$ is set to, say 1.5 or 2, and $|z_i|$ are the unknown zeros of $p(x)$. C. Carstensen in [C91a] proposes to choose $s_1, \ldots, s_n$ by using Gershgorin's discs, D. Bini [B96] proposes to choose the initial approximations along different circles whose radii are computed by means of Rouché theorem and Newton's polygon.
S.Fortune [F01] applies the QR algorithm to the Frobenius matrix $F(\mathbf{p})$ and uses the computed approximations to the eigenvalues as the initial approximations s_i. In spite of the order of n^3 flops involved, this single precision computation is quite fast, according to S.Fortune. Finally, a slower but reliable customary option is the continuation (or homotopy) approach, where one starts with a polynomial $p_{\tau_0}(x) = q(x) = \prod_j (x - s_j(\tau_0))$ with some fixed zeros $s_1(\tau_0), \ldots, s_n(\tau_0)$ and then recursively computes the zeros $s_j(\tau_i)$ for a sequence of polynomials $p_{\tau_i}(x) = \tau_i p(x) + (1 - \tau_i)q(x)$, $i = 0, 1, \ldots, K$, $\tau_0 < \tau_1 < \ldots < \tau_K = 1$ using the values $s_j = s_j(\tau_i)$ as the initial approximations to $t_j = s_j(\tau_{i+1})$, $j = 1, \ldots, n$. We refer the reader to [BF00], [KS94], [PHI98], [HSS01], and [BPa] on these and some other choices.

4 Generalized companion matrices and their eigenvectors

Definition 4.1. An $n \times n$ matrix $C = C(p(x))$ is a *generalized companion matrix* for a polynomial $p(x)$ in (2.1) if $\{z_1, \ldots, z_n\}$ is the set of the eigenvalues of C.

Thus root-finding for $p(x)$ amounts to the eigenvalue problem for C, where matrix methods can be applied. Next, we recall two most important classes of generalized companion matrices C and express their eigenvectors via $z_1, \dots, z_n$. In the next section we examine application of the inverse power method to these matrices, which can be viewed as an alternative or as a complement to the algorithms of Section 2.

Example 4.2. The Frobenius (companion) matrix $C = F(p(x))$ is given by

$$F(p(x)) = \begin{pmatrix} 0 & 1 & & \\ & & \ddots & \\ & & & 1 \\ -p_0 & -p_1 & \cdots & -p_{n-1} \end{pmatrix}$$

provided that $p(x) = x^n + \sum_{i=0}^{n-1} p_i x^i$. It has the eigenpairs $(z_i, \mathbf{v}_i)$, where

$$\mathbf{v}_i = (z_i^j)_{j=0}^{n-1}, \quad i = 1, \dots, n. \tag{4.1}$$

Here is another important subclass of generalized companion matrices.

Definition 4.3. (Cf. [E73],[C91],[C92].) For a polynomial $p(x)$ of (2.1) and n distinct values $s_1, \dots, s_n$, define a rank-one matrix $E_{\mathbf{d}}$ with diagonal entries $d_1, \dots, d_n$ of (2.4) and an associated $n \times n$ generalized companion matrix

$$C = C_{\mathbf{s},\mathbf{d}} = D_{\mathbf{s}} - E_{\mathbf{d}}. \tag{4.2}$$

where $D_{\mathbf{s}}$ is a diagonal matrix with diagonal entries $s_1, s_2, \dots, s_n$.

Definition 4.3 leaves us with some freedom in choosing the matrix $E_{\mathbf{d}}$. In particular, Fiedler in [F90] proposes $E_{\mathbf{d}} = \sigma \mathbf{f}\,\mathbf{f}^T$, $\mathbf{f} = (f_i)_{i=1}^n$, $\sigma f_i^2 = d_i$, $i = 1, \dots, n$, for a fixed scalar σ, whereas Elsner in [E73] proposes

$$E_{\mathbf{d}} = \mathbf{1}\mathbf{d}^T, \tag{4.3}$$

where $\mathbf{1} = (1, 1, \dots, 1)^T$.

The next well known result [C91] easily follows from the Lagrange interpolation formula (2.5).

Theorem 4.4. *For any pair of matrices C and $E_{\mathbf{d}}$ of Definition 4.3, we have*

$$\det(xI - C) = p(x).$$

Our next goal is the expressions for the eigenvectors of the matrix C in (4.2), (4.3) via the eigenvalues $z_1, \dots, z_n$.

Theorem 4.5. *Let C be the matrix in (4.2), (4.3) where s_i are pairwise distinct, $\mathbf{d} = (d_i)$, $d_i = p(s_i)/\prod_{j \neq i}(s_i - s_j)$, $i = 1, \dots, k \leq n$, and $p(x)$ is the polynomial (2.1) having pairwise distinct zeros $z_1, \dots, z_n$. Then, if $s_i \neq z_j$, for $i, j = 1, \dots, n$ for the right and left eigenvectors $\mathbf{u}_j$ and $\mathbf{v}_j$ of C, respectively, we have*

$$\begin{array}{ll} C\mathbf{v}_j = z_j \mathbf{v}_j & \mathbf{v}_j = (1/(s_i - z_j)) \\ \mathbf{u}_j^T C = z_j \mathbf{u}_j^T & \mathbf{u}_j = (d_i/(s_i - z_j)) \end{array} \qquad j = 1, \dots, k. \tag{4.4}$$

Proof. From $C\mathbf{v}_j = z_j\mathbf{v}_j$ we obtain $\mathbf{v}_j = (D - z_jI)^{-1}\mathbf{1}\mathbf{d}^T\mathbf{v}_j$. Whence, up to the normalization factor $\mathbf{d}^T\mathbf{v}_j$ we have $\mathbf{v}_j = (D - z_jI)^{-1}\mathbf{1} = (1/(s_i - z_j))$. Similarly, from $\mathbf{u}_j^T C = \mathbf{u}_j^T z_j$ we obtain $\mathbf{u}_j = (d_i/(s_i - z_j))$. □

Corollary 4.6. *Equation (2.6) holds.*

Proof. Equate the i-th coordinates on both sides of the vector equation $C\mathbf{v}_j = z_j\mathbf{v}_j$, substitute $\mathbf{v}_j$ from (4.4) and obtain (2.6). □

We have the following simple observation.

Theorem 4.7. *Given a scalar z such that $p(z) \neq 0$, a vector $\mathbf{x}$, and a Frobenius matrix $C = F(p(x))$ or a generalized companion matrix C of Definition 4.3, the vectors $C\mathbf{x}$ and $(C - zI)^{-1}\mathbf{x}$ can be computed by using $O(n)$ flops.*

Hereafter write $D = D_{\mathbf{s}}, E = E_{\mathbf{d}}$. Let us apply Sherman-Morrison-Woodbury formula ([GL96], page 50) to invert the matrix $C = D_s - \mathbf{1}\mathbf{d}^T$ of (4.2) and (4.3), that is,

$$(C - zI)^{-1} = (I + \tfrac{1}{1-\tau}(D - zI)^{-1}\mathbf{1}\mathbf{d}^T)(D - zI)^{-1}$$

$$\tau = \mathbf{d}^T D^{-1}\mathbf{1}.$$

Thus we have

$$(C - zI)^{-1}\mathbf{v} = (D - zI)^{-1}\mathbf{v} + \tfrac{\sigma}{1-\tau}(D - zI)^{-1}\mathbf{1},$$

$$\sigma = \mathbf{d}^T(D - zI)^{-1}\mathbf{1}, \quad \tau = \mathbf{d}^T(D - zI)^{-1}\mathbf{1}.$$

Therefore, we can compute $\mathbf{y} = (C - zI)^{-1}\mathbf{x}$ by performing n reciprocations, $4n$ multiplications and $4n$ additions.

Algorithm 4.8. [Shifted inverse power iteration for a generalized companion matrix]

STEP 1: compute $\mathbf{g} = (D - zI)^{-1}\mathbf{1}$;

STEP 2: compute $\mathbf{u} = \mathbf{g} * \mathbf{x}$ where $*$ denotes component-wise product of vectors;

STEP 3: compute $\tau = \sum_{i=1}^{n} d_i g_i$;

STEP 4: compute $\sigma = \sum_{i=1}^{n} d_i g_i$;

STEP 5: compute $\mathbf{y} = \mathbf{u} + \frac{\sigma}{1-\tau}\mathbf{g}$.

For complex data the overall cost amounts to $37n + O(1)$ arithmetic operations between real numbers.

5 The inverse power method for a single eigenvalue of a generalized companion matrix

Under the assumptions of Theorem 4.5, let z be a sufficiently close approximation to an eigenvalue z_j of a matrix C and let $\mathbf{v} = \sum_{i=1}^{n} a_i \mathbf{v}_i$, $\|\mathbf{v}\|_2 = 1$, where $\mathbf{v}_j$, $j = 1, \ldots, n$ are the eigenvectors of C and $a_i \neq 0$. Then the shifted inverse power iteration is defined as follows:

$$\mathbf{x}^{(0)} = \mathbf{v}, \quad \mathbf{y}^{(k)} = (C - zI)^{-1}\mathbf{x}^{(k-1)}, \quad \mathbf{x}^{(k)} = \mathbf{y}^{(k)}/\|\mathbf{y}^{(k)}\|_2, \tag{5.1}$$

$$z^{(k)} = \mathbf{x}^{(k)T} C \mathbf{x}^{(k)}, \quad k = 1, 2, \ldots. \tag{5.2}$$

The pair $(\mathbf{y}^{(k)}, z^{(k)})$ rapidly converges to an eigenvector/eigenvalue pair $(\mathbf{v}_j, z_j)$ (see [GL96, Section 7.6]) provided that for all $i \neq j$ the ratios $|z - z_j| \Big/ |z - z_i|$ are substantially less than 1 and the ratios $|a_j/a_i|$ are not close to 0. By Theorem 4.7, each iteration step uses $O(n)$ flops. Initial approximations z can be computed as in Section 3.

Practical statistics shows that random initial eigenvector is usually a good choice for $\mathbf{v}$ (see [GL96]). Having some initial information about the eigenvalues, we may further improve this choice based on (4.1) or (4.4). If s_i is close to z_i but not equal to z_i, then (4.4) suggests the choice of the i-th coordinate vector as $\mathbf{v}$. For $C = F(p(x))$, one may consider the choice of $\mathbf{v} = (\sum_{i=1}^{n} z_i^k)_{k=0}^{n-1}$, where the power sums of all the zeros z_i can be computed in $O(n \log n)$ flops. Consider also shifting to $\mathbf{v}$ for the reverse $p(x)$. If a crude approximation to a zero z of $p(x)$ is available, then (4.1) suggests $(\bar{z}^j)_j$ as $\mathbf{v}$.

As soon as a zero z_i of $p(x)$ is closely approximated, one may reapply the algorithm to the deflated polynomial $p(x)/(x - x_i)$. The deflation for $C = F(p(x))$ only requires $2n - 2$ multiplications and $n - 1$ subtractions.

Finally, Theorem 4.7 can be easily extended to the important class of semiseparable + diagonal matrices [EG97], [MCVH01]; consequently, the same inverse power algorithm can be extended to the eigencomputation for these matrices.

Remark 5.1. The convergence of the shifted inverse power algorithm can be accelerated if the shift value z is replaced at each step by the current approximation $z^{(k)}$ of the eigenvalue. In this way the local convergence to simple eigenvalues becomes superlinear.

Remark 5.2. In the shifted power method we may replace equation (5.2) with the following less expensive formula for the eigenvalues approximation:

$$z^{(k)} = (Cy^{(k)})_i / y_i^{(k)}$$

where the subscript i is such that $y_i \neq 0$. For the matrix $C = D - E$ of (4.2) and (4.3) the latter formula turns into

$$z^{(k)} = s_i - (\sum_{j=1}^{n} d_j y_j^{(k)})/y_i^{(k)}. \tag{5.3}$$

6 Deflation of a generalized companion matrix and approximating all its eigenvalues

Now, we consider a more specific implementation of the shifted inverse power method for the matrix $C = D - E$ of (4.2) and (4.3). Observe that for the matrix $C = D - E$ the deflation of each approximated eigenvalue can be performed automatically. Let $s_1, \ldots, s_n$ be the initial approximations to the eigenvalues and

$$d_i = \frac{p(s_i)}{\prod_{j \neq i}(s_i - s_j)}$$

be the W(D-K) corrections. Let z denote the computed eigenvalue of C, and without loss of generality assume that s_n is the initial approximation closest to z (recall that we can sort the initial approximations in any order).

Remark 6.1. Let $\hat{\mathbf{s}} = (\hat{s}_i) = (s_1, \ldots, s_{n-1}, z)^T$ and $\hat{\mathbf{d}} = (\hat{d}_i)$ be the vector of the W(D-K) corrections corresponding to $\hat{\mathbf{s}}$ such that

$$\hat{d}_i = \frac{p(\hat{s}_i)}{\prod_{j \neq i}(\hat{s}_i - \hat{s}_j)}. \tag{6.1}$$

Since $p(z) = 0$, then the last row of $C_{\hat{\mathbf{s}},\hat{\mathbf{d}}}$ is $(0, \ldots, 0, z)$. Therefore the $(n-1) \times (n-1)$ leading principal submatrix of $C_{\hat{\mathbf{s}},\hat{\mathbf{d}}}$ coincides with the generalized companion matrix associated with the deflated polynomial $p(x)/(x - z)$ and with the vector $(s_1, \ldots, s_{n-1})$. This matrix is defined by the $(n-1)$-dimensional vectors $(s_1, \ldots, s_{n-1})^T$, and by the W(D-K) corrections $(\hat{d}_1, \ldots, \hat{d}_{n-1})^T$. The components $s_1, \ldots, s_{n-1}$ are available whereas the values $\hat{d}_1, \ldots, \hat{d}_{n-1}$ must be computed. From (6.1) we immediately deduce that

$$\hat{d}_i = d_i \frac{s_i - s_n}{s_i - z}, \quad i = 1, \ldots, n-1, \tag{6.2}$$

which provides a simple means for performing deflation with $2(n-1)$ additions, $n-1$ divisions and $n-1$ multiplications.

According to Remark 6.1 we may approximate the eigenvalues of $C_{\mathbf{s},\mathbf{d}}$ by applying the shifted inverse power method to a sequence of $k \times k$ matrices $C^{(k)}$ for $k = n, n-1, \ldots, 2$, where $C^{(k)}$ is the matrix obtained by updating $C^{(k+1)}$ according to (6.1) once the inverse power method, applied to $C^{(k+1)}$ has arrived at convergence. The eigenvalue of the matrix $C^{(1)}$ is just $s_1 - d_1$. The resulting algorithm, with the dynamic choice of the shift value according to Remark 5.1, is described below:

Remark 6.2. It should be noted that such an application can be a strategy of an adversary, because the time cost increases roughly proportionally to the number of computed roots, that is, the best way to apply this method is to approximating a single root or a few roots. It is interesting that even playing by the adversary rules, we achieve substantial progress versus the W(D-K) iteration.

Algorithm 6.3. [Computing all eigenvalues of a generalized companion matrix]

INPUT: An integer n and the vectors $\mathbf{s} = (s_1, \ldots, s_n)^T$, $\mathbf{d} = (d_1, \ldots, d_n)^T$; the maximum number of iterations N; an error bound ϵ.

OUTPUT: Approximations $(\xi_1, \ldots, \xi_n)$ within ϵ to the eigenvalues of the matrix $C_{\mathbf{s},\mathbf{d}}$.

COMPUTATION:

sort $s_1, \ldots, s_n$, so that $|s_i| \leq |s_j|$ for $i > j$; apply the same permutation to $d_1, \ldots, d_n$.

