Computing multiple roots of inexact polynomials

Zhonggang Zeng
Northeastern Illinois University

Contents

List of Figures

List of Tables

Computing multiple roots of inexact polynomials *

Zhonggang Zeng†

July 14, 2003

Abstract

We present a combination of two novel algorithms that accurately calculate multiple roots of general polynomials. For a given multiplicity structure and initial root estimates, Algorithm I transforms the singular root-finding into a regular nonlinear least squares problem on a pejorative manifold, and calculates multiple roots simultaneously. To fulfill the input requirement of Algorithm I, we employ a numerical GCD-finder containing a successive singular value updating and an iterative GCD refinement, as the main engine of Algorithm II that calculates the multiplicity structure and the initial root approximation. The combined method calculates multiple roots with high accuracy without using multiprecision arithmetic even if the coefficients are inexact. This is perhaps the first blackbox-type root-finder with such capabilities. To measure the sensitivity of the multiple roots, a structure-preserving condition number is proposed and error bounds are given. Extensive computational experiments and the error analysis confirm that a polynomial being ill-conditioned in the conventional sense can be well conditioned with the multiplicity structure being preserved, and its multiple roots can be computed with remarkable accuracy.

1 Introduction

In this paper, we present a combination of two novel numerical algorithms that accurately calculate multiple roots of polynomials with coefficients possibly being inexact without using multiprecision arithmetic.

Polynomial root-finding is among the classical problems with longest and richest history. One of the most difficult issues in root-finding is computing multiple roots. In addition to requiring *exact* coefficients, multiprecision arithmetic may be needed when multiple roots are present [23]. In fact, using multiprecision has been a common practice in designing root-finding algorithms and softwares, such as those in [2, 13, 14]. Moreover, there is a so-called "attainable accuracy" in computing multiple roots [17, 23, 30]: to calculate an m-fold root to the precision of k correct digits, the accuracy of the polynomial coefficients *and* the machine precision must be at least mk digits. This "attainable accuracy" barrier also suggests the need of using multiprecision arithmetic. Multiprecision softwares such as [1] are available. However, when polynomial coefficients are truncated, multiple roots turn into clusters, and extending machine precision will never reverse clusters back to multiple roots. In the absence of accurate methods that are independent of multiprecision technology, multiple roots of perturbed polynomials would indeed be intractable.

*Mathematics Subject Classification 65F35, 65H05

†Dept of Math, Northeastern Illinois University, Chicago, IL 60625, email: zzeng@neiu.edu.

While multiple roots are considered hypersensitive in numerical computation, W. Kahan [19] proved that if the multiplicities are preserved, those roots can actually be well behaved. More precisely, polynomials with a fixed multiplicity structure form a pejorative manifold. A polynomial is ill-conditioned if it is near such a manifold. On the other hand, for the polynomial on the pejorative manifold, its multiple roots are insensitive to multiplicity-preserving perturbations, unless the polynomial is also near a submanifold of higher multiplicities. Therefore, to calculate multiple roots accurately, it is important to maintain the computation on a proper pejorative manifold.

In light of Kahan's theoretical insight, we propose Algorithm I in §3 that transforms the singular root-finding into a regular nonlinear least squares problem on a pejorative manifold. By projecting the given polynomial onto the manifold, the computation remains structure-preserving. As a result, the roots can be calculated simultaneously and accurately.

In applying Algorithm I, one needs to have a priori knowledge on the multiplicity structure of the polynomial and its initial root estimates. For this input requirement, we propose Algorithm II in §4. The algorithm employs a numerical GCD-finder which contains a successive singular value updating on the Sylvester discriminant matrices as well as an iterative refinement strategy for the recursive GCD computation. The resulting algorithm calculates the multiplicity structure and its initial root approximation for a given polynomial.

In §3.3, we propose a structure-preserving condition number that measures the sensitivity of multiple roots. A polynomial that is ill-conditioned in conventional sense can be well conditioned with the multiplicity structure being preserved, and its roots can be calculated far beyond the barrier of "attainable accuracy". This condition number can easily be calculated. Error bounds on the roots are given for inexact polynomials.

In §3.5 and §4.6, we present separate numerical results for Algorithm I and Algorithm II. The numerical results for the combined algorithm are shown in §5. Both algorithms and their combination are implemented as a Matlab package MultRoot which is electronically available from the author[1].

The combined algorithm is accurate, stable and reasonably efficient. Taking the coefficient vector as the *only* input, the output includes the roots and their multiplicities as well as the backward error, the estimated forward error, and the structure-preserving condition number. The most significant features of the algorithm are its remarkable accuracy and its robustness in handling inexact data. As shown in numerical examples, the code accurately identifies the multiplicity structure and multiple roots for polynomials with a coefficient accuracy being as low as 7 digits. With given multiplicities, Algorithm I converges even with data accuracy as low as 3 decimal digits. The code appears to be the first blackbox-type root-finder with such capability.

While numerical experiments reported in the literature rarely reach multiplicity ten, we successfully tested our algorithms on polynomials with root multiplicities as high as 400 without using multiprecision arithmetic. We are aware of no other reliable methods that calculate multiple roots accurately by using standard machine precision. Accurate results for multiple root computation we have seen in the literature, such as the methods of Farmer-Loizou [13],

[1] `http://www.neiu.edu/~zzeng/multroot.htm`

can be repeated only if multiprecision is used on exact polynomials. A zero-finder for general analytic functions with multiple zeros has been developed by Kravanja and Van Barel [21]. The method uses an accuracy refinement with modified Newton's iteration that also requires multiprecision for multiple roots unless the polynomial is already factored [33].

There exist general-purpose root-finders using $O(n^2)$ flops or less, such as those surveyed in [23]. However, the barrier of "attainable accuracy" may prevent those root-finders from calculating multiple roots accurately when the polynomials are inexact (e.g. see Figure 10 in §4.6) even if multiprecision is used. Our algorithms provide an option of reaching high accuracy on multiple roots at higher computing cost of $O(n^3)$ which may not be a lofty price to pay.

The idea of exploiting the pejorative manifold and the problem structure has been applied extensively for ill-conditioned problems. Besides Kahan's pioneer work 30 years ago, theories and computational strategies for the matrix canonical forms have been studied, such as [8, 10, 11, 22], to take advantage of the pejorative manifolds or varieties. At present, it is not clear if those methods can be applied to polynomials with multiple roots.

2 Preliminaries

2.1 Notations

In this paper, $\mathbf{R}^n$ and $\mathbf{C}^n$ denote the n dimensional real and complex vector spaces respectively. Vectors, always considered columns, are denoted by boldface lower case letters and matrices are denoted by upper case letters. Blank entries in a matrix are filled with zeros. The notation $(\cdot)^\top$ represents the transpose of $(\cdot)$, and $(\cdot)^H$ the Hermitian adjoint (i.e. conjugate transpose) of $(\cdot)$. When we use a (lower case) letter, say p, to denote a polynomial of degree n, then $p_0, p_1, \cdots, p_n$ are its coefficients as in

$$p(x) = p_0 x^n + p_1 x^{n-1} + \cdots + p_n.$$

The same letter in boldface (e.g. $\mathbf{p}$) denotes the coefficient (column) vector

$$\mathbf{p} = (\, p_0, p_1, \cdots, p_n \,)^\top$$

unless it is defined otherwise. The degree of p is $deg(p)$. For a pair of polynomials p and q, their greatest common divisor (GCD) is denoted by $GCD(p, q)$.

2.2 Basic definitions and lemmas

Definition 2.1 *Let* $p(x) = p_0 x^n + p_1 x^{n-1} + \cdots + p_n$ *be a polynomial of degree* n. *For any integer* $k > 0$, *the matrix*

$$C_k(p) = \overbrace{\begin{bmatrix} p_0 & & \\ p_1 & \ddots & \\ \vdots & \ddots & p_0 \\ p_n & & p_1 \\ & \ddots & \vdots \\ & & p_n \end{bmatrix}}^{k}$$

is called the k*-th order* **Cauchy matrix** *associated with* p.

Lemma 2.1 *Let f and g be polynomials of degrees n and m respectively with $h(x) = f(x)g(x)$. Then* $\mathbf{h}$ *is the* **convolution** *of* $\mathbf{f}$ *and* $\mathbf{g}$ *defined by*

$$\mathbf{h} = conv(\mathbf{f}, \mathbf{g}) = C_{m+1}(f)\mathbf{g} = C_{n+1}(g)\mathbf{f}.$$

Proof. A straightforward verification. Q.E.D.

Definition 2.2 *Let p be a polynomial of degree n and p' be its derivative. For $k = 1, 2, \cdots n$, the matrix of size $(n + k) \times (2k + 1)$*

$$S_k(p) = \left[\begin{array}{c|c} C_{k+1}(p') & C_k(p) \end{array} \right]$$

is called the k-th **Sylvester discriminant matrix**.

Lemma 2.2 *Let p be a polynomial of degree n and p' be its derivative with $u = GCD(p, p')$. For $j = 1, \cdots, n$, let ς_j be the smallest singular value of $S_j(p)$. Then the following are equivalent*

(a) $deg(u) = m$,

(b) p *has* $k = n - m$ *distinct roots,*

(c) $\varsigma_1, \varsigma_2, \cdots, \varsigma_{k-1} > 0, \qquad \varsigma_k = \varsigma_{k+1} = \cdots = \varsigma_n = 0.$

Proof. The equivalence between (a) and (b) is trivial to verify, and the assertion that (a) is equivalent to (c) is part of Proposition 3.1 in [24]. Q.E.D.

Lemma 2.3 *Let p be a polynomial of degree n and p' be its derivative with $u = GCD(p, p')$ and $deg(u) = m = n - k$. Let v and w be polynomials that satisfy*

$$u(x)v(x) = p(x), \quad u(x)w(x) = p'(x).$$

Then

(a) *v and w are co-prime, namely they have no common factors;*

(b) *the (column) rank of $S_k(p)$ is deficient by one;*

(c) *the normalized vector $\begin{bmatrix} \mathbf{v} \\ -\mathbf{w} \end{bmatrix}$ is the right singular vector of $S_k(p)$ associated with the smallest (zero) singular value ς_k;*

(d) *if $\mathbf{v}$ is known, the coefficient vector $\mathbf{u}$ of $u = GCD(p, p')$ is the solution to the linear system $C_{m+1}(v)\mathbf{u} = \mathbf{p}$.*

Proof. Assertion (a) is trivial. $S_k(p) \begin{bmatrix} \mathbf{v} \\ -\mathbf{w} \end{bmatrix} = C_{k+1}(p')\,\mathbf{v} - C_k(p)\mathbf{w} = 0$ because it is the coefficient vector of $p'v - pw \equiv (uw)v - (uv)w \equiv 0$. Let $\hat{\mathbf{v}} \in \mathbf{C}^{k+1}$ and $\hat{\mathbf{w}} \in \mathbf{C}^k$ be coefficient vectors of polynomials $\hat{v}$ and $\hat{w}$ respectively that also satisfy $C_{k+1}(p')\,\hat{\mathbf{v}} - C_k(p)\hat{\mathbf{w}} = 0$. Then we also have $(uw)\hat{v} - (uv)\hat{w} = 0$, namely $w\hat{v} = v\hat{w}$. Since v and w are co-prime, there is polynomial c such that $\hat{v} = cv$ and $\hat{w} = cw$ and c is obviously a constant. Therefore, the single vector $\begin{bmatrix} \mathbf{v} \\ -\mathbf{w} \end{bmatrix}$ forms the basis for the null space of $S_k(p)$. Consequently, both assertions (b) and (c) follow. The assertion (d) is a direct consequence of Lemma 2.1. Q.E.D.

Lemma 2.4 *Let* $A \in \mathbf{C}^{n\times m}$ *with* $n \geq m$ *be a matrix whose smallest two distinct singular values are* $\hat{\sigma} > \tilde{\sigma}$. *Let* $Q \begin{pmatrix} R \\ 0 \end{pmatrix} = A$ *be the QR decomposition of* A *where* $Q \in \mathbf{C}^{n\times n}$ *is unitary and* $R \in \mathbf{C}^{m\times m}$ *is upper triangular. From any vector* $\mathbf{x}_0 \in \mathbf{C}^m$ *that is not orthogonal to the right singular subspace of* A *associated with* $\tilde{\sigma}$, *we generate the sequences* $\{\sigma_j\}$ *and* $\{\mathbf{x}_j\}$, *by the inverse iteration*

$$\left\{ \begin{array}{lrcll} \text{Solve} & R^H \mathbf{y}_j & = & \mathbf{x}_{j-1} & \text{for } \mathbf{y}_j \in \mathbf{C}^m \\ \text{Solve} & R\,\mathbf{z}_j & = & \mathbf{y}_j & \text{for } \mathbf{z}_j \in \mathbf{C}^m \\ & & & & \\ \text{Calculate} & \mathbf{x}_j & = & \dfrac{\mathbf{z}_j}{\|\mathbf{z}_j\|_2}, & \sigma_j = \left\| R\mathbf{x}_j \right\|_2 \end{array} \right. \qquad j = 1, 2, \cdots. \tag{1}$$

Then $\lim_{j\to\infty} \sigma_j = \lim_{j\to\infty} \|A\mathbf{x}_j\|_2 = \tilde{\sigma}$ *and*

$$\sigma_j = \left\| A\mathbf{x}_j \right\|_2 = \tilde{\sigma} + O\left(\tau^j\right), \quad \text{where } \tau = \left(\frac{\tilde{\sigma}}{\hat{\sigma}}\right)^2.$$

If $\tilde{\sigma}$ *is simple, then* $\mathbf{x}_j$ *converges to the right singular vector* $\tilde{\mathbf{x}}$ *of* A *associated with* $\tilde{\sigma}$.

Proof. See [27] for straightforward verifications. Q.E.D.

