CSCI-5646 Project Report
Numerical Methods for Polynomial Root-finding Problem

Jianxiang Fan and Nir Boneh

1 Introduction

A monic polynomial of degree n is a function of the form

$$p_n(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + x^n$$

where the given coefficients $a_k \in \mathbb{R}$.

A root of the polynomial p_n is a value of $z \in \mathbb{C}$ for which $p_n(z) = 0$. If z is a root of p_n, then its complex conjugate $\bar{z}$ is also a root of p_n.

Abel-Ruffini theorem states that for $n \geq 5$ there does not exist an explicit formula for the roots of p_n. This motivates a number of numerical methods for finding roots of p_n. One class of methods is based on translating the problem of finding polynomial roots to the problem of finding eigenvalues of the companion matrix of the polynomial, while another is do iterative approximation directly.

For $p_n(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + x^n$, the associated companion matrix A is defined by

$$A = \begin{bmatrix} 0 & & & & -a_0 \\ 1 & 0 & & & -a_1 \\ & 1 & \ddots & & \vdots \\ & & \ddots & 0 & -a_{n-2} \\ & & & 1 & -a_{n-1} \end{bmatrix}$$

It is easy to show that the characteristic function of A is just $p_n(x)$, so the eigenvalues of companion matrix A are the roots of polynomial p_n. Those QR eigenvalue algorithms applied to the companion matrix are customary for finding polynomial roots.

2 The double-shift Francis's Algorithm

Since the companion matrix is already in an upper Hessenberg shape, the first step of Francis's algorithm will be skipped.

However, the single-shift Francis's algorithm is only practical when we know that all eigenvalues are real, which is not true for a companion matrix associated with a root-finding

problem. In this case, the double-shift version should be applied, which computes all eigenvalues, real or complex conjugate pairs.

Here is the basic idea of the double-shift Francis's algorithm. Suppose in some iteration, matrix A_0 is real and we apply a complex shift μ, the resulting matrix A_1 is, of course, complex. Then, if we do another iteration using the complex conjugate shift $\bar{\mu}$, the resulting matrix A_2 is again real. Francis found such a method [1] that would transform A_0 directly to A_2 and do two iterations of the QR algorithm implicitly, entirely in real arithmetic.

2.1 The Iteration with Double-shift

An iteration of Francis's algorithm of degree m begins by picking m shifts $\rho_1, \cdots, \rho_m$. When $m = 2$, take ρ_1, ρ_2 to be the eigenvalues of the 2×2 submatrix in the lower right corner of A. This can produce complex shifts, and they always occur in conjugate pairs, otherwise they are both real.

Now let $p(A) = (A - \rho_2 I)(A - \rho_1 I)$. We do not actually compute $p(A)$, since Francis's algorithm just needs the first column $x = p(A)e_1 = (A - \rho_2 I)(A - \rho_1 I)e_1$. We can compute $(A - \rho_1 I)e_1$ first by just taking the first column of $(A - \rho_1 I)$, and then do a matrix-vector multiplication to get $(A - \rho_2 I)(A - \rho_1 I)e_1$. The computation is especially cheap because A is upper Hessenberg. It is easy to check $(A - \rho_2 I)(A - \rho_1 I)e_1$ has nonzero entries in only its first 3 positions. The cost is a constant. Since the complex shifts ρ_1 and ρ_2 occur in conjugate pairs, $p(A)$ is real.

Let $\tilde{x} \in \mathbb{R}^3$ denote the vector consisting of the first 3 entries of x (the nonzero part). There is an 3×3 householder reflector $\tilde{Q}_0$ such that $\tilde{Q}_0 \tilde{x} = \alpha e_1$, where $\alpha = \pm \|x\|_2$. Let Q_0 be an $n \times n$ reflector given by

$$Q_0 = \begin{bmatrix} \tilde{Q}_0 & \\ & I \end{bmatrix}$$

Then $Q_0 x = \alpha e_1$.

