

Polynomial Roots by Optimization

By
Namir C. Shammas

Contents

Contents.....	1
Introduction	1
The Devil is in the Details.....	2
Back to the Drawing Board	3
A Funny Thing Happened on the Way to Heaven	8
A Quasi Lin-Bairstow Method	10
The Durand-Kerner Algorithm.....	13
Conclusions	16
Document History.....	17

Introduction

Numerical Analysts have put much effort in calculating the roots of polynomials. There seems to be three schools of thought that vary in the complexity of the problem:

1. Algorithms that find the real roots for real-coefficient polynomials.
2. Algorithms that find all roots for real-coefficient polynomials.
3. Algorithms that find all roots for complex-coefficient polynomials.

The first brand of problems can use classical root-finding algorithms to determine the real roots of polynomials. They should include a criterion (such as the maximum number of iterations) to determine when all the real roots have been obtained. By contrast, the third brand of problems require a programming language, such as MATLAB, that can handle complex math with ease. Moreover, such a language can switch between real and complex calculations in a seamless and transparent way.

This article deals with the second brand of problems. The most popular algorithm that calculates all real and complex roots of real-coefficient polynomials is the Lin-

Bairstow method. This method repeatedly factors out quadratic polynomials and determine their real and complex roots. The Lin-Bairstow method is quite robust and requires only real-math arithmetical operations. This feature makes it very attractive for system that don't support complex math or require using complex math libraries. You can implement the Lin-Bairstow on a whole diverse systems and programming languages.

This article looks at calculating the real and complex roots of real-coefficient polynomials. The approach I will be using involves optimization of the Vieta Formulas. These formulas describe mathematical properties of the roots and coefficients of the following general polynomial:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (1)$$

The Vieta formulas describe the relations between the coefficient polynomials with the roots x_1, \dots, x_n :

$$x_1 + x_2 + \dots + x_{n-1} + x_n = -a_{n-1} / a_n \quad (2)$$

$$(x_1 x_2 + x_1 x_3 + \dots + x_1 x_n) + (x_2 x_3 + x_2 x_4 + \dots + x_2 x_n) + \dots + x_{n-1} x_n = a_{n-2} / a_n$$

...

$$x_1 x_2 x_3 x_4 \dots x_n = (-1)^n a_0 / a_n$$

The basic approach that I propose is to optimize the roots (starting with guesses) until all the equations in set 2 are obeyed. The optimizing function would be the norm of the sum of difference squared between the left-hand side and right-hand side of equations in set 2.

$$F(\mathbf{x}, \mathbf{a}) = \sum_{i=1}^n (LHS(i) - RHS(i))^2 \quad (3)$$

The Devil is in the Details

The first step in solving the roots of real-coefficient polynomials is to develop programming code for equation 3. I had solicited the help of fellow calculator enthusiasts (and math minded folks) at www.hpmuseum.com. My math *challenge* was met quickly by Spanish member Valentin Albillo. He provided code for the HP-71B calculator in BASIC. The solution Albillo provided is stunningly simple—the work of a brilliant mathematician. I was able to find just another similar solution in

C++ code on the web. It seems that many programmers have shied away from coding the Vieta Formulas to yield equation 3. I was very fortunate that Albillo accepted my challenge and provided me with an essential tool for the optimization process.

I chose to use MATLAB for various reasons, including its ability to switch between real and complex math in a seamless and transparent way. This feature of MATLAB proved to be more valuable than I thought—more about that later in this article. To add more good news, the latest version of MATLAB includes the robust Particle Swarm Optimization (PSO) function `particleswarm()`. The PSO algorithm is a robust evolutionary method and having a finetuned version in MATLAB was a temptation I could not resist.

Armed with Albillo's code and the `particleswarm()` function I approached the problem of optimizing equation 3 in very general terms. I took into account that each root is basically complex—the real roots having imaginary values of 0. This meant that for a polynomial of order N , I had to optimize $2*(N+1)$ variables.