2.3 The Gauss-Newton iteration

The Gauss-Newton iteration is an effective method for solving nonlinear least squares problems. Let $G : \mathbf{C}^m \longrightarrow \mathbf{C}^n$ with $n > m$, and $\mathbf{a} \in \mathbf{C}^n$. The nonlinear system $G(\mathbf{z}) = \mathbf{a}$ for $\mathbf{z} \in \mathbf{C}^m$ is overdetermined with no conventional solutions in general. We thereby seek a *weighted least squares solution*. Let $W = diag(\omega_1, \cdots, \omega_n)$ be a diagonal weight matrix with positive weights ω_j's. Let $\left\| \cdot \right\|_W$ denote the weighted 2-norm:

$$\left\| \mathbf{v} \right\|_W \equiv \left\| W\,\mathbf{v} \right\|_2 \equiv \sqrt{\sum_{j=1}^{n} \omega_j^2 v_j^2}, \quad \text{for all} \quad \mathbf{v} = (v_1, \cdots, v_n)^\top \in \mathbf{C}^n. \tag{2}$$

Our objective is to solve the minimization problem $\min_{\mathbf{z}\in\mathbf{C}^m} \left\| G(\mathbf{z}) - \mathbf{a} \right\|_W^2$

Lemma 2.5 *Let* $F : \mathbf{C}^m \longrightarrow \mathbf{C}^n$ *be analytic with Jacobian being* $\mathcal{J}(\mathbf{z})$. *If there is a neighborhood* Ω *of* $\tilde{\mathbf{z}}$ *in* $\mathbf{C}^m$ *such that* $\left\| F(\tilde{\mathbf{z}}) \right\|_2 \leq \left\| F(\mathbf{z}) \right\|_2$ *for all* $\mathbf{z} \in \Omega$, *then* $\mathcal{J}(\tilde{\mathbf{z}})^H F(\tilde{\mathbf{z}}) = 0$.

Proof. The real case $F : \mathbf{R}^m \longrightarrow \mathbf{R}^n$ of the lemma is proved in [9]. The proof for the complex case is nearly identical, except for using the Cauchy-Riemann equation. Q.E.D.

By Lemma 2.5, let $J(\mathbf{z})$ be the Jacobian of $G(\mathbf{z})$. To find a local minimum of $\left\| F(\mathbf{z}) \right\|_2 \equiv \left\| W\Big[G(\mathbf{z}) - \mathbf{a}\Big] \right\|_2$ with $\mathcal{J}(\mathbf{z}) = W J(\mathbf{z})$, we look for $\tilde{\mathbf{z}} \in \mathbf{C}^m$ satisfying

$$\mathcal{J}(\tilde{\mathbf{z}})^H F(\tilde{\mathbf{z}}) = \Big[W J(\tilde{\mathbf{z}})\Big]^H W\Big[G(\tilde{\mathbf{z}}) - \mathbf{a}\Big] = J(\tilde{\mathbf{z}})^H\, W^2 \Big[G(\tilde{\mathbf{z}}) - \mathbf{a}\Big] = 0.$$

In other words, $G(\tilde{\mathbf{z}}) - \mathbf{a}$ is orthogonal, with respect to $\langle \mathbf{v}, \mathbf{w} \rangle \equiv \mathbf{v}^H W^2 \mathbf{w}$, to the tangent plane of the manifold $\Pi = \Big\{ \mathbf{u} = G(\mathbf{z}) \,\Big|\, \mathbf{z} \in \mathbf{C}^m \Big\}$ at $\tilde{\mathbf{u}} = G(\tilde{\mathbf{z}})$.

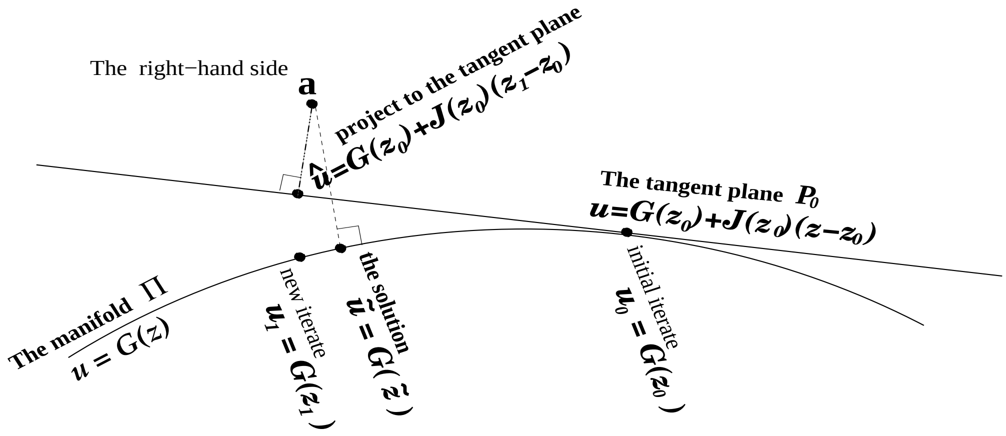

Figure 1: Illustration of the Gauss-Newton iteration

The Gauss-Newton iteration can be derived as follows (see Figure 1). To find a least squares solution $\mathbf{z} = \tilde{\mathbf{z}}$ to the equation $G(\mathbf{z}) = \mathbf{a}$, we look for the point $\tilde{\mathbf{u}} = G(\tilde{\mathbf{z}})$ that is the orthogonal projection of $\mathbf{a}$ onto Π. Let $\mathbf{u}_0 = G(\mathbf{z}_0)$ in Π be near $\tilde{\mathbf{u}} = G(\tilde{\mathbf{z}})$. We can approximate the manifold Π with the tangent plane $P_0 = \left\{ G(\mathbf{z}_0) + J(\mathbf{z}_0)(\mathbf{z} - \mathbf{z}_0) \;\middle|\; \mathbf{z} \in \mathbf{C}^m \right\}$. Then the point $\mathbf{a}$ is orthogonally projected onto the tangent plane P_0 at $\hat{\mathbf{u}} = G(\mathbf{z}_0) + J(\mathbf{z}_0)(\mathbf{z}_1 - \mathbf{z}_0)$ by solving the overdetermined linear system

$$G(\mathbf{z}_0) + J(\mathbf{z}_0)(\mathbf{z} - \mathbf{z}_0) = \mathbf{a} \quad \text{or} \quad J(\mathbf{z}_0)(\mathbf{z} - \mathbf{z}_0) = -[G(\mathbf{z}_0) - \mathbf{a}] \tag{3}$$

for its weighted least squares solution

$$\mathbf{z}_1 = \mathbf{z}_0 - \left[J(\mathbf{z}_0)_W^+\right][G(\mathbf{z}_0) - \mathbf{a}] \;\; \text{with} \;\; J(\mathbf{z}_0)_W^+ = \left[J(\mathbf{z}_0)^H W^2 J(\mathbf{z}_0)\right]^{-1} J(\mathbf{z}_0)^H W^2. \tag{4}$$

As long as $J(\mathbf{z}_0)$ is of full (column) rank, the pseudo inverse $J(\mathbf{z}_0)_W^+$ exists. Therefore $\mathbf{u}_1 = G(\mathbf{z}_1)$ is well defined and is expected to be a better approximation to $\tilde{\mathbf{u}} = G(\tilde{\mathbf{z}})$ than $\mathbf{u}_0 = G(\mathbf{z}_0)$. The Gauss-Newton iteration is then a recursive application of (4) (also see [6, 9]).

The convergence theory of the Gauss-Newton iteration has been well established for overdetermined systems in real spaces [9]. The following lemma is a straightforward generalization of Theorem 10.2.1 in [9] to complex spaces. Since the lemma itself as well as the proof are nearly identical to those in the real case in [9], we shall present the lemma without proof.

Lemma 2.6 *Let $\Omega \subset \mathbf{C}^m$ be a bounded open convex set and $F : D \subset \mathbf{C}^m \longrightarrow \mathbf{C}^n$ be analytic in an open set $D \supset \overline{\Omega}$. Let $\mathcal{J}(\mathbf{z})$ be the Jacobian of $F(\mathbf{z})$. Suppose that there exists $\tilde{\mathbf{z}} \in \Omega$ such that $\mathcal{J}(\tilde{\mathbf{z}})^H F(\tilde{\mathbf{z}}) = 0$ with $\mathcal{J}(\tilde{\mathbf{z}})$ full rank. Let σ be the smallest singular value of $\mathcal{J}(\tilde{\mathbf{z}})$. Let $\delta \geq 0$ be a constant such that*

$$\left\| \left[\mathcal{J}(\mathbf{z}) - \mathcal{J}(\tilde{\mathbf{z}}) \right]^H F(\tilde{\mathbf{z}}) \right\|_2 \leq \delta \left\| \mathbf{z} - \tilde{\mathbf{z}} \right\|_2 \quad \text{for all} \;\; \mathbf{z} \in \Omega. \tag{5}$$

If $\delta < \sigma^2$, then for any $c \in \left(\frac{1}{\sigma}, \frac{\sigma}{\delta}\right)$, there exists $\varepsilon > 0$ such that for all $z_0 \in \Omega$ with $\|\mathbf{z}_0 - \tilde{\mathbf{z}}\|_2 < \varepsilon$, the sequence generated by the Gauss-Newton iteration

$$\mathbf{z}_{k+1} = \mathbf{z}_k - \mathcal{J}(\mathbf{z}_k)^+ F(\mathbf{z}_k), \quad k = 0, 1, \cdots, \quad \text{where} \;\; \mathcal{J}(\mathbf{z}_k)^+ = [\mathcal{J}(\mathbf{z}_k)^H \mathcal{J}(\mathbf{z}_k)]^{-1} \mathcal{J}(\mathbf{z}_k)^H,$$

is well defined inside Ω*, converges to* $\tilde{\mathbf{z}}$*, and satisfies*

$$\left\|\mathbf{z}_{k+1}-\tilde{\mathbf{z}}\right\|_2 \le \frac{c\delta}{\sigma}\left\|\mathbf{z}_k-\tilde{\mathbf{z}}\right\|_2 + \frac{c\alpha\gamma}{2\sigma}\left\|\mathbf{z}_k-\tilde{\mathbf{z}}\right\|_2^2, \tag{6}$$

where $\alpha > 0$ *is the upper bound of* $\|\mathcal{J}(\mathbf{z})\|_2$ *on* $\overline{\Omega}$*, and* $\gamma > 0$ *is the Lipschitz constant of* $\mathcal{J}(\mathbf{z})$ *in* Ω*, namely,* $\left\|\mathcal{J}(\mathbf{z}+\mathbf{h})-\mathcal{J}(\mathbf{z})\right\|_2 \le \gamma\left\|\mathbf{h}\right\|_2$ *for all* $\mathbf{z},\ \mathbf{z}+\mathbf{h} \in \Omega$.

3 Algorithm I: root-finding with given multiplicities

In this section, we assume that the multiplicity structure of a given polynomial is known. We shall deal with the problem of determining this multiplicity structure in §4. A condition number will be introduced to measure the sensitivity of multiple roots. When the condition number is moderate, the multiple roots can be calculated accurately by our algorithm.

3.1 The pejorative manifold

Introduced heuristically by Kahan [19], a *pejorative manifold* is formed by polynomials with roots in a given multiplicity structure. Kahan showed that it may be a misconception to consider multiple roots as infinitely ill-conditioned, because they can be insensitive to perturbations that are constrained on the pejorative manifold. Kahan also suggested a possible approach based on the method of Lagrange multipliers to calculate the minimal distances of the given polynomial to all pejorative manifolds. However, it is not clear whether this approach is implementable in numerical computation. We shall propose a novel computational approach and develop a practical algorithm based, in part, on Kahan's theoretical insight. For this purpose, we need more precise formulation and further analysis of the manifold.

A polynomial of degree n corresponds to a vector (or point) in $\mathbf{C}^n$

$$p(x) = p_0 x^n + p_1 x^{n-1} + \cdots + p_n \quad \sim \quad \mathbf{a} = (a_1, \cdots, a_n)^\top \equiv \left(\frac{p_1}{p_0}, \cdots, \frac{p_n}{p_0}\right)^\top,$$

where "$\sim$" denotes this correspondence. For a partition of n, namely a fixed array of positive integers $\ell_1, \cdots, \ell_m$ with $\ell_1 + \cdots + \ell_m = n$, a polynomial p that has roots $z_1, \cdots, z_m$ with multiplicities $\ell_1, \cdots, \ell_m$ respectively can be written as

$$\frac{1}{p_0}p(x) = \prod_{j=1}^{m}(x-z_j)^{\ell_j} \quad = \quad x^n + \sum_{j=1}^{n} g_j(z_1, \cdots, z_m)\, x^{n-j}, \tag{7}$$

where each g_j is a polynomial in $z_1, \cdots, z_m$. We have the correspondence

$$p \sim G_\ell(\mathbf{z}) \equiv \begin{pmatrix} g_1(z_1,\cdots,z_m) \\ \vdots \\ g_n(z_1,\cdots,z_m) \end{pmatrix} \in \mathbf{C}^n, \qquad \text{where} \quad \mathbf{z} = \begin{pmatrix} z_1 \\ \vdots \\ z_m \end{pmatrix} \in \mathbf{C}^m. \tag{8}$$

We now define the pejorative manifold rigorously based on Kahan's heuristic description.

Definition 3.1 *An ordered array of positive integers* $\ell = [\ell_1, \cdots, \ell_m]$ *is called a* ***multiplicity structure*** *of degree* n *if* $\ell_1 + \cdots + \ell_m = n$*. For any such given multiplicity structure* ℓ*, the collection of vectors* $\Pi_\ell \equiv \left\{ G_\ell(\mathbf{z}) \,\middle|\, \mathbf{z} \in \mathbf{C}^m \right\} \subset \mathbf{C}^n$ *is called the* ***pejorative manifold*** *of multiplicity structure* ℓ*, where* $G_\ell : \mathbf{C}^m \longrightarrow \mathbf{C}^n$ *defined in (7) – (8) is called the* ***coefficient operator*** *associated with the multiplicity structure* ℓ*.*

For example, we consider polynomials of degree 3. First, for multiplicity structure $\ell = [1,2]$,

$$(x - z_1)(x - z_2)^2 = x^3 + (\, -z_1 - 2z_2 \,)\, x^2 + (\, 2z_1 z_2 + z_2^2 \,)\, x + (\, -z_1 z_2^2 \,).$$

A polynomial with one simple root z_1 and one double root z_2 corresponds to the vector

$$G_{[1,2]}(\mathbf{z}) \equiv \begin{pmatrix} -z_1 - 2z_2 \\ 2z_1 z_2 + z_2^2 \\ -z_1 z_2^2 \end{pmatrix} \in \mathbf{C}^3, \quad \text{with} \quad \mathbf{z} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \in \mathbf{C}^2. \tag{9}$$

The vectors $G_{[1,2]}(\mathbf{z})$ in (9) for all $\mathbf{z} \in \mathbf{C}^2$ form the pejorative manifold $\Pi_{[1,2]}$. Similarly,

$$\Pi_{[3]} = \left\{ (-3\,z,\, 3\,z^2,\, -z^3)^\top \;\middle|\; z \in \mathbf{C} \right\}$$

when $\ell = [3]$. $\Pi_{[3]}$ is a submanifold of $\Pi_{[1,2]}$ that contains all polynomials with a single triple root. Figure 2 shows the manifolds $\Pi_{[1,2]}$ (the wings) and $\Pi_{[3]}$ (the sharp edge) in $\mathbf{R}^3$.

