ECE 204 Project 3

Douglas Harder

February 8, 2024

1 Introduction

In this topic, you find all the real roots of a polynomial that has real coefficients and where the roots of that polynomial are distinct.

The need for restricting that the roots to those that are real and distinct include:

1. Finding complex roots requires the use of the `complex` library together with the type `std::complex<double>`.
2. There are convergence issues when there are roots with multiplicity greater than one, which reduces the rate of convergence of Newton's method from $O(h^2)$ to $O(h)$.

Guarantee: All polynomials you will be tested on have distinct real roots. You do not have to try to detect if a root is a double root.

1.1 Example 1

If you look at the polynomial $2x^5 - 2x^4 - 8x^3 - 10x^2 - 18x + 36$, with a little effort, you will realize that this polynomial has three distinct real roots: $-2, 1, 3$, and the other two roots form a complex conjugate pair. You can test this in Matlab:

```
>> roots( [2 -2 -8 -10 -18 36] )
    ans =
       3.00000 + 0.00000i
      -2.00000 + 0.00000i
      -0.50000 + 1.65831i
      -0.50000 - 1.65831i
       1.00000 + 0.00000i
```

Thus, this polynomial can be written as

$$(x-3)(x-1)(x+2)(2x^2+2x+6).$$

We want to find the roots $3, 1, -2$ and the remaining polynomial $2x^2 + 2x + 6$.

1.2 Example 2

The polynomial

$$1.3x^4 + 7.67x^3 - 22.997x^2 - 112.7763x + 152.4978$$

has four real roots, and thus, can be written as

$$1.3(x-1.2)(x+5.7)(x-3.5)(x+4.9)$$

where the roots are $1.2, 3.5, -4.9, -5.7$, and thus we want to find the four roots and the remaining constant polynomial 1.3.

Important: This may appear to be a difficult problem to solve, but we will walk you through the steps. Your primary focus will be working on one top-level function `roots(...)` and three helper functions that implement Newton's method, Horner's rule and polynomial division.

2 Coding requirements

You can fork the skeletons available at

```
https://replit.com/@dwharder/ECE-204-Project-3
```

You will write the following function, together with a few helper functions, in C++:

```
unsigned int roots( double coeffs[],
                    unsigned int degree );
```

The argument will be a polynomial represented by an array of coefficients, where `coeffs[k]` is the coefficient of x^k. The parameter `degree` is the degree of the polynomial, so it will be one less than the capacity of the array. The returned integer d is the degree of the remaining polynomial (that is, the original degree minus the number of real roots that were found). The first $d+1$ entries will represent the quotient polynomial (a polynomial with only complex roots, that is, the original polynomial divided by $x-r$ for each real root r), while all entries beyond index d will be the real roots that were found, sorted.

Important: Representations differ, so in this representation, the constant coefficient comes first, and as we step through the array, the degree of the term increases. This differs from Matlab where the highest term comes first. Thus, in Matlab, the polynomial $2x^5 - 2x^4 - 8x^3 - 10x^2 - 18x + 36$ is represented as

```
p = [2 -2 -8 -10 -18 36];
```

but in our C++ representation, it will be represented by:

```
double p[6]{ 36, -18, -10, -8, -2, 2 };
```

For example, to see how this function would work, you will note that the array

```
double my_poly[4]{4.3, 10.7, 7.0, 1.3};
```

represents the polynomial $4.3 + 10.7x + 7.0x^2 + 1.3x^3$, and thus has degree 3. To find the roots of this polynomial, we would call `roots( my_poly, 3 )`. When this function returns, the entries of the array would be

```
{1.3, -3.025505509096196, -1.725522469184636, -0.6335874063345515}
```

where the first entry is just the coefficient of x^3 and the remaining three entries are the real roots sorted. The function would return `0` because three roots were found, so the quotient polynomial is simply the constant polynomial `1.3`.

