Jenkins-Traub: A Globally Convergent Polynomial Root Finding Algorithm

Kathleen Brandes

May 2019

1 Introduction

Finding the roots of functions and polynomials is a standard operation used in many different problems and fields of study, such as optimizations or ODE shooting methods, and requires efficient computation of large numbers of function roots. As well, there are many problems in which an exact solution for computing the roots can not be solved for algebraically. For example, problems in physics, like the Van der Waals equation, use root calculations to intuit understanding of the physical system, yet the roots are unable to be solved for algebraically. Therefore, the need for efficient numerical methods to approximate the roots are required.

Numerical algorithms are used to compute approximations of the function's roots with accuracy bounded by a small epsilon, particularly when a closed form expression for the roots is either impossible to find or too time consuming to compute. There are many different variants of root finding algorithms, including iterative and bracketing methods. These algorithms must be able to find roots of high degree polynomials efficiently and accurately to be usable in the scope of optimizations or ODE solvers. The simpler root finding methods, such as bisection method, secant method, and Newton-Raphson, all share the problem that they are not convergent for all initial conditions or guesses and do not scale for larger degree polynomials. However, these methods require often fewer function evaluations per iteration. This paper will introduce the Jenkins-Traub root finding algorithm. The Jenkins-Traub root finding algorithm for real valued, high degree polynomial functions is capable of global convergence across all starting conditions for any distribution of the roots. Additionally, it has faster than quadratic convergence, however at the cost of more function evaluations.

2 Root Finding Methods

Many different root finding algorithms exist to solve all variations of the problem, such as nonlinear root finding, complex root finding, or polynomial root finding. For the case of solving for the roots, the problem can formally be stated as solving $\sum_{i=0}^{n} a_i x^{n-i} = 0$ for all possible values of x. For convenience, let $a_0 = 1$, which can always be obtained by dividing all coefficients by a_0 to normalize the polynomial. Methods that exist to solve this problem include bracketing methods (bisection algorithm), iterative methods (secant algorithm, Newton-Raphson algorithm), and various interpolation methods.

2.1 Bracketing Methods

Bracketing methods for root finding successively decrease a bracketed interval containing a root of the function until converging to the root. The primary bracketing method is the bisection method, which iteratively subdivides an interval $[a, b]$ with $f(a) = f(b)$. Each iteration creates two subintervals $[a, \frac{a+b}{2}]$ and $[\frac{a+b}{2}, b]$. The algorithm then evaluates the function at the midpoint $f(\frac{a+b}{2})$ to determine which interval has opposite signs at the endpoints; this interval will still contain the root since it will cross zero.

The bisection method has linear convergence to the root, which is very slow convergence considering polynomials with very high degree and many roots. Additionally, the algorithm relies on having an initial interval which contains a root, making it only locally convergent.

2.2 Iterative Methods

Most root-finding algorithms proceed with an iterative process, however the "iterative methods" for root finding algorithms use specific types of iteration which define auxiliary functions which are applied at the previous iteration's root approximation to get a new approximation. These algorithms terminate when a root approximation is found with a certain precision, so the change between the current approximation and the previous approximation is less than some epsilon.

The secant method is a common iterative method which uses the series of roots of secant lines to approximate the roots of function f. The secant method converges at the speed of the golden ratio, so between linear and quadratic convergence rates. This method will fail to converge to a root if the initial guess or interval are very incorrect.

Another common iterative method is Newton-Raphson method, which assumes the function will have a continuous derivative. Given this assumption, each iteration of the Newton-Raphson method computes the following update to an approximation of the function $f(x)$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

The algorithm terminates when the value x_{n+1} is within some epsilon distance of 0. In the optimal case, this method converges to the root quadratically, and it only involves 2 function evaluations per iteration of the algorithm. Additionally, the iterative formulation generalizes to higher dimension state spaces $\mathbf{x} \in R^n$. While this algorithm can provide quadratic convergence in most cases and is one of the faster root finding algorithms, it also has many practical considerations that make it difficult to use robustly. It relies on there existing a continuous derivative for the entire function; however, for the case of polynomial root finding, the derivative is easily known and computed. As well, there are multiple cases in which the algorithm will not converge, shown by [4], including:

- Derivative Issues the derivative does not exist at the root, or the derivative is discontinuous
- Initial Guess Issues the initial point causes stationary iteration by creating zero-valued update $\frac{f(x_n)}{f'(x_n)}$, or the initial point entering a value cycle $x_i, \ldots, x_{i+k}, x_{i+k+1}$ where $x_i = x_{i+k+1}$

In addition to these cases where the algorithm fails to converge, there are also times when the algorithm will only converge linearly such as functions with no second derivative or a zero-valued derivative.

The Jenkins-Traub algorithm computes roots for real and complex valued polynomials using an iterative procedure. The algorithm uses three stages, each of which performs a shift of the

polynomial by specific amount for each stage. The iteration of the algorithm is described in detail in Section 3 of the paper. As well, in Section 4, the paper will show that the Jenkins-Traub algorithm is globally convergent, meaning it will always find all roots of the polynomial. This contrasts with the other mentioned root finding methods which rely on a good initial guess to be able to converge to the root exactly. Section 5 will discuss the various details and tricks for creating an efficient implementation in Julia of the Jenkins-Traub algorithm for real and complex valued polynomials. The paper will then also compare the performance of the Jenkins-Traub implementation to the Newton-Raphson algorithm in Section 6 of the paper, considering cases where the algorithm is able to converge quadratically and also cases where it converges more slowly.