`For` $j = n, n-1, \ldots, 1$ `do`

`If` $|d_j| > |s_j|$ `then` set $z = s_j(1+\epsilon)$, $\mathbf{v} = \mathbf{e}_j$, `else` set $z = s_j - d_j$, $\mathbf{v} = ((s_j - z)/(s_i - z))$ (compare with (4.4))

Set $\nu = 0$, $err = 1$.

`While` $err > \epsilon$ `and` $\nu < N$ `do`

$\nu = \nu + 1$

Apply a single shifted inverse power step with $\mathbf{x} = \mathbf{v}$ according to Algorithm 4.8 which outputs vector $\mathbf{y}$

set $z_{new} = s_j - (\sum_i d_i y_i)/y_j$ (compare with (5.3))

set $err = |(z_{new} - z)/z_{new}|$, $z = z_{new}$

`End while`

`If` $err < \epsilon$ `then` output $\xi_j = z$, `else` output `Failure`.

Choose the s_i closest to ξ_j (compare with the next Remark 6.5), set $s_j = \xi_j$ and exchange s_j with s_i and d_j with d_i.

Update the Weierstrass (D-K) corrections $d_i = d_i * (s_i - s_j)/(s_i - z)$, $i = 1, \ldots, j-1$ by means of (6.2).

`End do`

Output $\xi_1 = s_1 - d_1$

Remark 6.4. The main goal of our approach is approximating the eigenvalues of C (which are the zeros of $p(x)$). Thus we may alternatively deflate by dividing $p(x)$ by $x - z$, which for $C = F(p(x))$ requires only $2n-2$ multiplications and $n-1$ subtractions, that is, even less than based on Remark 6.1.

Remark 6.5. The choice of the starting vector for the shifted inverse power iteration is mainly heuristic. In many cases a relatively large value of d_j does not correspond to a great distance from s_j to the closest eigenvalue. Moreover, if $|d_j| \leq |s_j|$ then the modulus of the W(D-K) correction is smaller than the modulus of the approximation and it is likely that the approximation s_j is not far from an eigenvalue. Once ξ_j have been computed, the algorithm selects the approximation s_i closest to ξ_j. Actually, this is not the best strategy. What we have really implemented is the closeness condition $\min(|s_i - \xi_j| + 2||s_i| - |\xi_j||)$. In fact, when we apply the shifted inverse power method to approximate

polynomial roots (see the next section), we may have "approximations" $s = \rho e^{i\theta}$ to some root $\rho' e^{i\theta'}$ where $\rho \approx \rho'$ are very large and θ may be much different from θ', say, $\theta = 0$, $\theta' = \pi$. In this case the approximation $s = \rho e^{i\theta}$ would be closer to an eigenvalue with a very small modulus than to $\rho' e^{i\theta'}$. However, the inverse power algorithm generally provides a better convergence to $\rho' e^{i\theta'}$ if it uses the shift $z = \rho e^{i\theta}$.

7 Approximating polynomial roots

The inverse power method applied to a generalized companion matrix can be used for implementing an efficient polynomial rootfinder in the same fashion of [F01].

We first recall the following useful result of [B96] which allows us to say if a given approximation ξ of a zero of $p(x)$ is a zero of a slightly perturbed polynomial. This will be used as stopping condition for terminating the iterations.

Theorem 7.1. *Let $p(x)$ be the polynomial in (2.1) and ξ a complex number. Denote with $fl(p(\xi))$ the value obtained by computing $p(\xi)$ by means of Horner rule,*

$$\begin{aligned} &u_0 = p_n, \\ &u_{i+1} = \xi u_i + p_{n-i-1}, \quad i = 0, \ldots, n-1, \\ &p(\xi) = u_n \end{aligned}$$

in floating point arithmetic with machine precision μ. If

$$|\mathrm{fl}(p(\xi))| \leq \delta \sum_{i=0}^{n} |a_i||\xi|^i, \qquad \delta = (12n+3)\mu, \tag{7.1}$$

then there exists a polynomial $\tilde{p}(x) = \sum_{i=0}^{n} \tilde{a}_i x^i$ such that $\tilde{a}_i = a_i(1+\epsilon_i)$, $|\epsilon_i| \leq \delta$, and $\tilde{p}(\xi) = 0$. If the inequality (7.1) is not satisfied, then for any polynomial $\tilde{p}(x)$ such that $|\epsilon_i| \leq \frac{1}{3}\delta$ it holds $\tilde{p}(\xi) \neq 0$.

If equation (7.1) is satisfied we say that ξ is a *δ-approximated zero of $p(x)$*.

We use also the criterion of [B96] for selecting initial approximation to the zeros of $p(x)$ based on Rouché theorem and on Newton's polygon. For more details on this criterion we refer the reader to [B96].

Algorithm 7.2. [Approximating polynomial zeros]

INPUT: The degree n and the coefficients $a_0, \ldots, a_n$ of the polynomial (2.1).

OUTPUT: Approximations $\xi_1, \ldots, \xi_n$ to the zeros of $p(x)$ such that (7.1) is satisfied for $\xi = x_i$, $i, 1, \ldots, n$.

COMPUTATION:

Compute initial approximations $s_1, \ldots, s_n$ to the zeros of $p(x)$ by means of the criterion of [B96]. Set $m = n$, $\delta = (12n+3)\mu$, where μ is the machine precision.

`While` $m > 0$ `do`

compute the W(D-K) corrections $d_1, \ldots, d_n$ defined by (2.4) and check if s_i is a δ-approximated zero of $p(x)$.

Sort s_i so that the δ-approximated components are at the bottom and the components which are not yet δ-approximated are ordered with nonincreasing modulus.

Denote with m the number of components which are not yet δ-approximated.

Apply the shifted inverse power method (Algorithm 6.3) to the $m \times m$ generalized companion matrix $C_{\mathbf{s},\mathbf{d}}$ defined by $s_1, \ldots, s_m$, $d_1, \ldots, d_m$, and output approximations $\xi_1, \ldots, \xi_m$. Set $s_i = \xi_i$, $i = 1, \ldots, m$.

`End While`

8 Numerical experiments

Algorithm 7.2 has been implemented in Fortran 90 (the program can be downloaded at `www.dm.unipi.it/~bini`) and compared with the simple W(D-K) iteration implemented in the Gauss-Seidel style. In order to avoid overflow in the computation of the product $\prod_{j=1,\ j\neq i}^{n}(x_i - x_j)$, we have replaced this product with its logarithm both in our and in the W(D-K) algorithms. The algorithms have been tested with the following set of polynomials:

- *Binomials with roots of the unity*: $x^n - 1$. All the zeros are well conditioned.
- *Mignotte-like polynomials*: $x^n + (100x - 1)^3$. This is a particular class of polynomials obtained by modifying certain polynomials introduced by Maurice Mignotte which almost reach the Mahler bound to the separation of the roots. The polynomials which we have considered in this class have three zeros clustered around $1/100$ and, therefore, very ill-conditioned, the remaining zeros are roughly displaced along a circle.
- *Polynomials with unbalanced zeros*: $x^n + 10^{100}x^{n-3} + 10^{100}x^3 + 10^{-200}$. Three zeros of this polynomial have very large moduli and three zeros have very small moduli. Here it is crucial to use the starting criterion of [B96].
- *Mandelbrot polynomials*: The polynomials are recursively defined by means of the relation

$$m_0(x) = 1$$
$$m_{i+1}(x) = z m_i(x)^2 - 1, \quad i = 0, 1, \ldots, k-1, \quad n = 2^k - 1.$$

The value of the Mandelbrot polynomial at a point is computed by means of the above relations in $O(\log_2 n)$ ops by a suitable subroutine. In order to avoid overflow/underflow the program computes the logarithm of $p(x)$. The stopping condition is obtained by computing an upper bound on the relative error in the floating point computation of $p(x)$, by using a criterion similar to the one of Theorem 7.1.

n	cpu	sweeps	w-iter.	cpu	iter.
20	0.03	1	52	0.02	130
50	0.05	1	190	0.05	682
100	0.09	1	251	0.1	658
200	0.35	2	684	0.4	1537
500	1.9	2	1767	15.2	22585
1000	9.5	2	4394	12.8	9675
2000	37.0	2	6012	349.1	126431

Table 2: Polynomials $x^n - 1$

n	cpu	sweeps	w-iter.	cpu	iter.
20	0.01	1	99	0.01	206
50	0.02	2	204	0.12	1364
100	0.11	2	333	0.14	900
200	0.4	2	844	0.6	2066
500	1.6	2	1165	7.7	11018
1000	9.3	2	4196	47.4	34671
2000	25.6	1	3053	342.4	44156

Table 3: Polynomials $x^n + (100x - 1)^3$

The zeros of this class of polynomials determine the Mandelbrot set. Therefore they are clustered in a fractal and this feature makes it difficult to approximate all the zeros.

The results are reported in tables 2-5 where n denotes the degree of the polynomial; cpu is the cpu time needed by our algorithm (on the left) and by W(D-K) algorithm (on the right) run over a laptop with a Celeron^{TM} cpu; sweeps denotes the number of times that a new generalized companion matrix is generated and that its eigenvalues are computed; w-iter denotes the overall weighted number of shifted inverse power iterations, where the weight is m/n with m the size of the matrix to which the iteration is applied; finally, iter denotes the overall number of iterations of the W(D-K) method (a whole sweep is performed on all the n approximations and covers n iterations).

According to the results of our experiments, our implementation of the inverse power rootfinder significantly accelerates the W(D-K) method, and some additional acceleration is possible with better implementation (see, e.g. Remarks 6.2 and 6.4). Moreover, the performance of our algorithm improves as the degree of the polynomial increases. This makes our approach a valid tool for replacing the QR iteration technique used by S. Fortune in the implementation of his polynomial rootfinder. In fact, our algorithm computes all the eigenvalues of a generalized companion matrix in $O(n^2)$ ops per step, whereas the QR iteration uses the order of n^3 ops per step.

n	cpu	sweeps	w-iter.	cpu	iter.
15	0.01	2	131	0.01	192
31	0.02	2	477	0.05	623
63	0.12	3	1050	0.21	2009
127	0.79	6	3993	1.24	7177
255	4.3	12	10685	9.8	28404
511	31.5	26	40555	78.7	116974
1023	265	58	167149	623	453734

Table 4: Mandelbrot polynomials

n	cpu	sweeps	w-iter.	cpu	iter.
20	0.02	2	72	0.02	224
50	0.04	2	189	0.07	514
100	0.09	2	253	0.11	598
200	0.4	2	597	0.39	1376
500	2.9	2	2403	3.6	5511
1000	10.4	2	3438	10.3	7834
2000	51.5	2	9103	94.6	36154

Table 5: Polynomials $x^n + 10^{100}x^{n-3} + 10^{100}x^3 - 10^{-200}$

9 Conclusion

In Section 2 we gave a unified derivation of the W(D-K) algorithm and its higher order extensions to give a new insight into this approach. In Corollary 4.6 in Section 4, we again revealed the correlation between this algorithm and a matrix approach to the same problem. To accelerate W(D-K) iteration, we then applied shifted inverse power iteration to the generalized companion matrices of (4.2) and Example 4.2. The acceleration was confirmed by our numerical experiments performed for the matrices in (4.2) and reported in Section 8. We note, however, that our method is substantially more powerful for approximating a single root or a few roots (see Remark 6.2) or even for the highly important task of splitting a polynomial over a fixed root-free annulus [BGM02]. A path to further improvement may also lie in application of the inverse power method to the Frobenius matrix of Example 4.2 due to simplicity of deflation in this case (see Remark 6.4).

This method can be tried for other classes of structured matrices, such as diagonal + semiseparable matrices and diagonal + Frobenius matrices. The latter class is associated with polynomials represented in Newton's (rather than Lagrange's) form [B81], and most of our study (except for Remark 6.4 on deflation) can be extended.

References

[B81] S. Barnett, Congenial matrices. *Linear Algebra Appl.* **41**, 277–298, 1981.

[B96] D. A. Bini, Numerical computation of polynomial zeros by means of Aberth's method, *Numerical Algorithms*, 13, no. 3-4, 179–200, 1996.

[BF00] D. A. Bini, G. Fiorentino, Design, Analysis, and Implementation of a Multiprecision Polynomial Rootfinder, *Numerical Algorithms*, **23**: 127–173, 2000.

[BGM02] D. A. Bini, L. Gemignani, and B. Meini. Computations with infinite Toeplitz matrices and polynomials. *Linear Algebra Appl*, 343/344, 21–61 2002.

[BPa] D. A. Bini, V. Y. Pan, *Polynomial and Matrix Computations, Vol.2: Fundamental and Practical Algorithms*, Birkhäuser, Boston, to appear.

[C91] C. Carstensen, On a Linear Construction of Companion Matrices, *Linear Algebra Appl.*, **149**: 191-214, 1991.

[C91a] C. Carstensen, Inclusion of the Roots of a Polynomial Based on Gershgorin's Theorem, *Numerische Math.*, **59**: 349-360, 1991.

[C92] C. Carstensen, On Grau's Method for Simultaneous Factorization of Polynomials, *SIAM J. of Numerical Analysis*, **29, 2**: 601-613, 1992.

[DS93] J.J. Dongarra, M. Sidani, A Parallel Algorithm for the Nonsymmetric Eigenvalue Problem, *SIAM J. Sci. Computing*, **14**: 242-269, 1993.

[E73] L. Elsner, A Remark on Simultaneous Inclusions of the Zeros of a Polynomial by Gershgörin Theorem, *Numer. Math.*, **21**: 425-427, 1973.

[EG97] Y. Eidelman, I.Gohberg, Inversion Formulas and Linear Complexity Algorithm for Diagonal Plus Semiseparable Matrices, *Computers and Math. (with Applications)*, **33**: 69-79, 1997.

[EM95] A. Edelman, H. Murakami, Polynomial Roots from Companion Matrix Eigenvalues, *Mathematics of Computation*, **64**: 763-776, 1995.

[F90] M. Fiedler, Expressing a Polynomial as the Characteristic Polynomial of a Symmetric Matrix, *Linear Algebra Appl.*, **141**: 265–270, 1990.

[F01] S. Fortune, Polynomial Root Finding Using Iterated Eigenvalue Computation, *Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC'01)*, 121–128, ACM Press, New York, 2001.

[GL96] G. H. Golub, C. F. Van Loan, *Matrix Computations*, 3rd edition, The Johns Hopkins University Press, Baltimore, Maryland, 1996.

[HSS01] J. Hubbard, D. Schleicher, S. Sutherland, How to Find All Roots of Complex Polynomials by Newton's Method, *Inventiones Math.*, **146**: 1-33, 2001.

[KS94] M-hi Kim, S. Sutherland, Polynomial Root-finding Algorithms and Branched Covers, *SIAM J. on Computing*, **23, 2**: 415–436, 1994.

[McN93] J. M. McNamee, Bibliography on Roots of Polynomials, *J. Comp. Appl. Math.*, **47**: 391-394, 1993.

[McN97] J. M. McNamee, A Supplementary Bibliography on Roots of Polynomials, *J. Computational Applied Mathematics*, **78, 1**, 1997, http://www.elsevier.nl/homepage/sac/cam/mcnamee/index.html.

[MCVH01] N. Mastronardi, S. Chandrasekaran, S. Van Huffel, Fast and Stable Two-Way Algorithm for Diagonal Plus Semi-Separable Systems of Linear Equations, *Numer. Linear Algebra Appl.*, **8, 1**: 7-12, 2001.

[MV95a] F. Malek, R. Vaillancourt, Polynomial Zerofinding Iterative Matrix Algorithms, *Computers and Math. (with Applications)*, **29, 1**: 1–13, 1995.

[MV95b] F. Malek, R. Vaillancourt, A Composite Polynomial Zerofinding Matrix Algorithm, *Computers and Math. (with Applications)*, **30, 2**: 37–47, 1995.