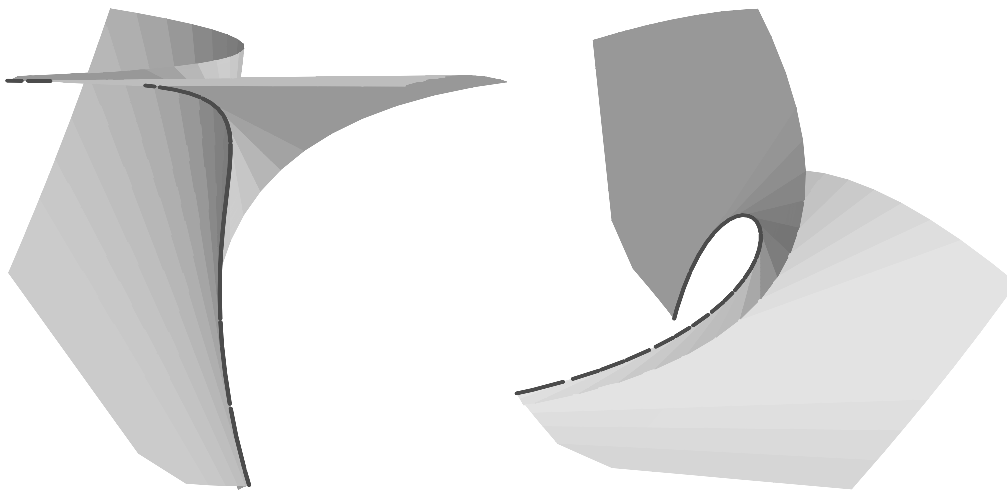

Figure 2: Pejorative manifolds of polynomials with degree 3 (view from two angles)

As a special case, $\Pi_{[1,1,\cdots,1]} = \mathbf{C}^n$ is the vector space of all monic polynomials with degree n.

3.2 Solving the nonsingular least squares problem

Let $\ell = [\ell_1, \cdots, \ell_m]$ be a multiplicity structure of degree n and Π_ℓ be the corresponding pejorative manifold. If the polynomial $p \sim \mathbf{a} \in \Pi_\ell$, then there is a vector $\mathbf{z} \in \mathbf{C}^m$ such that $G_\ell(\mathbf{z}) = \mathbf{a}$. In general, the polynomial system

$$\left\{ \begin{array}{rcl} g_1(z_1, \cdots, z_m) & = & a_1 \\ g_2(z_1, \cdots, z_m) & = & a_2 \\ \vdots & \vdots & \vdots \\ g_n(z_1, \cdots, z_m) & = & a_n \end{array} \right. \qquad \text{or} \qquad G_\ell(\mathbf{z}) = \mathbf{a} \tag{10}$$

is overdetermined except for the plain structure $\ell = [1, 1, \cdots, 1]$. Let $W = diag(\omega_1, \cdots, \omega_n)$ be a weight matrix and $\|\cdot\|_W$ denote the weighted 2-norm defined in (2). We seek a *weighted least squares solution* to (10) by solving the minimization problem

$$\min_{\mathbf{z} \in \mathbf{C}^m} \left\| G_\ell(\mathbf{z}) - \mathbf{a} \right\|_W^2 \;\equiv\; \min_{\mathbf{z} \in \mathbf{C}^m} \left\| W\Big(G_\ell(\mathbf{z}) - \mathbf{a}\Big) \right\|_2^2 \;\equiv\; \min_{\mathbf{z} \in \mathbf{C}^m} \left\{ \sum_{j=1}^{n} \omega_j^2 \left| g_j(\mathbf{z}) - a_j \right|^2 \right\}. \tag{11}$$

Two common types of weights can be used. To minimize the overall backward error of the roots, we set $W = diag(1, 1, \cdots, 1)$. On the other hand, the weights

$$\omega_j = \min\left\{1,\ |a_j|^{-1}\right\}, \quad j = 1, \cdots, n \tag{12}$$

lead to minimization of the relative backward error at every coefficient larger than one. All our numerical experiments for Algorithm I are conducted using the weights (12).

From Lemma 2.5, let $J(\mathbf{z})$ be the Jacobian of $G_\ell(\mathbf{z})$. In order to find a local minimum point of $F(\mathbf{z}) \equiv W\Big[G_\ell(\mathbf{z}) - \mathbf{a}\Big]$ with $\mathcal{J}(\mathbf{z}) = WJ(\mathbf{z})$, we look for $\tilde{\mathbf{z}} \in \mathbf{C}^m$ such that

$$\mathcal{J}(\tilde{\mathbf{z}})^H F(\tilde{\mathbf{z}}) = \Big[WJ(\tilde{\mathbf{z}})\Big]^H W\Big[G_\ell(\tilde{\mathbf{z}}) - \mathbf{a}\Big] = J(\tilde{\mathbf{z}})^H\, W^2\Big[G_\ell(\tilde{\mathbf{z}}) - \mathbf{a}\Big] = 0. \tag{13}$$

Definition 3.2 *Let $p \sim \mathbf{a}$ be a polynomial of degree n. For any given multiplicity structure ℓ of the same degree, the vector $\tilde{\mathbf{z}}$ satisfying (13) is called a* ***pejorative root vector*** *or simply* ***pejorative root*** *of p corresponding to the multiplicity structure ℓ and weight W.*

Our algorithms emanate from the following fundamental theorem by which one may convert the singular problem of computing multiple roots with standard methods to a regular problem by seeking the least squares solution of (10).

Theorem 3.1 *Let $G_\ell : \mathbf{C}^m \longrightarrow \mathbf{C}^n$ be the coefficient operator associated with a multiplicity structure $\ell = [\ell_1, \cdots, \ell_m]$. Then the Jacobian $J(\mathbf{z})$ of $G_\ell(\mathbf{z})$ is of full (column) rank if and only if the entries of $\mathbf{z} = (z_1, \cdots, z_m)^\top$ are distinct.*

Proof. Let $z_1, \cdots, z_m$ be distinct. To prove $J(\mathbf{z})$ is of full (column) rank, or the columns of $J(\mathbf{z})$ are linearly independent, write the j-th column of $J(\mathbf{z})$ as $J_j = \left(\frac{\partial g_1(\mathbf{z})}{\partial z_j}, \cdots, \frac{\partial g_n(\mathbf{z})}{\partial z_j}\right)^\top$. For $j = 1, \cdots, m$, let $q_j(x)$, a polynomial in x, be defined as follows,

$$\begin{aligned} q_j(x) &= \left(\frac{\partial g_1(\mathbf{z})}{\partial z_j}\right)x^{n-1} + \cdots + \left(\frac{\partial g_{n-1}(\mathbf{z})}{\partial z_j}\right)x + \left(\frac{\partial g_n(\mathbf{z})}{\partial z_j}\right) \\ &= \frac{\partial}{\partial z_j}\Big[x^n + g_1(\mathbf{z})x^{n-1} + \cdots + g_n(\mathbf{z})\Big] \;=\; \frac{\partial}{\partial z_j}\Big[(x-z_1)^{\ell_1}\cdots(x-z_m)^{\ell_m}\Big] \\ &= -\ell_j\,(x-z_j)^{\ell_j - 1}\left[\prod_{k\neq j}(x-z_k)^{\ell_k}\right]. \end{aligned} \tag{14}$$

If $c_1 J_1 + \cdots + c_m J_m = 0$ for constants $c_1, \cdots, c_m$, then

$$\begin{aligned} q(x) &\equiv c_1\, q_1(x) + \cdots + c_m\, q_m(x) \\ &= -\sum_{j=1}^m \left\{c_j \ell_j\,(x-z_j)^{\ell_j-1}\left[\prod_{k\neq j}(x-z_k)^{\ell_k}\right]\right\} = -\left[\prod_{\sigma=1}^m (x-z_\sigma)^{\ell_\sigma - 1}\right]\sum_{j=1}^m\left[c_j\ell_j\prod_{k\neq j}(x-z_k)\right] \end{aligned}$$

is a zero polynomial, yielding $r(x) = \sum_{j=1}^m c_j\ell_j\left[\prod_{k\neq j}(x-z_k)\right] \equiv 0$. Therefore, for $l = 1, \cdots, m$, $r(z_l) = c_l\left[\ell_l \prod_{k\neq l}(z_l - z_k)\right] = 0$ implies $c_l = 0$ since ℓ_l's are positive and z_k's are distinct. Therefore, J_j's are linearly independent.

On the other hand, suppose $z_1, \cdots, z_m$ are not distinct, say, for instance, $z_1 = z_2$. Then the first two columns of $J(\mathbf{z})$ are coefficients of polynomials $h_1(x)$ and $h_2(x)$ defined as

$$-\ell_1 (x-z_1)^{\ell_1-1}(x-z_2)^{\ell_2} \prod_{k=3}^{m}(x-z_k)^{\ell_k} \quad \text{and} \quad -\ell_2 (x-z_1)^{\ell_1}(x-z_2)^{\ell_2-1} \prod_{k=3}^{m}(x-z_k)^{\ell_k}$$

respectively. Since $z_1 = z_2$, these two polynomials differ by constant multiples ℓ_1 and ℓ_2. Therefore $J(\mathbf{z})$ is (column) rank deficient. Q.E.D.

With the system (10) being nonsingular from Theorem 3.1, the Gauss-Newton iteration

$$\mathbf{z}_{k+1} = \mathbf{z}_k - \left[J(\mathbf{z}_k)_W^+\right][G_\ell(\mathbf{z}_k) - \mathbf{a}], \quad k = 0, 1, \cdots \tag{15}$$

on Π_ℓ is well defined. Moreover, we have the convergence theorem based on Lemma 2.6.

Theorem 3.2 *Let* $\tilde{\mathbf{z}} = (\tilde{z}_1, \cdots, \tilde{z}_m)^\top \in \mathbf{C}^m$ *be a pejorative root of* $p \sim \mathbf{a}$ *associated with multiplicity structure* ℓ *and weight* W. *Assume* $\tilde{z}_1, \tilde{z}_2, \cdots, \tilde{z}_m$ *are distinct. Then there are* $\varepsilon, \epsilon > 0$ *such that, if* $\left\| \mathbf{a} - G_\ell(\tilde{\mathbf{z}}) \right\|_W < \varepsilon$ *and* $\left\| \mathbf{z}_0 - \tilde{\mathbf{z}} \right\|_2 < \epsilon$, *the iteration (15) is well defined and converges to the pejorative root* $\tilde{\mathbf{z}}$ *with at least a linear rate. If we have* $\mathbf{a} = G_\ell(\tilde{\mathbf{z}})$ *in addition, then the convergence is quadratic.*

Proof. Let $F(\mathbf{z}) = W\left[G_\ell(\mathbf{z}) - \mathbf{a}\right]$ and $\mathcal{J}(\mathbf{z})$ be its Jacobian. $F(\mathbf{z})$ is obviously analytic. From Theorem 3.1, the smallest singular value σ of $\mathcal{J}(\tilde{\mathbf{z}})$ is strictly positive. If $\mathbf{a}$ is sufficiently close to $G_\ell(\tilde{\mathbf{z}})$, then $\left\| F(\tilde{\mathbf{z}}) \right\|_2 = \left\| G_\ell(\tilde{\mathbf{z}}) - \mathbf{a} \right\|_W$ will be small enough, making (5) holds with $\delta < \sigma^2$. Therefore all conditions of Lemma 2.6 are satisfied and there is a neighborhood Ω of $\tilde{\mathbf{z}}$ such that if $\mathbf{z}_0 \in \Omega$, the iteration (15) converges and satisfies (6). If in addition $\mathbf{a} = G_\ell(\tilde{\mathbf{z}})$, then $F(\tilde{\mathbf{z}}) = 0$ and therefore $\delta = 0$ in (5) and (6). The convergence becomes quadratic. Q.E.D.

As a special case for the structure $\ell = [1, 1, \cdots, 1]$, equations in (10) form Viéte's system of n-variate polynomial system. Solving this system via Newton's iteration is equivalent to the Weierstrass (Durand-Kerner) algorithm [23]. When a polynomial has multiple roots, Viéte's system becomes singular at the non-distinct root vector. This singularity appears to be the very reason that causes the ill-conditioning of conventional root-finders: a wrong pejorative manifold is used.

3.3 The structure-preserving condition number

There are many insightful discussions on the numerical condition of polynomial roots in the literature such as [7, 15, 19, 28, 25, 29]. In general, a condition number can be characterized as the smallest number satisfying

$$\left[forward\_error\right] \leq \left[condition\_number\right] \times \left[backward\_error\right] + h.o.t., \tag{16}$$

where $h.o.t$ represents higher order terms of the backward error. For a polynomial with multiple roots, under *unrestricted* perturbation, the only condition number satisfying (16) is infinity. For a simple example, let polynomial $p(x) = x^2$. A backward error ε makes the perturbed polynomial $\tilde{p}(x) = x^2 + \varepsilon$ having roots $\pm\sqrt{\varepsilon}i$ with forward error $\sqrt{\varepsilon}$ in magnitude. The only "constant" c which accounts for $\sqrt{\varepsilon} \leq c\,\varepsilon$ for *all* $\varepsilon > 0$ must be infinity.

By changing the computational objective from solving a polynomial equation $p(x) = 0$ to the nonlinear least squares problem in the form of (11), the structure-altering noise is filtered out, and the multiplicity structure is preserved. With this shift in computing strategy, the sensitivity of the roots can be analyzed differently.

