A New and Efficient Program for Finding all Polynomial Roots

Markus Lang[*], Electrical and Computer Engineering Department

Rice University, Houston, TX 77251, lang@dsp.rice.edu

Bernhard-Christian Frenzel, Sonnebergerstr. 11

90765 Fürth, Germany

Technical Report # 9308

April 15, 1994

Abstract

Finding polynomial roots rapidly and accurately is an important problem in many areas of signal processing. We present a new program which is a combination of Muller's and Newton's method. We use the former for computing a root of the deflated polynomial which is a good estimate for the root of the original polynomial. This estimate is improved by applying Newton's method to the original polynomial. Test polynomials up to the degree 10000 show the superiority of our program over the best methods to our knowledge regarding speed and accuracy, i.e., Jenkins/Traub program and the eigenvalue method.

Furthermore we give a simple approach to improve the accuracy for spectral factorization in the case there are double roots on the unit circle. Finally we briefly consider the inverse problem of root finding, i.e., computing the polynomial coefficients from the roots which may lead to surprisingly large numerical errors.

[*]This research was partially supported by Alexander von Humboldt-Stiftung and by AFOSR under grant F49620-1-0006 funded by DARPA

Contents

1 Introduction

Finding polynomial roots rapidly *and* accurately is an important problem in various areas of signal processing such as spectral factorization, phase unwrapping, forming a cascade of second order systems etc. [1, 3, 4, 16, 18]. There exists a large number of different methods for finding all polynomial roots either iteratively or simultaneously. Most of them yield accurate results only for small degrees or can treat only special polynomials, e.g., polynomial with real roots.

One of the two best general purpose root finder is the Jenkins/Traub method [7]. It works with the polynomial itself. The required memory is proportional to $O(n)$. The maximum degree yielding reasonable accuracy is 60–80 (see Sec. 3).

The second method is called eigenvalue method and works with the so-called companion matrix formed with the polynomial coefficients. The polynomial roots are the eigenvalues of this companion matrix which can be found with high accuracy by use of the EISPACK routines [17]. These are the best known programs for solving general eigenvalue problems. The required memory and computation time are proportional to $O(n^2)$ and $O(n^3)$, respectively. As Toh and Trefethen point out in [19] one can only hope to get no worse conditioned problem than the underlying rootfinding problem by using the eigenvalue approach. This means one should solve the polynomial zerofinding problem and not the eigenvalue problem if the interesting parameters are the roots of a given polynomial.

We present a method for finding all polynomial roots of an arbitrary complex valued polynomial. It turns out to be faster and at least as accurate as the best known methods for nearly all polynomials we have used for testing. It basically consists of a combination of two well-known iterative methods, i.e., Muller's and Newton's method [14]. The first is numerically robust and yields an estimate for the root working with the actual deflated polynomial. In a second step this root is refined using the quadratically converging Newton's method for the original polynomial. We do not assume any special structure of the polynomial and take no extra precautions for multiple roots. The latter should be done by the user who can exploit additional knowledge about the roots. As an example we give a straightforward approach for spectral factorization where the problem of double roots on the unit circle may occur

The paper is organized as follows. In Sec. 2 we describe Muller's and Newton's method and some steps to stabilize the implemented program. The efficiency and reliability of all three methods are compared in Sec. 3 where we use several test polynomials with known roots. We discuss how to deal with double roots on the unit circle in the case of spectral factorization in Sec. 4. There we additionally consider the problem of multiplying the roots to get the original polynomial. This simple looking procedure may lead to completely perturbed polynomial coefficients even in well conditioned cases. The main results are summarized in Sec. 5

2 Description of the Procedure

We consider a complex valued polynomial

$$P(x) = \sum_{\nu=0}^{n} p_\nu x^\nu = p_n \prod_{\nu=1}^{n} (x - x_\nu), \quad p_n \neq 0 \tag{1}$$

of degree n. The problem we address is to find all n roots x_ν of $P(x)$ as accurately and fast as possible. We work with a given and fixed computation accuracy, i.e., IEEE-P754-floating point standard (accuracy $\approx 2.2 \cdot 10^{-16}$). The relative accuracy of the resulting roots has to be compared to this number. For the special case of real coefficients p_ν, complex roots x_ν must appear as complex conjugate pairs.

We have chosen a combination of Muller's and Newton's method since both of them can be used to find complex roots. In the following we give a pseudo code of our program:

1. Check polynomial and return if erroneous input.
2. If polynomial degree is 1 or 2 $\rightarrow$ compute roots; return.
3. Call `monic()`.
4. While degree of deflated polynomial > 2.
   - Call `muller()`.
   - Call `newton()`.
   - Call `poldefl()`.
5. Compute root(s) of deflated polynomial.

6. Call `newton()`.

In a first step the coefficients are formally checked, e.g., possible roots at zero are determined and deflated, leading zeros are cancelled etc. In the case of a first or second order polynomial the well-known explicit formulae are used. Only for degree $n > 2$ we choose our iterative procedure. At first the routine `monic()` yields a polynomial with $p_n = 1$. Then the routine `muller()` computes an estimate for a root of the actual, deflated polynomial $P_{defl}(x)$ which contains all roots of $P(x)$ but the roots found up to the actual iteration step (k).

In a second step the estimate resulting from `muller()` is used as the initial value of Newton's method. It is simple and known to have at least quadratical convergence near the solution (for single roots). We use the original polynomial to avoid errors introduced by the deflation process. Once Newton's method has converged the resulting root is deflated (and possibly its complex conjugated for real valued polynomials). This deflation procedure is repeated until the resulting polynomial is of degree two or one. An estimate for its root(s) can be obtained by using the well-known explicit formula and is refined by Newton's method.

2.1 Muller's method

We have chosen Muller's method for computing an initial estimate of the root because it has the following two properties: One is a good convergence to a reasonable estimate of a root. The second is the possibility to get complex roots even when initialized with real values in opposite to other methods, e.g., Newton's method. The convergence speed is super linear (1.84 for single roots). General convergence for this method has not been proven.

Muller's method extends the idea of the secant method which works with a linear polynomial to a quadratical polynomial. Given three previous estimates $x^{(k-2)}$, $x^{(k-1)}$, and $x^{(k)}$ for an unknown root we compute a new value by determining one of the roots of a parabola $\tilde{P}(x)$ which interpolates $P(x)$ in these three "old" points. This is illustrated by Fig. 1. The corresponding iteration formulae are [14]

$$h_k = x^{(k)} - x^{(k-1)} \tag{2}$$

$$r_k = h_k/h_{k-1} \tag{3}$$

$$A_k = r_k P(x^{(k)}) - r_k(1 + r_k)P(x^{(k-1)}) + r_k^2 P(x^{(k-2)}) \tag{4}$$

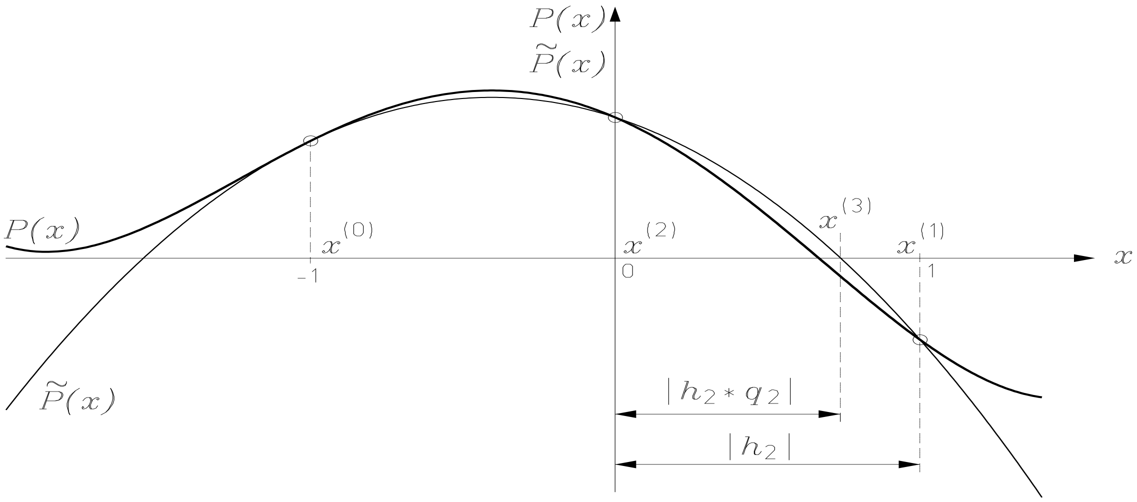

Figure 1: Iteration step of Muller's method.