[P95] V. Y. Pan, Optimal (up to Polylog Factors) Sequential and Parallel Algorithms for Approximating Complex Polynomial Zeros, *Proc. 27th Ann. ACM Symp. on Theory of Computing*, 741–750, ACM Press, New York, May, 1995.

[P97] V. Y. Pan, Solving a Polynomial Equation: Some History and Recent progress, *SIAM Review*, **39, 2**, 187–220, 1997.

[P01] V. Y. Pan, Univariate Polynomials: Nearly Optimal Algorithms for Factorization and Rootfinding, *Proc. Intern. Symposium on Symbolic and Algorithmic Computation (ISSAC'01)*, 253-267, ACM Press, New York, 2001.

[P02] V. Y. Pan, Univariate Polynomials: Nearly Optimal Algorithms for Factorization and Rootfinding, *Journal of Symbolic Computations*, **33, 5**, 701-733, 2002.

[PHI98] M. S. Petkovic, D. Herceg, S. Illic, Safe Convergence of Simultaneous Method for Polynomials Zeros, *Numer. Algorithms*, **17**: 313-331, 1998.

[TT94] K. C. Toh, L. N. Trefethen, Pseudozeros of Polynomials and Pseudospectra of Companion Matrices, *Numerische Math.*, **68**: 403–425, 1994.

Appendix

timings.txt

```
w-totit= weighted number of iterations: each iteration with matrix size i
         is weighted with the factor i/n


*polinomio x^n-1

n        cpu    sweeps    totit    totit/n    DK-cpu DK-it          MPS    PZ

20       0.03   1           52       2.5       0.02    130          0.01
50       0.05   1          190       3.8       0.05    682          0.02
100      0.09   1          251       2.5       0.1     658          0.04  0.04
200      0.35   2          684       3.4       0.4    1537          0.16  0.11
500      1.9    2         1767       1.8      15.2   22585          0.96  0.64
1000     9.5    2         4394       2.2      12.8    9675          3.91  2.5
2000    37.0    2         6012       3.0     349.1  126431         13.29  7.7
5000   187.5    1         7501       1.5                           95.6  65
10000 1045.1    2        20144       2.0                          412   275

*Polinomio x^n+(100 x-1)^3

n        cpu    sweeps  w-totit    totit/n    DK-cpu DK-it  it/n   MPS    PZ

20       0.0    1           99       5         0.01    206   10
50       0.04   2          204       4         0.12   1364   27     0.01
100      0.11   2          333       3.3       0.14    900    9     0.04
200      0.4    2          844       4.2       0.6    2066   10     0.16
500      1.59   2         1165       2.3       7.7   11018   20     0.82  0.5
1000     9.3    2         4196       4.2      47.4   34671   34.6   4.7   2.6
2000    25.6    1         3053       1.5      33.2   12036    2.4  13.7   5.4
5000   287.7    2        15085      3.0      342.4   44156    8.8 101.0  65
10000 1353.0    2        30740      3.0                           415


*Mandelbrot

n        cpu    sweeps  w-totit    totit/n    DK-cpu DK-it          MPS
15       0.01     2       131        8.7       0.01    192
```

```
31       0.02     2       477       15.3       0.05    623
63       0.12     3      1050       16.6       0.21   2009         0.19
127      0.79     6      3993       41.9       1.24   7177         1.06
255      4.3     12     10685       20.9       9.8   28404         7.19
511     31.5     26     40555       79.3      78.7  116974        50.8
1023   265       58    167149      163.3     623    453734       383

*Polinomio x^n + 10^100 x^(n-3) + 10^100 x^3-10^(-200)             MPS

n        cpu    sweeps    totit    totit/n    DK-cpu DK-it

20        0.02    2        72        3.6       0.02    224    11
50        0.04    2       189        3.8       0.07    514    10
100       0.09    2       253        2.5       0.11    598     6
200       0.4     2       597        3.0       0.39   1376     6
500       2.9     2      2403        4.8       3.6    5511    11   0.97
1000     10.4     2      3438        1.7      10.3    7834     4
2000     51.5     2      9103        4.5      94.6   36154
5000    349       2     14882        3.0     916.5  121225
10000
```

dksolve.f90

```
MODULE dk
  IMPLICIT NONE
  PRIVATE :: horner, rhorner, userpoly, &
       start, cnvex, cmerge, left, right, ctest
  PUBLIC  :: dksolve
  INTEGER, PARAMETER     :: dp = SELECTED_REAL_KIND(15, 60)
  REAL, PARAMETER        :: eps=1.d-16    ! used for stopping iterations
  LOGICAL, PARAMETER     :: dolog=.true. ! set true for debugging
  real (kind=dp) :: npe ! number of polynomial evaluations
CONTAINS
!*****************************************************************************
! All the software  contained in this library  is protected by copyright. *
! Permission  to use, copy, modify, and  distribute this software for any *
! purpose without fee is hereby granted, provided that this entire notice *
! is included  in all copies  of any software which is or includes a copy *
! or modification  of this software  and in all copies  of the supporting *
! documentation for such software.                                        *
!*****************************************************************************
!*****************************************************************************
! THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED *
! WARRANTY. IN NO EVENT, THE AUTHORS,  BE  LIABLE FOR ANY ERROR  IN THE   *
! SOFTWARE, ANY MISUSE  OF IT  OR ANY DAMAGE ARISING OUT OF ITS USE. THE  *
```

```
! ENTIRE RISK OF USING THE SOFTWARE LIES WITH THE PARTY DOING SO.         *
!*************************************************************************
! ANY USE  OF THE SOFTWARE  CONSTITUTES  ACCEPTANCE  OF THE TERMS  OF THE *
! ABOVE STATEMENT.                                                        *
!*************************************************************************
!
!   REFERENCE AUTHOR:
!
!       DARIO ANDREA BINI
!       UNIVERSITY OF PISA, ITALY
!       E-MAIL: bini@dm.unipi.it
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!                          DKSOLVE                                  !!!
!!!       APPROXIMATION OF POLYNOMIAL ROOTS BY MEANS OF               !!!
!!!                  DURAND-KERNER ITERATION                          !!!
!!!                           v 0.1 by                                !!!
!!!             D.A. Bini                                             !!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! The program implements the algorithm of Durand-Kerner for the !
! approximation of polynomial zeros.                              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! SUBROUTINES:
!       DKSOLVE: Implements the algorithm of Durand-Kerner
!       HORNER: Horner's rule for computing p(x)
!       RHORNER: Horner rule for computing x^np(x^(-1))
!       USERPOLY: user's polynomial
!--- The next subroutines, for the choice of starting
!    approximations, are taken from the package POLZEROS
!    by D. A. Bini, see NUMERICAL COMPUTATION OF POLYNOMIAL ZEROS
!    BY MEANS OF ABERTH'S METHOD, NUMERICAL ALGORITHMS, 13 (1996),
!    179-200, adjusted to Fortran90 by Alan Miller, CSIRO
!    Mathematical & Information Sciences, Private Bag 10, Clayton
!    South MDC, Victoria, Australia 3169.
!    Alan.Miller@mel.dms.csiro.au,
!    http://www.mel.dms.csiro.au/~alan
!
!       START
!       CNVX
!       LEFT
!       RIGHT
!       CMERGE
!       CTEST
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                       SUBROUTINE DKSOLVE                              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Approximates the zeros of the polynomial                              !
!           p(x)=p_0 + p_1*x + ... + p_n*x^n                            !
! by means of an algorithm based on Durand-Kerner iteration:            !
! according to the following steps:                                     !
! 1) chose starting approximations z(i), i=1,...,n:                     !
!    if the polynomial is assigned in terms of its coefficients         !
!    then use the Newton's polygon technique of Bini, Numer.            !
!    Algo. 13, 1996, 179-200. If the polynomial is assigned by          !
!    meas of a subroutine (user polynomial) choose the roots of         !
!    the unity shifted with a random rotation.                          !
! 2) compute the Durand-Kerner corrections                              !
!    w(i)=p(z(i))/prod_{j=1,n, j/=i}(z(i)-z(j)), and set                !
!    z(i)=z(i)-w(i), for i=1,...,n.                                     !
! 4) repeat from step 2 until convergence.                              !
! The convergence criterion is the one of D. Bini NUMERICAL             !
! COMPUTATION OF POLYNOMIAL ZEROS BY MEANS OF ABERTH'S METHOD,          !
! NUMERICAL ALGORITHMS, 13 (1996), 179-200. This criterion              !
! guarantees that the computed approximation is zero of a               !
! slightly perturbed polynomial.                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Input variables:                                                     !
!  n      : degree of p(x)                                              !
!  p(0:n) : complex coefficients p_i of the polynomial p(x)             !
!  usr    : if usr is true then p(x) is computed by means of            !
!           Horner rule, otherwise the subroutine  userpoly is         !
!           used (user polynomial). In this package, userpoly          !
!           computes the Mandelbrot polynomial defined by              !
!           m_0(x)=1, m_{k+1}(x)=x*m_k(x)^2+1                          !
!  Output variables                                                     !
!  z(1:n) : approximations of the zeros                                 !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Auxiliary variables and parameters:
!  again(1:n): again(i) is false if the approximation z(i) has          !
!              converged                                                !
!  nittot    : number of total<iterations of the power method          !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE dksolve(n, p, usr, z)
    IMPLICIT NONE
    INTEGER                             :: n
    COMPLEX (kind=dp), DIMENSION(0:n) :: p
    COMPLEX (kind=dp), DIMENSION(n)   :: z
    LOGICAL                             :: usr
```

```
    !
    INTEGER                          :: nsol, k, i, j, nz, &
         nit, nittot
    LOGICAL, DIMENSION(n)            :: again
    LOGICAL, DIMENSION(n+1)          :: h
    LOGICAL                          :: laux,ag
    REAL (kind=dp)                   :: mx1, mx2, abw
    COMPLEX (kind=dp)                :: tmp, s, zm1
    COMPLEX (kind=dp), DIMENSION(n)  :: w
    REAL (kind=dp), DIMENSION(n)     :: rad
    REAL (kind=dp), DIMENSION(0:n)   :: ap
    REAL (kind=dp), DIMENSION(n+1)   :: ap1
    INTEGER, PARAMETER :: nsweeps=5000
    npe=0.d0
    IF(dolog) OPEN(unit=2,file="log.log")
    ! initialize
    nittot=0
    nit=0
    w=0.d0
    nsol=0
    again=.TRUE.

    ! choose initial approximations
    h=.TRUE.
    IF(usr)THEN
       DO i=1,n
          z(i)=COS(6.28*i/n+0.223)+SIN(6.28*i/n+0.223)*(0.d0,1.d0)
       END DO
    ELSE
       ap1(1:n+1)=ABS(p(0:n))
       ap(0:n)=ap1(1:n+1)*(eps*n*4.d0)
       CALL  start(n, ap1, z, rad, nz, h)
    END IF

    DO k=1,nsweeps
       ! Compute the DK corrections
!       again=.TRUE. !!! ??
       DO i=1,n
       if(.not. again(i))exit
          IF(usr)THEN
             w(i)=0
          ELSE
             w(i)=LOG(p(n))
          END IF
          DO j=1,n
             IF (j.NE.i) THEN
```

```
                IF(z(i)-z(j)==0)THEN
                   WRITE(*,*)"WARNING 1 : confluent approximations"
                   STOP
                END IF
                w(i)=w(i)+LOG(z(i)-z(j))
             END IF
          END DO
          IF(usr)THEN
             CALL userpoly(n,z(i),s,again(i))
             w(i)=s-w(i)
          ELSE
             IF(ABS(z(i))<=1) THEN     !! |z(i)|<1
                CALL horner(n,p,ap,z(i),s,again(i))
                IF(s.NE.0)THEN
                   w(i)=LOG(s)-w(i)
                ELSE
                   w(i)=0.d0
                END IF
             ELSE
                zm1=1.d0/z(i)
                CALL rhorner(n,p,ap,zm1,s,again(i))
                IF(s.NE.0)THEN
                   s=LOG(s)+n*LOG(z(i))
                   w(i)=s-w(i)
                ELSE
                   w(i)=0
                END IF
             END IF
          END IF
          w(i)=-EXP(w(i))
!update the approximation
          z(i)=z(i)+w(i)
       END DO
       ! check for stop
       nsol=0
       DO j=1,n
          IF(.NOT. again(j)) THEN
             nsol=nsol+1
          END IF
       END DO

       ! sort z with approximated zeros at the end
       DO i=n,2,-1
          IF(again(i)) THEN
             DO j=i-1,1,-1
                IF(.NOT. again(j)) THEN
```

```
                  tmp=z(i)
                  z(i)=z(j)
                  z(j)=tmp
                  tmp=w(i)
                  w(i)=w(j)
                  w(j)=tmp
                  again(j)=.TRUE.
                  again(i)=.FALSE.
                  EXIT
               END IF
            END DO
         END IF
      END DO
      ! end of sorting


      IF(dolog)  THEN
         WRITE(2,*)"sweep ",k-1,"nsol=",nsol
         WRITE(*,*)"sweep ",k-1,"nsol=",nsol
         !do j=1,n
         !   write(4,*)z(j),again(j),abs(w(j))/abs(z(j))
         !end do
      END IF

      IF(nsol==n)GOTO 200

   END DO



200 CONTINUE

   IF(dolog) THEN
      WRITE(*,*)"sweep=",k-1," npe=",npe
      WRITE(2,*)"sweep=",k-1," npe=",npe
   END IF
  END SUBROUTINE dksolve

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                       SUBROUTINE HORNER                               !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE horner(n, p, ap, z, s, cnt)
    IMPLICIT NONE
    INTEGER                            :: n
    COMPLEX (kind=dp), DIMENSION(0:n) :: p
    REAL (kind=dp), DIMENSION(0:n)    :: ap
```

```
    COMPLEX (kind=dp)                 :: s, z
    LOGICAL                           :: cnt
    !
    INTEGER                           :: i
    REAL (kind=dp)                    :: as, az
    s=p(n)
    as=ap(n)
    az=ABS(z)
    npe=npe+1
    DO i=n-1,0,-1
       s=s*z+p(i)
       as=as*az+ap(i)
    END DO
    cnt=.TRUE.
    IF(ABS(s)<as)cnt=.FALSE.
  END SUBROUTINE horner

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                        SUBROUTINE RHORNER                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE rhorner(n, p, ap, z, s, cnt)
    IMPLICIT NONE
    INTEGER                           :: n
    COMPLEX (kind=dp), DIMENSION(0:n) :: p
    REAL (kind=dp), DIMENSION(0:n)    :: ap
    COMPLEX (kind=dp)                 :: s, z
    LOGICAL                           :: cnt
    !
    REAL (kind=dp)                    :: as, az
    INTEGER                           :: i
    npe=npe+1
    s=p(0)
    as=ap(0)
    az=ABS(z)
    DO i=1,n
       s=s*z+p(i)
       as=as*az+ap(i)
    END DO
    cnt=.TRUE.
    IF(ABS(s)<as)cnt=.FALSE.
  END SUBROUTINE rhorner

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                        SUBROUTINE USERPOLY                            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Compute Log(p(z)) where n=2^k-1 and p(x)=m_k(z) such that     !
```

```
!  m_0(z)=1, m_i(z)=z*m_{i-1}^2(z)+1, i=1,2,...,k.              !
!  the union of the zeros of this polynomial sequence generates !
!  the Mandelbrot set.                                          !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE userpoly(n, z, w, cnt)
    IMPLICIT NONE
    INTEGER           :: n
    COMPLEX (kind=dp) :: z, w
    LOGICAL           :: cnt
    !
    COMPLEX (kind=dp) :: lw, lz
    REAL (kind=dp)    :: alw, li, alw1, alz
    INTEGER           :: i,k
    real(kind=dp),parameter :: lu=-36.73680057+3, l6=1.791759
    npe=npe+1
    cnt=.TRUE.
    lw=0.d0

    alw=0.d0
    lz=LOG(z)
    alz=real(lz)
    li=-300
    k=LOG(n+1.d0)/LOG(2.d0)+0.5
    DO i=1,k
       lw=lz+lw*2
       IF(real(lw)<40)THEN
          lw=LOG(EXP(lw)-1)
       END IF
    alw1=REAL(lw)
li=-alw1+max(lu,alz+2*alw+max(log(2.d0)+li,log(3.d0)+lu)+log(2.d0))+log(2.d0)
!    li=max(li,lu)+l6+2*alw-alw1+alz
    alw=alw1
    END DO

!    alw1=REAL(lw)
!    lw=lz+lw*2
!    IF(ABS(lw)<37)THEN
!       lw=LOG(EXP(lw)-1)
!    END IF
    !   stop condition
!    IF(alw*2+REAL(lz)-20>REAL(lw))cnt=.FALSE.
   if(li>=1) cnt=.false.
    w=lw
  END SUBROUTINE userpoly
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!      SUBROUTINES FOR INITIAL APPROXIMATIONS                            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  SUBROUTINE start(n, a, y, radius, nz, h)
    IMPLICIT NONE
    INTEGER                             :: n
    INTEGER                             :: nz
    LOGICAL, dimension(n+1)             :: h  !!!!!!!!!!! n -> n+1
    COMPLEX (KIND(0.d0)),dimension(n) :: y
    REAL (KIND(0.d0))     :: small, big
    REAL (KIND(0.d0)), dimension(n+1) :: a
    REAL (KIND(0.d0)), dimension(n)   :: radius
    ! Local variables
    INTEGER                    :: i, iold, nzeros, j, jj
    REAL (KIND(0.d0))             :: r, th, ang, temp, xsmall, xbig
    REAL (KIND(0.d0)), PARAMETER :: pi2 = 6.2831853071796, sigma = 0.7

    SMALL=2.d0**(-1023)
    BIG=2.d0**1023

    xsmall = -1023*log(2.d0) !LOG(small)
    xbig = 1023*log(2.d0)  !LOG(big)
    nz = 0
    ! Compute the logarithm A(I) of the moduli of the coefficients of
    ! the polynomial and then the upper covex hull of the set (A(I),I)
    DO i = 1, n+1
       IF(a(i) /= 0) THEN
          a(i) = LOG(a(i))
       ELSE
          a(i) = -1.d30
       END IF
    END DO
    CALL cnvex(n+1, a, h)
    ! Given the upper convex hull of the set (A(I),I) compute the moduli
    ! of the starting approximations by means of Rouche's theorem
    iold = 1
    th = pi2/n
    DO i = 2, n+1
       IF (h(i)) THEN
          nzeros = i - iold
          temp = (a(iold) - a(i))/nzeros
          ! Check if the modulus is too small
          IF((temp < -xbig).AND.(temp >= xsmall))THEN
             WRITE(*,*)'WARNING:',nzeros,' ZERO(S) ARE TOO SMALL TO'
             WRITE(*,*)'REPRESENT THEIR INVERSES AS COMPLEX (KIND=dp) ::, THEY'
```

```
            WRITE(*,*)'ARE REPLACED BY SMALL NUMBERS, THE CORRESPONDING'
            WRITE(*,*)'RADII ARE SET TO -1'
            WRITE(2,*)'WARNING:',nzeros,' ZERO(S) ARE TOO SMALL TO '
            WRITE(2,*)'REPRESENT THEIR INVERSES AS COMPLEX (KIND=dp) ::, THEY'
            WRITE(2,*)'ARE REPLACED BY SMALL NUMBERS, THE CORRESPONDING'
            WRITE(2,*)'RADII ARE SET TO -1'
            nz = nz + nzeros
            r = 1.0D0/big
         END IF
         IF(temp < xsmall)THEN
            nz = nz + nzeros
            WRITE(*,*)'WARNING: ',nzeros,' ZERO(S) ARE TOO SMALL TO BE'
            WRITE(*,*)'REPRESENTED AS COMPLEX (KIND=dp) ::, THEY ARE SET TO 0'
            WRITE(*,*)'THE CORRESPONDING RADII ARE SET TO -1'
            WRITE(2,*)'WARNING: ',nzeros,' ZERO(S) ARE TOO SMALL TO BE'
            WRITE(2,*)'REPRESENTED AS COMPLEX (KIND=dp) ::, THEY ARE SET 0'
            WRITE(2,*)'THE CORRESPONDING RADII ARE SET TO -1'
         END IF
         ! Check if the modulus is too big
         IF(temp > xbig)THEN
            r = big
            nz = nz + nzeros
            WRITE(*,*)'WARNING: ', nzeros, ' ZEROS(S) ARE TOO BIG TO BE'
            WRITE(*,*)'REPRESENTED AS COMPLEX (KIND=dp) ::,'
            WRITE(*,*)'THE CORRESPONDING RADII ARE SET TO -1'
            WRITE(2,*)'WARNING: ',nzeros, ' ZERO(S) ARE TOO BIG TO BE'
            WRITE(2,*)'REPRESENTED AS COMPLEX (KIND=dp) ::,'
            WRITE(2,*)'THE CORRESPONDING RADII ARE SET TO -1'
         END IF
         IF((temp <= xbig).AND.(temp > MAX(-xbig, xsmall)))THEN
            r = EXP(temp)
         END IF
         ! Compute NZEROS approximations equally distributed in the disk of
         ! radius R
         ang = pi2/nzeros
         DO j = iold, i-1
            jj = j-iold+1
            IF((r <= (small)).OR.(r == big)) radius(j) = -1
            y(j) = r*(COS(ang*jj + th*i + sigma) + (0,1)*SIN(ang*jj + th*i + sigma))
         END DO
         iold = i
      END IF
   END DO
   RETURN
END SUBROUTINE start
```

```
  SUBROUTINE cnvex(n, a, h)
    IMPLICIT NONE
    INTEGER, INTENT(IN)        :: n
    LOGICAL, INTENT(OUT)       :: h(:)
    REAL (KIND(0.d0)), INTENT(IN) :: a(:)

! Local variables
    INTEGER :: i, j, k, m, nj, jc

    h(1:n) = .true.

    ! compute K such that N-2 <= 2**K < N-1
    k = INT(LOG(n-2.0D0)/LOG(2.0D0))
    IF(2**(k+1) <= (n-2)) k = k+1

    ! For each M=1,2,4,8,...,2**K, consider the NJ pairs of consecutive
    ! sets made up by M+1 points having the common vertex
    ! (JC,A(JC)), where JC=M*(2*J+1)+1 and J=0,...,NJ,
    ! NJ = MAX(0, INT((N-2-M)/(M+M))).
    ! Compute the upper convex hull of their union by means of subroutine CMERGE
    m = 1
    DO i = 0, k
       nj = MAX(0, INT((n-2-m)/(m+m)))
       DO j = 0, nj
          jc = (j+j+1)*m+1
          CALL cmerge(n, a, jc, m, h)
       END DO
       m = m+m
    END DO
    RETURN
  END SUBROUTINE cnvex


  SUBROUTINE left(h, i, il)
    IMPLICIT NONE
    INTEGER, INTENT(IN)  :: i
    INTEGER, INTENT(OUT) :: il
    LOGICAL, INTENT(IN)  :: h(:)

    DO il = i-1, 0, -1
       IF (h(il)) RETURN
    END DO
    RETURN
  END SUBROUTINE left
```

```
SUBROUTINE right(n, h, i, ir)
  IMPLICIT NONE
  INTEGER, INTENT(IN)  :: n, i
  INTEGER, INTENT(OUT) :: ir
  LOGICAL, INTENT(IN)  :: h(:)

  DO ir = i+1, n
     IF (h(ir)) RETURN
  END DO
  RETURN
END SUBROUTINE right



SUBROUTINE cmerge(n, a, i, m, h)
  IMPLICIT NONE
  INTEGER, INTENT(IN)        :: n, m, i
  LOGICAL, INTENT(IN OUT)    :: h(:)
  REAL (KIND(0.d0)), INTENT(IN) :: a(:)

  ! Local variables
  INTEGER :: ir, il, irr, ill
  LOGICAL :: tstl, tstr

  ! at the left and the right of the common vertex (I,A(I)) determine
  ! the abscissae IL,IR, of the closest vertices of the upper convex
  ! hull of the left and right sets, respectively
  CALL left(h, i, il)
  CALL right(n, h, i, ir)

  ! check the convexity of the angle formed by IL,I,IR
  IF (ctest(a, il, i, ir)) THEN
     RETURN
  ELSE
     ! continue the search of a pair of vertices in the left and right
     ! sets which yield the upper convex hull
     h(i) = .false.
     DO
        IF (il == (i-m)) THEN
           tstl = .true.
        ELSE
           CALL left(h, il, ill)
           tstl = ctest(a, ill, il, ir)
```

```
         END IF
         IF (ir == MIN(n, i+m)) THEN
            tstr = .true.
         ELSE
            CALL right(n, h, ir, irr)
            tstr = ctest(a, il, ir, irr)
         END IF
         h(il) = tstl
         h(ir) = tstr
         IF (tstl.AND.tstr) RETURN
         IF(.NOT.tstl) il = ill
         IF(.NOT.tstr) ir = irr
      END DO
    END IF

    RETURN
  END SUBROUTINE cmerge




  FUNCTION ctest(a, il, i, ir) RESULT(OK)
    IMPLICIT NONE
    INTEGER, INTENT(IN)        :: i, il, ir
    REAL (KIND(0.d0)), INTENT(IN) :: a(:)
    LOGICAL                    :: OK

    ! Local variables
    REAL (KIND(0.d0))             :: s1, s2
    REAL (KIND(0.d0)), PARAMETER :: toler = 0.4D0

    s1 = a(i) - a(il)
    s2 = a(ir) - a(i)
    s1 = s1*(ir-i)
    s2 = s2*(i-il)
    OK = .false.
    IF(s1 > (s2+toler)) OK = .true.
    RETURN
  END FUNCTION ctest


end MODULE dk
```