Philosophical discussion (skip if you want): In ECE 150, we always used `std::size_t` to be the type associated with an array index, so should `degree` not be of type `std::size_t`? You could, but mathematically, the degree of a polynomial is an integer, and a non-negative integer, so `unsigned int` is more natural. Also, not all polynomial representations use arrays. We could use a linked list instead of an array, where each entry of the linked list is a term in the polynomial. Thus, it makes more sense to use an integer data type.

For the purposes of this project, the minimum polynomial degree will be `0`, and the zero polynomial will be an array of capacity one with the one entry `0.0`. Other than this special case when the degree is zero, if the degree of the polynomial is `degree > 0`, then `array[degree]` may not be zero. Your code should include an assertion or throw an `std::invalid_argument` exception if the coefficient of the leading term is zero.

The output will replace the entries of the array as follows:

1. The coefficients of the polynomial that has no more real roots will occupy the entries from index `0` to `d`, representing a polynomial of degree d. If the original polynomial has only real roots, then the first entry of the array will be the coefficient of x^n where n is the degree of the original polynomial.
2. Indices `d + 1` through `degree` will store the roots of the polynomial sorted from smallest to largest.

For example, given the above polynomial, $4.3+10.7x+7.0x^2+1.3x^3$, there are three real roots, so the entries of the array would end up being

```
1.3, -3.025505509096197, -1.725522469184636, -0.6335874063345515
```

and the value `0` is returned. The first entry (index `0`) is the coefficient of x^{degree} and the remaining three entries are the three roots sorted. You can test your answer by finding the roots in Matlab:

```
format long
roots( [1.3 7.0 10.7 4.3] )
   ans =
      -3.025505509096191
      -1.725522469184640
      -0.6335874063345515
```

Important: Matlab stores the coefficients from the highest degree and places the constant coefficient last. The representation we will use has the constant coefficient first. Thus, you will have to manually reverse the order of the coefficients if you are checking your work in Matlab.

The roots shown above are correct to all sixteen digits, and as you can see, the Octave routine has errors in some cases in the least significant digits. You will only have to get six digits of accuracy in your solutions (your solution rounded to six decimal digits should equal the given solution rounded to six decimal digits). You may assume that Matlab has at most an error in the last two digits.
Remember that to have Matlab display sixteen significant digits, you must use `format long`. In C++, to have double-precision floating-point numbers displayed with sixteen significant digits, you must use `std::cout.precision( 16 );`.

2.1 The algorithm

Let N be the degree of the polynomial of the original polynomial. You will proceed as follows.

1. Assign $n \leftarrow N$.
2. You will pass the array storing the polynomial together with the degree n to a function implementing Newton's method on polynomials:
   (a) If Newton's method finds a root r, then you will first call the polynomial division algorithm to divide the polynomial by $x - r$. You will then store the root r at index n. Finally, you will decrement the degree n by one.
   (b) Otherwise, Newton's method did not find a root r and thus you must assume that all roots are complex (although, make sure your Newton's method works!) You will sort the roots stored at indices $n + 1$ up to N, and you will return n.

This program must terminate because either Newton's method finds a root or it doesn't. If it does, then after polynomial division, the degree n is decremented by 1, so this loop will run at most N times.

3 Polynomials and their derivatives

Newton's method uses $x_{\text{new}} \leftarrow x_{\text{old}} - \frac{f(x_{\text{old}})}{f^{(1)}(x_{\text{old}})}$. You will use Horner's rule to evaluate both the polynomial and its derivative at the value x_{old}. Recall that if a is our array, then the value of the polynomial at x is

$$\sum_{k=0}^{n} a_k x^k$$

and the value of the derivative at x is

$$\sum_{k=1}^{n} k a_k x^{k-1}$$

As you saw in class, Horner's rule can be implemented as

```
double result{ coeffs[degree] };

for ( unsigned int k{ degree - 1 }; k < degree; --k ) {
    result = result*x + coeffs[k];
}
```

To calculate the derivative, you will have to modify this slightly. As a suggestion, author two functions:

```
double  horner( double x, double coeffs[], unsigned int degree );
double dhorner( double x, double coeffs[], unsigned int degree );
```

The first should evaluate the polynomial at x, and the second should evaluate the derivative of the polynomial at x. For example, if the polynomial is $0.3 + 0.8x - 0.9x^2 + 0.2x^3$, then

```
// These two arrays are declared inline
std::cout <<  horner( 0.5, (double[]){ 0.3, 0.8, -0.9, 0.2 }, 3 )
          << std::endl;
std::cout << dhorner( 0.5, (double[]){ 0.3, 0.8, -0.9, 0.2 }, 3 )
          << std::endl;
```

These should print out `0.5` and `0.05`, respectively.

