Lecture 9

Polynomials

1 Introduction

The equation

$$p(x) = c_1 + c_2 x + c_3 x^2 + c_4 x^3 = 0 \tag{1}$$

is one equation in one unknown, and the root finding methods we developed previously can be applied to solve it. However this is a *polynomial* equation, and there are theoretical results that can be used to develop specialized root-finding methods that are more powerful than general-purpose methods.

For polynomials of order $n = 1,2,3,4$ there are analytic formulas for all roots. The single root of $c_1 + c_2 x = 0$ is $x = -c_1/c_2$. The quadratic formula gives the two roots of

$$c_1 + c_2 x + c_3 x^2 = 0$$

as

$$x = \frac{-c_2 \pm \sqrt{c_2^2 - 4 c_1 c_3}}{2 c_3}$$

The formulas for order 3 and 4 polynomials are too complicated to be of practical use. Therefore to find the roots of order 3 and higher polynomials we are forced to use numerical methods. Yet it is still the case that we have important theoretical results to guide us.

The fundamental theorem of algebra states that a polynomial of degree *n* has precisely *n* roots (some of which may be repeated). However, these roots may be real or complex. Most of the root finding algorithms we have studied so far apply only to a real function of a real variable. They can find the real roots of a polynomial (if there are any) but not complex roots.

If the polynomial has real coefficients $c_1, c_2, \ldots$ then complex roots (if any) come in complex-conjugate pairs. Let $z = x + i\,y$ be a general complex number with real part *x* and imaginary part *y*. If

$$c_1 + c_2 z + c_3 z^2 + \cdots + c_{n+1} z^n = 0$$

then taking the complex conjugate of both sides we have

$$c_1^* + c_2^* z^* + c_3^* (z^2)^* + \cdots + c_{n+1}^* (z^n)^* = 0 \tag{2}$$

where $z^* = x - i\,y$. Suppose the coefficients are real so that $c_k^* = c_k$. Since $(z^k)^* = (z^*)^k$ (2) becomes

$$c_1 + c_2 (z^*) + c_3 (z^*)^2 + \cdots + c_{n+1} (z^*)^n = 0$$

which tells us that if *z* is a root of the polynomial then so is z^*. Therefore complex roots must come in complex-conjugate pairs. From this we know that a polynomial of odd order has at least one real root since we must always have an even number of complex roots.

Another way to see this is in terms of root bracketing. For very large (real) values of x, an n^{th} order polynomial is dominated by its highest power term

$$c_1 + c_2 x + c_3 x^2 + \cdots + c_{n+1} x^n \sim c_{n+1} x^n$$

For n odd, x^n is positive for positive x and negative for negative x. Therefore the polynomial must change sign between $x \to -\infty$ and $x \to \infty$. Since polynomials are continuous functions we conclude that there must be a real root somewhere on the x axis.

For a complex root

$$p(z) = c_1 + c_2 (x + i\,y) + c_3 (x + i\,y)^2 + \cdots + c_{n+1} (x + i\,y)^n$$

Expanding each term into real and imaginary parts, along the lines of

$$(x + i\,y)^2 = x^2 - y^2 + i\,2\,x\,y$$
$$(x + i\,y)^3 = x^3 - 3\,x\,y^2 - i\,y\,(y^2 - 3\,x^2)$$

we end up with an equation of the form

$$p(z) = f(x, y) + i\,g(x, y) = 0$$

which is actually two real equations in two real unknowns

$$\begin{pmatrix} f(x, y) \\ g(x, y) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

and so not directly solvable using our $f(x) = 0$ algorithms.

2 Manipulating polynomials in Scilab

When we write out a polynomial such as (1) the variable x is simply a placeholder into which some number is to be inserted, so the polynomial is fully specified by simply listing its coefficients

$$[c_1, c_2, \ldots, c_{n+1}]$$

<u>*Here is a major difference between Scilab and Matlab*</u>. Scilab follows the convention that an array of polynomial coefficients begins with the constant term on the left and ends with the coefficient of x^n on the right. Matlab does the bookkeeping in the opposite direction. In Scilab the coefficients

```
c = [-15,23,-9,1]
```

correspond to the polynomial $-15 + 23\,x - 9\,x^2 + x^3$. In Matlab the same coefficients correspond to the polynomial $-15\,x^3 + 23\,x^2 - 9\,x + 1$. The Matlab convention more closely corresponds to the way we normally write polynomials (starting with the highest power). However, there are some advantages to the Scilab convention. Regardless, I want to make it clear that we will be following the Scilab convention, and most of the material in this section is particular to Scilab (although there are counterparts in Matlab).