$$
\begin{aligned}
B_k &= (2r_k+1)P(x^{(k)}) - (1+r_k)^2 P(x^{(k-1)}) + r_k^2 P(x^{(k-2)}) && (5)\\
C_k &= (1+r_k)P(x^{(k)}) && (6)\\
q_k &= \frac{-2C_k}{B_k \pm \sqrt{B_k^2 - 4A_kC_k}} && (7)\\
x^{(k+1)} &= x^{(k)} + h_k q_k. && (8)
\end{aligned}
$$

The new estimate $x^{(k+1)}$ is determined such that it is the one root of $\tilde{P}(x)$ closer to $x^{(k)}$. An example is given in Fig. 1 for $k = 2$. In the case the denominator vanishes q_k is chosen as $|q_k| = 1$ with arbitrary phase. The algorithm is initialized with $x^{(0)} = 1$, $x^{(1)} = -1$, and $x^{(2)} = 0$.

For the practical convergence of this method we have to include additional measures which are summarized in the following pseudo code of our program implementing Muller's method:

1. Call `initialize()`.
2. Repeat twice.
   (a) While iteration counter < ITERMAX **and** noise counter < NOISEMAX **and** root not found.
      - Call `root_of_parabola()`.
      - Call `iteration_equation()`.

- Call `compute_function()`.
- Call `check_x_value()`.

(b) Call `root_check()`.

(c) If root good enough return.

After initializing, the main part is repeated twice with different starting values if the first result is not good enough. The main loop stops whenever one of the following criteria is fulfilled: First we give a maximum number of iterations considering the case of very slow convergence. Second we stop when the iteration is dominated by noise. This means that we get only minor improvements in the range of the computer accuracy during a fixed number of successive iterations. Of course the program stops when

$$\left| \frac{x^{(k+1)} - x^{(k)}}{x^{(k+1)}} \right| < \epsilon \tag{9}$$

holds (`root_check()`), where ϵ is some small number depending on the computer accuracy.

The first step of the main loop is computing the roots of the parabola Eqs. (2)–(7). This is followed by the iteration equation (8). We have observed that values q_k computed according to Eq. (7) may yield too large changes of $x^{(k)}$ which possibly leads to another root and causes slow convergence. This can be circumvented (and is actually implemented in our program) by allowing a fixed maximum relative increase of $|q_k|$ from one iteration step to the next.

Before the new function value is evaluated we estimate $|P(x^{(k+1)})|$ to avoid overflow. This is done by checking $n \cdot \log_{10} |x^{(k+1)}|$. If an estimate indicates a value greater than the maximum possible computer number we choose

$$x^{(k+1)} = x^{(k)} + h_k q_k / 2 \tag{10}$$

instead of Eq. (8) and repeat this until no overflow occurs. This means we go back closer and closer to the old value $x^{(k)}$.

In a last step the actual value $|P(x^{(k+1)})|$ is compared to the best value until the current iteration step and it substitutes the latter if it is smaller.

The stopping criteria ϵ, ITERMAX, ... were determined on an experimental basis to get a program which is reliably and fast.

2.2 Newton's method

Newton's method is well-known and works with the following simple iteration formula

$$x^{(k+1)} = x^{(k)} - \Delta x^{(k)}, \quad \text{where } \Delta x^{(k)} = \frac{P(x^{(k)})}{P'(x^{(k)})}, \tag{11}$$

which is illustrated by Fig. 2.

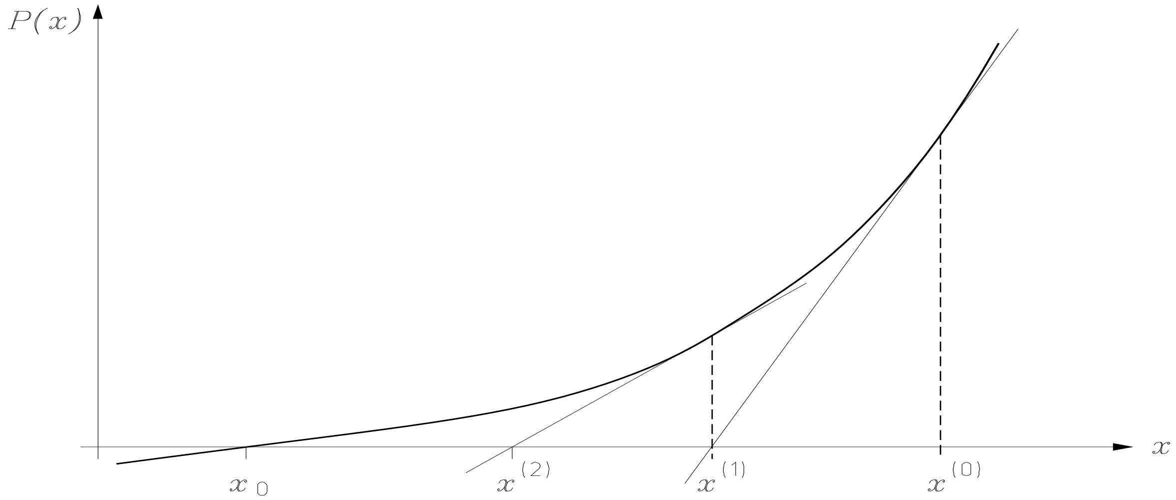

Figure 2: Iteration step of Newton's method.

It yields the root of the tangent through the point $(x^{(k)}, P(x^{(k)}))$ of the previous iteration step and is initialized with the result of Muller's method. Similar to this, additional measures must be included to improve the performance of the program. This can be seen by the following pseudo code of our implementation:

1. While iteration counter < ITERMAX.

   - Call `fdvalue()`.
   - If $|P(x^{(k+1)})| < |P(x_{min})| \rightarrow x_{min} = x^{(k+1)}$.
   - If $|P'(x^{(k+1)})| \neq 0$ **and** $|P(x^{(k+1)})|/|P'(x^{(k+1)})| < |\Delta x^{(k-1)}|$
     $\rightarrow \Delta x^{(k)} = P(x^{(k+1)})/P'(x^{(k+1)})$;
     else $\Delta x^{(k)} = \Delta x^{(k-1)}$.

- If $|\Delta x^{(k)}|/|x^{(k)}| < \epsilon$ **or** noise counter > NOISEMAX.
  - If in the case of a real valued polynomial $\mathrm{Im}(x) < \delta \rightarrow \mathrm{Im}(x_{min}) = 0$; return.
- $x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$.