dksolve_drv.f90

```
program driver
  use dk
  implicit none

  integer                                          :: n, i,  nsol
  character(len=25)                                :: ch
  real(kind(0.d0))                                 :: aux
  complex(kind(0.d0)), dimension(:), allocatable :: z, p
  logical                                          :: usr

  write(*,*)'type:'
  write(*,*)'  usr for the user (Mandelbrot) polynomial'
  write(*,*)'  0   for the polynomial z^n-1'
  write(*,*)'  1   for the polynomial z^n+(100z-1)^3'
  write(*,*)'file_name for the polynomial stored in the file filename'
  read(*,*)ch
  usr=.false.
  if(ch=='0')then  ! x^n-1
  write(*,*)'n='
  read(*,*)n
     allocate(p(0:n),z(n))
     p=0.d0
     p=0.d0
     p(n)=1
     p(0)=-1
  else if(ch=='1')then  ! x^n+ (100x-1)^3
     write(*,*)'n='
     read(*,*)n
     allocate(p(0:n),z(n))
     p=0.d0
     p(n)=1
     p(0)=-1
     p(1)=300
     p(2)=-30000
     p(3)=1000000
  else if(ch=='2')then  ! x^n+ 10^100 x^(n-3)+ 10^100 x^3-10^(-200)
     write(*,*)'n='
     read(*,*)n
     allocate(p(0:n),z(n))
     p=0.d0
     p(n)=1
     p(n-3)=1.d100
     p(3)=1.d100
     p(0)=1.d-200
```

```
  else if(ch=='usr') then ! user polynomial
     usr=.true.
     write(*,*)'n=    (n=2^k-1)'
     read(*,*)n
     allocate(p(0:n),z(n))
  else
     open(file=ch,unit=4)
     read(4,*)n
     allocate(p(0:n),z(n))
     do i=0,n
        read(4,*) aux
        p(i) = aux
     end do
  end if

  call dksolve(n, p, usr, z)
  open(unit=3, file='zeros')
  do i=1,n
     write(3,*)real(z(i)),aimag(z(i))
  end do
end program driver
```