To explicitly form a polynomial in the variable x with coefficients `c=[1,2,3]` we use the command

```
-->p = poly([1,2,3],'x','coeff')
 p  =
                2
    1 + 2x + 3x
```

Notice that Scilab outputs an ascii typeset polynomial in the variable of interest. To form a polynomial with roots at $x=1$, $x=2$ we use the command

```
-->q = poly([1,2],'x','roots')
 q  =
              2
    2 - 3x + x
```

Note that

$$(x-1)(x-2)=x^2-3x+2=2-3x+x^2$$

You can multiply and divide polynomials

```
-->p*q
 ans  =
             2    3    4
    2 + x + x - 7x + 3x

-->p/q
 ans  =
                2
    1 + 2x + 3x
    -----------
              2
    2 - 3x + x
```

In the second case we obtain a *rational function* of *x*. Now consider the somewhat redundant appearing command

```
-->x = poly(0,'x')
 x  =
    x
```

This assigns to the Scilab variable `x` a polynomial in the symbolic variable `x` having a single root at $x=0$, that is, it effectively turns `x` into a symbolic variable. Now we can enter expressions such as

```
-->h = 3*x^3-4*x^2+7*x-15
 h  =
                 2    3
  - 15 + 7x - 4x + 3x

-->g = (x-1)*(x-2)*(x-3)
 g  =
                 2   3
  - 6 + 11x - 6x + x
```

To evaluate a polynomial (or rational function) at a specific number we use the `horner` command

```
-->horner(h,3)
 ans  =
```

```
    51.


-->horner(h,1-%i)
 ans  =
  - 14. - 5.i
```

We can evaluate a polynomial (or rational function) at an array of numbers

```
-->v = [1,2,3]
 v  =
    1.    2.    3.

-->horner(h,v)
 ans  =
  - 9.    7.    51.
```

As always, we are only scratching the surface. See the Polynomials section of the Scilab Help Browser for more information.

3 Factoring and deflation

Another thing we know about polynomials is that they can always (in principle) be factored

$$c_1 + c_2 z + c_3 z^2 + \cdots + c_{n+1} z^n = c_{n+1}(z - r_1)(z - r_2)\cdots(z - r_n)$$

where $r_1, r_2, \ldots, r_n$ are the (in general complex) roots of the polynomial. If we find one root, r_1 say, we can divide out a factor of $(z - r_1)$ to get a polynomial of degree $n - 1$

$$\frac{c_1 + c_2 z + c_3 z^2 + \cdots + c_{n+1} z^n}{z - r_1} = b_1 + b_2 z + \cdots + b_n z^{n-1}$$

This is the process of *deflation*. If you form the ratio of two polynomials with a common factor, Scilab will cancel the common factor. Consider

```
-->p = (x-1)*(x-2)
 p  =
                2
    2 - 3x + x

-->q = (x-1)*(x-3)
 q  =
                2
    3 - 4x + x

-->p/q
 ans  =

  - 2 + x
    -----
  - 3 + x
```

Notice that the common factor of $x - 1$ has been canceled. This can be used for deflation

```
-->p/(x-2)
 ans  =
```

```
  - 1 + x
    -----
      1
```

We need to consider the effect of finite precision. Suppose we've calculated a polynomial root *r* numerically. We don't expect it to be exact. Will Scilab be able to factor it out of the polynomial? Look at the following

```
-->p = (x-sqrt(2))*(x-%pi)
 p  =
                                2
    4.4428829 - 4.5558062x + x

-->p/(x-%pi*(1+1e-6))
 ans  =
                                2
    4.4428829 - 4.5558062x + x
    -------------------------
        - 3.1415958 + x

-->p/(x-%pi*(1+1e-9))
 ans  =

  - 1.4142136 + x
    -------------
          1
```

The polynomial has a factor of $(x-\pi)$. In the first ratio we are trying to cancel a factor of $(x-\pi[1+10^{-6}])$. Now $\pi[1+10^{-6}]$ is very close to π, but not close enough for Scilab to consider them the same numbers, so no deflation occurs. On the other hand, in the second ratio Scilab treats $(x-\pi[1+10^{-9}])$ as numerically equivalent to $(x-\pi)$ and deflates the polynomial by that factor. The lesson is that a root estimate must be very accurate for it to be successfully factored out of a polynomial.