Now use Q_0 to perform a similarity transformation on A, we get $A \to Q_0^* A Q_0$. The left multiplication leaves the upper Hessenberg form of A intact, while right multiplication by Q_0 puts nonzeros in the $(3,1), (4,1)$ and $(4,2)$ positions of $Q_0^* A Q_0$.

$$\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ + & * & * & * & * & * \\ + & + & * & * & * & * \\ & & & * & * & * \\ & & & & * & * \end{bmatrix}$$

The rest of the Francis iteration consists of returning this matrix to upper Hessenberg form by chasing the bulge. This begins with a reflector Q_1 that acts only on rows 2 through 4 and creates zeros in positions $(3,1)$ and $(4,1)$. Applying Q_1^* on the left, we get a matrix $Q_1^* Q_0 A Q_0$. Completing the similarity transformation, we multiply by Q_1 on the right. This

recombines columns 2 through 4, and additional nonzero entries are created in positions $(5,2)$ and $(5,3)$.

$$\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ & * & * & * & * & * \\ & + & * & * & * & * \\ & + & + & * & * & * \\ & & & & * & * \end{bmatrix}$$

The bulge has been pushed one position to the right and downward. The next transformation will push the bulge over and down one more position, and so on.

The summary of an iteration of double-shift Francis's algorithm:

1. Pick shifts ρ_1 and ρ_2, which are eigenvalues of the 2×2 submatrix in the lower right corner of A
2. Compute $x = p(A)e_1 = (A - \rho_2 I)(A - \rho_1 I)e_1$.
3. Compute a householder reflector Q_0, such that $Q_0 x = \alpha e_1$, where $\alpha = \pm \|x\|_2$.
4. Do a similarity transformation $A \to Q_0^* A Q_0$, creating a bulge.
5. Return the matrix to upper Hessenberg form by chasing the bulge.

Let $\hat{A}$ denote the result of the iteration.

$$\hat{A} = Q_{n-2}^* \cdots Q_1^* Q_0^* A Q_0 Q_1 \cdots Q_{n-2}$$

Now let's see the time complexity. In each iteration, step 1 and step 2 takes constant FLOPs as we mentioned before.

Since

$$Q_k = \begin{bmatrix} I_k & & \\ & \tilde{Q}_k & \\ & & I_{n-k-3} \end{bmatrix}$$

where $\tilde{Q}_k$ is 3×3 Householder reflector, $k = 0, 1, \cdots, n-3$. And

$$Q_{n-2} = \begin{bmatrix} I_{n-2} & \\ & \tilde{Q}_{n-2} \end{bmatrix}$$

where $\tilde{Q}_{n-2}$ is 2×2 Givens rotator.

Therefore, step 3 is just computing a 3×3 Householder reflector, which takes constant time. And step 4 is first combining 3 rows and then combining 3 columns, which takes $15n + 27$ FLOPs. Step 5 is computing 3×3 Householder reflectors $\tilde{Q}_1, \tilde{Q}_2, \cdots, \tilde{Q}_{n-3}$ and then combining 3 rows and 3 columns for each time. The leading term of FLOPs is $\frac{15}{2}n^2$ $(15n + 15(n-1) + \cdots + 15 \cdot 4)$.

Overall each iteration with double-shift costs $\frac{15}{2}n^2$ FLOPs.

2.2 Deflation

The double-shift version doesn't always cause convergence to a triangular form; it is impossible for complex eigenvalues to emerge on the main diagonal of a real matrix. Instead pairs

of complex conjugate eigenvalues emerge in 2×2 blocks along the main diagonal of a block triangular matrix.

A customary deflation strategy is to deflate whenever a subdiagonal element satisfies $|a_{k+1,k}| \leq u(|a_{kk}| + |a_{k+1,k+1}|)$, where u is the unit roundoff.

If the size of the deflated matrix is 1×1, the element is the eigenvalue. If the size of the deflated matrix is 2×2, we will compute its eigenvalues, which can be two real numbers or two conjugate complex numbers.

3 Fast QR algorithms for Companion Matrices

As we have seen, the general routines for the double-shift Francis's require $O(n^2)$ FLOPs per iteration. Over the past 10 years, some authors have been intrigued by the hidden properties of companion matrix, and make improvements to $O(n)$ FLOPs per iteration [3,5].