My first and valiant attempt to optimize equation 3 used a general optimization approach and hoped for the best—obtaining some real roots and pairs of conjugate complex roots. The MATLAB code yielded results but none was correct! I had bit more than I could chew! I had expected my MATLAB code to behave more intelligently that its code delivered.

Back to the Drawing Board

After my first attempt yielding failure, I decided to study the problem more carefully. I decided that I needed to first extract the real roots and let equation 3 deal with a reduced polynomial that has pairs of conjugate complex roots. The code for the optimized MATLAB function should consider that there are pairs of conjugate complex roots. This approach reduced the number of optimized variables and forced the optimization process to consider the pairs of conjugate complex roots.

Listing 1 shows the MATLAB code.

```
Function [xroots,nroots] =  
newtonpolyrootsEx(polyCoeffs,toler,maxiter,maxIterOptim)  
% NEWTONPOLYROOTSEX extracts real and complex roots of a polynomial  
% in two stages:  
% Stage 1: Using Newton's method to determine the real roots of the  
% polynomial.  
% Stage 2: Using the optimization of the Vieta's formulas for the  
% reduced polynomials that have only complex roots.
```

```

%
% INPUT
% =====
% polyCoeffs - array of polynomial coefficients. The first element is the
% coefficient of the highest power, and so on.
% toler - the toleranced used in finding real roots using Newton's method.
% maxiter - the maximum number of iteration used with Newton's method.
% maxIterOptim - the maximum number of iteration used with optimizing
% the sum of squares of the differences between the LHS and RHS of the Vieta
% formulas.
%
% OUTPUT
% =====
% xroots - two-dimensional matrix of the roots. The values in column 1 are
% the real parts, and in column 2 are the imaginary parts.
% nroots - the number of roots.
%
% EXAMPLE
% =====
% c=[1 -2 44 -66 22 -11 -55]
%
% c =
%
%      1      -2      44     -66      22     -11     -55
%
% [xroots,nroots] = newtonpolyrootsEx(c,1e-10,100,1000)
% Optimization ended: relative change in the objective value
% over the last OPTIONS.MaxStallIterations iterations is less than
% OPTIONS.FunctionTolerance.
%
% xroots =
%
%      1.6089         0
%     -0.7138         0
%      0.2341     -6.5335
%      0.2341      6.5335
%      0.3184     -1.0095
%      0.3184      1.0095
%
%
% nroots =
%
%      6
%
%
% global polcoeff
% n = length(polyCoeffs) - 1;
% xroots = zeros(n,1); % max real roots, maybe?
% nroots = 0;
% x = complex(pi,pi);
% ----- Stage 1 -----
% Use Newton's method with real-only math to find the real
% roots of the polynomial. With each real root found, the code
% deflate the current polynomial.
while n>1
    polyCoeffsDeriv = polyder(polyCoeffs);
    for i=1:maxiter

```

```

        drv = polyval(polyCoeffsDeriv, x);
        diff = polyval(polyCoeffs, x)*drv/(drv^2 + 1e-10);
        x = real(x - real(diff));
        if abs(real(diff))<toler, break; end
    end
    if i<maxiter
        nroots = nroots + 1;
        xroots(nroots) = x;
        [polyCoeffs,~] = deconv(polyCoeffs,poly([x]));
        n = n - 1;
    else
        break;
    end
end
end

% ----- Stage 2 -----
% Determine the complex roots of the deflated polynomial by
% optimizing the norm of the differences between the LHS and
% RHS of the Vieta formulas.
if n >=2 && mod(n,2) == 0
    lb = -20 + zeros(n,1);
    ub = 20 + zeros(n,1);
    nvars = n;
    polcoeff = polyCoeffs;
    options = optimoptions('particleswarm', 'MaxIterations', maxIterOptim);
    options = optimoptions('particleswarm', 'FunctionTolerance', toler);
    numRetry = 50;
    while numRetry>0
        [x,~,exitflag] = particleswarm(@optimFun,nvars,lb,ub,options);
        if exitflag==1, break; end
        numRetry = numRetry - 1;
    end
    for i=1:2:n
        nroots = nroots + 1;
        xroots(nroots,1) = x(i);
        xroots(nroots,2) = x(i+1);
        nroots = nroots + 1;
        xroots(nroots,1) = x(i);
        xroots(nroots,2) = -x(i+1);
    end
end
end
end

function y = vf(X,A)
% VF Uses the Vieta's Formulas to calculate a function
% that returns the norm of the differences between the LHS
% and RHS Vieta formulas.
%
% Many thanks to Valentin Albillo (of the hp museum web site) for providing
% me with a most elegant and compact HP-71B calculator BASIC code.
n = length(X);
A = A / A(1);
%A = A(n+1:-1:1);
B = zeros(n+1,1);
B(n+1) = 1;
for i=1:n
    for j=n-i:n-1