4 Horner's method

Let's turn to the numerical mechanics of evaluating a polynomial. To compute

$$f(x)=c_1+c_2x+c_3x^2+\cdots+c_{n+1}x^n$$

for some value of *x*, it is generally not a good idea to directly evaluate the terms as written. Instead, consider the following factorization of a quadratic

$$c_1+c_2x+c_3x^2=c_1+(c_2+c_3x)x$$

a cubic

$$c_1+c_2x+c_3x^2+c_4x^3=c_1+[c_2+(c_3+c_4x)x]x$$

and a quartic

$$c_1+c_2x+c_3x^2+c_4x^3+c_5x^4=c_1+(c_2+[c_3+(c_4+c_5x)x]x)x$$

Notice that in all cases the expressions on the right involve only two simple operations: multiply by *x* or add a coefficient. There is no need to evaluate all the various powers $x, x^2, x^3, x^4, \ldots$. This is particularly important for low-level computational systems such as microcontrollers,

which typically do not have the hardware floating-point accelerator that a high-end cpu would.

Evaluating a polynomial using this factored form is called *Horner's method*. We can code it in a few lines. See the function `polyHorner` in the Appendix. As we've already seen, Scilab contains a built-in `horner` function.

Now consider polynomial deflation. Suppose we have a polynomial

$$p(x)=c_1+c_2x+c_3x^2+c_4x^3$$

and we know that *r* is a root. We want to factor $(x-r)$ from $p(x)$ to get a polynomial

$$q(x)=b_1+b_2x+b_3x^2$$

We must have

$$\begin{aligned} c_1+c_2x+c_3x^2+c_4x^3 &= (x-r)(b_1+b_2x+b_3x^2) \\ &= -rb_1-rb_2x-rb_3x^2 \\ &\quad +b_1x+b_2x^2+b_3x^3 \end{aligned}$$

Equating coefficients of like powers of *x* we have

$$\begin{aligned} c_4&=b_3 \\ c_3&=-rb_3+b_2 \\ c_2&=-rb_2+b_1 \\ c_1&=-rb_1 \end{aligned}$$

We can rearrange this to get

$$\begin{aligned} b_3&=c_4 \\ b_2&=c_3+rb_3 \\ b_1&=c_2+rb_2 \\ 0&=c_1+rb_1 \end{aligned}$$

Generalizing to an arbitrary order polynomial we have

$$\begin{aligned} b_n&=c_{n+1} \\ b_k&=c_{k+1}+rb_{k+1} \text{ for } k=n-1,n-2,\ldots,1 \end{aligned} \tag{3}$$

Additionally, the equation $c_1+rb_1=0$ must automatically be satisfied if *r* is a root. This algorithm appears in the Appendix as `polyDeflate`.

5 Finding the roots of a polynomial

We are now in a position to write a function that solves for all *n* roots of an n^{th} order polynomial. The algorithm is simply

Polynomial root-finding algorithm

```
for i=1 to n
  find a root r of p(z)
  remove a factor of (z-r) from p(z)
```

Newton's method actually works quite well at finding the complex zeros of a polynomial, and we can use the `rootNewton` function we developed previously without modification. For the purposes of calculating the derivative of a polynomial, note that if

$$p(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_{n+1} x^n$$

then

$$\begin{aligned} p'(x) &= c_2 + 2c_3 x + \cdots + n c_{n+1} x^{n-1} \\ &= b_1 + b_2 x + \cdots + b_n x^{n-1} \end{aligned}$$

and for $1 \le k \le n$

$$b_k = k c_{k+1}$$

In Scilab we can calculate these coefficients of the derivative as

```
b = c(2:n+1).*(1:n);
```

Now, one subtle point. If the coefficients c_k are all real, then so are the coefficients b_k. It follows that if x is real then $x - f(x)/f'(x)$ is real also. Therefore, if Newton's method starts on the real axis, it can never leave the real axis. For that reason we need to start with a complex value of x_0.

The function `polyRoots` shown in the Appendix is our implementation. We use `polyHorner` to evaluate the polynomials. Iteratively we use `rootNewton` to find a (any) root. Then we use `polyDeflate` to remove that root's factor and reduce the order of the polynomial.

