Polynomial Root Finding Implementation

Richard Schneider

January 3, 2021

1 Introduction

Since writing a historical analysis of Leopold Kronecker's constructive philosophy on the foundations of mathematics, where I examined his discussion of irrational number and the proof that they can be uniquely identified by integers, I have been curious as to how his algorithmic mathematics might be implemented with a computer. In this case, the proof shows that with a polynomial with no repeated roots and integer coefficients we can uniquely identify the intervals in which there are at most one real root and we can make such an interval arbitrarily small so that we can approximate the real roots. We will look at an overview of the proof, discuss the tasks that we will need to do, and implement these steps in Python.

2 Overview of Root Estimation

Kronecker's proof finds intervals of a maximum defined size that have the property that they contain at most one real root of a polynomial. This is very useful in the abstract discussion of this constructed number or other mathematical proofs, but that also generates a lot of intervals to check with a computer; whereas being able to restructure our search so that a divide-and-conquer method can be used might be faster (highly parallel vs. efficient). As to the steps of the proof, Kronecker uses Cauchy's Bound Theorem to limit the domain on which the polynomial could cross the x-axis (where there are roots) and also finds the discriminant of the polynomial, which based on the definition will be larger than the smallest interval between roots. A number s is then defined based on the coefficients of the polynomial so that an interval of size less than $\frac{1}{s}$ has the "at most one root" property. Then you would know that if an interval of that size has different signs on the endpoint, it crossed the x-axis and has a root. If you are interested in the actual proof for more information on how s is found or why it has the property you can read my analysis in the Rose-Hulman Undergraduate Mathematics Journal Vol. NUM (found here: LINK). So far we will want to use Cauchy's Bound Theorem, find the discriminant, and find s (not yet discussed); additionally, the procedure that Kronecker "replaces", a different root estimation theorem, has no guarantees about the precision of the estimate, but is more suited to a divide-and-conquer approach. This would be Sturm's Theorem, which provides a method for calculating the number of real roots of a polynomial are within an interval. For a better understanding of how our program will function, we fist need to look at the mathematics that we intend to implement.

2.1 Cauchy's Bound Theorem

This theorem provides a domain on which a polynomial could have real roots. It does this by finding the point at which the leading coefficient (multiplied by x^n) in larger than the rest of the terms. Thus, after this point, the sign of the polynomial will remain the same. For a polynomial $a_0 + a_1x + \cdots + a_nx^n$ this interval is $(-\mathbf{r}, \mathbf{r})$ where $\mathbf{r} = \max\left\{\left|\frac{a_0}{a_n}\right|, \left|\frac{a_1}{a_n}\right|, \ldots, \left|\frac{a_{n-1}}{a_n}\right|\right\} + 1 = \left|\frac{\max\{\mathbf{a}\}}{a_n}\right| + 1$. Using this theorem, we can have a starting domain to apply our algorithm to.

2.2 Sturm's Theorem

Sturm's Theorem counts the number of roots of a polynomial in an interval, which is useful for a quicker way to reduce our search space. There are other methods of counting roots based on Descartes' Rule of Signs that are faster (and are how most root finding algorithm's would work), but we will use this since we are not too concerned with speed yet. The definition of a Sturm Chain is required for the Theorem Bartlett 2013.

Definition 2.1 (Sturm Chain)**.** A **Sturm Chain** is a finite sequence of polynomials $p_0(x), p_1(x), \dots, p_m(x)$ of decreasing degree with the following properties:

1. $p_0(x)$ is square-free.
2. If a is a root of $p(x)$, then the sign of $p_1(a)$ is the same as the sign of $p'(a)$, and in particular is nonzero.
3. If a is a root of $p_i(x)$ for some i with $0 < i < m$, then both $p_{i-1}(a)$, $p_{i+1}(a)$ are nonzero. Moreover, the sign of $p_{i-1}(a)$ is opposite of the sign of $p_{i+1}(a)$.
4. $p_m(x)$'s sign is constant and nonzero for all x.