```

```

        B(j+1) = B(j+2)-B(j+1)*X(i);
    end
    B(n+1) =-B(n+1)* X(i);
end
B = B - A';
y = norm(B)^2;
end

function y = optimFun(x)
% The optimized function that uses the function vf(). This function creates
% complex variables from the array x.
global polcoeff
n = length(x);
xr = zeros(n,1);
j = 1;
for i=1:2:n
    xr(j) = complex(x(i),x(i+1));
    xr(j+1) = complex(x(i),-x(i+1));
    j = j + 2;
end
y = vf(xr,polcoeff);
end

```

Listing 1. MATLAB code to obtain the roots of polynomials by optimization.

Listing 1 shows the code for function `newtonpolyrootsEx()` and reveals that the process involves two main stages:

1. The first stage uses Newton's method in real math mode to obtain all the real roots of the targeted polynomial. The function `polyval()` calculates the value of the polynomial for a given value of x . The function `polyder()` yields the coefficients of the first derivative of a polynomial. The function `deconv()` deflates a polynomial given a linear equation (i.e. $ax + b$) based on a single real root.
2. Stage 2 works with the deflated polynomial that has complex roots. The function `particleswarm()` performs optimization on the function `optimFn()`. The code for this function builds pairs of complex conjugate roots before calling the Vieta Formula function `vf()`. The `vf()` function returns the value for equation 3. This stage must be able to handle complex math.

This dual scheme algorithm works well and allows the optimization part to systematically find the conjugate complex roots, thus reducing the number of optimizing variables.

Here is an example for solving for the roots of $P(x) = x^6 - 2x^5 + 44x^4 - 66x^3 + 22x^2 - 11x - 55 = 0$. I obtain the roots by calling function `newtonpolyrootsEx()` with the following arguments:

- The array `c` which contains the polynomial coefficients.
- The tolerance value 10^{-10} used with Newton's method.
- The maximum number of iterations of 100 used with Newton's method.
- The maximum number of optimization iterations of 1000.

```
>> c=[1 -2 44 -66 22 -11 -55]

c =

    1    -2    44   -66    22   -11   -55

>> [xroots,nroots] = newtonpolyrootsEx(c,1e-10,100,1000)
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than
OPTIONS.FunctionTolerance.

xroots =

    1.6089         0
   -0.7138         0
    0.2341   -6.5335
    0.2341    6.5335
    0.3184   -1.0095
    0.3184    1.0095

nroots =

     6

>> roots(c)

ans =

    0.2341 + 6.5335i
    0.2341 - 6.5335i
    1.6089 + 0.0000i
    0.3184 + 1.0095i
    0.3184 - 1.0095i
   -0.7138 + 0.0000i
```

The last MATLAB command calls the built-in `roots()` function to calculate the roots of the targeted polynomial and compare the results with function `newtonpolyrootsEx()`. The function `roots()` uses eigenvalues and eigenvectors to calculate the roots of polynomials.