3 Jenkins-Traub Algorithm Formulation

The Jenkins-Traub algorithm efficiently can compute all real valued roots through a three stage process. The algorithm finds roots one by one in a generally increasing size. After detecting a root, a, through the three staged process, the algorithm deflates the polynomial by dividing by the linear expression $x - a$, which reduces the polynomial and removes the corresponding linear factor for that root. Ultimately, the algorithm factorizes the polynomial completely which is exactly the root finding procedure. When all coefficients of the polynomial $P(x)$ are real-valued, this algorithm is known as RPOLY. Similarly, for polynomial $P(x)$ with real and complex valued coefficients, the algorithm is known as CPOLY, [5]. Section 4 will discuss how this algorithm guarantees convergence to every root of the polynomial.

The algorithm completes three stages to determine a single root; these stages are outlined in Jenkin's and Traub's paper, "A Three-Stage Variable-Shift Iteration for Polynomial Zeros and Its Relation to Generalized Rayleigh Iteration," [3]. Each stage computes shifts of the polynomial to yield a new polynomial, $H^{(\lambda}(x))$, of degree $n - 1$. A shift by an amount s_λ can be found by:

$$H^{(\lambda+1)}(x) = \frac{1}{1 - s_\lambda}\left(H^{(\lambda)}(x) - \frac{H^{(\lambda)}(s_\lambda)}{P(s_\lambda)}P(x)\right) \tag{2}$$

This shifted polynomial can then be normalized such that the leading coefficient on the highest degree polynomial is 1. Normalizing the polynomial makes division by the linear term simpler algorithmically and keeps the polynomial coefficients in a reasonable range. The algorithm can compute the roots of polynomials with both real and complex valued coefficients. In [2], an algorithm is presented to compute two roots at a time: either two distinct real roots or a complex conjugate pair of roots. This algorithm is 4 fold faster than the complex variant presented and implemented in this paper, however it forces more constraints on the expanse of the problem space.

3.1 Stage 1: Zero Shifts

Stage 1 of the Jenkins Traub algorithm is a no-shift process to accentuate smaller roots near zero. This stage starts with the polynomial $P(x)$ and degree n. Initialize with the equation:

$$H^{(0)}(x) = P'(x) \tag{3}$$

And then compute a sequence of M different degree $n - 1$ polynomials, as described in [6], with 0-shifts computed by:

$$H^{(\lambda+1)}(x) = \frac{1}{x}\left(H^{(\lambda)}(x) - \frac{H^{(\lambda)}(0)}{P(0)}P(x)\right),\ \lambda \in \{0, 1, \ldots, M-1\} \tag{4}$$

Typically, most implementations for RPOLY have $M = 5$ iterations of zero-shift stage to obtain accurate roots, based on numerical evidence from [6].

3.2 Stage 2: Fixed Shifts

Stage 2 of the Jenkins Traub algorithm is the fixed shift-process, which works to separate zeros of equal or nearly equal magnitude. If the multiplicity of the smallest magnitude zeros is larger than 2, this stage will shift the polynomial such that the smallest magnitude zero has either multiplicity of 1 or 2, as shown in [1].

The algorithm computes the fixed shift s to be a value close to the smallest root of the polynomial, so $|s| = \beta$ where $\beta \leq min(\rho_i)$ and ρ_i are the roots of the polynomial $P(x)$. A moduli of the minimum root can be lower bounded using a theorem by Cauchy stating that this lower bound will be a unique, positive β which is a zero of the polynomial $x^n + |a_1|x^{n+1} + \cdots + |a_{n-1}|x - |a_n|$, shown in [2]. In practice, this zero can be found with Newton-Raphson iteration. Given this value β, the fixed shift value s can be chosen by using a random number from a uniform distribution to pick a point on the circle of radius β. Therefore, we get an exact value of $s = \beta cos(2\pi * \texttt{rand})$, which will be the fixed shift for stage 2.

During each iteration of the fixed shift stage, compute the next shifted H polynomial using the fixed shift value s and also a normalized H polynomial, $\bar{H}$ which can be computed by dividing by the leading coefficient:

$$H^{(\lambda+1)}(x) = \frac{1}{x - s}\left(H^{(\lambda)}(x) - \frac{H^{(\lambda)}(s_\lambda)}{P(s_\lambda)}P(x)\right) \tag{5}$$

Additionally, each iteration calculates an approximation of the root is estimated with the equation:

$$t_\lambda = s - \frac{P(s)}{\bar{H}^{(\lambda)}(s)} \tag{6}$$

Throughout the iteration of stage 2, the algorithm should track $t_\lambda, t_{\lambda+1}$, and $t_{\lambda-1}$. This history of approximate roots is used to determine when to stop iterating for stage 2.

$$|t_{\lambda+1} - t_\lambda| \leq .5|t_\lambda| \tag{7}$$

$$|t_\lambda - t_{\lambda-1}| \leq .5|t_{\lambda-1}| \tag{8}$$

When both conditions are met, stage 2 can be complete and stage 3 will begin using the final $H^{(\lambda)}(x)$ used in iteration. If neither condition is met but the number of iterations exceeds a maximum threshold for allowed iterations, then stage 2 should be restarted with a different, randomly initialized fixed shift value.