This simple code actually works pretty well. Run on several hundred randomly generated 7th order polynomials it only failed about one percent of the time. However, those failures demonstrate why numerical analysis is an active field of research. In any type of problem there are almost always some "hard" cases which thwart a given algorithm. This motivates people to develop more advanced methods. For polynomial root finding some of the more advanced methods are Laguerre's method, Bairstow's method, the Durand–Kerner method, the Jenkins–Traub algorithm and the companion-matrix method. In its built-in root finding function Matlab uses the companion-matrix method while in Scilab you can use either the companion-matrix method (default) or the Jenkins–Traub algorithm.

5.1 Polishing the roots

Numerical root finding and polynomial deflation will be subject to round-off and finite tolerance errors. The more we deflate a polynomial the more error can be introduced into the coefficients. So, especially for a large-order polynomial, we should be suspicious of the roots we find with the `polyRoots` function. It is a very good idea to "polish" the roots using Newton's method on the original polynomial before performing deflation. This process should be very rapid since we are (presumably) very close to a true root. The function `polyRootsPolished` in the Appendix adds root polishing to `polyRoots`.

6 The roots function

In both Scilab and Matlab there is a built-in `roots` function for finding all roots of a

polynomial. It's as simple as

```
r = roots(p);
```

where `p` is a polynomial and `r` is an array containing all roots of `p`. For example, in Scilab

```
-->p = poly([1,2,3,4,5,6],'x','coeff')
 p  =
                2    3    4    5
    1 + 2x + 3x + 4x + 5x + 6x

-->r = roots(p)
 r  =

    0.2941946 + 0.6683671i
    0.2941946 - 0.6683671i
  - 0.6703320
  - 0.3756952 + 0.5701752i
  - 0.3756952 - 0.5701752i
```

Once you know the roots you can, if you wish, write the polynomial in factored form. If the polynomial has real coefficients then complex roots come in conjugate pairs. A product of the form $(z-r)(z-r^*)$ always reduces to a quadratic with real coefficients z^2+bz+c. To avoid factors with complex roots Scilab has the `polfact` command

```
-->polfact(p)
ans  =
                                  2                                  2
6  0.5332650 - 0.5883891x + x   0.4662466 + 0.7513904x + x   0.6703320 + x
```

This tells us that our polynomial can be factored (approximately) as

$$1+2x+3x^2+4x^3+5x^4+6x^5=6(0.533-0.588x+x^2)(0.466+0.751x+x^2)(0.670+x)$$

Some other useful polynomial/rational functions in Scilab are

```
derivat, factors, numer, denom, simp_mode, clean, order
```

As always, see the help browser for more details.

7 References

1. http://www.ece.rice.edu/dsp/software/FVHDP/horner2.pdf (retrieved 2014-06-04)

8 Appendix – Fundamental theorem of algebra

$$p(z)=c_1+c_2z+c_3z^2+\cdots+c_nz^{n-1}+z^n$$

$z=re^{j\theta}$

$z^n=r^ne^{jn\theta}$

$$p(0) = c_1$$

9 Appendix – Scilab code

9.1 Horner's method

```
0001  ////////////////////////////////////////////////////////////////////
0002  // polyHorner.sci
0003  // 2014-06-04, Scott Hudson
0004  // Horner's method for polynomial evaluation. c=[c(1),c(2),...,c(n+1)]
0005  // are coefficients of polynomial
0006  // p(z) = c(1)+c(2)*z+c(3)*z^2+...+c(n+1)*z^n
0007  // z is the number (can be complex) at which to evaluate polynomial
0008  ////////////////////////////////////////////////////////////////////
0009  function w=polyHorner(c, z)
0010    n = length(c)-1;
0011    w = c(n)+c(n+1)*z;
0012    for i=n-1:-1:1
0013      w = c(i)+w*z;
0014    end
0015  endfunction
```

9.2 Polynomial deflation

```
0001  ////////////////////////////////////////////////////////////////////
0002  // polyDeflate.sci
0003  // 2014-06-04, Scott Hudson
0004  // Given the coefficients c = [c(1),c(2),...,c(n+1)] of polynomial
0005  // p(z) = c(1)+c(2)*z+c(3)*z^2+...+c(n+1)*z^n
0006  // and root r, p(r)=0, remove a factor of (z-r) from p(z) resulting in
0007  // q(z) = b(1)+b(2)*z+...+b(n)*z^(n-1) of order one less than p(z)
0008  // Return array of coefficients b = [b(1),b(2),...,b(n)]
0009  ////////////////////////////////////////////////////////////////////
0010  function b=polyDeflate(c, r)
0011    n = length(c)-1;
0012    b = zeros(1,n);
0013    b(n) = c(n+1);
0014    for k=n-1:-1:1
0015      b(k) = c(k+1)+r*b(k+1);
0016    end
0017  endfunction
```