In the paper by Bini et al.[3], an explicit QR algorithm is applied to an input companion matrix $A \in \mathbb{C}^{n\times n}$ represented as a rank-one perturbation of a unitary matrix, namely, $A = U - \mathbf{z}\mathbf{w}^T$, where $U \in \mathbb{C}^{n\times n}$ is unitary and $\mathbf{z}, \mathbf{w} \in \mathbb{C}^n$. The computational improvement is achieved by exploiting the quasiseparable Hessenberg structure in the QR iterates inherited from the rank properties of the companion matrix A.

In another paper by Bini et al.[4], they modify the algorithm given in [3] to incorporate shift techniques, thus obtaining the implicit QR algorithm for the case where the initial matrix $A = A^{(0)} \in \mathbb{C}^{n\times n}$ is in companion form.

In the section, we'll give a brief introduction of their algorithm. Most of this section is based on the paper by Bini et al.[4]

3.1 $\mathcal{H}_n$ Class Matrix and its Parametrization

The companion matrix can be represented as

$$A = \begin{bmatrix} 0 & & & & 1 \\ 1 & 0 & & & 0 \\ & 1 & \ddots & & \vdots \\ & & \ddots & 0 & 0 \\ & & & 1 & 0 \end{bmatrix} - \begin{bmatrix} p_0 + 1 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix} \begin{bmatrix} 0 & 0 & \cdots & 1 \end{bmatrix}$$

Definition 3.1: $\mathcal{H}_n \subset \mathbb{C}^{n\times n}$ be the class of $n \times n$ upper Hessenberg matrices defined as rank-one perturbations of unitary matrices. That is, $H \in \mathcal{H}_n$ if there exist $U \in \mathbb{C}^{n\times n}$ unitary and $\mathbf{z}, \mathbf{w} \in \mathbb{C}^n$ such that

$$H = U - \mathbf{z}\mathbf{w}^T$$

The vectors $\mathbf{z}, \mathbf{w}$ are called the perturbation vectors of the matrix H.

Lemma 3.2: (Decomposition of U): If $A = U - \mathbf{z}\mathbf{w}^T \in \mathcal{H}_n$, there exist $n-2$ unitary matrix $\mathcal{V}_{n-1}, \mathcal{V}_{n-2}, \cdots, \mathcal{V}_2$, $n-2$ 3×3 unitary matrix $\mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_{n-2}$ and β_j, such that

$$\beta_n = w_n, \begin{bmatrix} \beta_k \\ 0 \end{bmatrix} = \mathcal{V}_k^* \begin{bmatrix} w_k \\ \beta_{k+1} \end{bmatrix}, \; k = n-1, n-2, \cdots, 2$$

$$U = V_{n-1} V_{n-2} \cdots V_2 F_1 F_2 \cdots F_{n-2}$$

where

$$V_k = \begin{bmatrix} I_{k-1} & & \\ & \mathcal{V}_k & \\ & & I_{n-k-1} \end{bmatrix}, k = 2, \cdots, n-1$$

$$F_k = \begin{bmatrix} I_{k-1} & & \\ & \mathcal{F}_k & \\ & & I_{n-k-2} \end{bmatrix}, k = 1 \cdots, n-2$$

The algorithm to get β, $\mathcal{V}_k$ and $\mathcal{F}_k$:

1 $\beta_n = w_n$
2 **for** $k = n-1 : -1 : 2$
3 $\quad [\mathcal{V}_k, \beta_k] = \mathbf{givens}(x_k, \beta_{k+1})$
4 $\quad U(k:k+1,:) = \mathcal{V}_k^* U(k:k+1,:)$
5 **for** $k = 1 : n-3$
6 $\quad \mathcal{F}_k = \mathbf{householder}(U(k:k+2,k))$
7 $\quad U(k:k+2,k:n) = \mathcal{F}_k^* U(k:k+2,k:n)$
8 $\mathcal{F}_{n-2} = U(n-2:n, n-2:n)$

Summing up, the matrix $A = U - \mathbf{z}\mathbf{w}^T \in \mathcal{H}_n$ is completely specified by the following parameters:

(1) the 2×2 unitary matrices $\mathcal{V}_k$, $k = 2, \cdots, n-1$
(2) the 3×3 unitary matrices $\mathcal{F}_k$, $k = 1, \cdots, n-2$
(3) the perturbation vectors $\mathbf{z}$ and $\mathbf{w}$.

These parameters are also called the *generating elements* of the matrix A.