psolve.f90

```
MODULE inversepowersolve
  IMPLICIT NONE
  PRIVATE :: invpower, horner, rhorner, userpoly, &
       start, cnvex, cmerge, left, right, ctest
  PUBLIC  :: psolve
  INTEGER, PARAMETER     :: dp = SELECTED_REAL_KIND(15, 60)
  REAL, PARAMETER        :: eps=1.d-16    ! used for stopping iterations
  LOGICAL, PARAMETER     :: dolog=.true. ! set true for debugging
  real(kind=dp) :: npe ! number of polynomial evaluations


CONTAINS
!*****************************************************************************
! All the software  contained in this library  is protected by copyright. *
! Permission  to use, copy, modify, and  distribute this software for any *
! purpose without fee is hereby granted, provided that this entire notice *
! is included  in all copies  of any software which is or includes a copy *
! or modification  of this software  and in all copies  of the supporting *
! documentation for such software.                                        *
!*****************************************************************************
!*****************************************************************************
```

```
! THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED *
! WARRANTY. IN NO EVENT, THE AUTHORS,  BE  LIABLE FOR ANY ERROR  IN THE   *
! SOFTWARE, ANY MISUSE  OF IT  OR ANY DAMAGE ARISING OUT OF ITS USE. THE  *
! ENTIRE RISK OF USING THE SOFTWARE LIES WITH THE PARTY DOING SO.         *
!*************************************************************************
! ANY USE  OF THE SOFTWARE  CONSTITUTES  ACCEPTANCE  OF THE TERMS  OF THE *
! ABOVE STATEMENT.                                                        *
!*************************************************************************
!
!   REFERENCE AUTHOR:
!
!       DARIO ANDREA BINI
!       UNIVERSITY OF PISA, ITALY
!       E-MAIL: bini@dm.unipi.it
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!                          PSOLVE                                 !!!
!!!       APPROXIMATION OF POLYNOMIAL ROOTS BY MEANS OF             !!!
!!!   INVERSE POWER METHOD AND STRUCTURED MATRIX COMPUTATION        !!!
!!!                          v 0.1 by                               !!!
!!!            D.A. Bini, L. Gemignani, V. Pan                      !!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! The program implements the algorithm of S. Fortune for the    !
! approximation of polynomial zeros where the computation of    !
! the  eigenvalues of the companion-like matrix is computed by  !
! means of the inverse power method according to the algorithm  !
! of D.A. Bini, L. Gemignani, V. Pan "Title..."                 !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! SUBROUTINES:
!       PSOLVE: Implements the algorithm of S. Fortune
!       INVPOWER: shifted inverse powers method
!       HORNER: Horner's rule for computing p(x)
!       RHORNER: Horner rule for computing x^np(x^(-1))
!       USERPOLY: user's polynomial
!--- The next subroutines, for the choice of starting
!    approximations, are taken from the package POLZEROS
!    by D. A. Bini, see NUMERICAL COMPUTATION OF POLYNOMIAL ZEROS
!    BY MEANS OF ABERTH'S METHOD, NUMERICAL ALGORITHMS, 13 (1996),
!    179-200, adjusted to Fortran90 by Alan Miller, CSIRO
!    Mathematical & Information Sciences, Private Bag 10, Clayton
!    South MDC, Victoria, Australia 3169.
!    Alan.Miller@mel.dms.csiro.au,
!    http://www.mel.dms.csiro.au/~alan
!
!       START
!       CNVX
```

```
!       LEFT
!       RIGHT
!       CMERGE
!       CTEST
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                       SUBROUTINE PSOLVE                                  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Approximates the zeros of the polynomial                                  !
!           p(x)=p_0 + p_1*x + ... + p_n*x^n                  !
! by means of an algorithm based on the approach of S. Fortune,             !
! according to the following steps:                                         !
! 1) chose starting approximations z(i), i=1,...,n:                         !
!    if the polynomial is assigned in terms of its coefficients             !
!    then use the Newton's polygon technique of Bini, Numer.                !
!    Algo. 13, 1996, 179-200. If the polynomial is assigned by              !
!    meas of a subroutine (user polynomial) choose the roots of             !
!    the unity shifted with a random rotation.                              !
! 2) compute the Durand-Kerner corrections                                  !
!    w(i)=p(z(i))/prod_{j=1,n, j/=i}(z(i)-z(j)), i=1,n                      !
! 3) compute the eigenvalues of the matrix D-ew^T, with                     !
!    D=Diag(z(1),...,z(n)), e=(1,...1)^T, and assign them to z;             !
! 4) repeat from step 2 until convergence.                                  !
! The convergence criterion is the one of D. Bini NUMERICAL                 !
! COMPUTATION OF POLYNOMIAL ZEROS BY MEANS OF ABERTH'S METHOD,              !
! NUMERICAL ALGORITHMS, 13 (1996), 179-200. This criterion                  !
! guarantees that the computed approximation is zero of a                   !
! slightly perturbed polynomial.                                            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Input variables:                                                         !
!  n      : degree of p(x)                                                  !
!  p(0:n) : complex coefficients p_i of the polynomial p(x)                 !
!  usr    : if usr is true then p(x) is computed by means of                !
!           Horner rule, otherwise the subroutine  userpoly is             !
!           used (user polynomial). In this package, userpoly              !
!           computes the Mandelbrot polynomial defined by                  !
!           m_0(x)=1, m_{k+1}(x)=x*m_k(x)^2+1                              !
!  Output variables                                                         !
!  z(1:n) : approximations of the zeros                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Auxiliary variables and parameters:
!  again(1:n): again(i) is false if the approximation z(i) has             !
!              converged                                                   !
```

```
!  nsweeps   : max number of times that steps 2,3,4 are         !
!              performed                                        !
!  nittot    : number of total<iterations of the power method   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE psolve(n, p, usr, z)
    IMPLICIT NONE
    INTEGER                              :: n
    COMPLEX (kind=dp), DIMENSION(0:n) :: p
    COMPLEX (kind=dp), DIMENSION(n)   :: z
    LOGICAL                              :: usr
    !
    INTEGER                           :: nsol, k, i, j, nz !, &
!         nit, nittot
    LOGICAL, DIMENSION(n)             :: again
    LOGICAL, DIMENSION(n+1)           :: h
    LOGICAL                           :: laux
    REAL (kind=dp)                    :: mx1, mx2, abw, nit, nittot
    COMPLEX (kind=dp)                 :: tmp, s, zm1
    COMPLEX (kind=dp), DIMENSION(n) :: w
    REAL (kind=dp), DIMENSION(n)    :: rad
    REAL (kind=dp), DIMENSION(0:n)  :: ap
    REAL (kind=dp), DIMENSION(n+1)  :: ap1
    INTEGER, PARAMETER :: nsweeps=100

    npe=0.d0
    IF(dolog) OPEN(unit=2,file="log.log")
    ! initialize
    nittot=0
    nit=0
    w=0.d0
    nsol=0
    again=.TRUE.

    ! choose initial approximations
    h=.TRUE.
    IF(usr)THEN
       DO i=1,n
          z(i)=COS(6.28*i/n+0.223)+SIN(6.28*i/n+0.223)*(0.d0,1.d0)
       END DO
    ELSE
       ap1(1:n+1)=ABS(p(0:n))
       ap(0:n)=ap1(1:n+1)*(eps*n*4.d0)
       CALL  start(n, ap1, z, rad, nz, h)
    END IF

    DO k=1,nsweeps
```

```
      ! Compute the DK corrections
!      again=.TRUE. !!! ??
      DO i=1,n
if(.not. again(i))exit  !!!!!!!!!!!!!
         IF(usr)THEN
            w(i)=0
         ELSE
            w(i)=LOG(p(n))
         END IF
         DO j=1,n
            IF (j.NE.i) THEN
               IF(z(i)-z(j)==0)THEN
                  WRITE(*,*)"WARNING 1 : confluent approximations"
                  STOP
               END IF
               w(i)=w(i)+LOG(z(i)-z(j))
            END IF
         END DO
         IF(usr)THEN
            CALL userpoly(n,z(i),s,again(i))
!if(ag==.false. .and. again(i))write(*,*)"*** i=",i,"z=",z(i)
            w(i)=s-w(i)
         ELSE
            IF(ABS(z(i))<=1) THEN     !! |z(i)|<1
               CALL horner(n,p,ap,z(i),s,again(i))
               IF(s.NE.0)THEN
                  w(i)=LOG(s)-w(i)
               ELSE
                  w(i)=0.d0
               END IF
            ELSE
               zm1=1.d0/z(i)
               CALL rhorner(n,p,ap,zm1,s,again(i))
               IF(s.NE.0)THEN
                  s=LOG(s)+n*LOG(z(i))
                  w(i)=s-w(i)
               ELSE
                  w(i)=0
               END IF
            END IF
         END IF
         w(i)=-EXP(w(i))
      END DO
      ! computed the DK corrections

      ! Check for convergence !???
```

```
       DO i=1,n
          IF (ABS(z(i)-w(i))==ABS(z(i)))then
!again(i)=.FALSE.
!write(*,*)"%%%% i=",i,"z=",z(i)
end if
       END DO

       ! sort z + w in nonincreasing modulus
       DO i=1,n-1
          mx1=ABS(z(i)+w(i))
          DO j=i+1,n
             mx2=ABS(z(j)+w(j))
             IF(mx1<mx2)THEN
                mx1=mx2
                tmp=z(i)
                z(i)=z(j)
                z(j)=tmp
                tmp=w(i)
                w(i)=w(j)
                w(j)=tmp
                laux=again(i)
                again(i)=again(j)
                again(j)=laux
             END IF
          END DO
       END DO


       ! check for stop
       nsol=0
       DO j=1,n
          IF(.NOT. again(j)) THEN
             nsol=nsol+1
          END IF
       END DO

       ! sort z with approximated zeros at the end
       DO i=n,2,-1
          IF(again(i)) THEN
             DO j=i-1,1,-1
                IF(.NOT. again(j)) THEN
                   tmp=z(i)
                   z(i)=z(j)
                   z(j)=tmp
                   tmp=w(i)
                   w(i)=w(j)
```

```
                w(j)=tmp
                again(j)=.TRUE.
                again(i)=.FALSE.
                EXIT
              END IF
            END DO
          END IF
        END DO
        ! end of sorting


        IF(dolog)  THEN
          WRITE(2,*)"sweep ",k-1,"nsol=",nsol,"nit=",nit
          WRITE(*,*)"sweep ",k-1,"nsol=",nsol,"nit=",nit
          !do j=1,n
          !   write(4,*)z(j),again(j),abs(w(j))/abs(z(j))
          !end do
        END IF

        IF(nsol==n)GOTO 200

        ! compute eigenvalues of D+we^T by means of shifted inverse power
        CALL invpower(n, p, ap, z, w, again, usr,  nit)
        nittot=nittot+nit
      END DO
200 CONTINUE

      IF(dolog) THEN
        WRITE(*,*)"sweep=",k-1,"nittot=",nittot,"npe/n=",npe/n
        WRITE(2,*)"sweep=",k-1,"nittot=",nittot,"npe/n=",npe/n
      END IF
  END SUBROUTINE psolve

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                       SUBROUTINE HORNER                               !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE horner(n, p, ap, z, s, cnt)
    IMPLICIT NONE
    INTEGER                             :: n
    COMPLEX (kind=dp), DIMENSION(0:n) :: p
    REAL (kind=dp), DIMENSION(0:n)    :: ap
    COMPLEX (kind=dp)                   :: s, z
    LOGICAL                             :: cnt
    !
    INTEGER                             :: i
    REAL (kind=dp)                      :: as, az
```

```
    npe=npe+1
    s=p(n)
    as=ap(n)
    az=ABS(z)

    DO i=n-1,0,-1
       s=s*z+p(i)
       as=as*az+ap(i)
    END DO
    cnt=.TRUE.
    IF(ABS(s)<as)cnt=.FALSE.
  END SUBROUTINE horner

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                       SUBROUTINE RHORNER                              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE rhorner(n, p, ap, z, s, cnt)
    IMPLICIT NONE
    INTEGER                              :: n
    COMPLEX (kind=dp), DIMENSION(0:n) :: p
    REAL (kind=dp), DIMENSION(0:n)    :: ap
    COMPLEX (kind=dp)                    :: s, z
    LOGICAL                              :: cnt
    !
    REAL (kind=dp)                       :: as, az
    INTEGER                              :: i
    npe=npe+1
    s=p(0)
    as=ap(0)
    az=ABS(z)
    DO i=1,n
       s=s*z+p(i)
       as=as*az+ap(i)
    END DO
    cnt=.TRUE.
    IF(ABS(s)<as)cnt=.FALSE.
  END SUBROUTINE rhorner

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                       SUBROUTINE USERPOLY                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Compute Log(p(z)) where n=2^k-1 and p(x)=m_k(z) such that             !
!  m_0(z)=1, m_i(z)=z*m_{i-1}^2(z)+1, i=1,2,...,k.                     !
!  the union of the zeros of this polynomial sequence generates        !
!  the Mandelbrot set.                                                 !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
  SUBROUTINE userpoly(n, z, w, cnt)
    IMPLICIT NONE
    INTEGER           :: n
    COMPLEX (kind=dp) :: z, w
    LOGICAL           :: cnt
    !
    COMPLEX (kind=dp) :: lw, lz
    REAL (kind=dp)    :: alw, li, alw1, alz
    INTEGER           :: i,k
    real(kind=dp),parameter :: lu=-36.73680057+3, l6=1.791759
    npe=npe+1
    cnt=.TRUE.
    lw=0.d0

    alw=0.d0
    lz=LOG(z)
    alz=real(lz)
    li=-300
    k=LOG(n+1.d0)/LOG(2.d0)+0.5
    DO i=1,k
       lw=lz+lw*2
       IF(real(lw)<40)THEN
          lw=LOG(EXP(lw)-1)
       END IF
    alw1=REAL(lw)
li=-alw1+max(lu,alz+2*alw+max(log(2.d0)+li,log(3.d0)+lu)+log(2.d0))+log(2.d0)
    alw=alw1
    END DO

!    alw1=REAL(lw)
!    lw=lz+lw*2
!    IF(ABS(lw)<37)THEN
!       lw=LOG(EXP(lw)-1)
!    END IF
    !   stop condition
!    IF(alw*2+REAL(lz)-20>REAL(lw))cnt=.FALSE.
   if(li>=1) cnt=.false.
    w=lw
  END SUBROUTINE userpoly

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                        SUBROUTINE INVPOWER                              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! It computes the eigenvalues of the matrix                               !
!           A=D+we^T, D=Diag(z1,...,zn), e=(1,...,1)^T                    !
! by means of the shifted inverse power where at each step,               !
```

```
! given the vector u, it is computed the vector v such that     !
! v = (D-lambda I + we^T)^-1 u                                  !
! The following expression, based on the Woodbury Sherman       !
! Morrison formula is used                                      !
!     v = (I-sD'we)D'u,  D'=(D-lambda I)^-1,  s=1/(1+e^TD'w)    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Input variables:                                             !
!  n      : degree of p(x)                                      !
!  p(0:n) : complex coefficients p_i of the polynomial p(x)     !
!  usr    : if usr is true then p(x) is computed by means of    !
!           Horner rule, otherwise the subroutine  userpoly is  !
!           used (user polynomial). In this package, userpoly   !
!           computes the Mandelbrot polynomial defined by       !
!           m_0(x)=1, m_{k+1}(x)=x*m_k(x)^2+1                    !
!  Output variables                                             !
!  z(1:n) : approximations of the zeros                         !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Auxiliary variables and parameters:
!  again(1:n): again(i) is false if the approximation z(i) has  !
!              converged                                        !
!  nsweeps   : max number of times that steps 2,3,4 are         !
!              performed                                        !
!  nittot    : number of total<iterations of the power method   !
!  dolog     : if true, the program prints in the screen and in !
!              the file log.log some useful information for     !
!              debugging                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
  subroutine invpower(n, p, ap, z, w, again, usr, nit)
    implicit none

    integer :: n
    complex (kind=dp), dimension(0:n) :: p
    real (kind=dp), dimension(0:n)    :: ap
    complex (kind=dp), dimension(n)   :: z, w
    logical, dimension(n)             :: again
    logical                           :: usr
    real(kind=dp)                     :: nit
    !
    integer                           :: i, j, iter, jmax
    complex (kind=dp), dimension(n)   :: u, v, xinv, uu
    real (kind=dp)                    :: err, mx, umax, tmp1, tmp2
    complex (kind=dp)                 :: smu, lambda, smu1, lambdan, s
    real (kind=dp), dimension(n)      :: aux
```

```
logical                          :: laux
integer, parameter :: nitmax=30


nit=0
! initial random vector for power method
do i=n,2,-1
   if(.not.again(i)) goto 100 !exit !!!!!!!!!
   do j=1,i
      if(i==j)then
         u(j)=1
      else
         u(j)=w(j)/(z(j)+w(j)-z(i))
         !u(j)=0.d0
      end if
   end do


   ! shifted inverse powers
   iter=1
   err=1
   if(abs(w(i))<abs(z(i))*0.5)then
      lambda=z(i) +w(i) ! choose as lambda the DK approximation
   else
      lambda=z(i)*(1.d0+1.d-10)
   end if
   ! start iterations
   do while(iter<nitmax .and. (err>eps*n) .and. abs(z(i)-lambda).ne.0)
      iter=iter+1
      nit=nit+i
      xinv(1:i)=(z(1:i)-lambda)
      !check confluent approx
      do j=1,i
         if(abs(xinv(j)).eq.0)then
            write(*,*)"WARNING 2 : confluent approximation"
            z(j)=z(j)*(1+0.123d-7)
            xinv(j)=z(j)-lambda
         end if
      end do
      xinv(1:i)=1.d0/xinv(1:i)
      if(iter==2) then !choose initial vector for shifted inverse power
         if(abs(w(i))<abs(z(i))*0.5)then
            u(1:i)=w(1:i)*xinv(1:i)
         else
            u(1:i)=0
            u(i)=1
```

```
      end if
   end if
   uu(1:i)=xinv(1:i)*u(1:i)
   smu=sum(uu(1:i))
   v(1:i)=xinv(1:i)*w(1:i)
   smu1=1.d0+sum(v(1:i))
   ! to avoid underflow and breakdown of lf95
   tmp1=abs(smu)
   tmp2=abs(smu1)
   if(tmp2.ne.0)then
      if(log(tmp1+1.d-301)-log(tmp2) .le. -300*log(10.d0))then
         smu=0.d0
      else
         smu=smu/smu1
      end if
      umax=0.d0
      jmax=i !!!!!!
      lambdan=lambda+u(jmax)/(uu(jmax)-smu*v(jmax))
      u(1:i)=uu(1:i)-smu*v(1:i)
   else
      lambdan=lambda
      u(1:i)=uu(1:i)
   end if
   !normalize the new u
   umax=0.d0
   do j=1,i
      if(abs(u(j)).gt.umax)then
         umax=abs(u(j))
         jmax=j
      end if
   end do
   mx=1.d0/abs(u(jmax))
   u(1:i)=u(1:i)*mx

   err=max(abs(lambda),abs(lambdan))
   if(err.ne.0)then
      err=abs(lambdan-lambda)/err
   else
      err=0
   end if

   lambda=lambdan
end do


! choose the diagonal entry closest to lambda
```

```
      aux(1:i)=abs(z(1:i)-lambda)+abs(abs(z(1:i))-abs(lambda))*2
      tmp1=aux(1)
      jmax=1
      do j=2,i
         if(aux(j)<tmp1)then
            jmax=j
            tmp1=aux(j)
         end if
      end do
      ! exchange components jmax and i of z and w
      smu=z(jmax)
      z(jmax)=z(i)
      z(i)=smu
      smu=w(jmax)
      w(jmax)=w(i)
      w(i)=smu
      laux=again(jmax)
      again(jmax)=again(i)
      again(i)=laux
      !update w
      do j=1,i-1
         s=z(j)-lambda
         if(s.ne.0)then
            w(j)=w(j)*(z(j)-z(i))/s
         else
            write(*,*)"WARNING 3 : Confluent approximation"
         end if
      end do
      z(i)=lambda
100 end do

    !if(again(1))then
    z(1)=z(1)+w(1)
    nit=nit/n
  end subroutine invpower


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!      SUBROUTINES FOR INITIAL APPROXIMATIONS                   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  SUBROUTINE start(n, a, y, radius, nz, h)
    IMPLICIT NONE
    INTEGER                           :: n
    INTEGER                           :: nz
    LOGICAL, dimension(n+1)           :: h  !!!!!!!!!!! n -> n+1
```

```
COMPLEX (KIND(0.d0)),dimension(n) :: y
REAL (KIND(0.d0))     :: small, big
REAL (KIND(0.d0)), dimension(n+1) :: a
REAL (KIND(0.d0)), dimension(n)   :: radius
! Local variables
INTEGER                   :: i, iold, nzeros, j, jj
REAL (KIND(0.d0))            :: r, th, ang, temp, xsmall, xbig
REAL (KIND(0.d0)), PARAMETER :: pi2 = 6.2831853071796, sigma = 0.7

SMALL=2.d0**(-1023)
BIG=2.d0**1023

xsmall = -1023*log(2.d0) !LOG(small)
xbig = 1023*log(2.d0)  !LOG(big)
nz = 0
! Compute the logarithm A(I) of the moduli of the coefficients of
! the polynomial and then the upper covex hull of the set (A(I),I)
DO i = 1, n+1
   IF(a(i) /= 0) THEN
      a(i) = LOG(a(i))
   ELSE
      a(i) = -1.d30
   END IF
END DO
CALL cnvex(n+1, a, h)
! Given the upper convex hull of the set (A(I),I) compute the moduli
! of the starting approximations by means of Rouche's theorem
iold = 1
th = pi2/n
DO i = 2, n+1
   IF (h(i)) THEN
      nzeros = i - iold
      temp = (a(iold) - a(i))/nzeros
      ! Check if the modulus is too small
      IF((temp < -xbig).AND.(temp >= xsmall))THEN
         WRITE(*,*)'WARNING:',nzeros,' ZERO(S) ARE TOO SMALL TO'
         WRITE(*,*)'REPRESENT THEIR INVERSES AS COMPLEX (KIND=dp) ::, THEY'
         WRITE(*,*)'ARE REPLACED BY SMALL NUMBERS, THE CORRESPONDING'
         WRITE(*,*)'RADII ARE SET TO -1'
         WRITE(2,*)'WARNING:',nzeros,' ZERO(S) ARE TOO SMALL TO '
         WRITE(2,*)'REPRESENT THEIR INVERSES AS COMPLEX (KIND=dp) ::, THEY'
         WRITE(2,*)'ARE REPLACED BY SMALL NUMBERS, THE CORRESPONDING'
         WRITE(2,*)'RADII ARE SET TO -1'
         nz = nz + nzeros
         r = 1.0D0/big
      END IF
```

```
          IF(temp < xsmall)THEN
             nz = nz + nzeros
             WRITE(*,*)'WARNING: ',nzeros,' ZERO(S) ARE TOO SMALL TO BE'
             WRITE(*,*)'REPRESENTED AS COMPLEX (KIND=dp) ::, THEY ARE SET TO 0'
             WRITE(*,*)'THE CORRESPONDING RADII ARE SET TO -1'
             WRITE(2,*)'WARNING: ',nzeros,' ZERO(S) ARE TOO SMALL TO BE'
             WRITE(2,*)'REPRESENTED AS COMPLEX (KIND=dp) ::, THEY ARE SET 0'
             WRITE(2,*)'THE CORRESPONDING RADII ARE SET TO -1'
          END IF
          ! Check if the modulus is too big
          IF(temp > xbig)THEN
             r = big
             nz = nz + nzeros
             WRITE(*,*)'WARNING: ', nzeros, ' ZEROS(S) ARE TOO BIG TO BE'
             WRITE(*,*)'REPRESENTED AS COMPLEX (KIND=dp) ::,'
             WRITE(*,*)'THE CORRESPONDING RADII ARE SET TO -1'
             WRITE(2,*)'WARNING: ',nzeros, ' ZERO(S) ARE TOO BIG TO BE'
             WRITE(2,*)'REPRESENTED AS COMPLEX (KIND=dp) ::,'
             WRITE(2,*)'THE CORRESPONDING RADII ARE SET TO -1'
          END IF
          IF((temp <= xbig).AND.(temp > MAX(-xbig, xsmall)))THEN
             r = EXP(temp)
          END IF
          ! Compute NZEROS approximations equally distributed in the disk of
          ! radius R
          ang = pi2/nzeros
          DO j = iold, i-1
             jj = j-iold+1
             IF((r <= (small)).OR.(r == big)) radius(j) = -1
             y(j) = r*(COS(ang*jj + th*i + sigma) + (0,1)*SIN(ang*jj + th*i +
sigma))
          END DO
          iold = i
       END IF
    END DO
    RETURN
  END SUBROUTINE start



  SUBROUTINE cnvex(n, a, h)
    IMPLICIT NONE
    INTEGER, INTENT(IN)        :: n
    LOGICAL, INTENT(OUT)       :: h(:)
    REAL (KIND(0.d0)), INTENT(IN) :: a(:)
```

```
! Local variables
    INTEGER :: i, j, k, m, nj, jc

    h(1:n) = .true.

    ! compute K such that N-2 <= 2**K < N-1
    k = INT(LOG(n-2.0D0)/LOG(2.0D0))
    IF(2**(k+1) <= (n-2)) k = k+1

    ! For each M=1,2,4,8,...,2**K, consider the NJ pairs of consecutive
    ! sets made up by M+1 points having the common vertex
    ! (JC,A(JC)), where JC=M*(2*J+1)+1 and J=0,...,NJ,
    ! NJ = MAX(0, INT((N-2-M)/(M+M))).
    ! Compute the upper convex hull of their union by means of subroutine CMERGE
    m = 1
    DO i = 0, k
       nj = MAX(0, INT((n-2-m)/(m+m)))
       DO j = 0, nj
          jc = (j+j+1)*m+1
          CALL cmerge(n, a, jc, m, h)
       END DO
       m = m+m
    END DO
    RETURN
  END SUBROUTINE cnvex


  SUBROUTINE left(h, i, il)
    IMPLICIT NONE
    INTEGER, INTENT(IN)  :: i
    INTEGER, INTENT(OUT) :: il
    LOGICAL, INTENT(IN)  :: h(:)

    DO il = i-1, 0, -1
       IF (h(il)) RETURN
    END DO
    RETURN
  END SUBROUTINE left



  SUBROUTINE right(n, h, i, ir)
    IMPLICIT NONE
    INTEGER, INTENT(IN)  :: n, i
    INTEGER, INTENT(OUT) :: ir
    LOGICAL, INTENT(IN)  :: h(:)
```

```
  DO ir = i+1, n
     IF (h(ir)) RETURN
  END DO
  RETURN
END SUBROUTINE right



SUBROUTINE cmerge(n, a, i, m, h)
  IMPLICIT NONE
  INTEGER, INTENT(IN)        :: n, m, i
  LOGICAL, INTENT(IN OUT)    :: h(:)
  REAL (KIND(0.d0)), INTENT(IN) :: a(:)

  ! Local variables
  INTEGER :: ir, il, irr, ill
  LOGICAL :: tstl, tstr

  ! at the left and the right of the common vertex (I,A(I)) determine
  ! the abscissae IL,IR, of the closest vertices of the upper convex
  ! hull of the left and right sets, respectively
  CALL left(h, i, il)
  CALL right(n, h, i, ir)

  ! check the convexity of the angle formed by IL,I,IR
  IF (ctest(a, il, i, ir)) THEN
     RETURN
  ELSE
     ! continue the search of a pair of vertices in the left and right
     ! sets which yield the upper convex hull
     h(i) = .false.
     DO
        IF (il == (i-m)) THEN
           tstl = .true.
        ELSE
           CALL left(h, il, ill)
           tstl = ctest(a, ill, il, ir)
        END IF
        IF (ir == MIN(n, i+m)) THEN
           tstr = .true.
        ELSE
           CALL right(n, h, ir, irr)
           tstr = ctest(a, il, ir, irr)
        END IF
        h(il) = tstl
```

```
        h(ir) = tstr
        IF (tstl.AND.tstr) RETURN
        IF(.NOT.tstl) il = ill
        IF(.NOT.tstr) ir = irr
      END DO
    END IF

    RETURN
  END SUBROUTINE cmerge




  FUNCTION ctest(a, il, i, ir) RESULT(OK)
    IMPLICIT NONE
    INTEGER, INTENT(IN)        :: i, il, ir
    REAL (KIND(0.d0)), INTENT(IN) :: a(:)
    LOGICAL                    :: OK

    ! Local variables
    REAL (KIND(0.d0))            :: s1, s2
    REAL (KIND(0.d0)), PARAMETER :: toler = 0.4D0

    s1 = a(i) - a(il)
    s2 = a(ir) - a(i)
    s1 = s1*(ir-i)
    s2 = s2*(i-il)
    OK = .false.
    IF(s1 > (s2+toler)) OK = .true.
    RETURN
  END FUNCTION ctest


end MODULE inversepowersolve
```