A Funny Thing Happened on the Way to Heaven

I have been using Newton's method since the middle of the 70s. Just about all the implementations I wrote use real math. I discovered that MATLAB allows Newton's method to obtain both real and complex roots in a simple and systematic way. Listing 2 shows the code for the MATLAB function `newtonpolyroots()`:

```
function [xroots,nroots] = newtonpolyroots(polyCoeffs,toler,maxiter)
% NEWTONPOLYROOTS extracts real and complex roots of a polynomial
% using Newton's method and real/complex math operations.
%
% INPUT
% =====
% polyCoeffs - array of polynomial coefficients. The first element is the
% coefficient of the highest power, and so on.
% toler - the tolerance used in finding real roots using Newton's method.
% maxiter - the maximum number of iteration used with Newton's method.
%
% OUTPUT
% =====
% xroots - two-dimensional matrix of the roots. The values in column 1 are
% the real parts, and in column 2 are the imaginary parts.
% nroots - the number of roots.
%
% c=[1 -2 44 -66 22 -11 -55]
%
% c =
%
%      1      -2      44     -66      22     -11     -55
%
% [xroots,nroots] = newtonpolyroots(c,1e-10,100)
%
% xroots =
%
%      0.3184 + 1.0095i
%      0.3184 - 1.0095i
%     -0.7138 + 0.0000i
%      1.6089 - 0.0000i
%      0.2341 - 6.5335i
%      0.2341 + 6.5335i
%
%
% nroots =
%
%      6
%
% n = length(polyCoeffs) - 1;
%
% check if any polynomial coefficient is complex
isComplexCoeff = false;
for i=1:n
    if abs(imag(polyCoeffs(i))) > 1e-10
        isComplexCoeff = true;
        break;
    end
end
```



```

    end
end

xroots = zeros(n,1); % max real roots, maybe?
nroots = 0;
x = complex(pi,pi);
while n>1
    polyCoeffsDeriv = polyder(polyCoeffs);
    for i=1:maxiter
        drv = polyval(polyCoeffsDeriv, x);
        diff = polyval(polyCoeffs, x)*drv/(drv^2 + 1e-10);
        x = x - diff;
        if abs(diff)<toler, break; end
    end
    if i<=maxiter
        nroots = nroots + 1;
        xroots(nroots) = x;
        [polyCoeffs,~] = deconv(polyCoeffs,poly([x]));
        n = n - 1;

        if abs(imag(x)) > 1e-7 && ~isComplexCoeff
            x = complex(real(x),-imag(x));
            nroots = nroots + 1;
            xroots(nroots) = x;
            [polyCoeffs,~] = deconv(polyCoeffs,poly([x]));
            n = n - 1;
        end
    end

    else
        break;
    end
end

if i<=maxiter && n==1
    nroots = nroots + 1;
    xroots(nroots) = -polyCoeffs(2)/polyCoeffs(1);
end
end
end

```

Listing 2. MATLAB code to obtain the roots of polynomials using Newton's method.

Listing 2 shows Newton's method operating in real and complex modes to calculate all the real and imaginary roots of a real-coefficient polynomial. The function sets the initial guess for the first root as $\text{complex}(\pi, \pi)$. The function also extracts the conjugate complex root when it finds a complex root. The function uses the current root as the guess for the next root.

Here is a sample session with function `newtonpolyroots()` to obtain the roots of the polynomial in the previous example. I obtain the roots by calling function `newtonpolyroots()` with the following arguments:

- The array c which contains the polynomial coefficients.
- The tolerance value 10^{-10} .
- The maximum number of iterations of 100.

```
>> [xroots,nroots] = newtonpolyroots(c,1e-10,100)

xroots =

    0.3184 + 1.0095i
    0.3184 - 1.0095i
   -0.7138 + 0.0000i
    1.6089 - 0.0000i
    0.2341 - 6.5335i
    0.2341 + 6.5335i

nroots =

     6

>> roots(c)

ans =

    0.2341 + 6.5335i
    0.2341 - 6.5335i
    1.6089 + 0.0000i
    0.3184 + 1.0095i
    0.3184 - 1.0095i
   -0.7138 + 0.0000i
```

A Quasi Lin-Bairstow Method

I actually started this paper by looking at integrating optimization with the Lin-Bairstow approach. This algorithm is based on the following equation:

$$P(x) - q(x) B(x) + r(x) \quad (4)$$

Where $q(x)$ (equal to $x^2 + q_1x + q_0$) is a quadratic polynomial and $r(x)$ is a linear equation (equal to $r_1x + r_0$). When $q(x)$ contains exact real or complex roots of $P(x)$, the coefficients of $r(x)$ are both zero. Optimization works by altering the two coefficients of $q(x)$ such that the sum $r_1^2 + r_0^2$ become zero or a very small positive number.