This might seem very complex for a structure, especially since we only have the chain's criteria and not a method of construction. If we consider a polynomial with no repeated roots (square-free) $p(x)$, then the Sturm Chain could be constructed as:

- $p(x) = p_0(x)$
- $p_1(x) = p'(x)$
- $\cdots$
- $p_n(x) = -\text{rem}(p_{n-2}(x), p_{n-1}(x))$, where rem is the remainder of **Polynomial Long Division** from $p_{n-2}(x)$ divided by $p_{n-1}(x)$
- Repeatedly find $p_i(x)$ until we arrive at some m such that $\text{rem}(p_{m-1}(x), p_{m-2}) = 0$.

You can find a proof that this procedure results in a Sturm Chain in the cited reference Bartlett 2013. This is not of particular interest to us, since we only need to be able to construct this. Sturm's Theorem then uses this object to count the number of roots in an interval.

Theorem 2.1 (Sturm's Thorem)**.** Take any square-free polynomial $p(x)$, and any interval (a, b) such that $p_i(a), p_i(b) \neq 0$, for any i. Let $p_0(x), \dots, p_m(x)$ denote the Sturm chain corresponding to $p(x)$. For any constant c, let $\sigma(c)$ denote the number of changes in sign in the sequence $p_0(c), \dots, p_m(c)$. Then $p(x)$ has $\sigma(a) - \sigma(b)$ distinct roots in the interval (a, b).

Again, for a proof of this, see the references. This allows us to use larger interval sizes and rule out parts of our search space based on whether or not there are real roots there. This theorem can become an algorithm through the following. By taking a polynomial, we can construct the Sturm Chain and then evaluate each of the polynomials at the interval endpoints and count the number of sign changes at each of the points. Then the subtraction of these two numbers would be the number of roots within the interval. If this is nonzero, we can create smaller intervals and repeat.

2.3 Multiplicity

Thus far, with both Kronecker's proof and Sturm's Theorem we have only considered polynomials with no repeated roots; however, it is also possible that a polynomial can have roots with a multiplicity greater than one. We must not consider this possibility. So if we have a repeated root, then we have a multiplicity of at least two; also, we know that a derivative will reduce the degree by one. So if $p(x) = (x-1)^2 = x^2 - 2x + 2$, then $p'(x) = 2x - 2 = 2(x-1)$. We can then conclude that if a polynomial has a repeated root, that its derivative also contains that root. Without a string proof, consider the following examples:

$$\begin{aligned}
\frac{d}{dx}((x-2)(x+3)) &= \frac{d}{dx}(x^2+2x-3)\\
&= 2x+2 = 2(x+1)\\
\frac{d}{dx}((x-1)^2(x-3)(x+4)) &= \frac{d}{dx}(x^4-x^3-13x^2+25x-12)\\
&= 4x^3-3x^2-26x+25\\
&= (x-1)(4x^2+x-25)\\
\frac{d}{dx}((x-1)^3(x+2)) &= \frac{d}{dx}(x^4-x^3-3x^2+5x-2)\\
&= 4x^3-3x^2-6x+5\\
&= (x-1)^2(4x+5)
\end{aligned}$$

So, we can remove the repeated roots of $p(x)$ by finding the **Greatest Common Divisor** between $p(x)$ and $p'(x)$, and then we find $p*(x) = \frac{p(x)}{\gcd(p(x),p'(x))}$ which we know is square-free and has the same roots as $p(x)$. This now covers the main parts of how the first part of our program will work. In the next section we will bring this together and discuss how the program will be formatted as well as the other functions that we will need, like differentiation and polynomial division.

