Analysis on Methods: Root Finding and Interpolation

Seth Shelnutt

October 22, 2012

Abstract

Numerical Analysis, it is defined as "The branch of mathematics that deals with the development and use of numerical methods for solving problems". Since before the Greeks man has often sought to put some physical process or data into a mathematical representation. Since the shift into the modern atomic era this field of mathematics has become ever more important as computers are able to perform these numerical methods at levels never before seen. Two topics of this field are root finding and interpolation. Various methods of these topics will be examined and tested for accuracy and speed, with a focus on proper statistical analysis.

Contents

1 Introduction

When implementing the field of numerical analysis there are several factors to first consider. The very first question to answer is will this be done by hand or on a computer? This has a large impact on what methods can be considered to evaluate the problems. In this case a computer will be used. This thing brings up the question of what language to use for this analysis. There are three main languages which are considered standard. First for cases which require the most optimization and speed C is used. For a easy to use high level "language" matlab is often chosen. The problem with matlab is two fold, first it is commercial and only available on certain platforms. Second is that it's less of a language and more of an interactive terminal, thus making complex programming difficult.

In the middle ground between these two languages lies Python. Python offers the robust mathematical support of matlab through numeric python (numpy), scientific python (scipy) and symbolic python (sympy). With the extension of Matplotlib, python also offers a diverse graphical package. Python is a higher level language than C, and offers many features such as garbage collecting and memory allocation. Code is run inside an interpreter which means it can be run on any platform python is support. By the open source nature of python, nearly any existing platform, be it mainstream or embedded is supported. Python also offers various methods of optimization, be it in the code, or the interpreter. Python also offers support for inline C code if needed. Overall python's rugged and robust offerings make it the ideal language for numerical analysis and thus will be used in this paper.

2 Root Finding

For the purpose of this section the object is to find the intersection of $e^{\frac{-x}{5}}$ and $\sin x$. That is the roots of $e^{\frac{-x}{5}} = \sin x$.

2.1 Visual Inspection

The first and simplest method for root finding is a visual inspection. With a visual inspection a graph is shown and a user selects an x value that is approximately a zero. Here matplotlib is used to create the graph of the two functions and the user is asked to click on the point in which the zero is approximated. From this the x value is given and the difference in $e^{\frac{-x}{5}}$ - $\sin x$. See Figure 1.

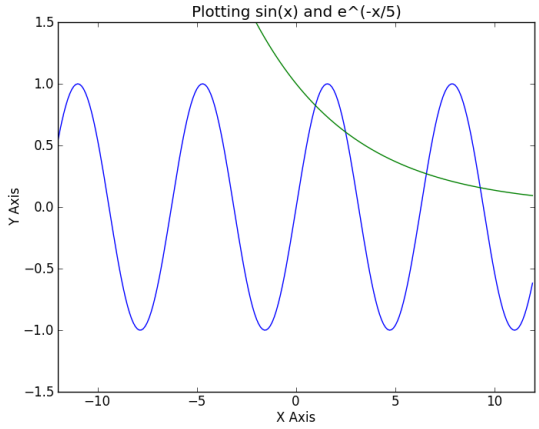

Figure 1:

Here the first of the four intersections is estimated to be, $x = 0.967742$ and the difference between the two functions is: **-0.000422834187542** . So depending on the interval and scale one can guess reasonably close, however the chances of visually selecting the intersection with a high degree of precision is unlikely. The remaining estimates on the intersections are at $x = \{2.479839, 6.572581, 9.274194\}$and the differences between the functions were **0.00552022494328** , **0.016768497586** and **-0.00646222207318**. By and large in most cases a higher degree of accuracy is sought after. Thus a more mathematical process such as bisection or Newton's method is to be examined.