9.3 Polynomial root solver

```
0001  ////////////////////////////////////////////////////////////////////
0002  // polyRoots.sci
0003  // 2014-06-04, Scott Hudson, for pedagocic purposes only!
0004  // Given an array of coefficients c=[c(1),c(2),...,c(n+1)]
0005  // defining a polynomial p(z) = c(1)+c(2)*z+...+c(n+1)*z^n
0006  // find the n roots using Newton's method (with complex arguments)
0007  // followed by polynomial deflation. The derivative polynomial is
0008  // b(1)+b(2)*z+b(3)*z^2+...+b(n)*z^(n-1) =
0009  // c(2)+2*c(3)*z+3*c(4)*z^2+...+n*c(n+1)*z^(n-1)
0010  ////////////////////////////////////////////////////////////////////
0011  function r=polyRoots(c)
0012    n = length(c)-1; //order of polynomial
0013    b = zeros(1,n); //coefficients of polynomial derivative
0014    b = c(2:n+1).*(1:n); //b(k) = c(k+1)*k
0015    deff('y=f(z)','y=polyHorner(c,z)'); //f(x) for Newton method
0016    deff('y=fp(z)','y=polyHorner(b,z)'); //fp(x) for same
0017    r = zeros(n,1);
0018    z0 = 1+%i; //initial search point, should not be real
0019    for i=1:n-1
0020      r(i) = rootNewton(z0,f,fp,1e-8);
0021      c = polyDeflate(c,r(i));
0022      m = length(c)-1; //order of deflated polynomial
0023      b = c(2:m+1).*(1:m); //b(k) = c(k+1)*k
0024    end
0025    r(n) = -c(1)/c(2); //last root is solution of c(1)+c(2)*z=0
0026  endfunction
```

9.4 Polynomial root solver with root polishing

```
////////////////////////////////////////////////////////////////////
// polyRootsPolished.sci
// 2014-06-04, Scott Hudson, for pedagocic purposes
// Given an array of coefficients c=[c(1),c(2),...,c(n+1)]
// defining a polynomial p(z) = c(1)+c(2)*z+...+c(n+1)*z^n
// find the n roots using Newton's method (with complex arguments)
// followed by polynomial deflation. The derivative polynomial is
// b(1)+b(2)*z+b(3)*z^2+...+b(n)*z^(n-1) =
// c(2)+2*c(3)*z+3*c(4)*z^2+...+n*c(n+1)*z^(n-1)
// Each root is polished before deflation
////////////////////////////////////////////////////////////////////
function r=polyRootsPolished(c)
  n = length(c)-1; //order of polynomial
  b = zeros(1,n);  //coefficients of polynomial derivative
  b = c(2:n+1).*(1:n); //b(k) = c(k+1)*k
  deff('y=f(z)','y=polyHorner(c,z)'); //f(x) for Newton method
  deff('y=fp(z)','y=polyHorner(b,z)');//fp(x) for same
  r = zeros(n,1);
  z0 = 1+%i; //initial search point, should not be real
  c0 = c;    //save original coefficients for polishing
  b0 = b;
  deff('y=f0(z)','y=polyHorner(c0,z)'); //f(x) or orig. poly
  deff('y=fp0(z)','y=polyHorner(b0,z)');//fp(x) for same
  for i=1:n-1
    r(i) = rootNewton(z0,f,fp,1e-4);
    r(i) = rootNewton(r(i),f0,fp0,1e-8); //polish root using original poly
    c = polyDeflate(c,r(i));
    m = length(c)-1; //order of deflated polynomial
    b = c(2:m+1).*(1:m); //b(k) = c(k+1)*k
  end
  r(n) = -c(1)/c(2); //last root is solution of c(1)+c(2)*z=0
  r(n) = rootNewton(r(n),f0,fp0,1e-8); //polish root using original poly
endfunction
```