3 Layout of Program

For the first part of the polynomial root finding program, we want to use Cauchy's Bound Theorem and Sturm's Theorem to reduce our search space. Cauchy's Bound is a straightforward application of the theorem in the previous section. On the other hand, Sturm's Theorem will require differentiation, polynomial division, evaluation of polynomials, and a function to count the number of sign changes of a Sturm Chain. These are not difficult to implement, but do require a bit of thinking. This section is just describing the algorithms for how this would be implemented, but since we write them in Python in the next section, that is where we will plan for floating point error and the precision of our estimates.

3.1 Polynomial Representation

We want to store a polynomial so that we can use if for the functions that we create. Polynomials will be stored as a list of coefficients starting with the leading term at index zero. For example, $5x^2 + 2x + 3$ will be represented by the list $[5, 2, 3]$.

3.2 Differentiation

We are finding the derivative of a polynomial (which is differentiable everywhere) using only the power rule. In case you don't remember this from your Calculus I class, we have the theorem here.

Theorem 3.1 (Power Rule). $\frac{d}{dx}(ax^n) = anx^{n-1}$

Thus to implement this we just find the degree of our polynomial based on the length of the list, multiple each by $degree - index$, and drop the last coefficient.

3.3 Evaluation of a Polynomial

The naive way of evaluating a polynomial would be to do $a \cdot x^n$ for each term and to sum them, but this is many more computations than needed ($\frac{n(n+1)}{2} + 2n - 1$ CPU cycles in fact). The faster method would be to use Horner's Method which evaluates $p(x)$ by multiplying the leading coefficient by x, adding the next coefficient, then multiplying by x. This occurs repeatedly, so the polynomial is rearranged to evaluate like:

$$p(x) = (((a_n \cdot x + a_{n-1}) \cdot x + \cdots) \cdot x) + a_0$$

This evaluates a polynomial at x with the least complexity.

3.4 Polynomial Long Division

Polynomial Division is something I always have to relearn whenever I need it. If you have $x^2 - 2x + 2$ and we want to divide it by $x - 1$, then will be finding a polynomial of degree $1 = 2 - 1$ such that multiplication by $x - 1$ gives us $x^2 - 2x + 2$. So the first term of this polynomial will be $\frac{x^2}{x} = x$. Multiplying x by $x - 1$ yields $x^2 - x$. And the last step of long division (subtraction) gives $(x^2 - 2x + 2) - (x^2 - x) = -x + 2$. The second repetition completes the polynomial as $x - 1$ with no remainder. This seems complicated, but for each round we are dividing the remaining leading coefficient of the dividend by the leading coefficient of the divisor to find the resulting polynomial's i^{th} term. We will then multiply this new coefficient by our divisor and subtract based on like terms. These steps are repeated until we have the finally result and the list of coefficient that we have been updating is then our remainder. Once we state the steps in the form of the coefficients and arithmetic it is easier to see how this would be implements. If this doesn't make sense at the moment, you can try doing a couple of these on paper by hand to see the pattern, or you can wait until we implement this in the next section.

3.5 Greatest Common Denominator

Similar to using the Euclidean Algorithm to find the gcd of two numbers, we can use it with polynomial division to find the greatest common denominator between two polynomials. Given polynomials f, g with $textrmdegf >= \deg g$ the divisor can be found by:

1. $r_0 = f$
2. $r_1 = g$
3. $r_n = \text{rem}(r_{n-2}, r_{n-1})$
4. Repeatedly find r_n until $r_{n+1} = 0$. The GCD of f and g is then r_n.

Since we have already discussed polynomial division, implementation of this will not require much more work.

3.6 Sturm's Theorem

The creation of the Sturm Chain is as exactly as discussed in the previous section, and counting the number of roots can be done through evaluation of each of the polynomials in this chain (already discussed), and counting how many sign changes (fairly straightforward).

4 Implementation: Part 1

Here we will take each of the parts of the first part of our algorithm and we will implement then as Python functions. We could make a polynomial class and give it different functions, but that is more of a structure concern that should be done if you want to integrate this into another program; whereas, we are only concerned with the one task here and will keep them separate.