4 Newton's method on polynomials

In class, we saw a Newton's method algorithm for a general function `f` and we had to pass the derivative as a a separate argument. It is suggested you implement a function of the form

```
double newton( double poly[], unsigned int degree );
```

You can hard code the constraints for ending the iteration, as this is an internal helper function.

1. First, if the degree is zero, then just signal that there is no root (see the next subsection).
2. Your function should assert that the entry `poly[degree] != 0.0`, for if that entry was zero, then the degree of the polynomial is not equal to the degree given by the second argument.
3. Next, apply Newton's method and you can pick any initial x_0 value you like; say, 201.23456 where 20123456 is your uWaterloo Student ID Number, but you can pick whatever value you want. You will run a maximum number of iterations, so to start, choose 100, although, as you try the problems posed in the questions, you may have to adjust this. Then, with each iteration, given `x0`, you will calculate the next iteration `x1`. You can then determine if `x1` is a good-enough approximation of the root based on the discussion in class. If you find a root, you will return a root. If you do not, you will have to, in some way, signal that a root was not found. See the next sub-section.

4.1 Failure to find a root

There are two ways to signal a failure to find a root: either by returning `NAN` or by throwing an exception.

If you return `NAN`, defined in the `cmath` library, then you can check if `newton` found a root by checking the value returned:

```
double root{ newton( ... ) };

if ( std::isnan( root ) ) {
    // No root was found: if your code is correct,
    // we just sort the entries in the array
    // storing the real roots, and then return
    // the degree of the remaining polynomial.
} else {
    // Divide the polynomial by 'x - root' (using
    // polynomial division), store the new root
    // in the index corresponding to the 'degree'
    // before we divided out 'x - root', and decrement
    // the degree of the remaining polynomial.
}
```

Important: Because not-a-number represents an indeterminate calculation, the authors of IEEE 754 specified that `NAN == NAN` always returns `false`. Thus, you must use `std::isnan(...)` to determine if the returned value was `NAN`.

Alternatively, you can include the `stdexcept` library and then throw a `std::runtime_error` error from within the `newton(...)` function; for example

```
throw std::runtime_error{ "No root was found." };
```

For example, you could try

```
try {
    double root{ newton( ... ) };

    // A root was found...
    // Divide the polynomial by 'x - root' (using
    // polynomial division), store the new root
    // in the index corresponding to the 'degree'
    // before we divided out 'x - root', and decrement
    // the degree of the remaining polynomial.
} catch ( std::runtime_error &e ) {
    // No root was found: if your code is correct,
    // we just sort the entries in the array
    // storing the real roots, and then return
    // the degree of the remaining polynomial.
}
```

5 Polynomial division

To divide the polynomial stored in an array by `x - root` where `root` is the root that was found by your implementation of Newton's method, you need to do polynomial division. In secondary school, you learned a polynomial division algorithm; however, if you are simply dividing by $x - r$, where r is the root, then life becomes a lot easier. Below is the algorithm that divides the polynomial `poly[]` by `x - r`, and you must assume that once this function returns, the entries of the array from index `0` to `degree - 1` contain the quotient (the result of dividing the polynomial by `x - r`), and you now have a blank location at index `degree`. It is into this blank location you will copy the most the new root you found.

```
double divide( double r, double poly[], unsigned int degree ) {
    double s{ poly[degree] };

    for ( unsigned int k{ degree }; k > 0; --k ) {
        double t{ poly[k - 1] + r*s };
        poly[k - 1] = s;
        s = t;
    }

    // The remainder is returned
    return s;
}
```

You may recall that the remainder is equal to the original polynomial evaluated at r. Thus, if r is a root, the remainder should be close to zero.