2. If in the case of a real valued polynomial $\mathrm{Im}(x) < \delta \rightarrow \mathrm{Im}(x_{min}) = 0$.

As in Muller's method we give a maximum number of iterations. The first step of the main loop is computing the function value and its derivative at the actual point $x^{(k+1)}$. This substitutes the minimum value up to the actual iteration step x_{min} if it yields a smaller function value. We do not permit too large changes by using the new improvement $\Delta x^{(k)}$ according to Eq. (11) only if it is smaller than the old one of the previous iteration step. Otherwise the latter one is used to avoid that the algorithm switches to another root.

The algorithm is stopped when it is dominated by noise (see Muller's method) or when

$$\tilde{e} = \left| \frac{\Delta x^{(k)}}{x_{min}} \right| < \epsilon, \tag{12}$$

where x_{min} is that $x^{(k)}$ leading to the minimum $|P(x)|$ up to the actual iteration step and ϵ again depends on the computer accuracy. In the case of a real valued polynomial (i.e., all p_ν are real) for every complex root also its complex conjugate is a root which can be deflated together. Consequently we have to decide whether a root with a very small imaginary part is to be seen as a real or a complex root. We assume to have a real root if the imaginary part is less than δ which we have chosen as half of the computer accuracy. To avoid this decision which may lead to errors one can simply multiply the real valued polynomial by the imaginary unit. In this case the program interpretes the polynomial as a complex valued polynomial and it consequently deflates only one root at each iteration step.

It is interesting that the quotient $\tilde{e}$ introduced in (12) can be used as an estimate of the relative accuracy of the actual root. Our program yields the corresponding maximum value of all computed roots which is discussed in the following section.

3 Test of the New Algorithm

3.1 General Considerations

After the short description of our implementation we have to prove the efficiency. We point out again that we do not try to construct a new algorithm with nice theoretical properties but a tool which works reliable and fast in many practical applications. Our approach to verify the performance is oriented at the ideas of Jenkins and Traub [10]. They propose to chose a lot of test polynomials with known roots testing programs for different weaknesses. A root finder is the better the smaller the difference between the correct roots and the determined ones. We use a normalized version of this criterion

$$e = \max_{\nu} \left| \frac{x_\nu - x_{\nu min}}{x_\nu} \right|, \tag{13}$$

where $x_{\nu min}$ is the value computed by the root finder and x_ν is the corresponding exact value. This number is computed for every polynomial. Additionally we determined the necessary CPU time on an HP Apollo workstation 9000/705. We compare the results of our program regarding to speed and accuracy with two of the best root finder programs to our knowledge. These are the Jenkins/Traub program [7] and the eigenvalue method based on EISPACK [17] in the version of MATLAB.

We do not show results of factoring actual transfer functions since we do not know the correct roots and cannot give an objective measure for the accuracy. However, we successfully computed the roots of many FIR filters. To give an example our program determined all roots of a degree 1000 FIR low-pass filter within 8.35s with an estimated error $\tilde{e} = 1.8 \cdot 10^{-16}$. The computed roots in the stopband which should have magnitude one have a maximum distance from the unit circle of $1.11 \cdot 10^{-16}$. That means they are exact within computer accuracy.

3.2 Test polynomials

The following polynomials used for the detection of different properties of a root finder were proposed by Jenkins and Traub [10]. We use several polynomials for each property but we leave out polynomials with random coefficients for the same reasons we did not examine the factorization of transfer functions. We conjecture that in this case one gets similar results to those of the last polynomials we present in this paper (equidistantly distributed roots on the unit circle) since polynomials with random coefficients tend to have a similar root distribution [2, 18].

3.2.1 Check of the Stopping Criterion

The first polynomial

$$P_1(x) = B(x - A)(x + A)(x - 1) \tag{14}$$

shows the effect of very large or small roots (A) and very large or small polynomial coefficients (B), respectively on the stopping criterion. Table 1 shows the result for all three methods, where e_E, e_J, e_N means the error of the eigenvalue method, the Jenkins/Traub method, and our new method according to Eq. (13). $\tilde{e}$ means the corresponding estimate computed by our program and

A	B	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
$1 \cdot 10^{-10}$	$1 \cdot 10^{-10}$	0	$3.009 \cdot 10^{-26}$	0	0	0.01	0.05	0.02
$1 \cdot 10^{-10}$	$1 \cdot 10^{10}$	0	$3.009 \cdot 10^{-26}$	0	0	0.01	0.02	0.01
$1 \cdot 10^{10}$	$1 \cdot 10^{-10}$	0	$1.177 \cdot 10^{-5}$	0	$3.584 \cdot 10^{-17}$	0.02	0.02	0.01
$1 \cdot 10^{10}$	$1 \cdot 10^{10}$	0	$-1.177 \cdot 10^{-5}$	0	$3.584 \cdot 10^{-17}$	0.01	0.03	0.01

Table 1: $P_1(x) = B(x - A)(x + A)(x - 1)$.

t_E, t_J, t_N, are the necessary CPU times in seconds. As can be seen our program computes all roots exactly just as the eigenvalue method but in opposite to the Jenkins/Traub method which yields relatively large errors for large values of A.

Furthermore we use the polynomial

$$P_2(x) = \prod_{\nu=0}^{n} (x - 10^{-\nu}) \tag{15}$$

with more and more zeros close to 0 for larger values n. The results for $n = 5, 7$ are summarized in Table 2. Again our program yields the most accurate results but this time together with the Jenkins/Traub method. The large value t_N can be explained by the fact that in the case $n = 7$ the main loop of Muller's method has to be computed twice.

From the test polynomials above it can be concluded that our method has the fewest problems concerning the stopping criterion.

n	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
5	$6.939 \cdot 10^{-16}$	$1.735 \cdot 10^{-16}$	$1.735 \cdot 10^{-16}$	$1.307 \cdot 10^{-16}$	0.01	0.01	0.03
7	$1.388 \cdot 10^{-15}$	$1.735 \cdot 10^{-16}$	$1.735 \cdot 10^{-16}$	$1.576 \cdot 10^{-16}$	0.02	0.02	0.10

Table 2: $P_2(x) = \prod_{\nu=0}^{n} (x - 10^\nu)$.

3.2.2 Check of Convergence

For the check of problems concerning the convergence we choose the polynomial

$$P_3(x) = \prod_{\nu=1}^{n} (x - \nu) \tag{16}$$

which has a surprisingly ill conditioned numerical behaviour (cf. Wilkinson [20]). The results are depicted in Table 3. We make the following observations: The accuracy of all three methods is comparable. For degrees greater than 8 our method always yields the best results. The estimate $\tilde{e}$ is close to the actual error. For larger degrees our method takes the most CPU time since the main loop in Muller's method is computed twice.

3.2.3 Multiple or Clustered Roots

Multiple roots or roots close to each other lead to numerically ill conditioned polynomials. Nearly every root finder has difficulties to compute these with high accuracy. As Wilkinson points out in [20] the limiting accuracy of a root with multiplicity m is $\varepsilon^{1/m}$ where ε means the computer accuracy. This means in our case (accuracy $\approx 2.2 \cdot 10^{-16}$) that a double root can be determined up to an accuracy of about 10^{-8} as long as no special methods are used in this case (cf. Sec. 4.1). Furthermore this means without additional knowledge it cannot be decided whether two roots differing by about 10^{-8} are two separated roots or incorrectly determined double roots. However, as Wilkinson points out in [20] (and is confirmed by our experience) this must not affect the accuracy of well conditioned roots even for a polynomial with several multiple roots (see below).