psolve_drv.f90

```
program driver
  use inversepowersolve
  implicit none

  integer                                          :: n, i,  nsol
  character(len=25)                                :: ch
  real(kind(0.d0))                                 :: aux
  complex(kind(0.d0)), dimension(:), allocatable :: z, p
```

```
logical                                          :: usr

write(*,*)'type:'
write(*,*)'  usr for the user (Mandelbrot) polynomial'
write(*,*)'  0   for the polynomial z^n-1'
write(*,*)'  1   for the polynomial z^n+(100z-1)^3'
write(*,*)'  2   for the polynomial '
write(*,*)'        z^n+1.d100*z^(n-3)+1.d100*x^3+1.d-200'
write(*,*)'file_name for the polynomial stored in the file filename'
read(*,*)ch
usr=.false.
if(ch=='0')then  ! x^n-1
write(*,*)'n='
read(*,*)n
   allocate(p(0:n),z(n))
   p=0.d0
   p=0.d0
   p(n)=1
   p(0)=-1
else if(ch=='1')then  ! x^n+ (100x-1)^3
   write(*,*)'n='
   read(*,*)n
   allocate(p(0:n),z(n))
   p=0.d0
   p(n)=1
   p(0)=-1
   p(1)=300
   p(2)=-30000
   p(3)=1000000
else if(ch=='2')then  ! x^n+ 10^100 x^(n-3)+ 10^100 x^3-10^(-200)
   write(*,*)'n='
   read(*,*)n
   allocate(p(0:n),z(n))
   p=0.d0
   p(n)=1
   p(n-3)=1.d100
   p(3)=1.d100
   p(0)=1.d-200
else if(ch=='usr') then ! user polynomial
   usr=.true.
   write(*,*)'n=    (n=2^k-1)'
   read(*,*)n
   allocate(p(0:n),z(n))
else
   open(file=ch,unit=4)
   read(4,*)n
```

```
     allocate(p(0:n),z(n))
     do i=0,n
        read(4,*) aux
        p(i) = aux
     end do
  end if

  call psolve(n, p, usr, z)
  open(unit=3, file='zeros')
  do i=1,n
     write(3,*)real(z(i)),aimag(z(i))
  end do
end program driver
```