Here is an example of how polynomial division works:

```
int main() {
    // This represents the polynomial
    //   5    4     3     2
    //  x  - x  - 4x  - 5x  - 9x + 18,
    // which has a real root at x = 3.
    double my_poly[6]{ 18.0, -9.0, -5.0, -4.0, -1.0, 1.0 };

    // The remainder should be zero, or very small
    std::cout << divide( 3.0, my_poly, 5 ) << std::endl;

    // For the above polynomial, the output is
    //     -6  1  2  2  1
    // representing the polynomial
    //   4     3     2
    //  x  + 2x  + 2x  + x - 6,
    for ( std::size_t k{ 0 }; k < 5; ++k ) {
            std::cout << my_poly[k] <<  "  ";
    }

    std::cout << std::endl;

    return 0;
}
```

Aside: If you want, try to compare how polynomial division was explained in secondary school, and then try to understand how the above (very simple, and very fast code) does the same. For example, you could try dividing the polynomial $x^5 - x^4 - 4x^3 - 5x^2 - 9x + 18$ by $x - 1$, $x - 3$ or $x + 3$, these representing the roots.

6 Sorting

You can use the `std::sort` function in the `algorithm` library within the C++ standard library. To sort the entries from index `m` to `n - 1` of an array `array`, you use the calling sequence `std::sort( array + m, array + n )`. Of course, `m <= n`, otherwise the sorting algorithm will go into an infinite loop.

For example, the following code sorts the entries of the array from index 5 to index 9 inclusive:

```
#include <iostream>
#include <algorithm>

int main();

int main() {
    double data[12]{
        0.40,0.19,0.02,0.80,0.43,0.84,0.41,1.0,0.39,0.68,0.77,0.73
    };

    std::sort( data + 5, data + 10 );

    for ( std::size_t k{ 0 }; k < 12; ++k ) {
        std::cout << data[k] << " ";
    }

    std::cout << std::endl;

    return 0;
}
```

The output is shown here, and the sorted entries from index `5` to index `9` are underlined.

```
0.4 0.19 0.02 0.8 0.43 0.39 0.41 0.68 0.84 1 0.77 0.73
                       ---------------------
```

7 Printing the result

The following function prints the result. It prints the remaining polynomial and the lists the real roots. You are encouraged to, but not required to, understand some of the steps taken in this routine which you may not be familiar with.

```
void print(
    double coeffs[],
    unsigned int complex_degree,
    unsigned int degree
) {
    // Store the current value of precision
    // and set the precision to 16 digits
    std::streamsize old_precision{ std::cout.precision( 16 ) };

    std::cout << "Remaining polynomial:" << std::endl;
    std::cout << "\t" << coeffs[0];

    // Show a "+" sign in front of all positive floating-point numbers
    std::cout << std::showpos;

    if ( complex_degree >= 1 ) {
        std::cout << coeffs[1] << "x";
    }

    for ( unsigned int k{ 2 }; k <= complex_degree; ++k ) {
        std::cout << coeffs[k] << "x^" << k;
    }

    std::cout << std::endl << "Real roots:" << std::endl;

    for ( unsigned int k{ complex_degree + 1 }; k <= degree; ++k ) {
        std::cout << "\t" << coeffs[k] << std::endl;
    }

    // Stop showing the leading "+"
    std::cout << std::noshowpos << std::endl;

    // Restore the original value of precision
    std::cout.precision( old_precision );
}
```