The other question we should have when dealing with computer estimation is floating point error because of how they are represented Goldberg 1991. Thus our estimation of zero will be if a number is less than $1 \cdot 10^-16$. This is an approximation for zero with one more decimal place than we might reasonably desire. For example, NASA's Jet Propulsion Laboratory uses an approximation of pi rounded off at the 15^{th} digit, since this still provides a very accurate calculation - given a circle with a diameter of 25 billion miles calculating the circumference with only 15 digits of pi will only be wrong by about 1.5 inches JPL Education 2016. Additionally, if we want to obtain "better" representations of our numbers, we can use the Fraction library and class found in Python so that all of our numbers are expressed as rational number. This will be slower since CPUs are built for integer or floating point arithmetic, but we have already stated that we are not too concerned with minimal complexity with our decision on how to approach root finding. Thus, we will use the rational representation. Moving onto the implementation of the first part of our algorithm we will write the functions to limit our search space.

The details described in the last section means that most of the functions can be created easily. My final file can be found on GitHub here: LINK, or at the end of this. Some of the functions are harder than others, so they are discussed here.

4.1 Polynomial Division

Polynomial Long Division is that repressed memory from Algebra II in High School and is something that really never comes up again. The algorithmic approach to this was discussed in the last section, but we will review the following function line by line to understand it better.

```python
from fractions import Fraction
from copy import deepcopy

def poly_div(f, g):
    '''Divide polynomial f by polynomial g. Returns the result and remainder.

    :param f: list-like of coefficients of a polynomial in descending order of degree
    :param g: list-like of coefficients of a polynomial in descending order of degree
    :return: tuple of two lists of polynomial coefficients - the result and the remainder'''

    degree_f = len(f) - 1
    degree_g = len(g) - 1
    assert f[0] != 0, "Must have nonzero leading coefficient"
    assert g[0] != 0, "Must have nonzero leading coefficient"
    assert degree_f >= degree_g, "Must divide by lesser degree"

    out = [0 for i in range(degree_f - degree_g + 1)]
    rem = deepcopy(f)
    for i in range(degree_f - degree_g + 1):
        # find the coef of the ith term in f divided by the leading term of g
        new_coef = Fraction(rem[i]) / Fraction(g[0])
        out[i] = new_coef
        # update subtraction
        for j in range(len(g)):
            rem[i + j] -= new_coef * Fraction(g[j])

    # f has been updated in-place to represent the remainder
    # we need to remove the front-zeros
    try:
        while abs(rem[0]) < 1e-16:
            rem.pop(0)
    except IndexError:
        # we have no remainder so f is empty and throws an index error
        rem = [0]

    return out, rem
```

Here we use the Python packages fraction and copy. Specifically, within those the class Fraction and method deepcopy. Fractions express our numbers are rationals and it allows for more accurate (and expensive) computation, and deepcopy creates a separate identical object so that way we can modify it without modifying the original since Python is neither pass-by-value nor pass-by-reference and can have some weird behavior without this consideration. We then declare our function for polynomial division of f and g, where those are both lists of coefficients like we have described before; and we have f as the dividend and g as the divisor, so we do some error checking to make sure that our function can divide these two polynomials - namely, we check that they have a nonzero leading coefficient and that the degree of the dividend is greater than or equal to that of the divisor. If not, then an assertion error is raised. Note: Assert statements can be turned off and should really only be for internal logic checking, so in the final file referenced, these are modified to raise actual exceptions.