3.3 Stage 3: Variable Shifts

Stage 3 of the Jenkins Traub algorithm is the variable shift stage, where the algorithm will ultimately find a value for which the polynomial can be evaluated to be zero within some epsilon bound. Compute the initial value for the variable shift polynomials:

$$s_L = t_L = s - \frac{P(s)}{\bar{H}^{(\lambda)}(s)} \tag{9}$$

where t_L is just the final root estimate from stage 2, found by computing a normalized version of the final H polynomial and evaluating it at the fixed shift value s from stage 2. Stage 3 iterates until it converges to a root, so $\lambda = L, L+1, \ldots$, and on each iteration it updates the shift value and the H polynomial:

$$H^{(\lambda+1)}(x) = \frac{1}{x - s_\lambda}\left(H^{(\lambda)}(x) - \frac{H^{(\lambda)}(s_\lambda)}{P(s_\lambda)}P(x)\right) \tag{10}$$

$$s_{\lambda+1} = s_\lambda - \frac{P(s_\lambda)}{\bar{H}^{(\lambda)}(s_\lambda)} \tag{11}$$

Termination of stage 3 occurs when a root is found, so $P(s_\lambda) < \epsilon$, or if the maximum number of allowed iterations occur without determining a root. In the latter scenario, restart from stage 2 with a different random fixed shift value, [5].

4 Proof of Convergence

The Jenkins-Traub algorithm is known to be a globally convergent algorithm for finding all roots of polynomials with real or complex valued coefficients. The proof of convergence follows that of [3], and starts by first defining the sufficient conditions for the convergence of the variable shift stage, as this stage will ultimately determine the roots. Then it will examine the entire 3 stage algorithm for convergence, using these conditions for finding a root.

Let the roots of polynomial $P(x)$ be ρ_i. Define the H polynomial at iteration L to be:

$$H^{(L)}(x) = \sum_i c_i^{(L)} P_i(x), \quad P_i(x) = \frac{P(x)}{(x - \rho_i)} \tag{12}$$

From this, we can derive the following theorem.

Theorem 1. *Assume the following conditions are met*
1) $|s_L - \rho_1| < .5R$ *where* $R = min_i|\rho_1 - \rho_i|$,
2) $c_1^{(L)} \neq 0$
3) $D_L = \sum_{i=2} \frac{|c_i^{(L)}|}{|c_1^{(L)}|} > \frac{1}{3}$
Then, $s_\lambda \to \rho_1$ *during the iteration of stage 3.*

Proof. For $\lambda \geq L$, the following definition follows from equation (12):

$$c_i^{(\lambda)} = c_i^{(L)} \Pi_{t=L}^{\lambda-1} \frac{1}{(\rho_i - s_t)} \tag{13}$$

$$H^{(\lambda)}(x) = \sum_i c_i^{(\lambda)} P_i(x) \tag{14}$$

The base case is $c_i^{(0)}$ equals the multiplicity of root ρ_i. Additionally, define ρ_1 to be the smallest root for the polynomial. For stage 3 to converge to the smallest root each iteration, we need $H^{(\lambda)}(x) \to c_1^{(\lambda)} \frac{p(x)}{(x-\rho_1)}$, so $\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}} \to 0$. Using these definitions and some algebraic manipulation, we can write the approximation for the root at each iteration as:

$$t_{\lambda+1} = s_\lambda - \frac{p(s_\lambda)}{\bar{H}^{(\lambda+1)}(s_\lambda)} = s_\lambda - \frac{p(s_\lambda)\sum_{i=1}^{j} c_i^{\lambda+1}}{\sum_{i=1}^{j} c_i^{\lambda+1}(\frac{p(x)}{(x-\rho_i)})} = s_\lambda - \frac{\frac{p(s_\lambda)}{s_\lambda-\rho_1}(s_\lambda-\rho_1)c_1^{\lambda+1}(1+\sum_{i=2}^{j}\left(\frac{c_j^{(\lambda+1)}}{c_1^{(\lambda+1)}}\right))}{\frac{p(s_\lambda)}{s_\lambda-\rho_1}c_1^{\lambda+1}(1+\sum_{i=2}^{j}\left(\frac{c_j^{(\lambda+1)}}{c_1^{(\lambda+1)}}\frac{(s_\lambda-\rho_1)}{(s_\lambda-\rho_i)}\right))}$$

Further manipulation yields the equation, where $r_i^{(\lambda)} = \frac{s_\lambda-\rho_1}{s_\lambda-\rho_i}$:

$$T_L = \frac{|s_{\lambda+1}-\rho_1|}{|s_\lambda-\rho_1|} = \frac{\sum_{i=2}^{j}(r_i^{(\lambda)})^2\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}} + \sum_{i=2}^{j}(r_i^{(\lambda)})\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}}}{1+\sum_{i=2}^{j}(r_i^{(\lambda)})^2\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}}} \tag{15}$$