pzeros.f90

```
MODULE poly_zeroes
IMPLICIT NONE
PRIVATE :: aberth, newton, start, cnvex, cmerge, left, right, ctest
PUBLIC  :: polzeros
INTEGER, PARAMETER     :: dp = SELECTED_REAL_KIND(15, 60)

CONTAINS

!*************************************************************************
!    NUMERICAL COMPUTATION OF THE ROOTS OF A POLYNOMIAL HAVING          *
!        COMPLEX COEFFICIENTS, BASED ON ABERTH'S METHOD.                *
!                       Version 1.4, June   1996                        *
!    (D. Bini, Dipartimento di Matematica, Universita' di Pisa)         *
!                          (bini@dm.unipi.it)                           *
!
!  ***********************************************************************
!  * All the software  contained in this library  is protected by copyright. *
!  * Permission  to use, copy, modify, and  distribute this software for any *
!  * purpose without fee is hereby granted, provided that this entire notice *
!  * is included  in all copies  of any software which is or includes a copy *
!  * or modification  of this software  and in all copies  of the supporting *
!  * documentation for such software.                                        *
!  ***********************************************************************
!  * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED *
!  * WARRANTY. IN NO EVENT, NEITHER  THE AUTHORS, NOR THE PUBLISHER, NOR ANY *
!  * MEMBER  OF THE EDITORIAL BOARD OF  THE JOURNAL  "NUMERICAL ALGORITHMS", *
!  * NOR ITS EDITOR-IN-CHIEF, BE  LIABLE FOR ANY ERROR  IN THE SOFTWARE, ANY *
!  * MISUSE  OF IT  OR ANY DAMAGE ARISING OUT OF ITS USE. THE ENTIRE RISK OF *
!  * USING THE SOFTWARE LIES WITH THE PARTY DOING SO.                        *
```

```
!  ************************************************************************
!  * ANY USE  OF THE SOFTWARE  CONSTITUTES  ACCEPTANCE  OF THE TERMS  OF THE *
!  * ABOVE STATEMENT.                                                       *
!  ************************************************************************
!
!   AUTHOR:
!
!       DARIO ANDREA BINI
!       UNIVERSITY OF PISA, ITALY
!       E-MAIL: bini@dm.unipi.it
!
!   REFERENCE:
!
!    -  NUMERICAL COMPUTATION OF POLYNOMIAL ZEROS BY MEANS OF
!       ABERTH'S METHOD
!       NUMERICAL ALGORITHMS, 13 (1996), PP. 179-200
!
!   This version, which is compatible with Lahey's free ELF90 compiler
!   by Alan Miller, CSIRO Mathematical & Information Sciences,
!   Private Bag 10, Clayton South MDC, Victoria, Australia 3169.
!   Alan.Miller @ mel.dms.csiro.au    http://www.mel.dms.csiro.au/~alan
!   Latest revision of ELF90 version - 5 May 1997
!
!************************************************************************
! Work performed under the support of the ESPRIT BRA project 6846 POSSO *
!************************************************************************
!**********         SUBROUTINES AND FUNCTIONS                **********
!************************************************************************
!  The following routines/functions are listed:                         *
!  POLZEROS  :  computes polynomial roots by means of Aberth's method   *
!    ABERTH  :  computes the Aberth correction                          *
!    NEWTON  :  computes p(x)/p'(x) by means of Ruffini-Horner's rule   *
!    START   :  Selects N starting points by means of Rouche's theorem  *
!    CNVEX   :  Computes the convex hull, used by START                 *
!    CMERGE  :  Used by CNVEX                                           *
!    LEFT    :  Used by CMERGE                                          *
!    RIGHT   :  Used by CMERGE                                          *
!    CTEST   :  Convexity test, Used by CMERGE                          *
!************************************************************************
!                                                                       *
!                                                                       *
!************************************************************************
!********************** SUBROUTINE POLZEROS ****************************
!************************************************************************
!                          GENERAL COMMENTS                             *
!************************************************************************
```

```
!  This routine approximates the roots of   the  polynomial            *
!  p(x)=a(n+1)x^n+a(n)x^(n-1)+...+a(1), a(j)=cr(j)+I ci(j), I**2=-1,    *
!  where a(1) and a(n+1) are nonzero.                                   *
!  The coefficients are complex*16 numbers. The routine is fast, robust *
!  against overflow, and allows to deal with polynomials of any degree. *
!  Overflow situations are very unlikely and may occurr if there exist  *
!  simultaneously coefficients of moduli close to BIG and close to      *
!  SMALL, i.e., the greatest and the smallest positive real*8 numbers,  *
!  respectively. In this limit situation the program outputs a warning  *
!  message. The computation can be speeded up by performing some side   *
!  computations in single precision, thus slightly reducing the         *
!  robustness of the program (see the comments in the routine ABERTH).  *
!  Besides a set of approximations to the roots, the program delivers a *
!  set of a-posteriori error bounds which are guaranteed in the most    *
!  part of cases. In the situation where underflow does not allow to    *
!  compute a guaranteed bound, the program outputs a warning message    *
!  and sets the bound to 0. In the situation where the root cannot be   *
!  represented as a complex*16 number the error bound is set to -1.     *
!***********************************************************************
!  The computation is performed by means of Aberth's method             *
!  according to the formula                                             *
!           x(i)=x(i)-newt/(1-newt*abcorr), i=1,...,n             (1)   *
!  where newt=p(x(i))/p'(x(i)) is the Newton correction and abcorr=     *
!  =1/(x(i)-x(1))+...+1/(x(i)-x(i-1))+1/(x(i)-x(i+1))+...+1/(x(i)-x(n)) *
!  is the Aberth correction to the Newton method.                       *
!***********************************************************************
!  The value of the Newton correction is computed by means of the       *
!  synthetic division algorithm (Ruffini-Horner's rule) if |x|<=1,      *
!  otherwise the following more robust (with respect to overflow)       *
!  formula is applied:                                                  *
!                    newt=1/(n*y-y**2 R'(y)/R(y))                (2)   *
!  where                                                               *
!                    y=1/x                                            *
!                    R(y)=a(1)*y**n+...+a(n)*y+a(n+1)            (2')  *
!  This computation is performed by the routine NEWTON.                 *
!***********************************************************************
!  The starting approximations are complex numbers that are             *
!  equispaced on circles of suitable radii. The radius of each          *
!  circle, as well as the number of roots on each circle and the        *
!  number of circles, is determined by applying Rouche's theorem        *
!  to the functions a(k+1)*x**k and p(x)-a(k+1)*x**k, k=0,...,n.        *
!  This computation is performed by the routine START.                  *
!***********************************************************************
!                              STOP CONDITION                           *
!***********************************************************************
! If the condition                                                      *
```

```
!                     |p(x(j))|<EPS s(|x(j)|)                      (3)  *
! is satisfied,    where      s(x)=s(1)+x*s(2)+...+x**n * s(n+1),       *
! s(i)=|a(i)|*(1+3.8*(i-1)),  EPS is the machine precision (EPS=2**-53  *
! for the IEEE arithmetic), then the approximation x(j) is not updated  *
! and the subsequent iterations (1)  for i=j are skipped.               *
! The program stops if the condition (3) is satisfied for j=1,...,n,    *
! or if the maximum number NITMAX of  iterations   has   been reached.  *
! The condition (3) is motivated by a backward rounding error analysis  *
! of the Ruffini-Horner rule, moreover the condition (3) guarantees     *
! that the computed approximation x(j) is an exact root of a slightly   *
! perturbed polynomial.                                                 *
!************************************************************************
!             INCLUSION DISKS, A-POSTERIORI ERROR BOUNDS                *
!************************************************************************
! For each approximation x of a root, an a-posteriori absolute error    *
! bound r is computed according to the formula                          *
!                   r=n(|p(x)|+EPS s(|x|))/|p'(x)|                 (4)  *
! This provides an inclusion disk of center x and radius r containing a *
! root.                                                                 *
!************************************************************************
!************************************************************************
!*************       MEANING OF THE INPUT VARIABLES         *************
!************************************************************************
!************************************************************************
!                                                                       *
!  -- N     : degree of the polynomial.                                 *
!  -- POLY  : complex vector of N+1 components, POLY(i) is the          *
!           coefficient of x**(i-1), i=1,...,N+1 of the polynomial p(x) *
!  -- EPS   : machine precision of the floating point arithmetic used   *
!            by the computer, EPS=2**(-53)  for the IEEE standard.      *
!  -- BIG   : the max real*8, BIG=2**1023 for the IEEE standard.        *
!  -- SMALL : the min positive real*8, SMALL=2**(-1074) for the IEEE.   *
!  -- NITMAX: the max number of allowed iterations.                     *
!************************************************************************
!************************************************************************
!*************      MEANING OF THE OUTPUT VARIABLES         *************
!************************************************************************
!************************************************************************
!  ROOT   : complex vector of N components, containing the              *
!           approximations to the roots of p(x).                        *
!  RADIUS : real vector of N components, containing the error bounds to *
!           the approximations of the roots, i.e. the disk of center    *
!           ROOT(i) and radius RADIUS(i) contains a root of p(x), for   *
!           i=1,...,N. RADIUS(i) is set to -1 if the corresponding root *
!           cannot be represented as floating point due to overflow or  *
!           underflow.                                                  *
```

```
!  ERR    : vector of (N+1) components detecting an error condition;    *
!           ERR(j)=.TRUE. if after NITMAX iterations the stop condition *
!                         (3) is not satisfied for x(j)=ROOT(j);        *
!           ERR(j)=.FALSE.  otherwise, i.e., the root is reliable,      *
!                         i.e., it can be viewed as an exact root of a  *
!                         slightly perturbed polynomial.                *
!           The vector ERR is used also in the routine convex hull for  *
!           storing the abscissae of the vertices of the convex hull.   *
!           ERR(N+1) is only used for the convex hull.                  *
!  ITER   : number of iterations peformed.                              *
!**********************************************************************
!**********************************************************************
!************    MEANING OF THE AUXILIARY VARIABLES         ************
!**********************************************************************
!**********************************************************************
!  APOLY  : real vector of N+1 components used to store the moduli of   *
!           the coefficients of p(x) and the coefficients of s(x) used  *
!           to test the stop condition (3).                             *
!  APOLYR : real vector of N+1 components used to test the stop         *
!           condition                                                   *
!**********************************************************************
!*****         WARNING:   2 is the output unit                    *******
!**********************************************************************
SUBROUTINE polzeros (n, poly, eps, big, small, nitmax, root, radius, err, &
                     iter)
IMPLICIT NONE
INTEGER, INTENT(IN)            :: n, nitmax
INTEGER, INTENT(OUT)           :: iter
COMPLEX (KIND=dp), INTENT(IN)  :: poly(:)
COMPLEX (KIND=dp), INTENT(OUT) :: root(:)
REAL (KIND=dp), INTENT(OUT)    :: radius(:)
REAL (KIND=dp), INTENT(IN)     :: eps, small, big
LOGICAL, INTENT(OUT)           :: err(:)

! Local variables
INTEGER                   :: i, nzeros
COMPLEX (KIND=dp)         :: corr, abcorr
REAL (KIND=dp)            :: amax, apoly(n+1), apolyr(n+1)
REAL (KIND=dp), PARAMETER :: zero = 0.0_dp

! Check consistency of data
IF (ABS(poly(n+1)) == zero)THEN
  WRITE(*,*)'Inconsistent data: the leading coefficient is zero'
  STOP
END IF
IF (ABS(poly(1)) == zero) THEN
```

```
  WRITE(*,*)'The constant term is zero: deflate the polynomial'
  STOP
END IF

! Compute the moduli of the coefficients
amax = 0
DO i = 1, n+1
  apoly(i) = ABS(poly(i))
  amax = MAX(amax, apoly(i))
  apolyr(i) = apoly(i)
END DO
IF((amax) >= (big/(n+1))) THEN
  WRITE(*,*)'WARNING: COEFFICIENTS TOO BIG, OVERFLOW IS LIKELY'
  WRITE(2,*)'WARNING: COEFFICIENTS TOO BIG, OVERFLOW IS LIKELY'
END IF
! Initialize
DO i = 1, n
  radius(i) = zero
  err(i) = .true.
END DO

! Select the starting points
CALL start(n, apolyr, root, radius, nzeros, small, big, err)

! Compute the coefficients of the backward-error polynomial
DO i = 1, n+1
  apolyr(n-i+2) = eps*apoly(i)*(3.8*(n-i+1) + 1)
  apoly(i) = eps*apoly(i)*(3.8*(i-1) + 1)
END DO
IF((apoly(1) == 0).OR.(apoly(n+1) == 0)) THEN
  WRITE(*,*)'WARNING: THE COMPUTATION OF SOME INCLUSION RADIUS'
  WRITE(*,*)'MAY FAIL. THIS IS REPORTED BY RADIUS = 0'
  WRITE(2,*)'WARNING: THE COMPUTATION OF SOME INCLUSION RADIUS'
  WRITE(2,*)'MAY FAIL. THIS IS REPORTED BY RADIUS = 0'
END IF
DO i = 1,n
  err(i) = .true.
  IF(radius(i) == -1) err(i) = .false.
END DO

! Starts Aberth's iterations
DO iter = 1, nitmax
  DO i = 1, n
    IF (err(i)) THEN
      CALL newton(n, poly, apoly, apolyr, root(i), small, radius(i), corr, &
                  err(i))
```

```
      IF (err(i)) THEN
        CALL aberth(n, i, root, abcorr)
        root(i) = root(i) - corr/(1 - corr*abcorr)
      ELSE
        nzeros = nzeros + 1
        IF (nzeros == n) RETURN
      END IF
    END IF
  END DO
END DO
RETURN
END SUBROUTINE polzeros



!*****************************************************************************
!                              SUBROUTINE NEWTON                             *
!*****************************************************************************
! Compute  the Newton's correction, the inclusion radius (4) and checks *
! the stop condition (3)                                                 *
!*****************************************************************************
! Input variables:                                                       *
!     N     : degree of the polynomial p(x)                              *
!     POLY  : coefficients of the polynomial p(x)                        *
!     APOLY : upper bounds on the backward perturbations on the          *
!             coefficients of p(x) when applying Ruffini-Horner's rule   *
!     APOLYR: upper bounds on the backward perturbations on the          *
!             coefficients of p(x) when applying (2), (2')               *
!     Z     : value at which the Newton correction is computed           *
!     SMALL : the min positive REAL (KIND=dp) ::, SMALL=2**(-1074) for the IEEE.   *
!*****************************************************************************
! Output variables:                                                      *
!     RADIUS: upper bound to the distance of Z from the closest root of *
!             the polynomial computed according to (4).                  *
!     CORR  : Newton's correction                                        *
!     AGAIN : this variable is .true. if the computed value p(z) is      *
!             reliable, i.e., (3) is not satisfied in Z. AGAIN is        *
!             .false., otherwise.                                        *
!*****************************************************************************
SUBROUTINE newton(n, poly, apoly, apolyr, z, small, radius, corr, again)
IMPLICIT NONE
INTEGER, INTENT(IN)            :: n
COMPLEX (KIND=dp), INTENT(IN)  :: poly(:), z
COMPLEX (KIND=dp), INTENT(OUT) :: corr
REAL (KIND=dp), INTENT(IN)     :: apoly(:), apolyr(:), small
```

```
REAL (KIND=dp), INTENT(OUT)    :: radius
LOGICAL, INTENT(OUT)           :: again

! Local variables
INTEGER           :: i
COMPLEX (KIND=dp) :: p, p1, zi, den, ppsp
REAL (KIND=dp)    :: ap, az, azi, absp

az = ABS(z)
! If |z|<=1 then apply Ruffini-Horner's rule for p(z)/p'(z)
! and for the computation of the inclusion radius
IF(az <= 1)THEN
  p = poly(n+1)
  ap = apoly(n+1)
  p1 = p
  DO i = n, 2, -1
    p = p*z + poly(i)
    p1 = p1*z + p
    ap = ap*az + apoly(i)
  END DO
  p = p*z + poly(1)
  ap = ap*az + apoly(1)
  corr = p/p1
  absp = ABS(p)
  ap = ap
  again = (absp > (small + ap))
  IF(.NOT.again) radius = n*(absp + ap)/ABS(p1)
  RETURN
ELSE
! If |z| > 1 then apply Ruffini-Horner's rule to the reversed polynomial
! and use formula (2) for p(z)/p'(z). Analogously do for the inclusion radius.
  zi = 1/z
  azi = 1/az
  p = poly(1)
  p1 = p
  ap = apolyr(n+1)
  DO i = n, 2, -1
    p = p*zi + poly(n-i+2)
    p1 = p1*zi + p
    ap = ap*azi + apolyr(i)
  END DO
  p = p*zi + poly(n+1)
  ap = ap*azi + apolyr(1)
  absp = ABS(p)
  again = (absp > (small+ap))
  ppsp = (p*z)/p1
```

```
  den = n*ppsp - 1
  corr = z*(ppsp/den)
  IF(again)RETURN
  radius = ABS(ppsp) + (ap*az)/ABS(p1)
  radius = n*radius/ABS(den)
  radius = radius*az
END IF
RETURN
END SUBROUTINE newton



!***************************************************************************
!                              SUBROUTINE ABERTH                           *
!***************************************************************************
! Compute  the Aberth correction. To save time, the reciprocation of       *
! ROOT(J)-ROOT(I) could be performed in single precision (complex*8)       *
! In principle this might cause overflow if both ROOT(J) and ROOT(I)       *
! have too small moduli.                                                   *
!***************************************************************************
! Input variables:                                                         *
!     N     : degree of the polynomial                                     *
!     ROOT  : vector containing the current approximations to the roots    *
!     J     : index of the component of ROOT with respect to which the     *
!             Aberth correction is computed                                *
!***************************************************************************
! Output variable:                                                         *
!     ABCORR: Aberth's correction (compare (1))                            *
!***************************************************************************
SUBROUTINE aberth(n, j, root, abcorr)
IMPLICIT NONE
INTEGER, INTENT(IN)             :: n, j
COMPLEX (KIND=dp), INTENT(IN)  :: root(:)
COMPLEX (KIND=dp), INTENT(OUT) :: abcorr

! Local variables
INTEGER                    :: i
COMPLEX (KIND=dp)          :: z, zj
REAL (KIND=dp), PARAMETER :: zero = 0.0_dp

abcorr = CMPLX(zero, zero, KIND=dp)
zj = root(j)
DO i = 1, j-1
  z = zj - root(i)
  abcorr = abcorr + 1/z
END DO
```

```
DO i = j+1, n
  z = zj - root(i)
  abcorr = abcorr + 1/z
END DO
RETURN
END SUBROUTINE aberth



!***********************************************************************
!                              SUBROUTINE START                        *
!***********************************************************************
! Compute  the starting approximations of the roots                    *
!***********************************************************************
! Input variables:                                                     *
!     N     :  number of the coefficients of the polynomial            *
!     A     :  moduli of the coefficients of the polynomial            *
!     SMALL : the min positive REAL (KIND=dp) ::, SMALL=2**(-1074) for the IEEE.   *
!     BIG   : the max REAL (KIND=dp) ::, BIG=2**1023 for the IEEE standard.        *
! Output variables:                                                    *
!     Y     :  starting approximations                                 *
!     RADIUS:  if a component is -1 then the corresponding root has a  *
!              too big or too small modulus in order to be represented *
!              as double float with no overflow/underflow              *
!     NZ    :  number of roots which cannot be represented without     *
!              overflow/underflow                                      *
! Auxiliary variables:                                                 *
!     H     :  needed for the computation of the convex hull           *
!***********************************************************************
! This routines selects starting approximations along circles center at *
! 0 and having suitable radii. The computation of the number of circles *
! and of the corresponding radii is performed by computing the upper    *
! convex hull of the set (i,log(A(i))), i=1,...,n+1.                    *
!***********************************************************************
SUBROUTINE start(n, a, y, radius, nz, small, big, h)
IMPLICIT NONE
INTEGER, INTENT(IN)            :: n
INTEGER, INTENT(OUT)           :: nz
LOGICAL, INTENT(OUT)           :: h(:)
COMPLEX (KIND=dp), INTENT(OUT) :: y(:)
REAL (KIND=dp), INTENT(IN)     :: small, big
REAL (KIND=dp), INTENT(IN OUT) :: a(:)
REAL (KIND=dp), INTENT(OUT)    :: radius(:)

! Local variables
INTEGER                   :: i, iold, nzeros, j, jj
```

```
REAL (KIND=dp)            :: r, th, ang, temp, xsmall, xbig
REAL (KIND=dp), PARAMETER :: pi2 = 6.2831853071796, sigma = 0.7

xsmall = LOG(small)
xbig = LOG(big)
nz = 0
! Compute the logarithm A(I) of the moduli of the coefficients of
! the polynomial and then the upper covex hull of the set (A(I),I)
DO i = 1, n+1
  IF(a(i) /= 0) THEN
    a(i) = LOG(a(i))
  ELSE
    a(i) = -1.d30
  END IF
END DO
CALL cnvex(n+1, a, h)
! Given the upper convex hull of the set (A(I),I) compute the moduli
! of the starting approximations by means of Rouche's theorem
iold = 1
th = pi2/n
DO i = 2, n+1
  IF (h(i)) THEN
    nzeros = i - iold
    temp = (a(iold) - a(i))/nzeros
! Check if the modulus is too small
    IF((temp < -xbig).AND.(temp >= xsmall))THEN
      WRITE(*,*)'WARNING:',nzeros,' ZERO(S) ARE TOO SMALL TO'
      WRITE(*,*)'REPRESENT THEIR INVERSES AS COMPLEX (KIND=dp) ::, THEY'
      WRITE(*,*)'ARE REPLACED BY SMALL NUMBERS, THE CORRESPONDING'
      WRITE(*,*)'RADII ARE SET TO -1'
      WRITE(2,*)'WARNING:',nzeros,' ZERO(S) ARE TOO SMALL TO '
      WRITE(2,*)'REPRESENT THEIR INVERSES AS COMPLEX (KIND=dp) ::, THEY'
      WRITE(2,*)'ARE REPLACED BY SMALL NUMBERS, THE CORRESPONDING'
      WRITE(2,*)'RADII ARE SET TO -1'
      nz = nz + nzeros
      r = 1.0D0/big
    END IF
    IF(temp < xsmall)THEN
      nz = nz + nzeros
      WRITE(*,*)'WARNING: ',nzeros,' ZERO(S) ARE TOO SMALL TO BE'
      WRITE(*,*)'REPRESENTED AS COMPLEX (KIND=dp) ::, THEY ARE SET TO 0'
      WRITE(*,*)'THE CORRESPONDING RADII ARE SET TO -1'
      WRITE(2,*)'WARNING: ',nzeros,' ZERO(S) ARE TOO SMALL TO BE'
      WRITE(2,*)'REPRESENTED AS COMPLEX (KIND=dp) ::, THEY ARE SET 0'
      WRITE(2,*)'THE CORRESPONDING RADII ARE SET TO -1'
    END IF
```

```
! Check if the modulus is too big
    IF(temp > xbig)THEN
      r = big
      nz = nz + nzeros
      WRITE(*,*)'WARNING: ', nzeros, ' ZEROS(S) ARE TOO BIG TO BE'
      WRITE(*,*)'REPRESENTED AS COMPLEX (KIND=dp) ::,'
      WRITE(*,*)'THE CORRESPONDING RADII ARE SET TO -1'
      WRITE(2,*)'WARNING: ',nzeros, ' ZERO(S) ARE TOO BIG TO BE'
      WRITE(2,*)'REPRESENTED AS COMPLEX (KIND=dp) ::,'
      WRITE(2,*)'THE CORRESPONDING RADII ARE SET TO -1'
    END IF
    IF((temp <= xbig).AND.(temp > MAX(-xbig, xsmall)))THEN
      r = EXP(temp)
    END IF
! Compute NZEROS approximations equally distributed in the disk of
! radius R
    ang = pi2/nzeros
    DO j = iold, i-1
      jj = j-iold+1
      IF((r <= (1.0D0/big)).OR.(r == big)) radius(j) = -1
      y(j) = r*(COS(ang*jj + th*i + sigma) + (0,1)*SIN(ang*jj + th*i + sigma))
    END DO
    iold = i
  END IF
END DO
RETURN
END SUBROUTINE start


!*********************************************************************
!                              SUBROUTINE CNVEX                      *
!*********************************************************************
! Compute  the upper convex hull of the set (i,a(i)), i.e., the set of  *
! vertices (i_k,a(i_k)), k=1,2,...,m, such that the points (i,a(i)) lie *
! below the straight lines passing through two consecutive vertices.    *
! The abscissae of the vertices of the convex hull equal the indices of *
! the TRUE  components of the logical output vector H.                  *
! The used method requires O(nlog n) comparisons and is based on a      *
! divide-and-conquer technique. Once the upper convex hull of two       *
! contiguous sets  (say, {(1,a(1)),(2,a(2)),...,(k,a(k))} and           *
! {(k,a(k)), (k+1,a(k+1)),...,(q,a(q))}) have been computed, then       *
! the upper convex hull of their union is provided by the subroutine    *
! CMERGE. The program starts with sets made up by two consecutive       *
! points, which trivially constitute a convex hull, then obtains sets   *
! of 3,5,9... points,  up to  arrive at the entire set.                 *
! The program uses the subroutine  CMERGE; the subroutine CMERGE uses   *
```

```
! the subroutines LEFT, RIGHT and CTEST. The latter tests the convexity *
! of the angle formed by the points (i,a(i)), (j,a(j)), (k,a(k)) in the *
! vertex (j,a(j)) up to within a given tolerance TOLER, where i<j<k.    *
!************************************************************************
SUBROUTINE cnvex(n, a, h)
IMPLICIT NONE
INTEGER, INTENT(IN)        :: n
LOGICAL, INTENT(OUT)       :: h(:)
REAL (KIND=dp), INTENT(IN) :: a(:)

! Local variables
INTEGER :: i, j, k, m, nj, jc

h(1:n) = .true.

! compute K such that N-2 <= 2**K < N-1
k = INT(LOG(n-2.0D0)/LOG(2.0D0))
IF(2**(k+1) <= (n-2)) k = k+1

! For each M=1,2,4,8,...,2**K, consider the NJ pairs of consecutive
! sets made up by M+1 points having the common vertex
! (JC,A(JC)), where JC=M*(2*J+1)+1 and J=0,...,NJ,
! NJ = MAX(0, INT((N-2-M)/(M+M))).
! Compute the upper convex hull of their union by means of subroutine CMERGE
m = 1
DO i = 0, k
  nj = MAX(0, INT((n-2-m)/(m+m)))
  DO j = 0, nj
    jc = (j+j+1)*m+1
    CALL cmerge(n, a, jc, m, h)
  END DO
  m = m+m
END DO
RETURN
END SUBROUTINE cnvex



!************************************************************************
!                              SUBROUTINE LEFT                           *
!************************************************************************
! Given as input the integer I and the vector H of logical, compute the *
! the maximum integer IL such that IL<I and H(IL) is TRUE.              *
!************************************************************************
! Input variables:                                                      *
!     H   : vector of logical                                           *
```

```
!     I   : integer                                                          *
!************************************************************************
! Output variable:                                                        *
!     IL  : maximum integer such that IL<I, H(IL)=.TRUE.                   *
!************************************************************************
SUBROUTINE left(h, i, il)
IMPLICIT NONE
INTEGER, INTENT(IN)  :: i
INTEGER, INTENT(OUT) :: il
LOGICAL, INTENT(IN)  :: h(:)

DO il = i-1, 0, -1
  IF (h(il)) RETURN
END DO
RETURN
END SUBROUTINE left



!************************************************************************
!                              SUBROUTINE RIGHT                          *
!************************************************************************
!************************************************************************
! Given as input the integer I and the vector H of logical, compute the *
! the minimum integer IR such that IR>I and H(IL) is TRUE.              *
!************************************************************************
!************************************************************************
! Input variables:                                                        *
!     N   : length of the vector H                                        *
!     H   : vector of logical                                             *
!     I   : integer                                                       *
!************************************************************************
! Output variable:                                                        *
!     IR  : minimum integer such that IR>I, H(IR)=.TRUE.                   *
!************************************************************************
SUBROUTINE right(n, h, i, ir)
IMPLICIT NONE
INTEGER, INTENT(IN)  :: n, i
INTEGER, INTENT(OUT) :: ir
LOGICAL, INTENT(IN)  :: h(:)

DO ir = i+1, n
  IF (h(ir)) RETURN
END DO
RETURN
END SUBROUTINE right
```

```
!*********************************************************************
!                              SUBROUTINE CMERGE                      *
!*********************************************************************
! Given the upper convex hulls of two consecutive sets of pairs       *
! (j,A(j)), compute the upper convex hull of their union              *
!*********************************************************************
! Input variables:                                                    *
!     N    : length of the vector A                                   *
!     A    : vector defining the points (j,A(j))                      *
!     I    : abscissa of the common vertex of the two sets            *
!     M    : the number of elements of each set is M+1                *
!*********************************************************************
! Input/Output variable:                                              *
!     H    : vector defining the vertices of the convex hull, i.e.,   *
!            H(j) is .TRUE. if (j,A(j)) is a vertex of the convex hull *
!            This vector is used also as output.                      *
!*********************************************************************
SUBROUTINE cmerge(n, a, i, m, h)
IMPLICIT NONE
INTEGER, INTENT(IN)        :: n, m, i
LOGICAL, INTENT(IN OUT)    :: h(:)
REAL (KIND=dp), INTENT(IN) :: a(:)

! Local variables
INTEGER :: ir, il, irr, ill
LOGICAL :: tstl, tstr

! at the left and the right of the common vertex (I,A(I)) determine
! the abscissae IL,IR, of the closest vertices of the upper convex
! hull of the left and right sets, respectively
CALL left(h, i, il)
CALL right(n, h, i, ir)

! check the convexity of the angle formed by IL,I,IR
IF (ctest(a, il, i, ir)) THEN
  RETURN
ELSE
! continue the search of a pair of vertices in the left and right
! sets which yield the upper convex hull
  h(i) = .false.
  DO
    IF (il == (i-m)) THEN
      tstl = .true.
```

```
    ELSE
      CALL left(h, il, ill)
      tstl = ctest(a, ill, il, ir)
    END IF
    IF (ir == MIN(n, i+m)) THEN
      tstr = .true.
    ELSE
      CALL right(n, h, ir, irr)
      tstr = ctest(a, il, ir, irr)
    END IF
    h(il) = tstl
    h(ir) = tstr
    IF (tstl.AND.tstr) RETURN
    IF(.NOT.tstl) il = ill
    IF(.NOT.tstr) ir = irr
  END DO
END IF

RETURN
END SUBROUTINE cmerge



!*****************************************************************************
!                              FUNCTION CTEST                                *
!*****************************************************************************
! Test the convexity of the angle formed by (IL,A(IL)), (I,A(I)),            *
! (IR,A(IR)) at the vertex (I,A(I)), up to within the tolerance              *
! TOLER. If convexity holds then the function is set to .TRUE.,              *
! otherwise CTEST=.FALSE. The parameter TOLER is set to 0.4 by default.      *
!*****************************************************************************
! Input variables:                                                           *
!     A       : vector of double                                             *
!     IL,I,IR : integers such that IL < I < IR                               *
!*****************************************************************************
! Output:                                                                    *
!     .TRUE. if the angle formed by (IL,A(IL)), (I,A(I)), (IR,A(IR)) at      *
!            the vertex (I,A(I)), is convex up to within the tolerance       *
!            TOLER, i.e., if                                                 *
!            (A(I)-A(IL))*(IR-I)-(A(IR)-A(I))*(I-IL)>TOLER.                  *
!     .FALSE.,  otherwise.                                                   *
!*****************************************************************************
FUNCTION ctest(a, il, i, ir) RESULT(OK)
IMPLICIT NONE
INTEGER, INTENT(IN)        :: i, il, ir
REAL (KIND=dp), INTENT(IN) :: a(:)
```

```
LOGICAL                    :: OK

! Local variables
REAL (KIND=dp)             :: s1, s2
REAL (KIND=dp), PARAMETER :: toler = 0.4D0

s1 = a(i) - a(il)
s2 = a(ir) - a(i)
s1 = s1*(ir-i)
s2 = s2*(i-il)
OK = .false.
IF(s1 > (s2+toler)) OK = .true.
RETURN
END FUNCTION ctest


END MODULE poly_zeroes
```