We are now into the implementation of the algorithm for polynomial division. We know that the resulting polynomial has a degree so that when multiplied by g it is the same degree as f. Meaning $\deg r = \deg f - \deg g$. Thus, we create our output list. Additionally, we will want to keep track of our remainder. The remainder is whatever is left over after having calculated the resulting polynomial and subtracted from the dividend $(f - (res \cdot g) = rem)$. We are continually subtracting from f, so we will make a duplicate copy for this logic and when we are done, this will be the remainder. Finally, we get to the repeating step. We have the leading coefficient of f, a, and we want a term c such that when multiplied by the leading coefficient of g, b, we have a; this will reduce the dividend entirely by one degree and leaves no remainder. Stated as an equation

$$\begin{aligned} ax^n &= c \cdot bx^m \\ \frac{ax^n}{bx^m} &= c \\ c &= \frac{a}{b}x^{n-m} \end{aligned}$$

And since we have stored the polynomials as lists, we don't have to worry about the x^{n-m} part. So, this factor c is just the division of leading coefficients. This factor is added to our resulting polynomial as the next coefficient, and then we move to the last part of this repetition. We need to subtract from our remainder (the updated copy of f) the full expansion of $cx^{n-m} \cdot g = cb_0 + cb_1x + \cdots + cb_mx^m$. To update our subtraction, we iterate through g, starting with the current leading terms, and update the coefficients of the remainder by subtracting cb_i. While we still have coefficients left in our resulting polynomial, we repeat this step until we have our final polynomial and remainder.

The last bit of logic takes our remainder polynomial and removes the leading coefficient if it is zero. And if we have all zeros, then we create a remainder $[0] \in \mathcal{P}_0$. The function returns the result and remainder as a tuple and we have successfully completed the definition. Note: we will write tests in a later section.

4.2 Application of Part 1

For the time being, this is not a function since we will have more to add to it, but it does bring together the necessary functions in the required steps for application of Sturm's Theorem. We start with a function $f = x^4 - x^3 - 13x^2 + 25x - 12$ so that $f = (x-1)^2(x-3)(x+4)$ and two parameters for how we approach the divide-and-conquer: the number of subintervals to create (num_ints) and the width of the final intervals (precision). The first thing we do is remove the multiple roots as discussed before in the multiplicity section, and we create $f^*(x) = \frac{f(x)}{\gcd(f,f')}$. For that, we can generate the Sturm Chain. Using Cauchy's Bound we have an interval for all of our roots, and if we apply Sturm's Theorem to this interval, we can count the number of real roots of the polynomial (logic to quit if zero is not shown here). We can then subdivide this interval into eight new ones. Then while the largest of our intervals has a greater width than the expected precision (our stopping condition), we will evaluate the Sturm Chain at the endpoints of the intervals, find the number of sign changes, and then calculate the number of roots within an interval. Parallel processing could improve the evaluation step, but this is our basic overview of steps.

```
num_ints = 8 # number of intervals
f = [1, -1 , -13, 25, -12]
num_ints = 8 # number of intervals
precision = 1e-15

# remove multiple roots
f_star = poly_div(f, gcd(f, differentiate(f)))[0]

# find Sturm Chain for function
seq = sturm_seq(f_star)

# find the bounds of our roots and step up inital intervals
lb, ub = cauchy_bound(f_star)
step = (ub - lb) / num_ints
num_roots = sturm_seq_eval(seq, lb) - sturm_seq_eval(seq, ub)
```

```
next_ints = [[lb + (i*step), lb + ((i+1)*step)] for i in range(num_ints)]
intervals = next_ints

while max([abs(x[1] - x[0]) for x in intervals]) > precision:
    intervals = next_ints

    # get root counts for intervals
    root_counts = []
    for x in intervals:
        root_counts.append(sturm_seq_eval(seq, x[0]) - sturm_seq_eval(seq, x[1]))

    # find smaller intervals
    next_ints = []
    for i in range(1, len(root_counts)):
        if root_counts[i] > 0:
            a = intervals[i][0]
            b = intervals[i][1]
            step = (b - a) / num_ints
            next_ints += [[a + (i*step), a + ((i+1)*step)] for i in range(num_ints)]

# find our candidate intervals
candidates = []
for j in root_counts:
    if j != 0:
        candidates.append((j, intervals[j][0], intervals[j][1]))
```

After looking at this, you might ask why is this Part 1 and not the full process since we have a precision for the estimate of the root such that $|x_i - r_i| < 1 \cdot 10^-15$; however, we have no guarantees about the the evaluation of f at our approximate root, and we also have some weird behavior when attempting to evaluate the Sturm Chain at a root (since that is not allowed by the theorem) if given as the endpoint of one of our intervals.