8 Matlab to the rescue...

You can use Matlab to check your answers by using the `roots(...)` command and the `deconv(...)` command. The de-convolution command does polynomial division, returning the quotient and the remainder.

```
% Create a random quartic
p = rand( 5, 1 )
     p =
        0.8005791898616001
        0.08654993971008877
        0.3271353390179731
        0.9076708563644759
        0.02914475992058279

% There are two real roots and two complex roots
rts = roots( p )
     rts =
        0.4278675579753484 + 1.009995359543043
        0.4278675579753484 - 1.009995359543043
       -0.9313567731141167 + 0.000000000000000i
       -0.03248749781605349 + 0.000000000000000i
% Divide the polynomial by 'x - rts(3)'
%  - Note the remainder is very small
[p, r] = deconv( p, [1 -rts(3)] )
     p =
        0.8005791898616001
       -0.6590749111817249
        0.9409692215366574
        0.03129279859439227
     r =
        0.0
        0.0
        0.0
        0.0
       -3.642919299551295e-16
% Divide the polynomial by 'x - rts(4)'
%  - Note the remainder is very small
[p, r] = deconv( p, [1 -rts(4)] )
     p =
        0.8005791898616001
       -0.6850837258639315
        0.9632258775844756
     r =
        0.0
        0.0
        0.0
        4.093947403305265e-16
```

Do not try this one first, but if you executed the following code with your program,

```
    double myp[5]{
        0.02914475992058279, 0.9076708563644759,
        0.3271353390179731, 0.08654993971008877,
        0.8005791898616001
    };
    unsigned int myd{ roots( myp, 4 ) };
    print( myp, myd, 4 );
```

then the output should be something similar to:

```
Remaining polynomial:
    0.963225877584476-0.685083725863931 6x+0.8005791898616001x^2
Real roots:
    -0.9313567731141168
    -0.0324874978160535
```

9 Sample output

To help you see if your code is working, we will provide a number of sample questions together with output. Again, you only have to have your answer correct to six decimal digits of precision.

```
// C++ Code
double q0a[1]{ -14.1 };
unsigned int c0a{ roots( q0a, 0 ) };
print( q0a, c0a, 0 );
// Expected output:
// Remaining polynomial:
//     -14.1
// Real roots:

double q1a[2]{ -5.0, 2.0 };
unsigned int c1a{ roots( q1a, 1 ) };
print( q1a, c1a, 1 );
// Expected output:
// Remaining polynomial:
//     2
// Real roots:
//     +2.5

double q2a[3]{ 5.0, 2.0, 2.0 };
unsigned int c2a{ roots( q2a, 2 ) };
print( q2a, c2a, 2 );
// Expected output (no real roots):
// Remaining polynomial:
//     5+2x+2x^2
// Real roots:

double q2b[3]{ 5.0, 2.0, -2.0 };
unsigned int c2b{ roots( q2b, 2 ) };
print( q2b, c2b, 2 );
// Expected output (two real roots):
// Remaining polynomial:
//     -2
// Real roots:
//     -1.1583123951777
//     +2.1583123951777

double q3a[4]{ -100.8, 8.4, -4.2, 4.2 };
unsigned int c3a{ roots( q3a, 3 ) };
print( q3a, c3a, 3 );
// Expected output (one real and two complex roots):
// Remaining polynomial:
//     33.6+8.4x+4.2x^2
// Real roots:
//     +3
```

In the last example, you will notice that if you expand the remaining polynomial and $x - 3$, you get the original polynomial:

$$(4.2x^2 + 8.4x + 33.6)(x - 3) = 4.2x^3 - 4.2x^2 + 8.4x - 100.8$$

10 Questions

For all questions, you only need match the roots or the coefficients of the remaining polynomial to six decimal digits of precision. You must enter your solutions as text. If you upload an image, your submission will receive a 0. Please remember that `2.9999999997235` does approximate `3.0000000000000` to six significant digits, because the former rounded to six significant digits is `3.00000`.