Let's consider the root vector $\mathbf{z}$ of $p \sim \mathbf{a} = G_\ell(\mathbf{z})$. The polynomial p is perturbed, with multiplicity structure ℓ being preserved, to be $\hat{p} \sim \hat{\mathbf{a}} = G_\ell(\hat{\mathbf{z}})$. In other words, both p and $\hat{p}$ are on the same pejorative manifold Π_ℓ. Then

$$\hat{\mathbf{a}} - \mathbf{a} = G_\ell(\hat{\mathbf{z}}) - G_\ell(\mathbf{z}) = J(\mathbf{z})(\hat{\mathbf{z}} - \mathbf{z}) + O\left(\|\,\hat{\mathbf{z}} - \mathbf{z}\,\|^2\right)$$

where $J(\mathbf{z})$ is the Jacobian of $G_\ell(\mathbf{z})$. Assuming the entries of $\mathbf{z}$ are distinct, by Theorem 3.1. $J(\mathbf{z})$ is of full rank. Consequently,

$$\begin{aligned} \left\| W(\hat{\mathbf{a}} - \mathbf{a}) \right\|_2 &= \left\| [W J(\mathbf{z})](\hat{\mathbf{z}} - \mathbf{z}) \right\|_2 + h.o.t.\,, \\ \text{namely,} \qquad \left\| \hat{\mathbf{a}} - \mathbf{a} \right\|_W &\geq \sigma_{min} \left\| \hat{\mathbf{z}} - \mathbf{z} \right\|_2 + h.o.t., \\ \text{or} \qquad \left\| \hat{\mathbf{z}} - \mathbf{z} \right\|_2 &\leq \left(\frac{1}{\sigma_{min}}\right) \left\| \hat{\mathbf{a}} - \mathbf{a} \right\|_W + h.o.t. \end{aligned} \tag{17}$$

where σ_{min} , the smallest singular value of $WJ(\mathbf{z})$, is strictly positive since W and $J(\mathbf{z})$ are of full rank. The distance $\|\,\hat{\mathbf{z}} - \mathbf{z}\,\|_2$ is the forward error and the weighted distance $\|\,\hat{\mathbf{a}} - \mathbf{a}\,\|_W$ measures the backward error. Therefore, the sensitivity of the root vector is asymptotically bounded by $\frac{1}{\sigma_{min}}$ times the size of the multiplicity-preserving perturbation. In this sense, the multiple roots are not infinitely sensitive.

Definition 3.3 *Let p be a polynomial and $\mathbf{z}$ be its pejorative root corresponding to a given multiplicity structure ℓ and weight W. Let G_ℓ be the coefficient operator associated with ℓ, J be its Jacobian, and σ_{min} be the smallest singular value of $WJ(\mathbf{z})$. Then the* ***condition number of*** $\mathbf{z}$ **with respect to the multiplicity structure** ℓ **and weight** W *is defined as*

$$\kappa_{\ell,w}(\mathbf{z}) = \frac{1}{\sigma_{min}}.$$

Remark: The condition number $\kappa_{\ell,w}(\mathbf{z})$ is structure dependent. The array $\ell = [\ell_1, \cdots, \ell_m]$ may or may not be the *actual* multiplicity structure. A polynomial has different condition numbers corresponding to different pejorative roots on various pejorative manifolds. For example, see Table 4 in §3.5.2.

We now estimate the error on pejorative roots of polynomials with inexact coefficients. In this case, the given polynomial $\hat{p}$ is assumed to be arbitrarily perturbed from p with both polynomials near a pejorative manifold Π_ℓ. In exact sense, neither polynomial possesses the structure ℓ. The nearby pejorative manifold causes both polynomials being ill-conditioned in conventional sense. Consequently, the exact roots of $\hat{p}$ can be far from those of p even if two polynomials are close to each other. However, the following theorem ensures that their pejorative roots, not exact roots, may still be insensitive to perturbation.

Theorem 3.3 *Let the polynomial $\hat{p} \sim \hat{\mathbf{b}}$ be an approximation to $p \sim \mathbf{b}$. Corresponding to the multiplicity structure ℓ and weight W, let $\mathbf{z}$ and $\hat{\mathbf{z}}$ be pejorative roots of p and $\hat{p}$ respectively,* *with components of* $\mathbf{z}$ *being distinct. If* $\|\mathbf{b}-\hat{\mathbf{b}}\|_W$ *and* $\|G_\ell(\mathbf{z})-\mathbf{b}\|_W$ *are sufficiently small, then*

$$\left\|\mathbf{z}-\hat{\mathbf{z}}\right\|_2 \leq 2\cdot\kappa_{\ell,w}(\mathbf{z})\cdot\left(\left\|G_\ell(\mathbf{z})-\mathbf{b}\right\|_W+\left\|\mathbf{b}-\hat{\mathbf{b}}\right\|_W\right)+h.o.t. \tag{18}$$

Proof. From (17),

$$\begin{aligned}\left\|\mathbf{z}-\hat{\mathbf{z}}\right\|_2 &\leq \kappa_{\ell,w}(\mathbf{z})\left\|G_\ell(\mathbf{z})-G_\ell(\hat{\mathbf{z}})\right\|_W+h.o.t.\\ &\leq \kappa_{\ell,w}(\mathbf{z})\left(\left\|G_\ell(\mathbf{z})-\mathbf{b}\right\|_W+\left\|\mathbf{b}-\hat{\mathbf{b}}\right\|_W+\left\|G_\ell(\hat{\mathbf{z}})-\hat{\mathbf{b}}\right\|_W\right)+h.o.t.\end{aligned}$$

Since $\|G_\ell(\hat{\mathbf{z}})-\hat{\mathbf{b}}\|_W$ is a local minimum, we have

$$\left\|G_\ell(\hat{\mathbf{z}})-\hat{\mathbf{b}}\right\|_W \leq \left\|G_\ell(\mathbf{z})-\hat{\mathbf{b}}\right\|_W \leq \left\|G_\ell(\mathbf{z})-\mathbf{b}\right\|_W+\left\|\mathbf{b}-\hat{\mathbf{b}}\right\|_W,$$

and the assertion of the theorem follows. Q.E.D.

By the above theorem, when a polynomial is perturbed, the error on the pejorative roots depends on the magnitude of the perturbation (i.e. $\|\mathbf{b}-\hat{\mathbf{b}}\|_W$), the distance to the pejorative manifold (namely $\|G_\ell(\mathbf{z})-\mathbf{b}\|_W$), as well as the condition number $\kappa_{\ell,w}(\mathbf{z})$. Although the (exact) roots may be hypersensitive, their pejorative roots are stable if $\kappa_{\ell,w}(\mathbf{z})$ is moderate.

For a polynomial p having multiplicity structure ℓ, we can now estimate the error of its multiple roots computed from its (inexact) approximation $\hat{p}$. The perturbation from p to $\hat{p}$ can be arbitrary, such as rounding up digits in coefficients. The (exact) roots of $\hat{p}$ are all simple in general and far from the multiple roots of p. The following corollary ensures that the *pejorative* root $\hat{\mathbf{z}}$ of $\hat{p}$ with respect to the multiplicity structure ℓ can be an accurate approximation to the multiple roots $\mathbf{z}$ of p.

Corollary 3.1 *Under the condition of Theorem 3.3, if* $\mathbf{z}$ *is the exact root vector of* p *with multiplicity structure* ℓ*, then*

$$\left\|\mathbf{z}-\hat{\mathbf{z}}\right\|_2 \leq 2\cdot\kappa_{\ell,w}(\mathbf{z})\cdot\left\|\mathbf{b}-\hat{\mathbf{b}}\right\|_W+h.o.t. \tag{19}$$

Proof. Since $\mathbf{z}$ is exact, $\|G_\ell(\mathbf{z})-\mathbf{b}\|_W=0$ in (18). Q.E.D.

The "attainable accuracy" barrier suggests that when multiplicity increases, the roots sensitivity intensifies. However, there is no apparent corelation between the magnitude of the multiplicities and the structure-constraint sensitivity. For example, consider polynomials

$$p_\ell(x)=(x+1)^{\ell_1}(x-1)^{\ell_2}(x-2)^{\ell_3}$$

with different multiplicities $\ell=[\ell_1,\ell_2,\ell_3]$. For the weight W defined in (12), Figure 3 lists the condition numbers for different multiplicities. As seen in this example, the magnitude of root error can actually be *less* than that of the data error when the condition number is less than one. The condition theory described above indicates that multiprecision arithmetic may *not* be a necessity, and the "attainable accuracy" barrier appears to be highly questionable.

multiplicities			condition
ℓ_1	ℓ_2	ℓ_3	number
1	1	1	3.1499
1	2	3	2.0323
10	20	30	0.0733
100	200	300	0.0146

Figure 3: The sensitivity and root multiplicities

In §3.5 and §5, more examples will show that our iterative algorithm indeed reaches the accuracy permissible by the condition number $\kappa_{\ell,w}(\mathbf{z})$, which can be calculated with negligible cost. The Jacobian $J(\mathbf{z})$ and its QR decomposition are required by the Gauss-Newton iteration, and can be recycled to calculate $\kappa_{\ell,w}(\mathbf{z})$. The inverse iteration described in Lemma 2.4 is suitable for finding the smallest singular value.

3.4 The numerical procedures

The iteration (15) requires evaluation of $G_\ell(\mathbf{z}_k)$ and $J(\mathbf{z}_k)$, where the components of $G_\ell(\mathbf{z})$ are defined in (7) and (8) as coefficients of the polynomial $p(x) = (x - z_1)^{\ell_1} \cdots (x - z_m)^{\ell_m}$. While the explicit formulas for each $g_j(z_1, \cdots, z_m)$ and $\frac{\partial g_j}{\partial z_i}$ can be symbolically (inefficiently in general) computed using softwares like Maple, we propose more efficient numerical procedures for computing $G_\ell(\mathbf{z})$ and $J(\mathbf{z})$ in Figure 4.

```
Algorithm EvalG:
input:  m, n, z = (z_1, ..., z_m)^T,
              ℓ = [ℓ_1, ..., ℓ_m]
output:  vector G_ℓ(z) ∈ C^n

    s = (1)
    for i = 1,2,...m do
        for l = 1,2,...,ℓ_i do
            s = conv(s, (1, -z_i)^T)
        end do
    end do
    g_j(z) =(j+1)-th component of s
        j = 1,...,n
```

```
Algorithm EvalJ:
input:  m, n, z = (z_1, ..., z_m)^T,
              ℓ = [ℓ_1, ..., ℓ_m]
output:  Jacobian matrix J(z) ∈ C^{n×m}

    u ~ ∏(x - z_j)^{ℓ_j - 1} by EvalG
    for j = 1,2,...,m do
        s = -ℓ_j u
        for l = 1,...,m, l ≠ j do
            s = conv(s, (1, -z_l)^T)
        end do
        j-th column of J(z) = s
    end do
```

Figure 4: Pseudo-codes for evaluating $G_\ell(\mathbf{z})$ and $J(\mathbf{z})$

The polynomial multiplication is equivalent to the vector convolution (Lemma 2.1). The polynomial $p(x) = (x - z_1)^{\ell_1} \cdots (x - z_m)^{\ell_m}$ can thereby be constructed from recursive convolution with vectors $(1, -z_j)^\top$, $j = 1, 2, \cdots, m$. As a result, $G_\ell(\mathbf{z})$ is computed through the nested loops shown in Figure 4 as Algorithm EvalG. It takes $n^2 + O(n)$ flops (additions and multiplications) to calculate $G_\ell(\mathbf{z})$.

The j-th column of the Jacobian $J(\mathbf{z})$, as shown in the proof of Theorem 3.1, can be considered the coefficients of the polynomial $q_j(x)$ defined in (14). The cost of computing $J(\mathbf{z})$ is no more than $mn^2 + O(n)$ flops. Each step of the Gauss-newton iteration takes $O(nm^2)$ flops. Therefore, for a polynomial of degree n with m distinct roots, the complexity of Algorithm I is $O(m^2 n + mn^2)$. The worst case occurs when $m = n$ and the complexity becomes $O(n^3)$. The complete pseudo-code of the Algorithm I is shown in Figure 5.

3.5 Numerical results for Algorithm I

Algorithm I is implemented as a Matlab code PejRoot. All the tests of PejRoot are conducted with IEEE double precision (16 decimal digits) without extension. In comparison, other algorithms and software may use unlimited machine precision in some cases.

```
Pseudo-code PejRoot  (Algorithm I):
 input:  m, n, a ∈ C^n, weight matrix W, initial iterate z_0,
             multiplicity structure ℓ, error tolerance τ
 output:  Roots z = (z_1, ···, z_m), or message of failure

    for k = 0, 1, ··· do
        Calculate G_ℓ(z_k) and J(z_k) with EvalG and EvalJ
        Compute the least squares solution Δz_k to the linear system
                [W J(z_k)](Δz_k) = W[G_ℓ(z_k) − a]
        Set z_{k+1} = z_k − Δz_k  and  δ_k = ||Δz_k||_2
        if k ≥ 1 then
            if δ_k ≥ δ_{k−1} then, stop, output failure message
            else if δ_k^2 / (δ_{k−1} − δ_k) < τ then,  stop, output z = z_{k+1}
            end if
        end if
    end do
```

Figure 5: Pseudo-code of Algorithm I

3.5.1 The effect of "attainable accuracy"

Conventional methods, such as Farmer-Loizou methods [13], are subject to the "attainable accuracy" barrier. We made a straightforward implementation of the Farmer-Loizou third order iteration suggested in [13] and apply it to the same example they used

$$p_1(x) = (x-1)^4(x-2)^3(x-3)^2(x-4).$$

Both iterations start from $\mathbf{z}_0 = (1.1, 1.9, 3.1, 3.9)$ using the standard IEEE double precision. The "attainable accuracy" of the roots are 4, 5, 8, 16 digits respectively. For 100 iteration steps, the Farmer-Loizou method produces iterates that bounce around the roots. In contrast, our iteration smoothly converges to the roots and reaches accuracy of 14 digits. The "attainable accuracy" barrier has no effect on our algorithm. The iterations are shown in Table 1 for three roots $x = 1, 2, 3$ with highest multiplicities.

	Farmer-Loizou third order iteration				PejRoot result		
steps	iterateas			steps	iterates		
1	1.0009	1.998	3.001	1	1.03	1.8	3.4
2	0.99997	1.9999992	3.000000008	2	0.997	1.98	2.6
3	0.01	3.4	2.9988	3	1.00009	2.05	2.8
4	0.8	2.3	3.000007	4	0.99994	1.994	2.98
5	0.998	2.007	3.0000001	5	1.000003	2.0001	2.9990
6	1.0000007	2.00000007	2.99996	6	0.999999997	2.000000005	2.9999990
...	...			7	1.00000000000000	2.0000000000002	2.999999999998
100	1.00000008	3.3	2.99999997	8	1.00000000000000	2.00000000000000	2.99999999999999

Table 1: Comparison with the Farmer-Loizou third order iteration in low multiplicity case. Three roots are shown with unimportant digits truncated

In the same problem, we increase the multiplicities 10 times as large, generating

$$p_2(x) = (x-1)^{40}(x-2)^{30}(x-3)^{20}(x-4)^{10}$$

with 16-digit accuracy in coefficients. In this test, our method still uses the standard 16-digit arithmetic and attains 14 correct digits on the roots, while Farmer-Loizou method uses 1000-digit operations in Maple and fails (Three roots iterations are shown in Table 2).