Consider our example $f = x^4 - x^3 - 13x^2 + 25x - 12 = (x-1)^2(x-3)(x+4)$. We can quickly find $f' = 4x^3 - 3x^2 - 26x + 25 = (x-1)(4x^2 + x - 25)$ and $\text{GCD}(f, f') = (x-1)$. So, we have $f* = (x-1)(x-3)(x+4) = x^3 - 13x + 12$. Our Sturm Chain is then calculated as follows:

$$\begin{aligned}
p_0(x) &= f*(x) = x^3 - 13x + 12 = (x-1)(x-3)(x+4)\\
p_1(x) &= f*'(x) = 3x^2 - 13\\
p_2(x) &= -\text{rem}(p_0, p_1) = \frac{26}{3}x - 12\\
p_3(x) &= -\text{rem}(p_1, p_2) = \frac{1225}{169}
\end{aligned}$$

If we then choose -4 and evaluate our Chain, we get $[0, 35, -46\frac{2}{3}, \frac{1225}{169}]$ which has one sign change. So then let's evaluate at -3, so that we have $[24, 14, -38, \frac{1225}{169}]$. Again one sign change, so in the interval $(-4, -3)$ we have zeros roots, which is in fact correct because that is an open interval, but this evaluation is obscured on a computer. Storing our numbers as rationals does mitigate this, but If we run the program above, we if fact identify four candidate intervals (when we can only have a maximum of three). We find:

- The interval $(\frac{-72057594037927939}{18014398509481984}, \frac{-144115188075855871}{36028797018963968}) \approx (-4.0, -4.0)$ has 2 roots.
- The interval $(\frac{-144115188075855871}{36028797018963968}, \frac{-18014398509481983}{4503599627370496}) \approx (-4.0, -4.0)$ has -1 root.
- The interval $(\frac{36028797018963959}{36028797018963968}, \frac{18014398509481983}{18014398509481984}) \approx (0.9999999999999998, 1.0)$ has 1 root.
- The interval $(\frac{54043195528445949}{18014398509481984}, \frac{108086391056891905}{36028797018963968}) \approx (3.0, 3.0)$ has 1 root.

This is really close to correct, and we can look at the intervals to correctly identify roots, but we should not have to do that. Upon further inspection, our number that both of the -4 intervals share is

$-3.9999999999999999722444243843710864894092082977294921875$ (according to WolframAlpha, since our float representation rounded us off to 4.0). Sturm's Theorem does not hold if we pass in a root of any of the polynomials in the Chain, and it is for an open interval. This is why I choose to look at evaluation of negative -4 above. So, how might we extend our implementation for arbitrary evaluation? We will extend our rational number evaluation to handle large integers, and we will use Kronecker's algorithm (or rather a modified version that we will cover in the next section) for further evaluation, since that only requires two polynomial evaluations and we have sufficiently narrowed our search space.

5 Kronecker's Algorithm

References

Bartlett, P. (2013). *Finding all the roots: Sturm's theorem.* University of California, Santa Barbara. http://web.math.ucsb.edu/~padraic/mathcamp_2013/root_find_alg/Mathcamp_2013_Root-Finding_Algorithms_Day_2.pdf

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *Computing Surveys.* https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

JPL Education. (2016). *How many decimals of pi do we really need?* https://www.jpl.nasa.gov/edu/news/2016/3/16/how-many-decimals-of-pi-do-we-really-need/