The decomposition of U by means of the elementary matrices $\mathcal{V}_k$ and $\mathcal{F}_k$ is not easy to be manipulated under the QR iteration applied to A. In this respect, a more suited condensed form of U is its quasiseparable parametrization. The representation gives an explicit description of each entry of U as the product of certain vectors and matrices of small size. For every matrix from the class U_n there exist vectors $\boldsymbol{g}_j \in \mathbb{C}^2, 1 \le j \le n, \boldsymbol{h}_j \in \mathbb{C}^2, 1 \le j \le n$, and matrices $B_j \in \mathbb{C}^{2\times 2}, 2 \le j \le n$, such that

$$u_{i,j} = \boldsymbol{g}_i^T B_{i,j}^{\times} \boldsymbol{h}_j, \text{ for } ji \ge 0$$

where $B_{i,j}^{\times} = B_{i+1} \cdots B_j$ for $i+1 \le j \le n$ and $B_{i,j}^{\times} = I_2$ if $i = j$. The elements $\boldsymbol{g}_j$, $\boldsymbol{h}_j (j = 1, \cdots, n)$, $B_j (j = 2, \cdots, n)$ are called *upper generators* of the matrix U.

Theorem 3.3: Suppose $\{\mathcal{V}_k\}_2^{n-1}, \{\mathcal{F}_k\}_1^{n-2}, \mathbf{z}, \mathbf{w}$ are the generating elements of an $\mathcal{H}_n$ class matrix $A = U - \mathbf{z}\mathbf{w}^T \in \mathcal{H}_n$. The entries $u_{i,j}$, $\max\{1, i-2\} \le j \le n, 1 \le i \le n$, satisfy the following relations

$$u_{i,j} = \boldsymbol{g}_i^T B_{i,j}^{\times} \boldsymbol{h}_j \text{ for } j - i \ge 0$$

$$u_{i,j} = \sigma_j \text{ for } 1 \le i = j + 1 \le n$$

where the vectors $\boldsymbol{h}_k$ and the matrices B_k are determined by the formulas

$$\boldsymbol{h}_k = \mathcal{F}_k(1:2,1), \quad B_{k+1} = \mathcal{F}_k(1:2,2:3), \quad 1 \le k \le n-2$$

and the vectors $\boldsymbol{g}_k$ and the numbers σ_k are computed recursively

$$\Gamma_1 = (0 \quad 1), \quad \boldsymbol{g}_1^T = (1 \quad 0)$$

$$\begin{bmatrix} \sigma_k & \boldsymbol{g}_{k+1}^T \\ * & \Gamma_{k+1} \end{bmatrix} = \mathcal{V}_{k+1} \begin{bmatrix} \Gamma_k & 0 \\ 0 & 1 \end{bmatrix} \mathcal{F}_k, \quad (k = 1, \cdots, n-2)$$

$$\sigma_{n-1} = \Gamma_{n-1} \boldsymbol{h}_{n-1}, \quad \boldsymbol{g}_n^T = \Gamma_{n-1} B_{n-1}$$

with the auxiliary variables $\Gamma_k \in \mathbb{C}^{1\times 2}$.

3.2 Single-shift Iteration

The procedure for the computation of the generating elements of $A^{(1)}$ such that

$$A - \alpha I = QR$$

$$A^{(1)} \leftarrow Q^* A Q$$

is outlined below.

Given the generating elements $\mathcal{V}_k$ $(k = 2, \cdots, n-1)$, $\mathcal{F}_k$ $(k = 1, \cdots, n-2)$, $\mathbf{z}$, $\mathbf{w}$ of A together with the shift parameter α, the algorithm calculates the generating elements $\mathcal{V}_k^{(1)}$ $(k = 2, \cdots, n-1)$, $\mathcal{F}_k^{(1)}$ $(k = 1, \cdots, n-2)$, $\mathbf{z}^{(1)}$, $\mathbf{w}^{(1)}$ of $A^{(1)}$. We'll just briefly outline the four steps of each iteration here. The long implementation detail of each step can be

(1) Using algorithm from Theorem 3.3 compute upper generators $\boldsymbol{g}_i$, $\boldsymbol{h}_i$ $(i = 1, \cdots, n)$, B_k $(k = 2, \cdots, n)$ and σ_k $(k = 1, \cdots, n-1)$.