We have used the following polynomials.

$$P_4(x) = (x - 0.1)^3(x - 0.5)(x - 0.6)(x - 0.7) \tag{17}$$

$$P_5(x) = (x - 0.1)^4(x - 0.2)^3(x - 0.3)^2(x - 0.4) \tag{18}$$

$$P_6(x) = (x - 0.1)(x - 1.001)(x - 0.998)(x - 0.99999) \tag{19}$$

n	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
1	0.000	0.000	0.000	$2.220 \cdot 10^{-16}$	0.01	0.01	0.00
2	0.000	$2.220 \cdot 10^{-16}$	0.000	$2.220 \cdot 10^{-16}$	0.02	0.01	0.00
3	$5.921 \cdot 10^{-16}$	$6.518 \cdot 10^{-20}$	$2.220 \cdot 10^{-16}$	$4.534 \cdot 10^{-25}$	0.02	0.02	0.00
4	$1.021 \cdot 10^{-14}$	$1.136 \cdot 10^{-19}$	$1.776 \cdot 10^{-15}$	$1.776 \cdot 10^{-15}$	0.02	0.02	0.01
5	$5.003 \cdot 10^{-14}$	$1.510 \cdot 10^{-14}$	$1.554 \cdot 10^{-15}$	$1.776 \cdot 10^{-14}$	0.01	0.02	0.02
6	$2.236 \cdot 10^{-13}$	$4.269 \cdot 10^{-14}$	$5.695 \cdot 10^{-14}$	$1.089 \cdot 10^{-13}$	0.02	0.02	0.01
7	$8.777 \cdot 10^{-13}$	$1.305 \cdot 10^{-13}$	$6.370 \cdot 10^{-13}$	$2.425 \cdot 10^{-13}$	0.01	0.02	0.02
8	$1.180 \cdot 10^{-11}$	$5.340 \cdot 10^{-13}$	$1.217 \cdot 10^{-12}$	$1.929 \cdot 10^{-12}$	0.02	0.02	0.03
9	$1.062 \cdot 10^{-10}$	$9.705 \cdot 10^{-12}$	$5.760 \cdot 10^{-12}$	$7.370 \cdot 10^{-12}$	0.03	0.03	0.04
10	$4.365 \cdot 10^{-11}$	$8.977 \cdot 10^{-11}$	$1.604 \cdot 10^{-11}$	$4.312 \cdot 10^{-11}$	0.02	0.02	0.05
11	$5.448 \cdot 10^{-10}$	$3.513 \cdot 10^{-10}$	$1.363 \cdot 10^{-10}$	$1.772 \cdot 10^{-10}$	0.02	0.02	0.07
12	$6.751 \cdot 10^{-9}$	$8.332 \cdot 10^{-10}$	$1.976 \cdot 10^{-10}$	$2.205 \cdot 10^{-9}$	0.02	0.02	0.08
13	$1.121 \cdot 10^{-7}$	$1.397 \cdot 10^{-8}$	$4.252 \cdot 10^{-9}$	$4.507 \cdot 10^{-9}$	0.03	0.03	0.09
14	$4.421 \cdot 10^{-8}$	$4.147 \cdot 10^{-8}$	$1.372 \cdot 10^{-8}$	$5.249 \cdot 10^{-8}$	0.03	0.02	0.12
15	$3.848 \cdot 10^{-7}$	$2.725 \cdot 10^{-7}$	$9.540 \cdot 10^{-8}$	$5.714 \cdot 10^{-8}$	0.03	0.03	0.13

Table 3: $P_3(x) = \prod_{\nu=1}^{n} (x - \nu)$.

$$P_7(x) = (x - 0.001)(x - 0.01)(x - 0.1)(x - 0.1 + A \cdot \mathrm{j})(x - 0.1 - A \cdot \mathrm{j})(x - 1)(x - 10) \quad (20)$$

The results are depicted in Table 4 and 5.

Again the accuracy of all three methods is comparable where the Jenkins/Traub method has advantages for $P_7(x)$. The eigenvalue method always yields the worst results. The resulting accuracy lies in the expected range of the limiting accuracy.

We want to remark that although Steiglitz and Dickinson conjecture the polynomial $P(x) = (x^{100} - 1)^3$ "$\cdots$ should stop any root solver in its tracks." our program is able to find all roots of $P(x) = (x^{100} - 1)^3(x + 0.5)(x - 0.5)(x - 2)(x - 3)$ with a maximum error of $4 \cdot 10^{-7}$ (for the multiple roots). All single roots could be computed up to computer accuracy which confirms the proposition above about the effect of multiple roots.

ν	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
4	$2.124 \cdot 10^{-5}$	$3.435 \cdot 10^{-6}$	$8.912 \cdot 10^{-6}$	$3.248 \cdot 10^{-6}$	0.01	0.04	0.04
5	$7.076 \cdot 10^{-4}$	$1.441 \cdot 10^{-5}$	$5.626 \cdot 10^{-4}$	$1.492 \cdot 10^{-4}$	0.03	0.03	0.09
6	$1.529 \cdot 10^{-5}$	$2.918 \cdot 10^{-4}$	$1.002 \cdot 10^{-5}$	$4.116 \cdot 10^{-6}$	0.01	0.03	0.04

Table 4: Polynomials $P_\nu(x)$, $\nu = 4, 5, 6$.

A	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
0	$1.627 \cdot 10^{-5}$	$4.430 \cdot 10^{-8}$	$6.978 \cdot 10^{-6}$	$2.406 \cdot 10^{-6}$	0.01	0.03	0.04
$1 \cdot 10^{-10}$	$1.627 \cdot 10^{-5}$	$4.384 \cdot 10^{-8}$	$6.978 \cdot 10^{-6}$	$2.406 \cdot 10^{-6}$	0.02	0.01	0.04
$1 \cdot 10^{-9}$	$1.364 \cdot 10^{-5}$	$4.063 \cdot 10^{-8}$	$8.620 \cdot 10^{-6}$	$1.841 \cdot 10^{-6}$	0.02	0.02	0.04
$1 \cdot 10^{-8}$	$1.610 \cdot 10^{-5}$	$1.013 \cdot 10^{-7}$	$4.988 \cdot 10^{-6}$	$1.145 \cdot 10^{-6}$	0.02	0.02	0.04
$1 \cdot 10^{-7}$	$9.612 \cdot 10^{-6}$	$9.902 \cdot 10^{-7}$	$4.863 \cdot 10^{-6}$	$3.632 \cdot 10^{-6}$	0.02	0.04	0.03
$1 \cdot 10^{-6}$	$2.407 \cdot 10^{-5}$	$9.989 \cdot 10^{-6}$	$2.918 \cdot 10^{-6}$	$1.289 \cdot 10^{-6}$	0.03	0.02	0.04

Table 5: Polynomial $P_7(x)$.

3.2.4 Stability of Deflation

Stability of the deflation process means that the roots of the deflated polynomial are close to those of the original. Since we refine all roots using the original polynomial our method is expected to yield good results. We choose the polynomials

$$P_8(x) = (x - A)(x - 1)(x - \frac{1}{A}) \tag{21}$$

$$P_9(x) = \prod_{\nu=1-M}^{M-1} (x - \mathrm{e}^{\mathrm{j}\frac{\nu\pi}{2M}}) \cdot \prod_{\nu=M}^{3M} (x - 0.9 \cdot \mathrm{e}^{\mathrm{j}\frac{\nu\pi}{2M}}), \tag{22}$$

where the latter has a distribution of roots similar to the transfer function of an FIR lowpass filter. Table 6 and 7 show the results and Fig. 3 depicts the accuracy and the CPU time for $P_9(x)$ depending on the degree n.