From equation (15), we can show convergence by proof by induction that there exists a τ_L such that $T_L \le \tau_L < 1$. First, observe that $|r_i^{(L)}| = \frac{|s_L-\rho_1|}{|s_L-\rho_i|} < 1$, so we can rewrite the expression for T_L from equation (15):

$$T_L \le \frac{\sum_{i=2}^{j}\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}} + \sum_{i=2}^{j}\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}}}{1-\sum_{i=2}^{j}\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}}}$$

where this expression is just $T_L \le \frac{2D_L}{1-D_L} < \frac{2/3}{1-1/3} = 1$ using the definitions and assumptions from the theorem statement. Let $\tau_L = \frac{2D_L}{1-D_L}$, which shows there exists a $T_L \le \tau_L < 1$. Considering the iteration of stage 3 of the algorithm, then $T_L, T_{L+1}, \ldots, T_{\lambda-1} \le \tau_L < 1$ such that for $t = L, L+1, \ldots, \lambda$, and permits the inequalities $|s_t-\rho_1| \le |s_L-\rho_1| < .5R$ and $|s_t-\rho_i| \ge |\rho_1-\rho_i| - |s_t-\rho_1| > .5R$. Therefore, similar to before, we observe that for all $t = L, L+1, \ldots, \lambda$, then

$$|r_i^{(t)}| = \frac{|s_t-\rho_1|}{|s_t-\rho_i|} < 1 \tag{16}$$

Additionally, observe that $\frac{c_j^{(\lambda+1)}}{c_1^{(\lambda+1)}} = \frac{|s_\lambda-\rho_1|}{|s_\lambda-\rho_i|}\frac{c_j^{(\lambda)}}{c_1^{(\lambda)}}$, so we get the inequality for all $\lambda \ge L$:

$$\sum_{i=2}^{j}\frac{c_i^{(\lambda)}}{c_1^{(\lambda)}} \le D_L \tag{17}$$

From equations (15), (16), and (17), we can see by induction that $T_L \le \tau_L < 1$ for $\lambda \ge L$, completing the proof of convergence for stage 3. □

Using the information from Theorem 1, we can see that the sequence of variable shifts $\{s_\lambda\}$ will be well defined for $\lambda \ge L$. Knowing these necessary conditions for stage 3 to converge to a root, we can prove the global convergence of the entire algorithm to the smallest root on each iteration as Jenkins and Traub did as well in "A Three-Stage Variable-Shift Iteration for Polynomial Zeros and Its Relation to Generalized Rayleigh Iteration", [3].

Theorem 2. *For $i = 2, \ldots, j$, let s meet the constraint $|s-\rho_1| < |s-\rho_i|$. Then, for all fixed L sufficiently large, the algorithm will converge, so $s_\lambda \to \rho_1$.*

Proof. From equation (4) computing the H polynomials during the zero-shift stage and from the method of choosing β in stage 2 such that $|s| = \beta$ and $|s - \rho_1| < |s - \rho_i|$, we get:

$$H^{(L)}(x) = \sum_{i=1}^{j} m_i \rho_i^{-M} (\rho_i - s)^{-L+M} P_i(x) = \sum_i c_i^{(L)} P_i(x) \tag{18}$$

where M is the number of iterations for the update at stage 1, and m_i is the multiplicity of root ρ_i. This is exactly the expression for the H polynomial at iteration L as shown in equation (12). It is clear that $c_1^(L) \neq 0$. Additionally, we get:

$$D_L = \sum_{i=2}^{j} \frac{c_i^{(L)}}{c_1^{(L)}} = \sum_{i=2}^{j} \frac{m_i}{m_1} \frac{\rho_1^M}{\rho_i^M} \frac{(\rho_1 - s)^{L-M}}{(\rho_i - s)^{L-M}}$$

Using the theorem statement that $|s - \rho_1| < |s - \rho_i|$, for a fixed M, we can find a L large enough such that

$$D_L = \sum_{i=2}^{j} \frac{|c_i^{(L)}|}{|c_1^{(L)}|} < \frac{1}{3} \tag{19}$$

and also

$$|s - \rho_1| \frac{2D_L}{1 - D_L} < .5 min_i |\rho_1 - \rho_i| \tag{20}$$

Given this value of L, all conditions required by Theorem 1 are met, and therefore the convergence of the algorithm follows as such. □

These proofs demonstrate that the Jenkins-Traub algorithm is indeed globally convergent and will find all roots of any degree polynomial with real or complex coefficients. Additionally, Jenkins and Traub show in [3] that the algorithm will converge quadratically with an error constant approaching zero at a rate of $C(\lambda) = \frac{|s_{\lambda+1} - \rho_1|}{|s_\lambda - \rho_1|^2} \leq \frac{2}{min_i |\rho_1 - \rho_i|} \tau^{\lambda(\lambda-1)/2}$ for $\lambda \geq L$.

5 Jenkins Traub Implementation

The technical implementation of the Jenkins-Traub algorithms follows the algorithm presented in [3] to find the real roots of any degree polynomial given inputs of the number of iterations, a tolerance for exactness of the roots, and an ordered list of the coefficients of the polynomial. The algorithm implementation can accurately compute all real valued roots of the polynomial. Optimizations to the code in Julia from the baseline functionality and accuracy make the Jenkins-Traub algorithm implementation very efficient computing roots for all polynomials, even those with degree $n > 20$.