Farmer-Loizou third order iteration				PejRoot result			
steps	iterateas			steps	iterates		
1	0.47	-.33	3.02	1	1.004	1.98	3.05
2	32.92	-4.65	2.69	2	1.0001	1.998	3.003
3	4.75	-1.80	1.75	3	0.9999998	2.000006	2.99997
4	205.96	.40	1.54	4	0.999999999994	2.00000000001	2.9999999990
...	...			5	1.00000000000000	2.00000000000001	2.99999999999997
100	5.99	1.10	0.30	6.	1.00000000000000	2.00000000000001	2.99999999999998

Table 2: Comparison with the Farmer-Loizou third order iteration in high multiplicity case. Three roots are shown with unimportant digits truncated

The true accuracy barrier of our Algorithm I is the condition number $\kappa_{\ell,w}(\mathbf{z})$. Matlab constructed the test polynomial with a relative coefficient error of 4.56×10^{-16}, the condition number is 29.3. The root error is approximately 1×10^{-14}, which is within the error bound $2 \times (29.3) \times (4.56 \times 10^{-16}) = 2.67 \times 10^{-14}$ established in Corollary 3.1.

There are state-of-art root-finding packages available using multiprecision, such as MPSolve implemented by Bini et al [2] and Eigensolve by Fortune [14]. If the given polynomial is exact (e.g. polynomial with rational coefficients), those packages in general are capable of calculating all roots to the desired accuracy via extending the machine precision according to the "attainable accuracy". For inexact polynomials, the accuracy of those packages on multiple roots are limited no matter how many digits the machine precision is extended. For example, consider the polynomial

$$p(x) = \left(x - \sqrt{2}\right)^{20} \left(x - \sqrt{3}\right)^{10}.$$

The coefficients are calculated to 100-digit accuracy. The "attainable accuracy" for the roots $\sqrt{2}$ and $\sqrt{3}$ are 5 and 10 digits respectively. MPSolve and Eigensolve output nearly identical results in accordance with this "attainable accuracy". In contrast, our software using only 16 digits precision in coefficients without extending the machine precision, still outputs roots of 15 digits accuracy along with accurate multiplicities (see Table 3).

MPSolve and Eigensolve results with 100-digit input accuracy (unimportant digits are truncated)	MultRoot results with 16-digit input/machine precision	
1.41412 - 0.000013i	THE CONDITION NUMBER:	0.90775
1.41412 + 0.000013i	THE BACKWARD ERROR:	6.66E-016
... ...	THE ESTIMATED FORWARD ROOT ERROR:	1.21e-15
1.73205077 - 0.0000000094i	computed roots	multiplicities
1.73205077 + 0.0000000094i	1.732050807568876	10
... ...	1.414213562373096	20

Table 3: Comparison with multiprecision root-finders MPSolve and Eigensolve

3.5.2 Clustered multiple roots

Consider $f(x) = (x - 0.9)^{18}(x - 1)^{10}(x - 1.1)^{16}$. The roots are highly multiple and clustered. The Matlab function roots produces 44 ill-conditioned roots scattered in a box of 2.0×2.0. (see Figure 6). In contrast, the Algorithm I code PejRoot obtains all three multiple roots for at least 14 digits in accuracy by taking two additional iteration steps on the information of multiplicity structure and the initial iterate provided by our Algorithm II in §4.

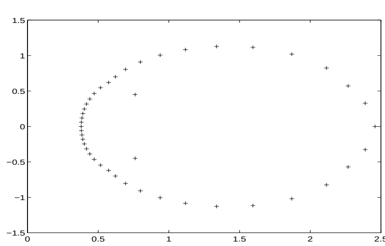

Figure 6: The root cluster from three multiple roots calculated by Matlab root-finder `roots`

step	z_1	z_2	z_3
0	0.89999999993	0.9999999993	1.0999999998
1	0.999999999999991	1.00000000000001	1.10000000000001
2	0.999999999999991	1.000000000000001	1.10000000000001

The backward accuracy can easily be verified to be less than 1.36×10^{-15}. The condition number is 60.4. Therefore, with perturbation at the 16-th digit of the coefficients, 14 correct digits constitute the best possible accuracy that can be expected from any method.

An important feature of Algorithm I is that it does *not* require the correct multiplicity structure. Computation with different structures is often needed and is permissible with Algorithm I. If the computation is on a "wrong" pejorative manifold, then either the condition number or the backward error becomes large. Table 4 is a partial list of pejorative roots under different multiplicity structures. As shown in Table 4, if the computing objective is *unconstraint minimization* of the backward error, like standard methods, then we would get simple, clustered, and incorrect roots as shown in Figure 6. On the other hand, if the objective is to minimize backward error, subject to a proper sensitivity constraint, say $\kappa_{\ell,w}(\mathbf{z}) > 100$, then the only solution is $\mathbf{z} \approx (0.9, 1.0, 1.1)^\top$ with correct multiplicity structure [18, 10, 16]. In short, computing multiple roots of inexact polynomials is a *constraint optimization* problem.

multiplicity structure	pejorative roots	backward error (relative)	condition number
[1,1,...,1]	(see Figure 6)	.0000000000000006	1390704851032436
[18,10,16]	(.9000, 1.0000, 1.1000)	.000000000000002	60.4
[17,11,16]	(.8980, .9934, 1.1006)	.0000004	53.8
[14,16,14]	(.8890, .9892, 1.1090)	.000003	29.0
[10,24,10]	(.8711, .9906, 1.1315)	.000008	26.7
[2, 40, 2]	(.7390, .9917, 1.3277)	.00009	23.6
[1, 43]	(.5447, 1.0054)	.004	1.3
[44]	(.9925)	.04	.0058

Table 4: Partial list of multiple roots on different pejorative manifolds

3.5.3 Roots with huge multiplicities

The accuracy as well as stability of Algorithm I seem independent of the multiplicities of the roots. For instance, let's consider the polynomial of degree 1000

$$g(x) = [x - (0.3 + 0.6i)]^{100}\, [x - (0.1 + 0.7i)]^{200}\, [x - (0.7 + 0.5i)]^{300}\, [x - (0.3 + 0.4i)]^{400}.$$

The multiplicities of the roots are 100, 200, 300 and 400. These multiplicities are "huge" compared to other numerical examples, usually with multiplicities less than ten, used in the root-finding literature. In addition to such high multiplicities, we perturb the sixth digits of all coefficients of g by multiplying (1 ± 10^{-6}) on each one of them. Using any conventional approach, this perturbation will result in a total loss of forward accuracy, even if multiprecision is used. The code PEJROOT of Algorithm I takes a few seconds under Matlab to calculate all roots up to 7 digits accuracy.

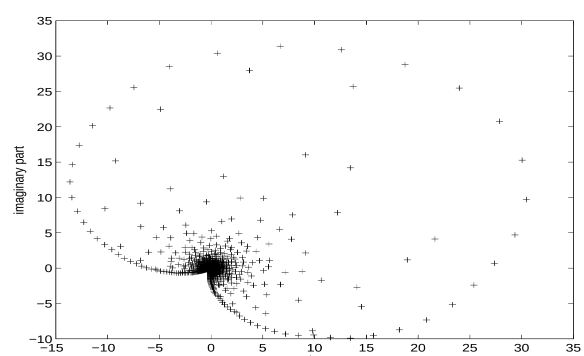

Figure 7: Result for the degree 1000 polynomial by Matlab function `roots`

Taking the condition number 0.58 into account, this accuracy is optimal. On the same machine, Matlab function `roots` takes about 15 minutes to produce 1000 incorrect roots, as shown in Figure 7.

z_1		z_2		z_3		z_4	
.289	+.601i	.100	+.702i	.702	+.498i	.301	+.399i
.309	+.602i	.097	+.698i	.698	+.499i	.299	+.400i
.293	+.596i	.101	+.7003i	.7002	+.5005i	.3007	+.4003i
.3003	+.5994i	.09994	+.70008i	.69996	+.50003i	.29996	+.40007i
.300005	+.600006	.099998	+.6999992i	.69999992	+.4999993i	.2999992	+.3999992i
.3000002	+.60000005i	.09999995	+.69999998i	.69999997	+.49999998i	.29999997	+.400000002i

4 Algorithm II: the multiplicity structure and initial root estimates

While Algorithm I can be used on any particular pejorative manifold, of course, the "correct" multiplicity structure is preferred if it is attainable. We present Algorithm II that calculates the multiplicity structure of a given polynomial as well as the initial root approximation for Algorithm I.

4.1 Remarks on the univariate GCD computation

For a given polynomial p with $u = GCD\,(p, p')$, it is known as early as 1769 by Lagrange that $v = p/u$ has the same distinct roots as p, and all roots of v are simple. If v is obtainable, its simple roots can be calculated using standard root-finders. Based on this observation, the following recursive process is a natural approach to completely factor the polynomial p.

$$\left\{\begin{array}{l} u_0 = p \\ \text{for } j = 1, 2, \cdots, \ \text{ while } deg(u_{j-1}) > 0 \text{ do} \\ \qquad \text{calculate } u_j = GCD\left(u_{j-1}, u'_{j-1}\right), \qquad v_j = \dfrac{u_{j-1}}{u_j} \\ \qquad \text{calculate the (simple) roots of } v_j(x) \\ \text{end do} \end{array}\right. \tag{20}$$

This process is also known as early as 1863 by Gauss. Other squarefree factorization processes, such as Yun's algorithm [31], have also been proposed in the context of Computer Algebra.

The difficulty in carrying out the process (20) is the GCD computation. The classical Euclidean GCD Algorithm requires recursive polynomial division which may not be numerically stable (see §4.2.3). Therefore, implementations of (20) based on the Euclidean GCD-finder [3, 26] may fail to reach desirable reliability or accuracy (see numerical comparison in §4.6).

Numerical GCD computation has been studied extensively [4, 5, 12, 16, 20, 24]. However, a reliable blackbox-type software is still not available. In [5], Corless, Gianni, Trager and Watt proposed a novel approach using the singular value decomposition in finding the degree of the GCD, and suggested the possibility of solving a GCD system similar to (22) below as a least squares problem, along with several other possibilities including using the Euclidean algorithm.

There are several unresolved issues in the approach of Corless et al, especially in the stage of calculating the GCD after determining its degree. Among the possible avenues suggested, they seem to prefer using iterative methods to solve the least squares problem similar to (22) below. However, their least squares system is underdetermined by one equation. Moreover, with no clearly decided initial iterate being given, one can only leave this crucial ingredient to guessing or some sort of expensive global search [4]. From [5] and its follow-up work such as [4, 20] it is also not clear which standard optimization algorithm should be selected. We shall demonstrate that the Gauss-Newton iteration, absent from the above works, is apparently the simplest, most efficient and most suitable method in solving the GCD system (22), and it is at least locally convergent.

The key to carrying out the procedure (20) is the capability to factor an arbitrary polynomial f and its derivative f' with a GCD triplet (u, v, w):

$$\left\{ \begin{array}{rcl} u(x)\, v(x) & = & f(x) \\ u(x)\, w(x) & = & f'(x) \end{array} \right. , \quad u \text{ is monic, } v \text{ and } w \text{ are co-prime.} \tag{21}$$

In light of the Corless-Gianni-Trager-Watt approach, which calculates all singular values of a single Sylvester matrix $S_{n-1}(f)$, we employ a successive updating process that calculates only the smallest singular values of the Sylvester matrices $S_j(f)$, $j = 1, 2, \cdots$ and stop at the first rank deficient matrix $S_k(f)$. With this $S_k(f)$, not only the degrees of the GCD triplet u, v, w are available, we also obtain coefficients of v and w automatically from the resulting right singular vector. In combination with a least squares division in §4.2.3 instead of the unstable long division, we can generate an excellent approximation to the GCD triplet, and obtain an initial iterate that is not clearly indicated in the approach of Corless et al. Consequently, a blackbox-type software computing $GCD(f, f')$ is developed for the the process (20).

The discussion on GCD computation is limited to $GCD(f, f')$ because our objective is mainly root-finding in this paper. With minor modifications, our GCD-finder can easily be adapted to the general GCD problem of arbitrary polynomial pairs.

4.2 Calculating the greatest common divisor

Algorithm II is based on the following GCD-finder for an arbitrary polynomial f:

STEP 1. Find the degree m of $GCD(f, f')$.
STEP 2. Set up the system (21) in accordance with the degree m.
STEP 3. Find an initial approximation to u, v and w for the GCD system (21).
STEP 4. Use the Gauss-Newton iteration to refine the GCD triplet (u, v, w).

We shall describe each step in detail.

4.2.1 Finding the degrees of the GCD triplet

Let f be a polynomial of degree n. By Lemma 2.2, the degree of $u = GCD(f, f')$ is $m = n - k$ if and only if the the k-th Sylvester discriminant matrix is the first one being rank-deficient. Therefore, $m = deg(u)$ can be identified by calculating the sequence of the smallest singular values ς_j of $S_j(f)$, $j = 1, 2, \cdots$, until reaching ς_k that is *approximately* zero. Since only one singular pair (i.e. the singular value and the right singular vector) is needed, the inverse iteration described in Lemma 2.4 is suitable for this purpose. Moreover, we can reduce the computing cost even further by recycling and updating the QR decomposition of $S_j(f)$'s along the way. More specifically, let

$$f(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_n, \quad f'(x) = b_0 x^{n-1} + b_1 x^{n-2} + \cdots + b_{n-1}.$$

We rotate the columns of $S_j(f)$ to form $\hat{S}_j(f)$ in such a way

$$
\left(\begin{array}{ccc|ccc}
\multicolumn{3}{c}{\overbrace{\hphantom{b_{n-1}\ \ddots\ b_{n-1}}}^{j+1}} & \multicolumn{3}{c}{\overbrace{\hphantom{a_n\ \ddots\ a_n}}^{j}} \\
b_0 & & & a_0 & & \\
b_1 & \ddots & & a_1 & \ddots & \\
\vdots & \ddots & b_0 & \vdots & \ddots & a_0 \\
\vdots & & b_1 & \vdots & & a_1 \\
b_{n-1} & & \vdots & a_n & & \vdots \\
 & \ddots & \vdots & & \ddots & \vdots \\
 & & b_{n-1} & & & a_n
\end{array}\right)
\rightarrow
\overbrace{\left(\begin{array}{cccccccc}
b_0 & a_0 & & & & & & \\
b_1 & a_1 & b_0 & a_0 & & & & \\
\vdots & \vdots & b_1 & a_1 & \ddots & & & \\
b_{n-1} & a_{n-1} & \vdots & \vdots & & b_0 & a_0 & \\
 & a_n & b_{n-1} & a_{n-1} & & b_1 & a_1 & b_0 \\
 & & & a_n & \ddots & \vdots & \vdots & b_1 \\
 & & & & & b_{n-1} & a_{n-1} & \vdots \\
 & & & & & & a_n & b_{n-1}
\end{array}\right)}^{2j+1}
$$

that the odd and even columns of $\hat{S}_j(f)$ consist of the coefficients of f' and f respectively. Consequently, the matrix $\hat{S}_{j+1}(f)$ is simply formed by adding a zero row at the bottom and two columns in the right on $\hat{S}_j(f)$. Updating the QR decomposition of each $\hat{S}_j(f)$ requires only $O(n)$ additional flops. The inverse iteration (1) requires $O(j^2)$ flops at each $S_j(f)$.