A	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
$1 \cdot 10^3$	$2.274 \cdot 10^{-16}$	$2.483 \cdot 10^{-16}$	0.000	$1.139 \cdot 10^{-16}$	0.01	0.02	0.00
$1 \cdot 10^6$	$2.328 \cdot 10^{-16}$	$2.220 \cdot 10^{-16}$	$2.118 \cdot 10^{-16}$	$1.164 \cdot 10^{-16}$	0.02	0.01	0.00
$1 \cdot 10^9$	$4.441 \cdot 10^{-16}$	$2.068 \cdot 10^{-16}$	$2.068 \cdot 10^{-16}$	$1.192 \cdot 10^{-16}$	0.01	0.02	0.00

Table 6: $P_8(x) = (x - A)(x - 1)(x - \frac{1}{A})$.

M	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
5	$2.357 \cdot 10^{-15}$	$1.959 \cdot 10^{-15}$	$2.758 \cdot 10^{-16}$	$1.820 \cdot 10^{-16}$	0.16	0.03	0.05
10	$3.903 \cdot 10^{-15}$	$2.227 \cdot 10^{-12}$	$3.084 \cdot 10^{-16}$	$1.825 \cdot 10^{-16}$	0.92	0.05	0.15
12	$4.242 \cdot 10^{-15}$	$1.771 \cdot 10^{-1}$	$4.934 \cdot 10^{-16}$	$1.364 \cdot 10^{-16}$	1.43	0.07	0.19
15	$6.690 \cdot 10^{-15}$	$1.552 \cdot 10^{-1}$	$8.723 \cdot 10^{-16}$	$9.812 \cdot 10^{-17}$	2.63	0.11	0.27
20	$2.534 \cdot 10^{-14}$	$2.396 \cdot 10^{-1}$	$1.061 \cdot 10^{-15}$	$1.825 \cdot 10^{-16}$	6.27	0.18	0.39
25	$1.300 \cdot 10^{-13}$	$1.613 \cdot 10^{-1}$	$4.939 \cdot 10^{-15}$	$1.875 \cdot 10^{-16}$	12.18	0.30	0.51
50	$1.803 \cdot 10^{-9}$	$3.510 \cdot 10^{2}$	$2.481 \cdot 10^{-13}$	$1.121 \cdot 10^{-14}$	95.34	26.11	2.31

Table 7: $P_9(x) = \prod_{\nu=1-M}^{M-1} (x - \mathrm{e}^{\mathrm{j}\frac{\nu\pi}{2M}}) \cdot \prod_{\nu=M}^{3M} (x - 0.9 \cdot \mathrm{e}^{\mathrm{j}\frac{\nu\pi}{2M}})$

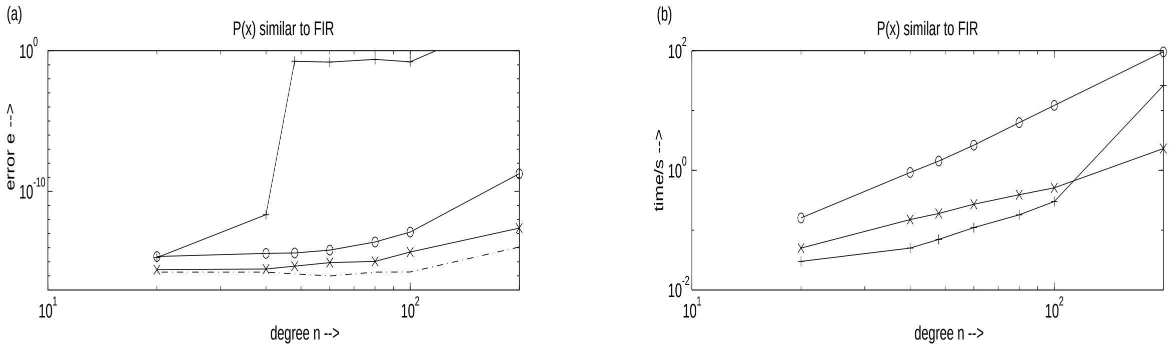

Figure 3: Results of Jenkins/Traub (+), eigenvalue (∘), and our method (×) for $P_9(x)$. (a) Actual and estimated accuracy e and $\tilde{e}$ ($- \cdot -$); (b) CPU time.

Our method always yields the most accurate results. It is better than a factor of 1000 for $P_9(x)$, $n = 200$ ($M = 200$) and furthermore much faster for increasing degree up to a factor of 45 compared to the eigenvalue method. It is remarkable that the Jenkins/Traub program yields much less accurate results even for $P_9(x)$, $n = 40$ ($M = 10$) and completely useless values for larger degrees. As we will see this is typical for the program.

3.2.5 High Order, Well Conditioned Polynomials

As a last example we consider high degree polynomials up to $n = 10000$ for the very well conditioned polynomials

$$P_{10}(x) = x^n - 1 \tag{23}$$

$$P_{11}(x) = x^n + 1. \tag{24}$$

The results are summarized in Table 8 and 9 and graphically depicted in Figs. 4 and 5.

n	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
10	$8.083 \cdot 10^{-16}$	$2.428 \cdot 10^{-15}$	$2.289 \cdot 10^{-16}$	$1.579 \cdot 10^{-16}$	0.04	0.02	0.02
20	$1.226 \cdot 10^{-15}$	$5.431 \cdot 10^{-15}$	$5.979 \cdot 10^{-16}$	$1.247 \cdot 10^{-16}$	0.08	0.06	0.04
50	$2.109 \cdot 10^{-15}$	$6.785 \cdot 10^{-12}$	$1.144 \cdot 10^{-15}$	$2.220 \cdot 10^{-16}$	0.34	0.11	0.11
70	$2.047 \cdot 10^{-15}$	$3.049 \cdot 10^{-1}$	$6.280 \cdot 10^{-16}$	$5.155 \cdot 10^{-17}$	0.72	0.24	0.17
100	$2.559 \cdot 10^{-15}$	$9.769 \cdot 10^{-1}$	$1.024 \cdot 10^{-15}$	$1.868 \cdot 10^{-16}$	1.77	0.43	0.26
200	$3.443 \cdot 10^{-15}$	–	$1.180 \cdot 10^{-15}$	$2.220 \cdot 10^{-16}$	11.88	–	0.67
500	$3.360 \cdot 10^{-15}$	–	$8.968 \cdot 10^{-16}$	$2.069 \cdot 10^{-16}$	315.22	–	2.59
1000	–	–	$1.024 \cdot 10^{-15}$	$2.202 \cdot 10^{-16}$	–	–	8.81
2000	–	–	$1.106 \cdot 10^{-15}$	$2.059 \cdot 10^{-16}$	–	–	30.96
10000	–	–	$1.047 \cdot 10^{-15}$	$2.196 \cdot 10^{-16}$	–	–	944.09