(2) Set $\beta_n = z_n$ and for $k = n-1, \cdots, 3$ compute

$$\beta_k = \mathcal{V}_k^*(1, 1:2) \begin{bmatrix} z_k \\ \beta_{k+1} \end{bmatrix}$$

(3) Using upper generators, σ_k and shift α to compute the Givens rotation matrices $\mathcal{G}_k$ $(k = 1, \cdots, n-1)$ and the updated perturbation vectors $\mathbf{z}^{(1)}$,$\mathbf{w}^{(1)}$

(4) Using β_k and the Givens rotation matrices $\mathcal{G}_k$ $(k = 1, \cdots, n-1)$ to compute the generating elements $\mathcal{V}_k^{(1)}$ $(k = 2, \cdots, n-1)$, $\mathcal{F}_k^{(1)}$ $(k = 1, \cdots, n-2)$

In our project, we implement the single-shift version in Matlab based on the paper by Bini et al. But there are still remaining problems we'll state below.

3.3 Problems in Implementation

In this section, we'll talk about some remaining problems in our implementation of "fast" QR algorithm.

Deflation is an important concept in the practical implementation of the QR iteration. The subdiagonal and diagnal elements of matrix A can be represented by upper generators:

$$a_{k+1,k} = \sigma_k - z_{k+1}w_k$$

$$a_{k,k} = \boldsymbol{g}_k^T \boldsymbol{h}_k - z_k w_k$$

Therefore, we can deflate whenever a subdiagonal element satisfies $|a_{k+1,k}| \leq u(|a_{kk}| + |a_{k+1,k+1}|)$ as we usually do. But a problem is, when we are going to deflate, it seems we still need to reconstruct matrix U from all $\mathcal{V}_i$'s and $\mathcal{F}_i$'s, which breaks the general scheme of "fast" QR and also loses accuracy during the reconstruction of U. The paper does talk about deflation, but we are still confusing about this part.

The double-shift technique follows the scheme mentioned in the single-shift version. In this case, the matrices Q_k are of the type $Q_k = diag\{I_{k-1}, \mathcal{G}_k, I_{n-k-2}\}$, where this time $\mathcal{G}_k$ is a 3×3 Householder reflector and $1 \leq k \leq n-2$. However, we still have some problems during the implementation of the double-shift version.

Theoretically, the single-shift version of "fast" QR algorithm takes $243n$ FLOPs. The big constant factor, lots of extra memory manipulation and difficulty in implementation may keep it from practical use.

4 Iterative approximation

There exists another class of root-finding algorithms which directly generates the approximation of a root z for p_n at each iteration, such as Newton-Horner method and Muller's method.

4.1 Newton-Horner Method

The Newton Horner method combines two mathematical methods in order to find an approximation for the roots of a polynomial function, the Horner and Newton methods. In order to explain how the method works, it is necessary to first elaborate on the workings of each of those methods individually.

The Newton method also known as the Newton-Raphson method is a method for finding a better approximation of the roots of a function with each iteration. Suppose you have a guess x_0 , for one of the roots of a function $f(x)$, then to form a better approximation use the following formula:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The reason this works is because x_1 is the intersection at the x-axis with the tangent-line of the function at x_0. Typically this will be a better approximation of the function root then the initial guess.

$$x_{n+1} = x_n - \frac{f(x_0)}{f'(x_0)}$$

The formula is then repeated until a good enough approximation is found for the root. A good stopping criterion for the Newton method is to test how close the value is to zero when plugging x_n into the function.

The Horner method main objective is to find the solution to a polynomial function given input x, and significantly reduce the amount of computation required to evaluate the polynomial. It achieves this by rewriting the polynomial, suppose you have polynomial function $p(x)$, then according to Horner, the equation can be rewritten as $p(x) = q(x) \cdot (x - a) + constant$. Where the solution to $p(x)$ when $x = a$ is equal to the constant. This can be shown in the following example: Suppose we have the following polynomial, $p(x) = x^3 - 2x^2 - 5x + 6$. If we wanted to evaluate the polynomial when $x = 2$, we can use Horner's method to rewrite the polynomial as $p(x) = (x^2 - 5)(x - 2) - 4$ and according to it $p(2) = -4$. Horner method can be written in code as the following function:

```
1 function px = Horner(x)
2     px = c(1);
3     for i = 2:n
4          px = px * x + c(i);
5     end
6 end
```