The implementation takes advantage of the problem formulation, particularly that the algorithm is finding the roots of a polynomial. Therefore, we can express derivatives, which is used in stage 1 of the algorithm, efficiently as a product of the powers and the coefficients, truncated to remove the constant term that becomes 0 when taking the derivative:

```
powers = [length(coefficients)-i for i in 1:length(coefficients)]
derivative = powers .* coefficients
derivative = derivative[1:length(derivative)-1]
```

Additionally, dividing a polynomial expression by a linear term $(x-a)$ can be done efficiently while simultaneously evaluating the polynomial at a. Synthetic division can be used to calculate the new polynomial $\frac{p(x)}{(x-a)}$, and then since it is a polynomial we can efficiently utilize the remainder of this division to evaluate the polynomial at a rather than having to compute an entire function evaluation again. In julia, this algorithm can be implemented as such:

```
syn_div = zeros(length(coefficients)-1)
syn_div[1] = coefficients[1]
for i=2:length(coefficients)-1
        syn_div[i] = coefficients[i] + syn_div[i-1]*a
end
```

```
eval = coefficients[length(coefficients)] + syn_div[length(syn_div)]*a
```

The implementation in Julia also uses vectorized, element wise operations when applicable to further optimize the algorithm for computational efficiency. Additionally, the Jenkins-Traub algorithm will terminate iteration within any of the three stages if a root approximation is found to be within the desired precision of the algorithm, rather than computing a fixed number of iterations like Newton-Raphson typically is implemented to do. Therefore, it has the potential to be even more efficient if the estimated variable shift values in stage 3 converge quickly or if an earlier stage shifted polynomial finds a root. The implementation in Julia is in the code listing in Appendix 1, and also can be found at `https://github.com/kbrandes45/numerical_methods_MITclass/blob/master/final_project_root_finding/jenkins_traub_rpoly.jl` which includes the analysis computations as well as the actual algorithms for Jenkins-Traub.

6 Results of the Jenkins-Traub Algorithm

The Jenkins-Traub algorithm implemented in Julia was analyzed with respect to error convergence, number of function evaluations, and solve times. For each of these metrics, it will be compared to the Newton-Raphson root finding algorithm, which is another iterative method commonly used for polynomial root finding since derivatives are known and continuous.

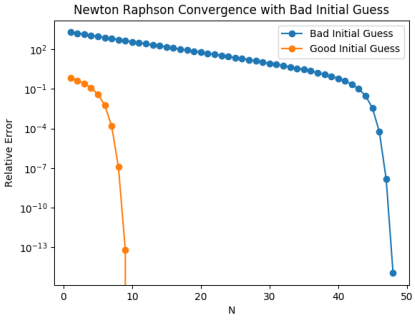

Figure 1: Effects of initial guess for Newton Raphson convergence.

To analyze and compare the two root finding algorithms, it is assumed that Newton-Raphson is given a reasonable initial guess; without a proper guess, the convergence to the correct root takes significantly more iterations and occasionally will not occur. Figure 1 shows the effects of an initial guess on the convergence rate of the Newton Raphson algorithm to the exact root of a test function; a "good" initial guess will be one where $\texttt{abs}(exactRoot - guess) < 5$ and a "bad" initial has $\texttt{abs}(exactRoot - guess) > 5$.

The convergence rates of the two root finding algorithms are analyzed for three different test functions of different polynomial degree. The specific test functions used were polynomials with all real roots of degree 6 (small), degree 15 (medium), and degree 26 (large). Evaluating the convergence of the two methods experimentally, Jenkins Traub algorithm can converge to the exact value of a single root significantly faster than Newton-Raphson for all tested degrees of polynomials, as seen in Figures 2, 3, 4 which analyze the relative error between the approximated root and the exact value of the root for both algorithms computing a single root with a fixed number of iterations. The three test polynomials confirm the expected convergence behaviors for the root

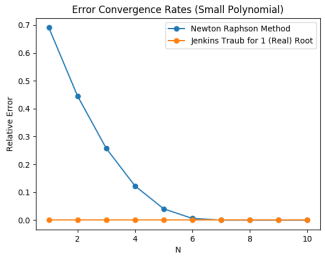

Figure 2: Convergence Rate of Small (degree 6) Polynomial

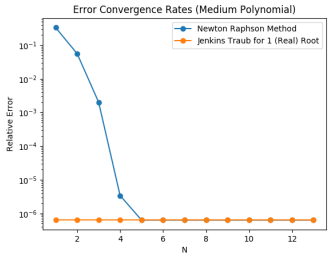

Figure 3: Convergence Rate of Medium (degree 15) Polynomial

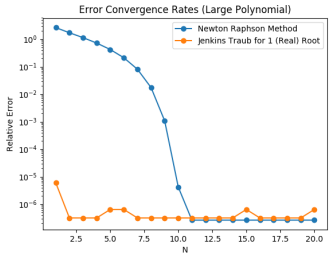

Figure 4: Convergence Rate of Large (degree 26) Polynomial

finding algorithms. Newton-Raphson has quadratic convergence to a single root, as mentioned in Section 2, and Jenkins-Traub is shown to have faster than quadratic convergence for finding a single root, as shown in [3].