Let θ be a given *zero singular value threshold*. We shall discuss more about this parameter in §4.4. With successive QR updating and the inverse iteration, the process of finding the degrees of the GCD triplet (u, v, w) can be summarized as follows.

> Calculate the QR decomposition of the $(n+1) \times 3$ matrix $\hat{S}_1(f) = Q_1 R_1$
> `For` $j = 1, 2, \cdots$ `do`
>   use the inverse iteration (1) to find the smallest singular value ς_j of $\hat{S}_j(f)$ and
>   the corresponding right singular vector $\mathbf{y}_j$
>   `if` $\varsigma_j \le \theta \big\| \mathbf{f} \big\|_2$ `then` $k = j$, $m = n - k$, extract $\mathbf{v}$ and $\mathbf{w}$ from $\mathbf{y}_j$, exit
>   `else` update $\hat{S}_j(f)$ to $\hat{S}_{j+1}(f) = Q_{j+1} R_{j+1}$
>   `end if`
> `end do`

4.2.2 The quadratic GCD system

Let $m = n - k$ be the degree of $GCD(f, f')$ calculated in `STEP 1`. We now formulate the GCD system (21) of `STEP 2` in vector form with unknown vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$:

$$
\left[\begin{array}{l} u_0 \\ conv(\mathbf{u}, \mathbf{v}) \\ conv(\mathbf{u}, \mathbf{w}) \end{array}\right] = \left[\begin{array}{c} 1 \\ \mathbf{f} \\ \mathbf{f}' \end{array}\right], \qquad \text{for} \quad \left(\begin{array}{c} \mathbf{u} \\ \mathbf{v} \\ \mathbf{w} \end{array}\right) \in \mathbf{C}^{m+1} \times \mathbf{C}^{k+1} \times \mathbf{C}^{k}. \tag{22}
$$

Here, the convolution $conv(\cdot, \cdot)$ is defined in Lemma 2.1. The following lemma ensures this quadratic system is nonsingular.

Lemma 4.1 *The Jacobian of the quadratic system (22) is*

$$
J(\mathbf{u}, \mathbf{v}, \mathbf{w}) = \left[\begin{array}{ccc} \mathbf{e}_1^\top & & \\ C_{m+1}(v) & C_{k+1}(u) & \\ C_{m+1}(w) & & C_k(u) \end{array}\right], \quad \textit{where} \quad \mathbf{e}_1 = (1, 0, \cdots, 0)^\top \in \mathbf{C}^{m+1}. \tag{23}
$$

If $u = GCD(f, f')$ *with* $(\mathbf{u}, \mathbf{v}, \mathbf{w})$ *satisfying (22), then* $J(\mathbf{u}, \mathbf{v}, \mathbf{w})$ *is of full (column) rank.*

Proof. It is straightforward to verify (23) by using Lemma 2.1. To prove $J(\mathbf{u},\mathbf{v},\mathbf{w})$ is of full rank, we assume the existence of polynomials $q(x) = \sum_{j=0}^{m} q_j x^{m-j}$, $r(x) = \sum_{j=0}^{k} r_j x^{k-j}$ and $s(x) = \sum_{j=0}^{k-1} s_j x^{k-j-1}$ such that

$$J(\mathbf{u},\mathbf{v},\mathbf{w}) \begin{pmatrix} \mathbf{q} \\ \mathbf{r} \\ \mathbf{s} \end{pmatrix} = 0, \quad \text{or} \quad \begin{cases} q_0 & = & 0 \\ vq + ur & = & 0 \\ wq + us & = & 0 \end{cases}. \tag{24}$$

Here, as before, $\mathbf{q}$, $\mathbf{r}$ and $\mathbf{s}$ are coefficient vectors of q, r and s respectively. From (24), we have $vq = -ur$ and $wq = -us$. So, $wvq - vwq = -uwr + uvs = 0$, namely $-wr + vs = 0$ or $wr = vs$. Since v and w are co-prime, there is a polynomial t such that $r = tv$ and $s = tw$. Consequently, $vq = -ur = -utv$ leads to $q = -tu$. Because $deg(q) = deg(tu) \le m$, $deg(u) = m \ge 0$ and $u_0 = 1$, the degree of t must be zero. So the polynomial t is a constant. Using the first equation in (24) and $u_0 = 1$, we have $q_0 = -tu_0 = -t = 0$. It follows that $q = -tu = 0$, $r = tv = 0$ and $s = tw = 0$. Consequently, $J(\mathbf{u},\mathbf{v},\mathbf{w})$ is of full rank. Q.E.D.

The equation $u_0 = 1$, absent in [5], plays a crucial role for the system (22) to be nonsingular, and this nonsingularity warrants the local convergence of the Gauss-Newton iteration.

Theorem 4.1 *Let $\tilde{u} = GCD(f, f')$ with $\tilde{v}$ and $\tilde{w}$ satisfying (22), and let W be a weight matrix. Then there exists $\varepsilon > 0$ such that for all $\mathbf{u}_0$, $\mathbf{v}_0$, $\mathbf{w}_0$ satisfying $\left\| \mathbf{u}_0 - \tilde{\mathbf{u}} \right\|_2 < \varepsilon$, $\left\| \mathbf{v}_0 - \tilde{\mathbf{v}} \right\|_2 < \varepsilon$ and $\left\| \mathbf{w}_0 - \tilde{\mathbf{w}} \right\|_2 < \varepsilon$, the Gauss-Newton iteration*

$$\begin{bmatrix} \mathbf{u}_{j+1} \\ \mathbf{v}_{j+1} \\ \mathbf{w}_{j+1} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_j \\ \mathbf{v}_j \\ \mathbf{w}_j \end{bmatrix} - J(\mathbf{u}_j, \mathbf{v}_j, \mathbf{w}_j)_W^{+} \begin{bmatrix} \mathbf{e}_1^\top \mathbf{u}_j & - & 1 \\ conv(\mathbf{u}_j, \mathbf{v}_j) & - & \mathbf{f} \\ conv(\mathbf{u}_j, \mathbf{w}_j) & - & \mathbf{f}' \end{bmatrix}, \quad j = 0, 1, \cdots \tag{25}$$

converges to $[\tilde{\mathbf{u}}, \tilde{\mathbf{v}}, \tilde{\mathbf{w}}]^\top$ quadratically. Here $J(\cdot)_W^{+} = [J(\cdot)^H W^2 J(\cdot)]^{-1} J(\cdot)^H W^2$ is the weighted pseudo-inverse of the Jacobian $J(\cdot)$ as defined in (23).

Proof. A straightforward verification by using Lemma 2.6 and Lemma 4.1. Q.E.D.

4.2.3 Setting up the initial iterate

We now need initial iterates $\mathbf{u}_0, \mathbf{v}_0, \mathbf{w}_0$ for the Gauss-Newton iteration (25). In `STEP 1`, when the singular value ς_k is calculated, the associated singular vector $\mathbf{y}_k$ consists of $\mathbf{v}_0$ and $\mathbf{w}_0$ that are approximations to $\mathbf{v}$ and $\mathbf{w}$ in (22) respectively (see Lemma 2.3). Because of the column rotation in §4.2.1, the odd and even entries of $\mathbf{y}_k$ form $\mathbf{v}_0$ and $\mathbf{w}_0$ respectively. For the initial approximation $\mathbf{u}_0$, notice that in theory the long division yields,

$$f(x) = v_0(x) q(x) + r(x) \tag{26}$$

with $u_0(x) = q(x)$ and $r(x) = 0$. The process itself may not be numerically stable. By Lemma 2.3, the long division (26) with $r(x) = 0$ is equivalent to solving the linear system

$$C_{m+1}(v_0)\, \mathbf{u}_0 = \mathbf{f} \tag{27}$$

for a least squares solution $\mathbf{u}_0$ that minimizes $\left\| conv(\mathbf{u}_0, \mathbf{v}_0) - \mathbf{f} \right\|_2$. This "least squares division" is more accurate than the long division (26). In fact, the long division (26) is equivalent to solving the $(n+1)\times(n+1)$ lower triangular linear system

$$L_{m+1}(v_0)\begin{pmatrix}\mathbf{q}\\ \mathbf{r}\end{pmatrix}=\mathbf{f},\quad \text{with}\quad L_{m+1}(v_0)=\left(\begin{array}{c|c} C_{m+1}(v_0) & \begin{array}{c}0_{(m+1)\times(n-m)}\\ I_{(n-m)\times(n-m)}\end{array}\end{array}\right). \tag{28}$$

The following theorem indicates that solving (27) for $\mathbf{u}_0$ may be more preferable than using the long division (26).

Theorem 4.2 *Let $\kappa(A)$ denote the condition number of an arbitrary matrix A with respect to the matrix 2-norm. Then $\kappa\left(C_{m+1}(v)\right)\le\kappa(L_{m+1}(v))$ for any polynomial v and $m>0$.*

Proof. For any matrix A, $\kappa(A)=\frac{\sigma_{max}(A)}{\sigma_{min}(A)}$, where $\sigma_{max}(A)=\max_{\|\mathbf{x}\|_2=1}\left\|A\mathbf{x}\right\|_2$ and $\sigma_{min}(A)=\min_{\|\mathbf{x}\|_2=1}\left\|A\mathbf{x}\right\|_2$ are the largest and smallest singular values of A respectively. Therefore

$$\begin{aligned}\sigma_{max}(C_{m+1}(v)) &= \max_{\|\mathbf{u}\|_2=1}\left\|C_{m+1}(v)\,\mathbf{u}\right\|_2=\max_{\|\mathbf{q}\|_2=1,\mathbf{r}=0}\left\|C_{m+1}(v)\,\mathbf{q}+\mathbf{r}\right\|_2\\ &= \max_{\|\mathbf{q}\|_2=1,\mathbf{r}=0}\left\|L_{m+1}(v)\begin{pmatrix}\mathbf{q}\\ \mathbf{r}\end{pmatrix}\right\|_2 \le \max_{\|y\|_2=1}\left\|L_{m+1}(v)\mathbf{y}\right\|_2=\sigma_{max}(L_{m+1}(v)).\end{aligned}$$

Similarly, $\sigma_{min}(C_{m+1}(v))\ge\sigma_{min}(L_{m+1}(v))$, and consequently, $\kappa(C_{m+1}(v))\le\kappa(L_{m+1}(v))$. Q.E.D.

The magnitude gap between the condition numbers $\kappa(C_{m+1}(v))$ and $\kappa(L_{m+1}(v))$ can be tremendous for seemingly harmless v and moderate m. Actually, $L_{m+1}(v)$ can be pathetically ill-conditioned, making the long division (26) virtually a singular process, while $C_{m+1}(v)$ is still well conditioned. For example, consider a simple polynomial $v(x)=x+25$. When m increases, $\kappa(L_{m+1}(v))$ grows exponentially but $\kappa(C_{m+1}(v))$ stays as nearly a constant, see Table 5. In

	$m=1$	$m=5$	$m=10$	$m=20$
$\kappa(C_{m+1}(v))$	1	1.0668	1.0791	1.0823
$\kappa(L_{m+1}(v))$	627	1.01×10^{7}	9.92×10^{13}	9.46×10^{27}

Table 5: The comparison between the conditions of (26) and (27) for $v(x)=x+25$.

fact, we have not encountered a truly ill-conditioned least squares division (27) in our extensive numerical experiments. On the other hand, the example shown in Table 6 is quite common. In which $\mathbf{f}=conv(\mathbf{u},\mathbf{v})$ is rounded up at the eighth digit after decimal point. The difference between the long division (Matlab `deconv`) and the least squares division is quite substantial.

Extracting $\mathbf{v}_0$ and $\mathbf{w}_0$ from the singular vector and solving (27) for $\mathbf{u}_0$, we shall use them as the initial iterates for the Gauss-Newton iteration (25) that refines the GCD triplet. Moreover, the linear system (27) is banded, with bandwidth being one plus the number of distinct roots. Therefore, the cost of solving (27) is insignificant in the overall complexity.

4.2.4 Refining the GCD with the Gauss-Newton iteration

The Gauss-Newton iteration is expected to reduce the residual

$$\left\|\begin{pmatrix}conv(\mathbf{u}_j,\mathbf{v}_j)\\ conv(\mathbf{u}_j,\mathbf{w}_j)\end{pmatrix}-\begin{pmatrix}\mathbf{f}\\ \mathbf{f}'\end{pmatrix}\right\|_W=\left\|W\begin{pmatrix}conv(\mathbf{u}_j,\mathbf{v}_j) & - & \mathbf{f}\\ conv(\mathbf{u}_j,\mathbf{w}_j) & - & \mathbf{f}'\end{pmatrix}\right\|_2 \tag{29}$$

approx. coef. of $f(x)$	coefficients of $v(x)$	known coef.'s of $f(x) \div v(x)$	least squares division	long division
1.00000000	1.00000000	1.00000000	0.9999999999	1.00000000
23.35360257	23.01829201	0.33531056	0.3353105599	0.33531056
29.89831582	22.05776405	0.12227539	0.1222753902	0.122275385
10.75803809		0.54726624	0.5472662398	0.5472663
15.57240922		0.27815340	0.2781534002	0.278151
18.76038493		0.28629915	0.2862991496	0.28634
13.73079603		1.00523653	1.0052365305	1.004
30.45600101		1.00205392	1.0020539195	1.02
46.21275197		0.97391204	0.9739120403	0.5
44.89871211		0.37785145	0.3778514500	11.
30.17981700				
8.33455813				

Table 6: A numerical comparison between long division and least squares division

at each step until it is numerically unreducible. We stop the iteration when this residual no longer decreases. The diagonal weight matrix W is used to scale the GCD system (22) so that the entries of $W \begin{bmatrix} \mathbf{f} \\ \mathbf{f}' \end{bmatrix}$ are of similar magnitude. Each step of the Gauss-Newton iteration requires solving an overdetermined linear system

$$\left[\, WJ(\mathbf{u}_j, \mathbf{v}_j, \mathbf{w}_j) \,\right] \mathbf{z} = W \begin{bmatrix} \mathbf{e}_1^\top \mathbf{u}_j & - & 1 \\ conv(\mathbf{u}_j, \mathbf{v}_j) & - & \mathbf{f} \\ conv(\mathbf{u}_j, \mathbf{w}_j) & - & \mathbf{f}' \end{bmatrix}$$

for its least squares solution $\mathbf{z}$, and requires a QR decomposition of the Jacobian $WJ(\mathbf{u}_j, \mathbf{v}_j, \mathbf{w}_j)$ and a backward substitution for an upper triangular linear system. This Jacobian is a sparse matrix with a special sparsity structure that can largely be preserved during the process. Figure 8 shows the typical sparsity of $WJ(\mathbf{u}, \mathbf{v}, \mathbf{w})$ along with its triangularization. When f is a polynomial of degree n, a straightforward QR decomposition of $WJ(\mathbf{u}, \mathbf{v}, \mathbf{w})$ costs $O(n^3)$ flops. Taking the sparsity of $WJ(\mathbf{u}, \mathbf{v}, \mathbf{w})$ into account, it can be verified that the sparse QR decomposition costs $O(mk^2 + m^2k + k^3)$, where, as before, k is the number of distinct roots and $m = n - k$. The complexity is significantly reduced to between $O(n^2)$ and $O(n^3)$.