Table 8: $P_{10}(x) = x^n - 1$

Again the accuracy of the Jenkins/Traub program drastically decreases for relatively small degrees ($n = 50$). It cannot be used for degrees $n > 60 \dots 70$. On the other hand our method yields the best results with nearly computer accuracy up to the degree $n = 10000$. The eigenvalue method is slightly worse regarding the accuracy but could be used only for degrees up to about 500. This is

n	e_E	e_J	e_N	$\tilde{e}$	t_E	t_J	t_N
10	$7.022 \cdot 10^{-16}$	$1.466 \cdot 10^{-15}$	$5.661 \cdot 10^{-16}$	$1.784 \cdot 10^{-16}$	0.04	0.03	0.02
20	$1.422 \cdot 10^{-15}$	$7.462 \cdot 10^{-15}$	$1.159 \cdot 10^{-15}$	$1.610 \cdot 10^{-16}$	0.05	0.05	0.04
50	$1.355 \cdot 10^{-15}$	$1.082 \cdot 10^{-5}$	$1.024 \cdot 10^{-15}$	$1.964 \cdot 10^{-16}$	0.34	0.14	0.11
70	$1.490 \cdot 10^{-15}$	$3.569 \cdot 10^{-1}$	$6.280 \cdot 10^{-16}$	$1.110 \cdot 10^{-16}$	0.76	0.25	0.18
100	$2.253 \cdot 10^{-15}$	$3.680 \cdot 10^{-1}$	$1.180 \cdot 10^{-15}$	$7.390 \cdot 10^{-17}$	1.74	0.45	0.32
200	$2.811 \cdot 10^{-15}$	–	$1.180 \cdot 10^{-15}$	$2.073 \cdot 10^{-16}$	11.77	–	0.67
500	$3.581 \cdot 10^{-15}$	–	$1.024 \cdot 10^{-15}$	$2.059 \cdot 10^{-16}$	316.12	–	2.64
1000	–	–	$1.106 \cdot 10^{-15}$	$2.144 \cdot 10^{-16}$	–	–	9.04
2000	–	–	$1.043 \cdot 10^{-15}$	$1.794 \cdot 10^{-16}$	–	–	31.48
10000	–	–	$1.024 \cdot 10^{-15}$	$2.219 \cdot 10^{-16}$	–	–	1036.80

Table 9: $P_{11}(x) = x^n + 1$

because of the necessary large memory to store the companion matrix $n^2 \cdot 8$Byte (double precision) which is $2 \cdot 10^6$Byte for $n = 500$ and $8 \cdot 10^8$Byte for $n = 10000$. The CPU time for our method is always the smallest as can be seen in Figs. 4(b) and 5(b). Especially for $n = 500$ it is 2.59s ($P_{10}(x)$) and 2.64s ($P_{11}(x)$) compared to 315.2s ($P_{10}(x)$) and 316.1s ($P_{11}(x)$). As in all examples above the estimate $\tilde{e}$ is close to the actual accuracy of our method.

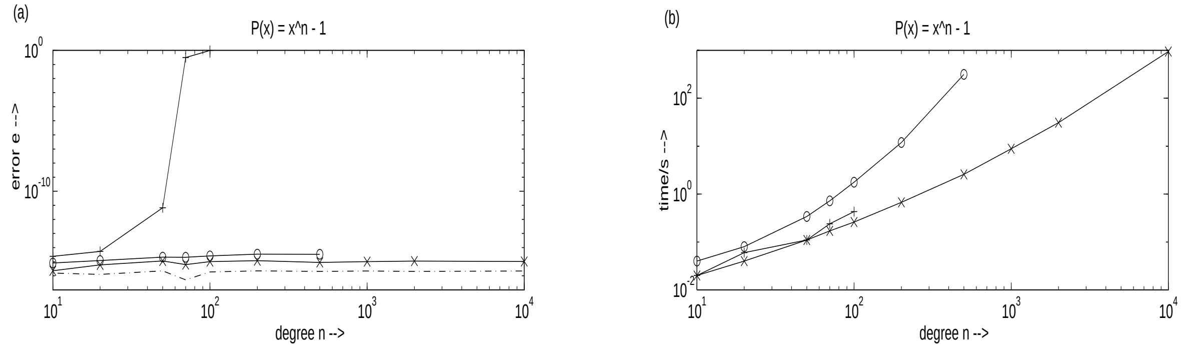

Figure 4: Results of Jenkins/Traub (+), eigenvalue ($\circ$), and our method ($\times$) for $P_{10}(x)$. (a) Actual and estimated accuracy e and $\tilde{e}$ ($- \cdot -$); (b) CPU time.

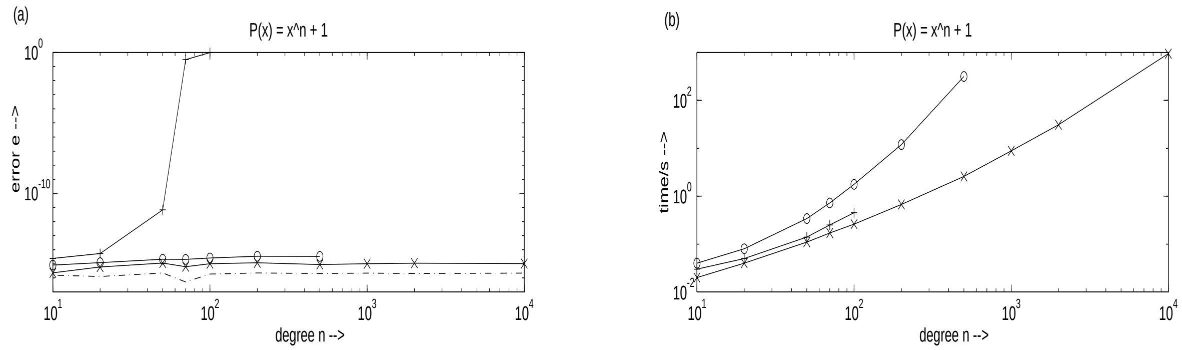

Figure 5: Results of Jenkins/Traub (+), eigenvalue ($\circ$), and our method ($\times$) for $P_{11}(x)$. (a) Actual and estimated accuracy e and $\tilde{e}$ ($- \cdot -$); (b) CPU time.

4 Discussion

4.1 Spectral Factorization

The problem of spectral factorization is to find all roots of a symmetric polynomial $H(x)$ (a polynomial where the existence of a root at x_ν implies the existence of a root at $(x_\nu^*)^{-1}$) with the additional property that all roots on the unit circle have even multiplicity. In the following we assume they have multiplicity two. After finding the roots of $H(x)$ one is interested to form a minimum phase polynomial or in general a polynomial $P(x)$ of degree n such that

$$H(x) = P(x) \cdot x^n P^*((x^*)^{-1}) \tag{25}$$

holds.

As we mentioned earlier, in general the limiting accuracy for double roots is half the computer accuracy. However, in the special case where we know their modulus it is possible to use a simple procedure to compute them up to computer accuracy. This can be done by the following procedure where we assume that all roots on the unit circle are well separated which holds in nearly all practical cases.

In a first step we compute all zeros of the original polynomial $H(x)$ and separate them into those lying inside, outside, and on the unit circle. To separate these three regions we chose an annulus with thickness 100 times the expected accuracy where we use the estimate $\tilde{e}$ of our program. In a second step we compute the roots of $H'(x)$. It has the same roots on the unit circle as the original polynomial $H(x)$ but with multiplicity 1. Consequently, they can be computed with higher accuracy. For this polynomial we are only interested in the roots lying on the unit circle where we now chose a much thinner separating annulus corresponding to the higher accuracy. If the number of roots of $H(x)$ on the unit circle is twice the number of those of $H'(x)$ these can be immediately used as an improved estimate.

We have implemented this approach into a MATLAB file (see Appendix) and give an example by using the polynomial $P(x) = P_9(x)$ with degree $n = 100$. We computed $H(x)$ according to Eq. (25) and the corresponding minimum phase part according to the method above. The maximum error of the roots of the resulting polynomial compared to the original one is $1.4 \cdot 10^{-14}$, which is hardly worse than the achieved accuracy of the original polynomial (cf. Table 7, $M = 25$). The roots of $H(x)$ on the unit circle could be determined only with accuracy $6 \cdot 10^{-9}$ so that there is an improvement of a factor $5 \cdot 10^5$.

We point out that the operation of computing the polynomial coefficients from the roots, we call it coefficient finding, must be done with care. Otherwise the errors introduced by it can be much larger than the given accuracy. This is investigated in the following section.

4.2 Coefficient Finding

As we have seen coefficient finding is a necessary step when we do spectral factorization. The fact that computing the polynomial coefficients from the roots can lead to very large errors although the polynomial has the best numerical condition one can think of, seems not to be well-known. The only related reference we could find is [13]. We want to give an intuition of what problems may arise and how these can be overcome.