1. (2 points) There are no roots of a constant polynomial, so if the degree is zero, then the array should be returned unchanged, as the first entry contains the constant coefficient. What is your output for the following two function calls?

```
double p0a[1]{ 0.7 };
unsigned int d0a{ roots( p0a, 0 ) };
print( p0a, d0a, 0 );

double p0b[1]{ 0.0 };
unsigned int d0b{ roots( p0b, 0 ) };
print( p0b, d0b, 0 );
```

2. (2 points) There is one root to a linear polynomial, so if the degree is one, it should return that one root. The remaining polynomial should be the constant polynomial equal to the coefficient of the leading term. What is your output for the following two function calls?

```
double p1a[2]{ 0.7, 0.5 };
unsigned int d1a{ roots( p1a, 1 ) };
print( p1a, d1a, 1 );

double p1b[2]{ 0.7, -0.5 };
unsigned int d1b{ roots( p1b, 1 ) };
print( p1b, d1b, 1 );
```

3. (2 points) There are two roots to a quadratic polynomial, so if the degree is two, then it will either find two roots, or none. Remember that we are excluding double roots from our testing. One of these two is a polynomial with two real roots, and one is one with two complex roots. What is your output for the following two function calls?

```
double p2a[3]{ 0.3, 0.1, 0.5 };
unsigned int d2a{ roots( p2a, 2 ) };
print( p2a, d2a, 2 );

double p2b[3]{ 0.7, -0.5, -0.2 };
unsigned int d2b{ roots( p2b, 2 ) };
print( p2b, d2b, 2 );
```

4. (2 points) There are three roots to a cubic polynomial, so if the degree is three, then it will either find three roots or one. Remember that we are excluding double roots from our testing. One of these two is a polynomial with three real roots, and one is one with only one real root (the other two being complex). What is your output for the following two function calls?

```
double p3a[4]{ -0.2, 0.3, -0.9, 0.3 };
unsigned int d3a{ roots( p3a, 3 ) };
print( p3a, d3a, 3 );
```

```
double p3b[4]{ 0.1, 0.3, -0.9, 0.3 };
unsigned int d3b{ roots( p3b, 3 ) };
print( p3b, d3b, 3 );
```

5. (3 points) There are four roots to a quartic polynomial, so if the degree is four, then it will either find four, two or no real roots. Remember that we are excluding double roots from our testing. One of these three is a polynomial with four real roots, one is with two real roots, and the third does not have any real roots. What is your output for the following three function calls?

```
double p4a[5]{ -12.0, -8.4, 2.4, 0.0, 1.2 };
unsigned int d4a{ roots( p4a, 4 ) };
print( p4a, d4a, 4 );

double p4b[5]{ -14.4, -19.2, -1.2, 4.8, 1.2 };
unsigned int d4b{ roots( p4b, 4 ) };
print( p4b, d4b, 4 );

double p4c[5]{ -54.0, -37.8, -29.7, -8.1, -2.7 };
unsigned int d4c{ roots( p4c, 4 ) };
print( p4c, d4c, 4 );
```

6. (3 points) There are five roots to a quintic polynomial. What is your output for the following three function calls?

```
double p5a[6]{ -157.50, 68.25, 29.75, 12.25, -5.25, -3.5 };
unsigned int d5a{ roots( p5a, 5 ) };
print( p5a, d5a, 5 );

double p5b[6]{ -16.2, 47.25, -22.275, -19.575, 8.1, 2.7 };
unsigned int d5b{ roots( p5b, 5 ) };
print( p5b, d5b, 5 );


double p5c[6]{
  -739.3056852825, 379.9560444975, -216.18149622,
   63.3701740,    -19.2249,         4.1
};
unsigned int d5c{ roots( p5c, 5 ) };
print( p5c, d5c, 5 );
```

7. (1 point) This polynomial claims to be of degree two, but the coefficient of the leading term is 0.0. Copy the error your assertion or exception prints when this is executed.

```
double p7[3]{ 0.7, -0.5, 0.0 };
unsigned int d7{ roots( p7, 2 ) };
print( p7, d7, 2 );
```

8. (4 points) In this last question, you will enter your source code for the functions you wrote. You will be marked on documenting your code, so your comments should assume that the reader knows C++, but you want your comments to explain why the code is doing what it is.

11 Acknowledgements

Nazanin Rahmati, who noted that there were three typos and that one polynomial was not being represented correctly in ASCII form.