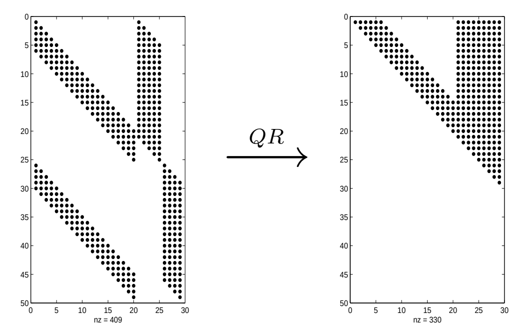

Figure 8: Sparsity of $WJ(\mathbf{u}, \mathbf{v}, \mathbf{w})$ and its triangularization

4.3 Computing the multiplicity structure

For a given polynomial p, the procedure (20) produces a sequence of square-free polynomial $\left\{ v_j \right\}_{j=1}^{s}$. Let $d_j = deg(v_j)$, $j = 1, 2, \cdots, s$, with $k = d_1$ being the number of distinct roots of p. It is straightforward to verify that the multiplicity structure is $\ell = [\ell_1, \cdots, \ell_k]$ with

$$\ell_j = \max \left\{ t \,\middle|\, d_t \geq k - j + 1 \right\}, \quad j = 1, 2, \cdots, k. \tag{30}$$

Moreover, an l-fold root of $p(x)$ appears l times as a simple root of each $v_m(x)$, $m = 1, 2, \cdots, l$. Using the multiplicity structure determined by (30), we can group the numerically "identical" roots of $\left\{ v_j \right\}_1^s$ and obtain the initial root approximation.

4.4 Control parameters

We use three control parameters for the recursive GCD computation. The default values of those parameters given below are selected under the assumption that the IEEE standard double precision of 16 decimal digits is used. The first control parameter is the *zero singular value threshold* θ for identifying the zero singular value. The default choice is $\theta = 10^{-8}$. When the smallest singular value ς_l of $\hat{S}_l(u_{m-1})$ is less than $\theta \left\| \mathbf{u}_{m-1} \right\|_2$, it will be *tentatively* considered as a zero (pending confirmation from the residual information produced by the Gauss-Newton iteration). Then the Gauss-Newton iteration is initiated to further reduce the residual as in (29) to its numerical limit. We use the second control parameter, the *initial residual tolerance* ϱ, to decide if the refined residual is acceptable. Our default choice is $\varrho = 10^{-10}$. We accept the GCD triplet (u_m, v_m, w_m) when the residual

$$\rho_m = \left\| \begin{pmatrix} conv(\mathbf{u}_m, \mathbf{v}_m) - \mathbf{u}_{m-1} \\ conv(\mathbf{u}_m, \mathbf{w}_m) - \mathbf{u}'_{m-1} \end{pmatrix} \right\|_W \leq \varrho \left\| \mathbf{u}_{m-1} \right\|_2. \tag{31}$$

Otherwise, we continue to update $S_l(u_{m-1})$ to $S_{l+1}(u_{m-1})$ and check $\varsigma_{l+1}, \cdots$.

The third parameter is the *residual tolerance growth factor* ϕ. Whenever a GCD triplet (u_m, v_m, w_m) and ρ_m are calculated, The error in (u_m, v_m, w_m) may cause the residual ρ_{m+1} of $(u_{m+1}, v_{m+1}, w_{m+1})$ to grow. Therefore, the tolerance ϱ may need adjustment. Our default growth factor is 100. After obtaining ρ_m, the residual tolerance ϱ is adjusted to be $\max\left\{ \varrho, \ \ \phi\, \rho_m \right\}$. Notice that the growth factor is applied to residual ρ_m rather than the residual tolerance ϱ. The residual tolerance ϱ itself may not grow at every step.

From our computing experience, the default control parameters works well for "normal" polynomials, such as those with unclustered roots of moderate multiplicities. For difficult problems, one may manually adjust the parameters. The overall Algorithm II shown in Fig. 9 is implemented as a Matlab code GcdRoot and included in the MultRoot package.

4.5 Remarks on the convergence of Algorithm II

There are two iterative components in Algorithm II. One of them is the inverse iteration (1). By Lemma 2.4, the iteration converges for all starting vector $\mathbf{x}_0$, unless $\mathbf{x}_0$ is orthogonal to the intended singular vector $\mathbf{y}$. The probability of the occurrence of this orthogonality is zero. But even if it occurs, roundoff errors in the numerical computation will quickly destroy the orthogonality during iteration. Therefore, the inverse iteration (1) always converges in practice. The other iterative component is the Gauss-Newton iteration (25) whose local convergence is ensured in Theorem 4.1. Therefore, as long as the rank decision on the Sylvester matrices is accurate and the error on the initial approximation of the GCD triplet is small, Algorithm II will produce correct multiplicity structure and an excellent root approximation.

However, due to the nature of the problem, there is no guarantee that the original multiplicity structure can be identified from an inexact polynomial. When a polynomial is perturbed to a place that has equal distances to two or more different pejorative manifolds, it is somewhat unrealistic to expect any method to recover reliably from the perturbation. Therefore, we have conducted extensive numerical experiments in addition to the results exhibited in this paper. As reported in our software release note [32], we made a comprehensive test suit of 104 polynomials based on Jenkins-Traub Testing Principles [18]. These polynomials include

Pseudo-code GcdRoot (Algorithm II)

input: The polynomial p of degree n, singular threshold θ,
residual tolerance ϱ, residual growth factor ϕ.
(If only p is provided, set $\theta = 10^{-8}$, $\varrho = 10^{-10}$, $\phi = 100$)

output: the root estimates $(z_1, \cdots, z_k)^\top$ and multiplicity structure $[\ell_1, \cdots, \ell_k]$

Initialize $u_0 = p$
for $m = 1, 2, \cdots, s$, until $deg(u_s) = 0$ do
  for $l = 1, 2, \cdots$ until residual $\rho < \varrho \left\| \mathbf{u}_{m-1} \right\|_2$ do
    calculate the singular pair $(\varsigma_l, \mathbf{y}_l)$ of $\hat{S}_l(u_{m-1})$ by iteration (1)
    if $\varsigma_l < \theta \left\| \mathbf{u}_{m-1} \right\|_2$ then
      set up the GCD system (22) with $f = u_{m-1}$ (see Section 4.2.2)
      extract $v_m^{(0)}, w_m^{(0)}$ from $\mathbf{y}_l$ and calculate $u_m^{(0)}$ (see Section 4.2.3)
      apply the Gauss-Newton iteration (25) from
        $u_m^{(0)}, v_m^{(0)}, w_m^{(0)}$ to obtain u_m, v_m, w_m
      extract the residual $\rho = \rho_m$ as in (31)
    end if
  end do
  adjust the residual tolerance ϱ to be $\max\{\varrho, \, \phi\rho_j\}$, and set $d_m = deg(v_m)$
end do
set $k = d_1$, $\ell_j = \max\left\{ t \,\middle|\, d_t \geq k - j + 1 \right\}$, $j = 1, 2, \cdots, k.$
match the roots of $v_m(x)$, $m = 1, 2, \cdots, s$ according to the multiplicities ℓ_j's.

Figure 9: Pseudo-code of Algorithm II

all the test examples we have seen in the literature that have been used by experts to test robustness, stability, accuracy and efficiency of root-finders intended for multiple roots. On all the polynomials with multiple roots in the test suit, our package MultRoot consistently outputs accurate root/multiplicity results near machine precision. They are far beyond the "attainable accuracy" barrier that other algorithms are subject to. The report [32] along with the test suit is electronically available in the author's homepage[2].

4.6 Numerical results for Algorithm II

The effectiveness of Algorithm II can be shown by the polynomial

$$p(x) = (x-1)^{20}(x-2)^{15}(x-3)^{10}(x-4)^5 \tag{32}$$

generated by Matlab function poly, with coefficients rounded up at 16 digits. Using the default control parameters, Algorithm II code GcdRoot correctly identifies the multiplicity structure. The roots are approximated to an accuracy of 10 digits or more. With this result as input to Algorithm I code PejRoot, we obtained all multiple roots in the end with at least 14 correct digits (Table 7).

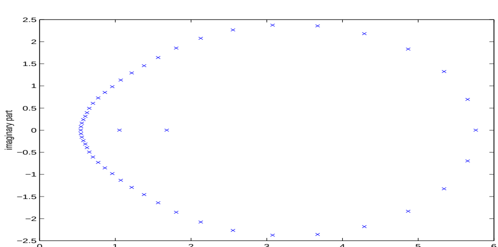

Figure 10: MPSolve results for the polynomial (32) using multiprecision

[2] http://www.neiu.edu/ zzeng/multroot.htm

```
Algorithm II (code GcdRoot) result:              |    Algorithm I (code PejRoot) result
                                                 |      THE BACKWARD ERROR:                     6.16e-016
The backward error is 6.057721e-010              |      THE ESTIMATED FORWARD ROOT ERROR:    9.46e-014
                                                 |
        computed roots          multiplicities   |         computed roots          multiplicities
                                                 |
        4.000000000109542              5         |         3.999999999999985              5
        3.000000000176196             10         |         3.000000000000011             10
        2.000000000030904             15         |         1.999999999999997             15
        1.000000000000353             20         |         1.000000000000000             20
```

Table 7: Roots of $p(x)$ in (32) computed in two stages

Polynomials with such high multiplicities are extremely difficult by any standard for root-finding. The magnitude of its coefficients stretches from 1 to 10^{21}. Remarkably, our algorithms have no difficulty finding all its multiple roots. To the best of our knowledge, there are no other methods that can calculate multiple roots for such polynomials. Since the coefficients are inexact, multiprecision root-finders also fail to calculate the roots with meaningful accuracy. Figure 10 shows the computed roots by MPSolve [2] using virtually unlimited number of digits in machine precision. Those results are quite remote from the roots 1, 2, 3, 4.

The Euclidean method has also been used to find GCD in order to identify the multiplicities [3, 26]. Uhlig's PZERO [26] is a Matlab implementation based on the Euclidean method. The drawback of the Euclidean method is its reliance on recursive long division that is numerically unstable (see §4.2.3). Here we compare our code GCDROOT with PZERO on the polynomials

$$p_k(x) = (x-1)^{4k}(x-2)^{3k}(x-3)^{2k}(x-4)^{k} \quad \text{for } k = 1, 2, \cdots, 8.$$

When the multiplicities increase, the root accuracy deteriorates with PZERO, which successfully identifies the multiplicity structure for $k = 1$ and $k = 2$ but fails to do so afterwards. In comparison, GCDROOT consistently attains at least 11 digits in root accuracy with increasing multiplicities. The multiplicity structures are identified correctly for k up to 7 and multiplicities up to 28. For the current implementation, the limitation of GCDROOT on this sequence is for $k \leq 7$, whereas the root accuracy will stay the same for even larger k.

k	code	$x_1 = 1$		$x_2 = 2$		$x_3 = 3$	
$k=1$	PZERO	1.00000000001	(4)	1.99999999998	(3)	3.000000000005	(2)
	GCDROOT	0.999999999999990	(4)	1.99999999999998	(3)	3.0000000000005	(2)
$k=2$	PZERO	1.0000000001	(8)	2.000000002	(6)	3.000000004	(4)
	GCDROOT	0.9999999999998	(8)	1.999999999983	(6)	2.99999999991	(4)
$k=3$	PZERO	0.9999999897	(**13**)	1.99999990	(**8**)	2.9999998	(**5**)
	GCDROOT	0.9999999999997	(12)	1.999999999997	(9)	2.9999999998	(6)
$k=4$	PZERO	0.9999995	(**21**)	1.999994	(**6**)	2.999990	(**7**)
	GCDROOT	1.0000000000003	(16)	2.00000000002	(12)	3.0000000001	(8)
$k=5$	PZERO	1.0000009	(**28**)	2.00001	(**8**)	3.00002	(**6**)
	GCDROOT	1.0000000000004	(20)	2.00000000003	(15)	3.0000000002	(10)
$k=6$	PZERO	− − − −	(**1**)	− − − −	(**1**)	− − − −	(**1**)
	GCDROOT	1.0000000000002	(24)	2.00000000001	(18)	3.00000000004	(12)
$k=7$	PZERO	− − − −	(**1**)	− − − −	(**1**)	− − − −	(**1**)
	GCDROOT	1.0000000000001	(28)	2.00000000001	(21)	3.00000000006	(14)

Table 8: Partial results on $p_k(x) = (x-1)^{4k}(x-2)^{3k}(x-3)^{2k}(x-4)^{k}$ and comparison between PZERO and GCDROOT. Numbers in parenthesis are computed multiplicities. Wrong multiplicities are in boldface.