Where c is the list of coefficient and n is the degree of the polynomial. This essentially evaluates the polynomial using the Horner form which is $q(x) \cdot (x - a) + constant$. This uses significantly less flops then the normal computation, it uses about $2n$ flops. Where a normal fully evaluated computation would use about $n^2/2 + n/2$ flops. Horner Method can also be modified to find the value of the derivative of the polynomial function as can be seen in the following function, where pprimex is the derivative value.

```
1 function [px, pprimex] = Horner(x)
2     pprimex = 0.0;
3     px = c(1);
4     for i = 2:n
5         pprimex = pprimex * x + px;
6         px = px * x + c(i);
7     end
8 end
```

Since Horner method can compute both the value of $f'(x)$ and $f(x)$ then it can be combined with the Newton method in order to form the Newton-Horner Method. The only

problem is that the Newton method is only used to compute a single root based on the original guess. We can achieve a search for all the roots by a process called deflation. Upon finding a root r_1 then it is possible to form a new function $p_2(x) = p(x)/(x - r_1)$, where it contains all the roots of $p(x)$ except r_1. Next we can find r_2 by running the Newton-Horner method on $p_2(x)$ then do it on $p_3(x)$ removing r_2 and so on. In the implementation of Newton-Horner that was used in our project, every time deflation is performed, the guess is set to the root that was just found, on the premise that the next root will be close to it. The process is repeated to the degree of the polynomial times and by doing so, every root of the original function can be found. This also accounts for a root multiplicity, where if a root has a multiplicity for example 6 it will be found 6 times.

We can also find complex roots by using an initial guess that is a complex number. In the Matlab code this is forced to happen by taking the initial guess and adding an imaginary number of the same size to it forming a complex number.

4.2 Muller's Method

The Muller Method is a root finding method that is based off the secant method for finding roots. The secant method works by using two initial guesses x_0 and x_1 for the roots. Using these two guesses the algorithm forms a line between $f(x_0)$ and $f(x_1)$. It then creates a new guess x_2 by using the point where that line intersects with the x-axis. It then repeats the process with x_1 and x_2 and so on and so on till a stopping criterion is reached.

Muller Method works the same way, but instead of using a linear interpolation, it uses quadratic interpolation. That is it start with three initial guesses x_0, x_1, and x_2 forms a parabola then picks one of the roots that intersects with the parabola using the quadratic method. Then repeats with x_1, x_2, and x_3 parabola, and so on. Just like the Newton-Horner method, deflation can be used to find all roots. Every time inflation occurs, the three initial guesses are reused.

5 Numerical Analysis

5.1 Convergence Rate of the Algorithms

For Newton's Method, let the error in the kth iteration $e_k = x_k - x^*$, where x_k is the computed solution in kth iteration and x^* is the exact solution to which the sequence of iterates $\{x_k\}$ converges. Using Taylor's Theorem, we obtain

$$\begin{aligned} e_{k+1} = x_{k+1} - x^* &= x_k - \frac{f(x_k)}{f'(x_k)} - x^* \\ &= e_k - \frac{1}{f'(x_k)}[f(x^*) - f'(x_k)(x^* - x_k) - \frac{1}{2}f''(\xi_k)(x_k - x^*)^2] \\ &= e_k + \frac{1}{f'(x_k)}[-f'(x_k)e_k + \frac{1}{2}f''(\xi_k)e_k^2] = \frac{f''(\xi_k)}{2f'(x_k)}e_k^2 \end{aligned}$$

where ξ_k is between x_k and x. If $f'(x^*) \neq 0$, then Newton's Method converges quadratically, with asymptotic error constant $|f''(x^*)/2f'(x^*)|$. If $f'(x^*)$ is very small, or zero, then convergence can be very slow or may not even occur. [6]

Let's do an experiment for $f(x) = x^2 - 3$. We examine the error $e_k = x_k - x^*$, where $x^* = \sqrt{3}$ is the exact solution. We have

k	x_k	$\lvert e_k \rvert$
0	1.0	0.73205080756888
1	2.0	0.26794919243112
2	1.75	0.01794919243112
3	1.73214285714286	0.00009204957398
4	1.73205081001473	0.00000000244585

In this example, the asymptotic error constant is $|f''(\sqrt{3})/2f'(\sqrt{3})| \approx 0.2886751$. Examining the numbers in the table above, we can see that the number of correct decimal places approximately doubles with each iteration, which is typical of quadratic convergence. Furthermore, we have $|e_4|/|e_3|^2 \approx 0.2886598$, so the actual behavior of the error is consistent with the behavior that is predicted by theory.