10 Appendix – Matlab code

10.1 Horner's method

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% polyHorner.sci
%% 2014-06-04, Scott Hudson
%% Horner's method for polynomial evaluation. c=[c(1),c(2),...,c(n+1)]
%% are coefficients of polynomial
%% p(z) = c(1)+c(2)*z+c(3)*z^2+...+c(n+1)*z^n
%% z is the number (can be complex) at which to evaluate polynomial
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function w = polyHorner(c,z)
  n = length(c)-1;
  w = c(n)+c(n+1)*z;
  for i=n-1:-1:1
    w = c(i)+w*z;
  end
end
```

10.2 Polynomial deflation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% polyDeflate.sci
%% 2014-06-04, Scott Hudson
%% Given the coefficients c = [c(1),c(2),...,c(n+1)] of polynomial
%% p(z) = c(1)+c(2)*z+c(3)*z^2+...+c(n+1)*z^n
%% and root r, p(r)=0, remove a factor of (z-r) from p(z) resulting in
%% q(z) = b(1)+b(2)*z+...+b(n)*z^(n-1) of order one less than p(z)
%% Return array of coefficients b = [b(1),b(2),...,b(n)]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function b = polyDeflate(c,r)
  n = length(c)-1;
  b = zeros(1,n);
  b(n) = c(n+1);
  for k=n-1:-1:1
    b(k) = c(k+1)+r*b(k+1);
  end
end
```

10.3 Polynomial root solver

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% polyRoots.sci
%% 2014-06-04, Scott Hudson, for pedagocic purposes
%% Given an array of coefficients c=[c(1),c(2),...,c(n+1)]
%% defining a polynomial p(z) = c(1)+c(2)*z+...+c(n+1)*z^n
%% find the n roots using Newton's method (with complex arguments)
%% followed by polynomial deflation. The derivative polynomial is
%% b(1)+b(2)*z+b(3)*z^2+...+b(n)*z^(n-1) =
%% c(2)+2*c(3)*z+3*c(4)*z^2+...+n*c(n+1)*z^(n-1)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function r = polyRoots(c)
  n = length(c)-1; %%order of polynomial
  b = zeros(1,n); %%coefficients of polynomial derivative
  b = c(2:n+1).*(1:n); %%b(k) = c(k+1)*k
    function y = f(z)
      y = polyHorner(c,z);
    end
    function y = fp(z)
      y = polyHorner(b,z);
    end
  r = zeros(n,1);
  z0 = 1+1i; %%initial search point, should not be real
  for i=1:n-1
    r(i) = rootNewton(z0,@f,@fp,1e-8);
    c = polyDeflate(c,r(i));
    m = length(c)-1; %%order of deflated polynomial
    b = c(2:m+1).*(1:m); %%b(k) = c(k+1)*k
  end
  r(n) = -c(1)/c(2); %%last root is solution of c(1)+c(2)*z=0
end
```

10.4 Polynomial root solver with root polishing

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% polyRootsPolished.sci
%% 2014-06-04, Scott Hudson, for pedagocic purposes
%% Given an array of coefficients c=[c(1),c(2),...,c(n+1)]
%% defining a polynomial p(z) = c(1)+c(2)*z+...+c(n+1)*z^n
%% find the n roots using Newton's method (with complex arguments)
%% followed by polynomial deflation. The derivative polynomial is
%% b(1)+b(2)*z+b(3)*z^2+...+b(n)*z^(n-1) =
%% c(2)+2*c(3)*z+3*c(4)*z^2+...+n*c(n+1)*z^(n-1)
%% Each root is polished before deflation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function r = polyRootsPolished(c)
  n = length(c)-1; %%order of polynomial
  b = zeros(1,n);  %%coefficients of polynomial derivative
  b = c(2:n+1).*(1:n); %%b(k) = c(k+1)*k
  function y = f(z)
    y = polyHorner(c,z);
  end
  function y = fp(z)
    y = polyHorner(b,z);
  end
  r = zeros(n,1);
  z0 = 1+1i; %%initial search point, should not be real
  c0 = c;    %%save original coefficients for polishing
  b0 = b;
  function y = f0(z)
    y = polyHorner(c0,z);
  end
  function y = fp0(z)
    y = polyHorner(b0,z);
  end
  for i=1:n-1
    r(i) = rootNewton(z0,@f,@fp,1e-4);
    r(i) = rootNewton(r(i),@f0,@fp0,1e-8); %%polish root using original poly
    c = polyDeflate(c,r(i));
    m = length(c)-1; %%order of deflated polynomial
    b = c(2:m+1).*(1:m); %%b(k) = c(k+1)*k
  end
  r(n) = -c(1)/c(2); %%last root is solution of c(1)+c(2)*z=0
  r(n) = rootNewton(r(n),@f0,@fp0,1e-8); %%polish root using original poly
end
```