5 Numerical results for the combined method

5.1 The effect of inexact coefficients

In application, input data are expected to be inexact. The following experiment tests the effect of data error on the accuracy as well as robustness of both Algorithm I and II. For

$$p(x) = \left(x - \frac{10}{11}\right)^5 \left(x - \frac{20}{11}\right)^5 \left(x - \frac{30}{11}\right)^5$$

in general form, every coefficient is rounded up to k-digit accuracy, where $k = 10, 9, 8, \cdots$.

number of correct digits	control parameters ϱ, θ	code	$x_1 = 0.\dot{9}\dot{0}$	$x_2 = 1.\dot{8}\dot{1}$	$x_3 = 2.\dot{7}\dot{2}$	backward error
$k = 10$	$\varrho = 1e-9$	GcdRoot	0.90909090	1.8181818	2.7272727	1.7e-08
	$\theta = 1e-7$	PejRoot	0.909090909	1.81818181	2.7272727	2.4e-10
$k = 9$	$\varrho = 1e-8$	GcdRoot	0.909090	1.81818	2.72727	7.0e-06
	$\theta = 1e-6$	PejRoot	0.9090909	1.8181818	2.727272	2.3e-09
$k = 8$	$\varrho = 1e-7$	GcdRoot	0.90909	1.8182	2.727	1.3e-04
	$\theta = 1e-5$	PejRoot	0.9090909	1.818181	2.72727	2.3e-08
$k = 7$	$\varrho = 1e-6$	GcdRoot	0.9090	1.82	2.7	1.3e-02
	$\theta = 1e-4$	PejRoot	0.90909	1.81818	2.7272	2.3e-07
$k = 6$	$----$	PejRoot	0.9090	1.8181	2.727	3.7e-06
$k = 5$	$----$	PejRoot	0.909	1.818	2.72	2.4e-05
$k = 4$	$----$	PejRoot	0.90	1.81	2.7	1.9e-04
$k = 3$	$----$	PejRoot	0.9	1.8	2.8	1.8e-03

Table 9: Effect of coefficient error on computed roots

For this sequence of problems, Algorithm II code GcdRoot correctly identifies the multiplicity structure if the coefficients have at least 7 accurate digits. If the multiplicities are manually given rather than computed by GcdRoot, Algorithm I code PejRoot continues to converge even when data accuracy is down to 3 digits. For lower data accuracy, the residual tolerance ϱ in GcdRoot needs to be adjusted accordingly. Table 9 shows the results of both programs.

As shown in this test, both methods allow inexact coefficients to certain extent. As usual, Algorithm I is more robust than Algorithm II, but Algorithm I depends on a structure identifier.

5.2 The effect of nearby multiple roots

When two or more multiple roots are nearby, it can be difficult to identify the correct multiplicity structure. We test the example

$$p_\varepsilon(x) = (x - 1 + \varepsilon)^{20}(x-1)^{20}(x+0.5)^5$$

for decreasing root gap $\varepsilon = 0.1,\ 0.01, \cdots$, making the root $x_1 = 0.9,\ 0.99,\ 0.999, \cdots$ along with fixed roots $x_2 = 1$ and $x_3 = -0.5$. When root gap decreases, the control parameters may need adjustment. In this test, we use the default parameters for all cases except for $\varepsilon = 0.0001$, in which case, the residual growth factor $\phi = 5$. GcdRoot is used to find the initial input for PejRoot. Computing results are shown for both programs in Table 10.

When the default growth factor stays the same as the default $\phi = 100$ and the gap $\varepsilon \leq 0.0001$, GcdRoot outputs a multiplicity structure $[40, 5]$. Namely, GcdRoot treats the two nearby 20-fold roots 1 and $1 - \varepsilon$ as a single 40-fold one. From the computed backward error and

root gap ε	code	$x_1 = 1 - \varepsilon$	$x_2 = 1$	$x_3 = -0.5$	backward error	condition number
$\varepsilon = 0.1$	GcdRoot	0.89999999999	0.99999999999	-0.49999999999999	9.7e-10	
	PejRoot	0.9000000000000	0.9999999999999	-0.50000000000000	2.7e-13	.7
$\varepsilon = 0.01$	GcdRoot	0.98999999	0.99999999	-0.50000000000000	3.2e-07	
	PejRoot	0.989999999999	1.000000000000	-0.49999999999999	1.0e-12	6.7
$\varepsilon = 0.001$	GcdRoot	0.99900	1.00000	-0.49999999999999	1.9e-04	
	PejRoot	0.99899999999	1.00000000000	-0.500000000000000	4.1e-13	62.5
$\varepsilon = 0.0001$	GcdRoot	0.9997	0.99996	-0.4999999999999	1.1e-02	
	PejRoot	0.999900000	0.999999999	-0.50000000000000	4.0e-12	621.7
$\varepsilon = 0.00001$	PejRoot	0.999989990	1.0000000	-0.50000000000000	4.0e-10	5791.8

Table 10: Effect of decreasing root gap on computed roots

the condition number, this may not necessarily be incorrect. See Table 11. When backward error becomes 10^{-12} and condition number is tiny (0.0066), they are numerically accurate! In contrast, using the "correct" multiplicity structure $[20, 20, 5]$, PejRoot outputs roots with backward error 10^{-10} and a large condition number 5791.8 (last line in Table 10).

root gap ε	code	$x_1 = 1 - \varepsilon$	$x_2 = 1$	$x_3 = -0.5$	backward error	condition number
$\varepsilon = 0.0001$	GcdRoot	0.99994999	0.99994999	-0.5000000000	5.7e-08	
	PejRoot	0.999949999	0.999949999	-0.500000000	2.2e-08	0.0066
$\varepsilon = 0.00001$	GcdRoot	0.9999949999	0.9999949999	-0.500000000000	1.1e-10	
	PejRoot	0.99999499999	0.99999499999	-0.50000000000	4.0e-12	0.0066

Table 11: If the control parameter is not adjusted, tiny root gap makes computed roots identical. However, from the backward errors and the condition number, they are not necessarily wrong answers.

By adjusting the control parameters, GcdRoot can find different pejorative manifolds that are close to the given polynomial. PejRoot then calculates corresponding pejorative roots. The selection of the most suitable solution should be application dependent.

5.3 A large inexact problem

Implementing the combination of two methods, we have produced a Matlab code MultRoot. We conclude this report by testing this code on our final test problem. First of all, twenty complex numbers are randomly generated and used as roots

$.5 \pm i,\ -1 \pm .2i,\ -.1 \pm i,\ -.8 \pm .6i,\ -.7 \pm .7i,\ 1.4,\ -.4 \pm .9i,\ .9,\ -.8 \pm .3i,\ .3 \pm .8i,\ .6 \pm .4i$

to generate a polynomial f of degree 20. We then round all coefficients to 10 decimal digits. The coefficients are shown in the right. We construct multiple roots by squaring f repeatedly. Namely,

$$g_k(x) = [\, f(x)\,]^{2^k}, \quad k = 1, 2, 3, 4, 5.$$

At $k = 5$, g_5 has a degree 640 and twenty complex roots of multiplicity 32. Since the machine precision is 16 digits, the polynomials g_k are inexact. Using the default control parameters, our combined program encounters no difficulty in calculating all the roots as well as finding accurate multiplicities. The worst accuracy of the roots is 11-digit. Here is the final result.

```
coefficients of f
    1
   -0.7
   -0.19
    0.177
   -0.7364
   -0.43780
   -0.952494
   -0.2998258
   -0.00322203
   -0.328903811
   -0.4959527435
   -0.9616679762
    0.4410459281
    0.1090273141
    0.6868094008
    0.0391923826
    0.0302248540
    0.6603775863
   -0.1425784968
   -0.3437618593
    0.4357949015
```

```
 THE STRUCTURE PRESERVING CONDITION NUMBER:           0.0780464
 THE BACKWARD ERROR:                            6.38e-012
 THE ESTIMATED FORWARD ROOT ERROR:              9.96e-013

 computed roots                         multiplicities |  computed roots                        multiplicities
                                                       |
 0.499999999999399 + 1.000000000006247 i        32     |  1.400000000000303 + 0.000000000000000 i      32
 0.499999999999399 - 1.000000000006247 i        32     | -0.399999999999482 + 0.899999999996264 i      32
-1.000000000003141 + 0.200000000004194 i        32     | -0.399999999999482 - 0.899999999996264 i      32
-1.000000000003140 - 0.200000000004193 i        32     |  0.899999999996995 - 0.000000000000000 i      32
-0.099999999996612 + 1.000000000001018 i        32     | -0.799999999987544 + 0.299999999995441 i      32
-0.099999999996612 - 1.000000000001018 i        32     | -0.799999999987544 - 0.299999999995441 i      32
 0.800000000001492 + 0.600000000001814 i        32     |  0.299999999995789 + 0.799999999976189 i      32
 0.800000000001492 - 0.600000000001815 i        32     |  0.299999999995789 - 0.799999999976189 i      32
-0.699999999997635 + 0.699999999997984 i        32     |  0.599999999989084 + 0.399999999997279 i      32
-0.699999999997635 - 0.699999999997984 i        32     |  0.599999999989084 - 0.399999999997279 i      32
```

Acknowledgment. The author wishes to thank Frank Uhlig and Peter Kravanja for freely sharing their codes and knowledge. The author is also grateful to Ross Lippert, Hans Stetter, Joab Winkler and anonymous referees for their valuable suggestions on the presentation. We are indebted to Barry Dayton for finding out an error in the early version of the manuscript.

References

[1] D. H. Bailey, *A Fortran-90 based multiprecision system*, ACM Trans. Math. Software, 21 (1995), pp. 379–387.

[2] D. Bini and G. Fiorentino, *Numerical computation of polynomial roots using MPSolve – version 2.0.* manuscript, Software and paper available at `ftp://ftp.dm.unipi.it/pub/mpsolve/`, 1999.

[3] L. Brugnanao and D. Trigiante, *Polynomial roots: the ultimate answer?*, Linear Alg. and Its Appl., 225 (1995), pp. 207–219.

[4] P. Chin, R. M. Corless, and G. F. Corless, *Optimization strategies for the approximate GCD problem*, in ISSAC 1998, New York, 1998, ACM Press, pp. 228–235.

[5] R. M. Corless, P. M. Gianni, B. M. Trager, and S. M. Watt, *The singular value decomposition for polynomial systems*, in Proc. ISSAC 1995, New York, 1995, ACM Press, pp. 195–207.

[6] J.-P. Dedieu and M. Shub, *Newton's method for over-determined system of equations*, Math. Comp., 69 (1999), pp. 1099–1115.

[7] J. W. Demmel, *On condition numbers and the distance to the nearest ill-posed problem*, Numer. Math., 51 (1987), pp. 251–289.

[8] J. W. Demmel and B. Kagström, *The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$: robust software with error bounds and applications. part I & part II*, ACM Trans. Math. Software, 19 (1993), pp. 161–201.

[9] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall Series in Computational Mathematics, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[10] A. Edelman, E. Elmroth, and B. Kagström, *A geometric approach to perturbation theory of matrices and and matrix pencils. part I: Versal deformations*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 693–705.

[11] ——, *A geometric approach to perturbation theory of matrices and and matrix pencils. part II: a stratification-enhanced staircase algorithm*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 667–699.

[12] I. Z. Emiris, A. Galligo, and H. Lombardi, *Certified approximate univariate GCDs*, J. Pure Appl. Algebra, 117/118 (1997), pp. 229–251.

[13] M. R. Farmer and G. Loizou, *An algorithm for the total, or partial, factorization of a polynomial*, Math. Proc. Camb. Phil. Soc., 82 (1977), pp. 427–437.

[14] S. Fortune, *An iterated eigenvalue algorithm for approximating roots of univariate polynomials*, J. Symbolic Comput., 33 (2002), pp. 627–646.

[15] W. Gautschi, *Questions of numerical condition related to polynomials*, in MAA Studies in Mathematics, Vol. 24, Studies in Numerical Analysis, G. H. Golub, ed., USA, 1984, The Mathematical Association of America, pp. 140–177.

[16] V. Hribernig and H. J. Stetter, *Detection and validation of clusters of polynomial zeros*, J. Symb. Comput., 24 (1997), pp. 667–681.

[17] M. Igarashi and T. Ypma, *Relationships between order and efficiency of a class of methods for multiple zeros of polynomials*, J. Comput. Appl. Math., 60 (1995), pp. 101–113.

[18] M. A. Jenkins and J. F. Traub, *Principles for testing polynomial zerofinding programs*, ACM Trans. Math. Software, 1 (1975), pp. 26–34.

[19] W. Kahan, *Conserving confluence curbs ill-condition.* Technical Report 6, Computer Science, University of California, Berkeley, 1972.

[20] N. K. Karmarkar and Y. N. Lakshman, *On approximate polynomial greatest common divisors*, J. Symb. Comput., 26 (1998), pp. 653–666.

[21] P. Kravanja and M. Van Barel, *Computing Zeros of Analytic Functions, Lecture Notes in Mathematics, 1727*, Springer-Verlag, 2000.

[22] R. A. Lippert and A. Edelman, *The computation and sensitivity of double eigenvalues*, in Advances in computational mathematics, Lecture Notes in Pure and Appl. Math. 202, New York, 1999, Dekker, pp. 353–393.

[23] V. Y. Pan, *Solving polynomial equations: some history and recent progress*, SIAM Review, 39 (1997), pp. 187–220.

[24] D. Rupprecht, *An algorithm for computing certified approximate GCD of n univariate polynomials*, J. Pure and Appl. Alg., 139 (1999), pp. 255–284.

[25] H. J. Stetter, *Condition analysis of overdetermined algebraic problems*, in Computer Algebra in Scientific Computing–CASC 2000, e. a. V.G. Ganzha, ed., Springer, 2000, pp. 345–365.

[26] F. Uhlig, *General polynomial roots and their multiplicities in $O(n)$ memory and $O(n^2)$ time*, Linear and Multilinear Algebra, 46 (1999), pp. 327–359.

[27] S. Van Huffel, *Iterative algorithms for computing the singular subspace of a matrix associated with its smallest singular values*, Linear Alg. Appl., 154-156 (1991), pp. 675–709.

[28] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1963.

[29] J. R. Winkler, *Condition numbers of a nearly singular simple root of a polynomial*, Appl. Numer. Math., (2001), pp. 275–285.

[30] T. J. Ypma, *Finding a multiple zero by transformations and Newton-like methods*, SIAM Review, 25 (1983), pp. 365–378.

[31] D. Y. Y. Yun, *On square-free decomposition algorithms*, in Proceedings of 1976 ACM Symposium of Symbolic and Algebraic Computation (ISSAC'76), R. Janks, ed., ACM Press, Yorktown Heights, New York, 1976, pp. 26–35.

[32] Z. Zeng, *Multroot – a matlab package computing polynomial roots and multiplicities.* manuscript, can be accessed at `http://www.neiu.edu/~zzeng/Papers/zrootpak.ps`, 2003.

[33] ——, *On ill-conditioned eigenvalues, multiple roots of polynomials, and their accurate computation.* MSRI Preprint No. 1998-048, (1998) [3].

[3] `http://www.neiu.edu/~zzeng/Papers/multiple.ps`