We again consider the polynomial $P_{10}(x)$ with all its roots $\mathrm{e}^{\mathrm{j}\nu 2\pi/n}$, $\nu = 1 \ldots n$ on the unit circle. Let us assume we compute the polynomial coefficients in a straightforward manner using the order implied by the roots given above. The resulting coefficients corresponding to the terms $x \ldots x^{n-1}$ are expected to be zero within computer accuracy. However, for $n = 20, 50, 100, 200$ the maximum values of these coefficients are $8.7 \cdot 10^{-13}$, $1.6 \cdot 10^{-5}$, $8 \cdot 10^7$, and $5.5 \cdot 10^{108}$. This at the first glance surprising result can be explained as follows.

In the process of computing the polynomial coefficients we have intermediate polynomials with roots only on a sector of the unit circle. These are known to be ill conditioned [11]. Furthermore they have coefficients with a large dynamical range. This last property leads to errors in the next step where the intermediate polynomial has to be convolved with a first order polynomial and numbers of different orders have to be added. The last observation shows an easy way to circumvent this problem. One merely has to sort the roots in such a way that each intermediate polynomial has approximately equidistantly distributed roots. These lead to coefficients with a small dynamic range of the coefficients.

If n is a power of two this can be easily done by interpreting the indices $\nu = 1 \ldots n$ of the roots as binary numbers, reverse the order of the digits and interpret this again as a decimal number. This procedure is known as *bit reversal* or *van der Corput sequence* [15, 6]. In the case n is not a power of two we simply continue the sequence $1 \ldots n$ to the next larger power of two. Then we proceed as described and cancel all values larger than n from the resulting sequence. As an example consider the vector $[1 \dots 20]$ which becomes

$$[1\ 17\ 9\ 5\ 13\ 3\ 19\ 11\ 7\ 15\ 2\ 18\ 10\ 6\ 14\ 4\ 20\ 12\ 8\ 16].$$

Using this "reindexing", the error for $n = 20, 50, 100, 200$ is 10^{-15}, $4 \cdot 10^{-15}$, $6 \cdot 10^{-15}$, $3 \cdot 10^{-14}$, which is a drastical improvement compared to the results above.

Although this bit reversal is primarily suited for equidistantly distributed roots on the unit circle it can also be successfully used for coefficient finding of an FIR filter since these often have a similar root distribution. However, there is a generalization called Leja ordering for arbitrary root locations [13] which is computationally more expensive but yields slightly improved results for FIR filters compared to the bit reversal. The corresponding MATLAB functions can be found in the Appendix.

5 Conclusion

The following conclusions can be drawn from the examples given in Sec. 3: All three methods yield comparable results regarding speed and accuracy when working with "difficult" low degree ($n < 20$) polynomials. For these the CPU time is on a low and comparable level. The accuracy of our method is always better than that of the eigenvalue method and better than that of the Jenkins/Traub method in most cases. For larger degrees ($n > 30 \dots 40$) the accuracy of the Jenkins/Traub method drastically decreases and it yields useless results for $n > 60 \dots 70$. The accuracy of our method is better than that of the eigenvalue method in every case sometimes by more than a factor of 1000 (cf. $P_9(x)$). Furthermore the CPU time of our method increases much slower than that of the eigenvalue method, e.g., it is faster than a factor of hundred for $n = 500$. This fact and the linear increasing memory (depending on n) compared to a quadratical needed for the eigenvalue method makes it possible to find roots in considerable time even for high degree polynomials ($\approx$ 1000s for a degree 10000 polynomial).

To summarize the results, our program seems to have no drawbacks (comparable to good methods for low degrees) but essential advantages regarding accuracy and speed which is especially true for for large degrees. A C version of the program can be obtained by the authors.

Furthermore we gave a powerful approach for spectral factorization which considerably improves the accuracy. Finally we have considered the inverse problem to factorization, i.e., finding the polynomial coefficients from the roots. We showed that large errors can result from this process and we gave a simple method to minimize them nearly up to computer accuracy.

A M-File for Spectral Factorization

```
function [p_out] = factorize(p_in,string)
% function [p_out] = factorize(p_in,string)
%
%    Input:   p_in, string
%
%    Output:  p_out
%
%    Description: Program yields (spectral) factorization of the polynomial
%                 p_in regarding to the the unit circle. For reliable
%                 results p_in should have single roots off the unit circle
%                 and double roots on the unit circle. If the optional input
%                 variable string contains at least one 'a' the maximum phase
%                 portion is determined (taking the double roots of p_in on the
%                 unit circle once + all roots of p_in outside the unit circle).
%                 In all other cases the function yields the minimum phase
%                 portion.
%
%    subroutines: ransort.m bitrev.m isodd.m rootsl.mex
%
%

%File Name: factorize.m
%Last Modification Date: %G% %U%
%Current Version: %M% %I%
%File Creation Date: Tue Sep  7 09:29:06 1993
%Author: Markus Lang  <lang@dsp.rice.edu>
%
%Copyright: All software, documentation, and related files in this distribution
%           are Copyright (c) 1993  Rice University
%
%Permission is granted for use and non-profit distribution providing that this
%notice be clearly maintained. The right to distribute any portion for profit
%or as part of any commercial product is specifically reserved for the author.
%
%Change History:
%

p_in = p_in(:)';
n = length(p_in) - 1;
```

```
% find roots of original and differentiated polynomial
[r_orig,e] = rootsl(p_in);  r_orig = r_orig(:).';  error_orig = e;
[r_dif, error_dif] = rootsl((n:-1:1).*p_in(n+1:-1:2));  r_dif = r_dif(:).';

% find roots on, inside and outside the unit circle (original pol.)
% and sort by angle
ind_orig_u = find(abs(1-abs(r_orig))<100*error_orig);
ind_orig_i = find(abs(r_orig)<abs(1-100*error_orig));
ind_orig_o = find(abs(r_orig)>abs(1+100*error_orig));
r_orig_u = r_orig(ind_orig_u);  [dum,ind] = sort(angle(r_orig_u));
r_orig_u = r_orig_u(ind);  n_orig_u = length(r_orig_u);
r_orig_i = r_orig(ind_orig_i);  [dum,ind] = sort(angle(r_orig_i));
r_orig_i = r_orig_i(ind);  n_orig_i = length(r_orig_i);
r_orig_o = r_orig(ind_orig_o);  [dum,ind] = sort(angle(r_orig_o));
r_orig_o = r_orig_o(ind);  n_orig_o = length(r_orig_o);

% check multiplicity of roots on unit circle
if isodd(n_orig_u);
   disp('There are roots on unit circle with odd multiplicity!!');
   return
end

% check whether all roots could be sorted
if n_orig_u+n_orig_i+n_orig_o ~= n;
   disp('Not all roots could be sorted');
   return
end

% find roots on the unit circle (differentiated pol.) and sort by angle
ind_dif_u = find(abs(1-abs(r_dif))<100*error_dif);
r_dif_u = r_dif(ind_dif_u);  [dum,ind] = sort(angle(r_dif_u));
r_dif_u = r_dif_u(ind);  n_dif_u = length(r_dif_u);

% check whether the numbers of roots on the unit circle of both polynomials
% fit together
if 2*n_dif_u ~= n_orig_u
   disp('The numbers of roots on unit circle do not fit together!!');
   return
end

% check whether maximum or minimum phase polynomial is desired
r_out = [r_dif_u r_orig_i];
if exist('string') == 1;
   if length(find(string=='a')) > 0;
     r_out = [r_dif_u r_orig_o];
   end
end
```

```
% compute  output polynomial
[dum,ind] = sort(angle(r_out)); r_out = r_out(ind);
p_out = poly(r_out(ransort(length(r_out))));
```

```
function odd = isodd(x)
% function odd = isodd(x)
%
% function yields a matrix odd which is 1 for every odd
% value of abs(x) and 0 else.
%
% ml, 20.8.1992
%
% Copyright Lehrstuhl fuer Nachrichtentechnik Erlangen, FRG
% e-mail: int@nt.e-technik.uni-erlangen.de

[m,n] = size(x);
odd = zeros(m,n);
ind = find((x-1)/2==fix((x-1)/2));
odd(ind) = 1 + odd(ind);
```