There is a wealth of classical and evolutionary algorithms one can choose from. I have tested a good variety of classical optimization algorithms. The results is that many of these algorithms give correct roots with certain polynomials but not others,

depending on the combination and sequences of polynomial coefficients. In other words, many of the classical optimization algorithms are not robust as one might have hoped.

Listing 3 shows the `slb()` function that implements the Quasi Lin-Bairstow method using, for example, the `particleswarm()` optimization function.

```
function [xroots,nroots] = slb(polyCoeffs,toler,maxIter,maxRetry)
%SLB implements a quasi Lin-Bairstow method by using the particleswarm
%function for PSo optimization.
    global polcoeff

    if nargin<2, toler=1e-10; end
    if nargin<3, maxIter=10000; end
    if nargin<4, maxRetry=1; end

    rng('shuffle')
    polcoeff = polyCoeffs/polyCoeffs(1);
    order=length(polcoeff)-1;
    xroots=zeros(order,2);
    nroots=0;
    while order>=2
        lb = [-20 -20];
        ub = [20 20];
        nvars = 2;
        options = optimoptions('particleswarm', 'MaxIterations', maxIter);
        options = optimoptions('particleswarm', 'FunctionTolerance', toler);
        numRetry = maxRetry;
        while numRetry>0
            [x,~,exitflag] = particleswarm(@optimFun,nvars,lb,ub,options);
            if exitflag==1, break; end
            numRetry = numRetry - 1;
        end
        xaug = [1 x]; % augment x
        [r1,i1,r2,i2]=quadratic(xaug);
        xroots(nroots+1,1)=r1;
        xroots(nroots+1,2)=i1;
        xroots(nroots+2,1)=r2;
        xroots(nroots+2,2)=i2;
        nroots = nroots + 2;
        order = order - 2;
        [q,~] = deconv(polcoeff, xaug);
        polcoeff = q;
        if exitflag~=1
            fprintf('Exit flag is %i ', exitflag)
            fprintf('for roots (%f,i%f) and (%f,i%f)\n', r1, i1, r2, i2);
        end
    end

    if order == 1
        xroots(nroots+1,1)=-polcoeff(2)/polcoeff(1);
        xroots(nroots+1,2)=0;
```

```

        nroots = nroots + 1;
    end
end

function [r1,i1,r2,i2]=quadratic(coeff)
    discr=coeff(2)^2-4*coeff(1)*coeff(3);
    if discr>=0
        i1=0;
        i2=0;
        r1=(-coeff(2)+sqrt(discr))/2/coeff(1);
        r2=(-coeff(2)-sqrt(discr))/2/coeff(1);
    else
        r1 =-coeff(2)/2/coeff(1);
        r2 = r1;
        i1 = sqrt(abs(discr))/2/coeff(1);
        i2 = -i1;
    end
end

function y = optimFun(x)
    global polcoeff
    xaug = [1 x];
    [~,r] = deconv(polcoeff,xaug);
    y = norm(r);
end

```

Listing 3. MATLAB code to obtain the roots of polynomials using the quasi Lin-Bairstow method.