pzeros_drv.f90

```
program driver
  use poly_zeroes
  implicit none

  integer                                          :: n, i,  nsol,nitmax,iter
  character(len=25)                                :: ch
  real(kind(0.d0))                                 :: aux,eps,small,big
  complex(kind(0.d0)), dimension(:), allocatable :: z, p
  logical                                          :: usr
real(kind=dp),dimension(:),allocatable::rad
logical,dimension(:),allocatable::er

  write(*,*)'type:'
  write(*,*)'  usr for the user (Mandelbrot) polynomial'
  write(*,*)'  0   for the polynomial z^n-1'
  write(*,*)'  1   for the polynomial z^n+(100z-1)^3'
  write(*,*)'file_name for the polynomial stored in the file filename'
  read(*,*)ch
  usr=.false.
  if(ch=='0')then  ! x^n-1
  write(*,*)'n='
  read(*,*)n
     allocate(p(0:n),z(n))
     p=0.d0
     p=0.d0
```

```
     p(n)=1
     p(0)=-1
  else if(ch=='1')then  ! x^n+ (100x-1)^3
     write(*,*)'n='
     read(*,*)n
     allocate(p(0:n),z(n))
     p=0.d0
     p(n)=1
     p(0)=-1
     p(1)=300
     p(2)=-30000
     p(3)=1000000
  else if(ch=='2')then  ! x^n+ 10^100 x^(n-3)+ 10^100 x^3-10^(-200)
     write(*,*)'n='
     read(*,*)n
     allocate(p(0:n),z(n))
     p=0.d0
     p(n)=1
     p(n-3)=1.d100
     p(3)=1.d100
     p(0)=1.d-200
  else if(ch=='usr') then ! user polynomial
     usr=.true.
     write(*,*)'n=    (n=2^k-1)'
     read(*,*)n
     allocate(p(0:n),z(n))
  else
     open(file=ch,unit=4)
     read(4,*)n
     allocate(p(0:n),z(n))
     do i=0,n
        read(4,*) aux
        p(i) = aux
     end do
  end if
eps=epsilon(1.0d0)
small=tiny(1.0d0)
big=huge(1.0d0)
nitmax=50
allocate(rad(n),er(n))
call polzeros(n,p,eps,big,small,nitmax,z,rad,er,iter)
  open(unit=3, file='zeros')
  do i=1,n
     write(3,*)real(z(i)),aimag(z(i))
  end do
end program driver
```