B M-File for Bit Reversal

```
function [ind] = ransort(n)

% function [ind] = ransort(n)
%
% function computes an index vector ind of length n which contains
% all integers 1:n but in a quasi random order.
%
% m-file bitrev must be available
% author: m. lang 17.12.91

% This is very advantageous for example for the function poly:
% compute z = roots([1 zeros(1:m) 1]); p = poly(z).
% There is a large error in p, which stems from the second operation
% poly. If the roots z are sorted with increasing angle in zs, then
% ph = poly(zs(ransort(n))) yields much less error than p.
%
% ransort uses bitreversal which is known as van der corput sequence
% in the theory of uniform distributed sequences.

m = ceil(log2(n));
```

```
ind = 1:2^m;
ind =  bitrev(ind);
ind = ind(find(ind<=n));
```

```
function re = bitrev(rfeld)
%
% BITREV   Y = BITREV(X) returns the vector X in bitreversed order
%
%
%       Author : Raimund Meyer 15.12.87
%       Revised: Rai           26.08.88

% Copyright (C) 1987-1991 Lehrstuhl fuer Nachrichtentechnik,
% University of Erlangen, FRG

dim = length(rfeld);
dim1 = dim - 1;
j = 1;
for i = 1:dim1
    if i < j
       xt       = rfeld(j);
       rfeld(j) = rfeld(i);
       rfeld(i) = xt;
    else
    end
    k = dim/2;
    while k < j
       j = j - k;
       k = k/2;
    end
    j = j + k;
end
re = rfeld;
```

C M-File for Leja Ordering

```
function [x_out] = leja(x_in)
% function [x_out] = leja(x_in)
%
%    Input:   x_in
%
%    Output:  x_out
%
%    Program orders the values x_in (supposed to be the roots of a
%    polynomial) in such a way that computing the polynomial coefficients
```

```
%    by using the m-file poly yields most accurate results.
%    Try, e.g.,
%                z=exp(j*(1:100)*2*pi/100);
%    and compute
%                p1 = poly(z);
%                p2 = poly(leja(z));
%    which both should lead to the polynomial x^100-1. You will be
%    surprised!
%
%    ref.: Nachtigal, Reichel, Trefethen. "A Hybrid GMRES Algorithm for
%          Nonsymmetric Linear Systems. SIAM J. Matr. Anal. and Appl., 13,
%          796-825, July 1992.

%File Name: leja.m
%Last Modification Date: %G% %U%
%Current Version: %M% %I%
%File Creation Date: Mon Nov  8 09:53:56 1993
%Author: Markus Lang  <lang@dsp.rice.edu>
%
%Copyright: All software, documentation, and related files in this distribution
%           are Copyright (c) 1993  Rice University
%
%Permission is granted for use and non-profit distribution providing that this
%notice be clearly maintained. The right to distribute any portion for profit
%or as part of any commercial product is specifically reserved for the author.
%
%Change History:
%

x = x_in(:).'; n = length(x);

a = x(ones(1,n+1),:);
a(1,:) = abs(a(1,:));
[dum1,ind] = max(a(1,1:n));
dum2 = a(:,1);  a(:,1) = a(:,ind);  a(:,ind) = dum2;
x_out(1) = a(n,ind);
a(2,2:n) = abs(a(2,2:n)-x_out(1));

for l=2:n-1
  [dum1,ind] = max(prod(a(1:l,l:n)));  ind = ind+l-1;
  if l~=ind
    dum2 = a(:,l);  a(:,l) = a(:,ind);  a(:,ind) = dum2;
  end
  x_out(l) = a(n,l);
  a(l+1,(l+1):n) = abs(a(l+1,(l+1):n)-x_out(l));
end
x_out = a(n+1,:);
```

References

[1] Aliphas, Amnon, S. Shankar Narayan, and Allen M. Peterson. Finding the zeros of linear phase fir frequency sampling filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31:729–734, June 1983.

[2] Arnold, G. Über die Nullstellenverteilung zufälliger Polynome. *Mathematische Zeitschrift*, 92:12–18, 1966.

[3] Chen Xiangkun, and Thomas W. Parks. Design of optimal minimum phase fir filters by direct factorization. *EURASIP Signal Processing*, 10:369–383, 1986.

[4] Daubechies, Ingrid. *Ten Lectures on Wavelets*. SIAM, Philadelphia, PA, 1992.

[5] Frenzel, Bernhard. *Untersuchung von Algorithmen zur Bestimmung von Polynomnullstellen (Examination of Algorithms for Finding Polynomial Roots, in German)*. Studienarbeit (senior project), Institute of Communication Theory, University of Erlangen, Erlangen, 1993.

[6] Hlawka, E. *The Theory of Uniform Distribution*. AB Academic Publisher, Berkhamsted, 1984.

[7] Jenkins, M. A. Algorithm 493 zeros of a real polynomial. *ACM Transactions on Mathematical Software*, 1:178–, June 1975.

[8] Jenkins, M. A. and J. F. Traub. A three-stage algorithm for real polynomials using quadratic iteration. *SIAM Journal on Numerical Analysis*, 7:545–566, 1970.

[9] Jenkins, M. A. and J. F. Traub. Zeros of a complex polynomial. *Communications of the ACM*, 15:97–99, February 1972.

[10] Jenkins, M. A. and J. F. Traub. Principles of testing polynomial zerofinding programs. *ACM Transactions on Mathematical Software*, 1:26–34, March 1975.

[11] Lang, Markus. *Ein Beitrag zur Phasenapproximation mit Allpässen (Phase Approximation by Allpass Filters, in German).* PhD thesis, University of Erlangen-Nürnberg, Erlangen, Germany, 1993.

[12] Muller, D. E. A method for solving algebraic equations using an automatic computer. *Math. Tables Aids Comput*, 10:208–215, 1956.

[13] Nachtigal, Noël M., Lothar Reichel, and Lloyd N. Trefethen. A hybrid GMRES algorithm for nonsymmetric linear systems. *SIAM Journal on Matrix Analysis and Applications*, 13:796–825, July 1992.

[14] Press, William H. et al. *Numerical Recipes in C.* Cambridge University Press, Cambridge, 1992.

[15] Reichel, Lothar, and Gerhard Opfer. Chebyshev-Vandermonde systems. *AMS Mathematics of Computation*, 57:703–721, October 1991.

[16] Schmidt, C. E. and L. R. Rabiner. A study of techniques for finding the zeros of linear phase fir digital filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25:96–98, February 1977.

[17] Smith, B. T. et al. *Matrix Eigensystem Routines — EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science.* Springer, 1976.

[18] Steiglitz, Kenneth, and Bradley Dickinson. Phase unwrapping by factorization. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 30:984–991, December 1982.

[19] Toh, Kim-Chuan, and Lloyd N. Trefethen. Pseudozeros of polynomials and pseudospectra of companion matrices. Technical Report 93-1360, Department of Computer Science, Cornell University, Ithaca, N. Y., 1993.

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