Here is a sample session with function `slb()` to obtain the roots of the polynomial in the previous examples. I obtain the roots by calling function `slb()` with the following arguments:

- The array `c` which contains the polynomial coefficients.
- The tolerance value 10^{-10} .
- The maximum number of iterations of 1000.
- The maximum number of 50 retries with the `particleswarm()` function.

```

>> [xroots,nroots] = slb(c,1e-10,1000,50)
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than
OPTIONS.FunctionTolerance.
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than
OPTIONS.FunctionTolerance.
Optimization ended: relative change in the objective value
over the last OPTIONS.MaxStallIterations iterations is less than
OPTIONS.FunctionTolerance.

xroots =

    0.3184    1.0095
    0.3184   -1.0095

```

```

1.6089      0
-0.7138      0
0.2341     4.4660
0.2341    -4.4660

nroots =

     6

```

The Durand-Kerner Algorithm

Before I wrap things up with this study, I want to share with you an interesting algorithm, that I came across, for calculating the roots of polynomials. The Durand-Kerner algorithm simultaneously calculates the real and complex roots of real-coefficient AND complex-coefficient polynomials. The approach resembles the Gauss-Seidel method for solving a system of linear equations. The algorithm uses a set of N equations to calculate N roots for the polynomial $f(x)$. Here is an example for the algorithm solving the roots of a fourth order polynomial. Notice that the updated root values ARE used in subsequent equations, within the same iteration.

$$\begin{aligned}
 p_n &= p_{n-1} - f(p_{n-1}) / [(p_{n-1} - q_{n-1})(p_{n-1} - r_{n-1})(p_{n-1} - s_{n-1})] \\
 q_n &= q_{n-1} - f(q_{n-1}) / [(q_{n-1} - p_n)(q_{n-1} - r_{n-1})(q_{n-1} - s_{n-1})] \\
 r_n &= r_{n-1} - f(r_{n-1}) / [(r_{n-1} - p_n)(r_{n-1} - q_n)(r_{n-1} - s_{n-1})] \\
 s_n &= s_{n-1} - f(s_{n-1}) / [(s_{n-1} - p_n)(s_{n-1} - r_n)(s_{n-1} - q_n)]
 \end{aligned} \tag{5}$$

Listing 4 shows the MATLAB code for the `durand_kerner()` function.

```

function xroots = durand_kerner(polyCoeffs,toler)
% DURAND_KERNER calculates the roots of a polynomial simultaneously. The
% method is relatively simple and easy to implement. The Durand-Kerner
% algorithm resembles the Gauss-Seidel method for solving simultaneous linear
% equations.

n = length(polyCoeffs) - 1;
xroots = zeros(n,1);
for i=1:n
    xroots(i) = complex(0.4,0.9)^i;
end
bStop = false;
while ~bStop
    lastRoots=xroots;
    for i=1:n
        prod = 1;
        for j=1:n
            if i~=j
                prod = prod * (xroots(i) - xroots(j));
            end
        end
        xroots(i) = xroots(i) - polyval(polyCoeffs,xroots(i))/prod;
    end
end

```

```

end

bStop = true;
for i=1:n
    if abs(abs(lastRoots(i)) - abs(xroots(i)))>toler
        bStop = false;
        break;
    end
end
end
end

end

```

Listing 4. The MATLAB code for the durand_kerner() function.

Looking at Listing 4, notice how the guesses for the roots are initialized. The function uses a for loop to initialize the initial, and somewhat arbitrary, guesses as $\text{complex}(0.4, 0.9)^i$ where i is the loop control variable. The imaginary parts for real roots converge to 0.

Here is a sample session with function `durand_kerner()` to obtain the roots of the polynomial in the previous examples. I obtain the roots by calling the function with the following arguments:

- The array `c` which contains the polynomial coefficients.
- The tolerance value 10^{-10} .

```

>> xroots = durand_kerner(c,1e-10)

xroots =

    0.2341 + 6.5335i
   -0.7138 + 0.0000i
    0.3184 + 1.0095i
    0.2341 - 6.5335i
    0.3184 - 1.0095i
    1.6089 + 0.0000i

```

Here is a version of the Durand-Kerner algorithm that resembles the Jacobi algorithm for linear equations. Notice that the updated root values are NOT used in subsequent equations. They will instead used in the next iterations.