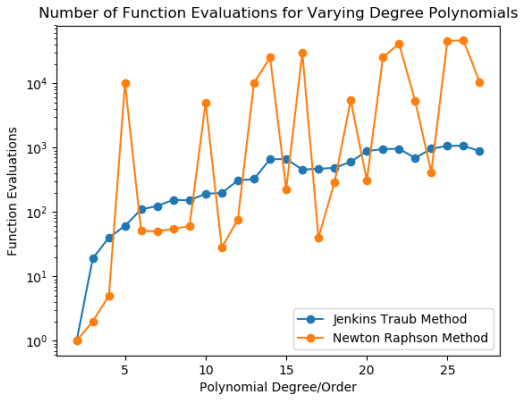

Figure 5: Total Number of Function Evaluations to Compute All Roots for Increasing Degree Polynomials

In addition to accuracy, numerical methods rely on efficient computation. To analyze the speed, the algorithms Newton-Raphson and Jenkins-Traub are compared on both the number of function evaluations and also the total time of the algorithm (computed by Julia) to compute all rootsof the polynomial. To analyze the number of function evaluations used by each algorithm, both algorithms iterated until all roots, x^*, achieved absolute numerical accuracy of $\texttt{abs}(f(x^*)) < 1e-10$. The algorithms were tested and compared for functions of degrees $d \in [2, 27]$. Considering a single iteration of the algorithms, Jenkins Traub takes roughly three times as many function evaluation than Newton Raphson. However, Newton Raphson requires more iterations in total to compute all the roots of the polynomial, especially for higher degree polynomials where the roots are more spread out and guessing the initial roots is not always in the margin of ± 5 from the actual root. This effect can be seen in Figure 5, where on certain polynomials, Newton Raphson requires far fewer function evaluations to compute all roots,

whereas on other polynomials, it takes many more function evaluations. As well, Jenkins-Traub provides a more reliable steady increase in function evaluations since it is not impacted as significantly by external factors, such as an initial guess. Figure 5 also shows that for polynomials of very low degree ($d < 5$), Newton-Raphson is more efficient at computing the roots to the same degree of precision, demonstrating that the overhead of the multiple function evaluations in Jenkins-Traub is unnecessary for such small polynomials.

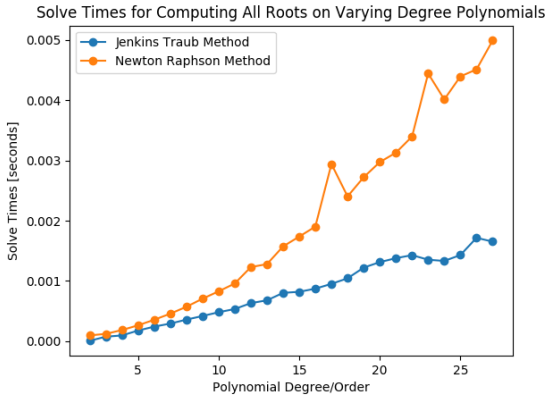

Figure 6: Solve Times for Computing All Roots for Increasing Degree Polynomials

In addition to the total number of function evaluations, the two algorithms were also compared on the amount of time needed to compute all roots of the polynomial. Again, the algorithms were tested and compared for functions of degrees $d \in [2, 27]$ with each iteration requiring roots to have numerical accuracy within $1e - 10$ of the real root. The results of timing the algorithms, shown in Figure 6, demonstrate that Jenkins-Traub is much more efficient overall at computing all roots of a polynomial. The Jenkins-Traub algorithm is significantly more efficient that Newton-Raphson for high degree polynomials, despite requiring more function evaluations than Newton-Raphson. In testing the two algorithms, for polynomials of degree $d > 15$, Newton-Raphson often hit the maximum threshold for the number of iterations imposed within the code rather than obtaining the root to the exact precision required. Therefore, solve times for that algorithm on high degree polynomials are likely even slower than is shown in Figure 6.

Additionally, the Jenkins-Traub algorithm was tested for global convergence and accuracy by defining polynomials with a large spread for the roots and high degrees as a closed form polynomial $p(x) = (x-\rho_1)*(x-\rho_2)*\cdots*(x-\rho_j)$, then computing the coefficients by expanding the polynomial, and solving for roots using the Jenkins-Traub algorithm. The Jenkins-Traub algorithm was able to successfully compute all roots for multiple different test polynomials with varying degree and spread of roots. In practice, this shows the ability for global convergence.

Overall, the 3-stage Jenkins-Traub algorithm performs significantly better than the Newton-Raphson algorithm for finding all roots for polynomials of high degrees. This is advantageous for many different applications, such as optimizations mentioned in Section 1, that require root finding for high degree polynomials, especially since finding a closed form solution for all roots is time-consuming to compute and other algorithms will perform slower in computing all roots to the same accuracy. Additionally, the Jenkins Traub algorithm is globally convergent and not reliant on an accurate initial guess for convergence, which makes the algorithm better than other iterative and bracketing methods for use in applications requiring root finding.

References

[1] Jenkins, M. A., and J. F. Traub. "An Algorithm for an Automatic General Polynomial Solver" Ruschlikon, Switzerland, 1967. `https://www.researchgate.net/publication/266847412_`

An_algorithm_for_an_automatic_general_polynomial_solver.

[2] Jenkins, M. A., and J. F. Traub. "A Three-Stage Algorithm for Real Polynomials Using Quadratic Iteration." SIAM Journal on Numerical Analysis, vol. 7, no. 4, 1970, pp. 545–566. JSTOR, www.jstor.org/stable/2949376.

[3] Jenkins, M. A., and J. F. Traub. "A Three-Stage Variable-Shift Iteration for Polynomial Zeros and Its Relation to Generalized Rayleigh Iteration." Numerical Math. Volume 14., 11 July 1968. https://Link.springer.com/Content/Pdf/10.10072FBF02163334.Pdf

[4] Kaw, Autar, and Egwu Kalu. "Numerical Methods with Applications" 1st Edition, 2008.