For the double-shift Franciss algorithm, the shifts ρ_1 and ρ_2 that we choose are eigenvalues of the 2×2 submatrix in the lower right corner of A. This strategy achieves cubic convergence in the generic case and at least quadratic convergence in the worst case. [7]

5.2 Conditioning of Polynomial Root-finding

The problem of polynomial root-finding can be in general ill-conditioned. For example, consider solving the quadratic equation $f(x) = x^2 - 2x + 1 = 0$, whose roots are $x_1 = 1$, $x_2 = 1$. Now perturb the coefficient 2 by 10^{-5}. The computed roots of the perturbed polynomial $\hat{f}(x) = x^2 - 2.00001x + 1$ are $x_1 = 1.0032$ and $x_2 = 0.9968$. The relative error in the data is 5×10^{-6}, while the relative errors in x_1 and x_2 are 0.0032. We can see a small perturbation in the data changed the roots substantially.

Another well-known example about the ill-conditioned polynomial root-finding is Wilkinson's polynomial $w(x) = \prod_{i=1}^{20}(x-i) = x^{20} - 210x^{19} + 20615x^{18} - \cdots$. The roots of $w(x)$ are $1, 2, \cdots, 20$. Now perturb coefficient of x^{19} from -210 to $-210 - 2^{-23}$. The change amount is approximately 1.12×10^{-7}. Several roots of the perturbed polynomial become very different from the original roots. For example, the root $x_{16} = 16$ changes to $16.73 + 2.81i$. [8]

Theorem 5.1 [8]: Let a_i be the ith coefficient of a polynomial $f(x)$ and let δa_i and δx_j denote small perturbation of a_i and the jth root x_j. Then the condition number of root x_j with respect to the perturbation of the single coefficient a_i is

$$\frac{|\delta x_j|}{|x_j|} \Big/ \frac{|\delta a_i|}{|a_i|} = \frac{|a_i x_j^{i-1}|}{|f'(x_j)|}$$

It can be verified that the condition number of $x_{16} = 16$ is $O(10^{10})$!

This teaches us a very important lesson: it is not a good idea to compute the eigenvalues of a matrix by finding coefficients of the characteristic polynomial, and then solving its roots with some other numerical methods.

6 Reference

[1] J. G. F. Francis, *The QR transformation. II*, Comput. J. 4 (1961/1962), 332-345.

[2] David S. Watkins, *Francis's Algorithm*, The American Mathematical Monthly 118(5), May (2011)

[3] D. A. Bini, Y. Eidelman, L. Gemignani and I. Gohberg, *Fast QR Eigenvalue Algorithms for Hessenberg Matrices Which Are Rank-One Perturbations of Unitary Matrices*, SIAM. J. Matrix Anal. & Appl., 29(2), (2007), 566-585.

[4] D. A. Bini, P. Boito, Y. Eidelman, L. Gemignani and I. Gohberg, *A Fast Implicit QR Eigenvalue Algorithms for Companion Matrices*, Linear Algebra Appl., April (2010)

[5] D. Bindel, S. Chandresekaran, J. Demmel, D. Garmire, and M. Gu, *A Fast and Stable Nonsymmetric Eigensolver for Certain Structured Matrices*, Technical report, University of California, Berkeley, CA, (2005)

[6] A. Quarteroni, R. Sacco, F. Saleri *Numerical Mathematics*, Second Edition, TAM37, Springer, Berlin (2007)

[7] Lloyd N. Trefethen, David Bau III, *Numerical Linear Algebra*, SIAM: Society for Industrial and Applied Mathematics (1997)

[8] Biswa Nath Datta, Numerical Linear Algebra and Applications, Second Edition, SIAM: Society for Industrial and Applied Mathematics (2010)