$$\begin{aligned}
 p_n &= p_{n-1} - f(p_{n-1}) / [(p_{n-1} - q_{n-1})(p_{n-1} - r_{n-1})(p_{n-1} - s_{n-1})] \\
 q_n &= q_{n-1} - f(q_{n-1}) / [(q_{n-1} - p_{n-1})(q_{n-1} - r_{n-1})(q_{n-1} - s_{n-1})] \\
 r_n &= r_{n-1} - f(r_{n-1}) / [(r_{n-1} - p_{n-1})(r_{n-1} - q_{n-1})(r_{n-1} - s_{n-1})] \\
 s_n &= s_{n-1} - f(s_{n-1}) / [(s_{n-1} - p_{n-1})(s_{n-1} - r_{n-1})(s_{n-1} - q_{n-1})]
 \end{aligned} \tag{6}$$

Listing 5 shows the MATLAB code for the `durand_kerner2()` function.

```

function xroots = durand_kerner2(polyCoeffs,toler)
% DURAND_KERNER calculates the roots of a polynomial simultaneously. The
% method is relatively simple and easy to implement. The Durand-Kerner
% algorithm resembles the Jacobi method for solving simultaneous linear
% equations.

n = length(polyCoeffs) - 1;
xroots = zeros(n,1);
for i=1:n
    xroots(i) = complex(0.4,0.9)^i;
end
bStop = false;
while ~bStop
    lastRoots=xroots;
    for i=1:n
        prod = 1;
        for j=1:n
            if i~=j
                prod = prod * (lastRoots(i) - lastRoots(j));
            end
        end
        xroots(i) = lastRoots(i) - polyval(polyCoeffs,lastRoots(i))/prod;
    end

    bStop = true;
    for i=1:n
        if abs(abs(lastRoots(i)) - abs(xroots(i)))>toler
            bStop = false;
            break;
        end
    end
end

end

function y = optimFun(x)
    global polcoeff
    xaug = [1 x];
    [~,r] = deconv(polcoeff,xaug);
    y = norm(r);
end

```

Listing 5. The MATLAB code for the durand_kerner2() function.

Here is a sample session with function durand_kerner2() to obtain the roots of the polynomial in the previous examples. I obtain the roots by calling the function with the following arguments:

- The array c which contains the polynomial coefficients.
- The tolerance value 10^{-10} .

```
>> xroots = durand_kerner2(c,1e-10)
```

```
xroots =  
  
    1.6089 - 0.0000i  
    0.2341 + 6.5335i  
    0.3184 - 1.0095i  
   -0.7138 - 0.0000i  
    0.3184 + 1.0095i  
    0.2341 - 6.5335i
```

Conclusions

The Lin-Bairstow remains a versatile and robust algorithm used to calculate the real and complex roots for real-coefficient polynomials. The algorithm can obtain the results using only real math calculations. This feature makes it a valuable algorithm.

The Quasi-Lin-Bairstow method that I covered works as good as the underlying server optimization algorithm. There seems to be a property of the collection of polynomial coefficients, that I am currently unaware of, that determines when the optimization algorithm give correct answers. Thus, the Quasi-Lin-Bairstow method that I discussed is a bit iffy!

Newton's method, implemented in a programming language like MATLAB, also proves to be a versatile algorithm that can reliably obtain real and complex roots. Newton's method works for real-coefficient AND complex-coefficient polynomials when using MATLAB. Without the ability to switch between real and complex math, Newton's method is only good for solving for the real roots of real-coefficient polynomials.

The compound Newton-Vieta Formulas algorithm that I have proposed also provides a reliable algorithm that can work with a wide variety of real-coefficient polynomials. While the stage that uses Newton's method does not require complex math, the stage that uses the Vieta formulas does require complex math. It remains a personal choice to either use just Newton's method with real/complex math or use the compound Newton-Vieta Formulas algorithm.

The Durand-Kerner algorithm is able to calculate all real and complex roots for real-coefficient AND complex-coefficient polynomials when using programming languages like MATLAB. It is a valuable tool that you can add to your software toolbox. This paper shows that both flavors (the Gauss-Seidel-like and Jacobi-like versions) of the Durand-Kerner method work well.

Document History

<i>Date</i>	<i>Comment</i>	<i>Version</i>
August 6, 2018	Initial release.	1.0.0