[5] Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2007), Numerical Recipes: The Art of Scientific Computing, 3rd ed., Cambridge University Press, page 470.

[6] Ralston, Anthony, and Philip Rabinowitz. "A First Course in Numerical Analysis: Second Edition (Dover Books on Mathematics)." 1978. McGraw-Hill Book Company.

7 Appendix 1: Code Listing for Jenkins Traub

```
function rpoly_stage1(p_coeffs, num_iters, epsilon)
       #make zero-shift polynomial sequences; approximately 5 iterations

       #Take the derivative
       powers = [length(p_coeffs)-i for i in 1:length(p_coeffs)]
       k_0 = powers .* p_coeffs
       k_0 = k_0[1:length(k_0)-1]

       #normalize the derivative function
       K_polys = k_0 ./ k_0[1]
       counter = 0

       #Iterate through fixed number of 0-shifts; terminate if root is found
       for i in 1:num_iters
              K_polys, x, root_found = next_step(p_coeffs, K_polys, 0, epsilon,
                  false)
              counter+=1
              if (root_found)
                     break
              end
       end
       return K_polys, counter
end

function rpoly_stage2(p_coeffs, K_curr, lin_const, num_iters, epsilon)
       #fixed shift stage 2
```

```
        t_next = t = Inf
        counter = 0

        #normalize coefficients and operate off of this
        if (K_curr[1] != 1)
                K_polys = K_curr ./ K_curr[1]
        else
                K_polys = K_curr
        end

        #Iteratively compute fixed-shifts until a sufficient root approximation is
             found (or maximum number of iterations exceeded)
        root_found = false
        while (true)
                t, t_prev = t_next, t
                K_polys, t_next, root_found = next_step(p_coeffs, K_polys,
                    lin_const, epsilon, true)
                if (root_found)
                        break
                end

                #Termination conditions combaring the difference in root
                    approximations
                counter += 1
                cond1 = abs(t-t_prev) <= .5*abs(t_prev) #is this a strictly less
                    than or a less than
                cond2 = abs(t_next-t) <= .5*abs(t)

                if ((cond1 && cond2) || (counter > num_iters))
                        break
                end
        end

        return K_polys, counter, root_found

end


function rpoly_stage3(p_coeffs, K_curr, lin_const, num_iters, epsilon)
        #stage 3 variable shifts

        #normalize coefficients and operate off of this
        if (K_curr[1] != 1)
                K_curr = K_curr ./ K_curr[1]
        end
```

```
        #Set initial estimate s_L for the root
        sl = lin_const - synth_div_eval(p_coeffs, lin_const)[2]/synth_div_eval(
            K_curr, lin_const)[2]
        sl_prev = Inf
        counter = 0

        K_polys = K_curr

        #Iterate until the absolute difference root approximations for 2
            iterations is < epsilon
        #Or also terminate if exceed maximum number of allowed iterations
        while ((abs(sl - sl_prev) > epsilon) && (counter < num_iters))
                p_prime, p_eval_at_sl = synth_div_eval(p_coeffs, sl)
                k_next, k_curr_at_sl = synth_div_eval(K_polys, sl)
                k_next = pushfirst!(k_next, 0)

                #check if we find a root by seeing if evaluated polynomial is 0
                if (abs(p_eval_at_sl) < epsilon)
                        return K_polys[length(K_polys)], sl, counter, true
                end

                k_next_iter = p_prime .- (p_eval_at_sl / k_curr_at_sl) .* k_next
                counter+=1
                sl, sl_prev = (sl - p_eval_at_sl / synth_div_eval(k_next_iter, sl)
                    [2]), sl
                K_polys = k_next_iter
        end
        if (counter > num_iters)
                #Failed to find a root for this iteration, returning false
                return 0,0,0,false
        end
        return K_polys[length(K_polys)], sl, counter, true
end

function newton(start, func_coeffs, deriv_coeffs, epsilon, num_iters)
        #Iterative newton raphson algorithm that terminates when aboslute error
            difference is < epsilon
        x_curr = start
        x_next = x_curr - evaluate_poly(func_coeffs, x_curr)/evaluate_poly(
            deriv_coeffs, x_curr)
        counter = 0
        while ((abs(x_next - x_curr)>epsilon) && (counter < num_iters))
                x_curr = x_next
                x_next = x_curr - evaluate_poly(func_coeffs, x_curr)/evaluate_poly(
                    deriv_coeffs, x_curr)
                counter += 1
```

```
        end
        return x_curr
end

function jenkins_traub_one_iter(p_coeffs, num_iterations, epsilon)
        #Compute the smallest root for the current coefficients list using the 3
            stage Jenkins Traub alg

        #Check if the constant coeff is a zero --> implies the root is zero
        if ((length(p_coeffs) > 0) && (p_coeffs[length(p_coeffs)]==0))
                return p_coeffs, 0, 0
        end

        #check if the current poly is only a constant.
        if (length(p_coeffs) < 2)
                return
        end

        #normalize coefficients and operate off of this
        if (p_coeffs[1] != 1)
                p_coeffs = p_coeffs ./ p_coeffs[1]
        end

        if (length(p_coeffs)==2)
                #linear function,so root is just the constant. return none for
                    coefs since nothing is left
                return 0.0, -p_coeffs[length(p_coeffs)], 0
        end
        total_func_evals = 0
        k_curr, num_iters = rpoly_stage1(p_coeffs, 5, epsilon)
        total_func_evals += 2*num_iters #next_step has 2 func evals
        println("Stage 1 with ",num_iters)

        #get a guess for initial linear constant s; approximate using standard
            method of a cauchy polynomial + netwons iter
        # new polynomial has coeffs that are the moduli of p_coeffs, but the last
            root is -1.
        all_pos = [abs(i) for i in p_coeffs] #TODO: need an element wise absolute
            value
        all_pos[length(all_pos)] = -all_pos[length(all_pos)]

        #Take the derivative
        powers = [length(all_pos)-i for i in 1:length(all_pos)]
        deriv = powers .* all_pos
        deriv = deriv[1:length(deriv)-1]
```

```
        beta = newton(1.0, all_pos, deriv, .01, 500)

        while (true)
                phi_rand = 2.0*pi*rand(1)[1]
                sl = cos(phi_rand)*beta

                k_curr, num_iters2, root_found = rpoly_stage2(p_coeffs, k_curr, sl,
                     num_iterations, epsilon)
                println("Done with stage 2 at ",num_iters2," iterations")
                total_func_evals += num_iters2*2 #next_step has 2 evals

                k_curr, sl, num_iters3, success = rpoly_stage3(p_coeffs, k_curr, sl
                    , num_iterations, epsilon)
                total_func_evals += num_iters3*3 #next_step and one extra eval
                println("Stage 3 finsihed with total iterations ",num_iters3)
                if (!success)
                        #didn't converge, try increasing maxiterations
                        num_iterations = 2*num_iterations
                else
                        return k_curr, sl, total_func_evals
                end
        end

end

function get_all_roots_JT(p_coeffs, epsilon, num_iterations)
        #Return the total set of coefficients for a polynomial and the root

        #Remove any leading zero coefficients since they are simply extra terms
            and will mess up normalization
        while ((length(p_coeffs) > 0) && (p_coeffs[1]==0))
                p_coeffs = p_coeffs[2:length(p_coeffs)]
        end

        if (length(p_coeffs) < 2)
                # function was just a constant, no roots
                return
        end

        roots = []
        total_function_evals = 0
        #Iteratively run Jenkins-Traub algorithm
        for i in 1:length(p_coeffs)-1
                upd_coeffs, s, num_f_evals = jenkins_traub_one_iter(p_coeffs,
                    num_iterations, epsilon)
```

```
                #update the polynomial by dividing out the linear term that factors
                     in the root
                p_coeffs, eval = synth_div_eval(p_coeffs, s)

                #increase count on number of times the polynomial func is evaluated
                total_function_evals += num_f_evals+1

                #store the root found for the iteration
                roots = push!(roots, s)
        end
        return roots, total_function_evals
end

function evaluate_poly(p_coeffs, val)
        #Stand alone polynomial evaluation method (no synthetic division)
        highest_power = length(p_coeffs)
        sum = 0
        for i in 1:highest_power
                power = highest_power-i
                sum+= p_coeffs[i]*val^power
        end
        return sum
end

function synth_div_eval(p_coeffs, lin_const)
        #Assumes leading coeff is 1.0 Decrease length of list by 1 (since dividing
             by x+/-c)


        #To only divide the entire function by x, then set lin_const = 0. This
            involves only truncating the function. Evaluation is merely the
            constant term
        if (lin_const == 0)
                return p_coeffs[1:length(p_coeffs)-1], p_coeffs[length(p_coeffs)]
        end

        #In all other cases, perform regular synthetic division
        syn_div = zeros(length(p_coeffs)-1)
        syn_div[1] = p_coeffs[1]
        for i=2:length(p_coeffs)-1
                syn_div[i] = p_coeffs[i] + syn_div[i-1]*lin_const
        end

        #Evaluate linear portion of polynomial -- taking advantage of remainder of
             the poly division
        eval = p_coeffs[length(p_coeffs)] + syn_div[length(syn_div)] .* lin_const
```

```
        return syn_div, eval
end

function next_step(p_coeffs, K_current, const_to_eval, epsilon, generate_t)
        #Update polynomial then evaluate at const; epsilon = threshold for roots

        p_prime, p_eval_at_c = synth_div_eval(p_coeffs, const_to_eval)
        k_next, k_curr_at_c = synth_div_eval(K_current, const_to_eval)

        root_found = false

        #check if we find a root by seeing if evaluated polynomial is 0
        if (abs(p_eval_at_c) < epsilon)
                root_found = True
        end

        # If evaluated shifted polynomial is too small, then increase it slightly
            so not dividing by 0
        if (abs(k_curr_at_c) < epsilon)
                k_curr_at_c += epsilon/100.0
        end

        t = 0.0
        if (generate_t == true)
                #update root approximation for the new shifted polynomial (not
                    needed for all stages)
                t = const_to_eval - p_eval_at_c/k_curr_at_c
        end

        #shift by zero once to represent the divide by 0
        k_deriv = pushfirst!(k_next, 0.0)

        #update the polynomial
        final = p_prime .- (p_eval_at_c / k_curr_at_c) .* k_deriv

        return final, t, root_found
end
```