2.2 Bisection

The bisection method stems from bisecting an interval in each iteration. A function and an interval is given. The interval is then bisected and a sign change is looked for on either side of midpoint. On the side that there is a sign change a new interval is formed between that end point and the midpoint. This then forms a new interval in which the next iteration takes place on. Mathematically we are doing: Given a bound $[x_1, x_2]$, $x_3 = \frac{x_1+x_2}{2}$, $f(x_1)f(x_3)\begin{cases} < 0 & x_2 = x_3 \\ > 0 & x_1 = x_3 \\ = 0 & f(x_3) = 0 \end{cases}$.

Now that a formula is established there are two lingering questions. What is the convergence rate and what is error in this method? The maximum error is given by $\frac{\|b-a\|}{2^i}$, where i is the number of iterations run. This yields the convergence rate of $\frac{b-a}{2^i}$, or simply $\frac{1}{2}$. This resultant linear convergence is good, but when finding the zero of $e^{\frac{-x}{5}} = \sin x$, between 0 and 2 with python it takes 21 iterations and 0.000659 seconds. The zero is estimated by $x = 0.968319892883$ While in this small instance that does not seem like a significant amount, on larger and more complex systems this could end up taking significant time and number of iterations. A quadratic approach was developed by Newton to quicken root finding and will be discussed in the next section.

2.3 Newton's Method

Newton first developed his method for root finding in De analysi per aequationes numero terminorum infinitas written in 1669 and in De metodis fluxionum et serierum infinitarum written in 1671. Both of these description vary from the modern Newton's method as Newton failed to make the connection to Calculus and instead is based on a pure algebraic implementation. The modern interpretation of Newton's method and the one implemented in this project is $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ where you test for $f(x_{n+1}) = 0$. Here as with the bisection the function the zero is being determined is $e^{\frac{-x}{5}} - \sin x = 0$. The convergence

of Newton's method is what gives it an advantage over the bisection method. The convergence is given by $lim_{n\to\infty}\frac{x_{n+1}-\alpha}{(x_n-\alpha)^2} = \frac{f^2(a)}{2}$. Thus Newton's method yield quadratic convergence on the root.

When implementing Newton's method in python intriguing results are found. Starting at the same initial point as in bisection, $x_1 = 0$, it takes only 4 iterations to find the zero at $x = 0.968319798371$. How it takes 0.1072 seconds to calculate this. This is an astonishing 162 times longer than the bisection method. Even though the number of iteration was decreased by over 80% the time it takes to calculate the derivative yields in a more timely process. The the optimization section a more detailed analysis of this time differential and methods to improve speed will be examined.

3 Interpolation

Interpolation is another of the most commonly used applications of numerical analysis. Interpolation is defined as the construction of a new set of data points, or equation, within a defined ranged give known data points. There are several methods and techniques for interpolation ranging in accuracy, speed, and power. The function that is being interpolated in this case is $f(x) = \frac{1}{1+x^2}$ over the range $[-5, 5]$.

3.1 Lagrange Polynomials

Lagrange polynomials were first published by Waring in 1779, only to be rediscovered by Euler in 1783, and published by Lagrange in 1795. Lagrange polynomials are unique and given by $P_n(x) = \sum_{i=0}^{n} f(x_i)L_n(x)$, where $L_n(x) = \prod_{j=k,j=0}^{n} \frac{x-x_k}{x_j-x_k}$.

When constructing interpolating polynomials, there is an inherit problem with having a better fit and having a smooth fitting function. The more data points in the interpolation, the higher the degree of the resulting polynomial, and therefore the greater oscillation it will exhibit between the data points. At the same time the accuracy at the data points will be very high.

As always when estimate is used, the error value is important. How good and how close is our estimations? The error can be calculated by $f(x) - P_n(x) = \prod_{x=0}^{n} \frac{x-x_n}{(n+1)!} f^{n+1}(\xi)$.

For the function $f(x) = \frac{1}{1+x^2}$ over the range $[-5, 5]$, three separate Lagrange polynomials will be examined. First with a degree of n=5, then n=10 and lastly n=20. With each of these n+1 points will be selected on an even distribution between $[-5, 5]$. See figure 2 for a graph showing all three polynomials.

When n = 5, the number of data points present is six. Using evenly space points, it takes 0.3169 seconds to calculate the polynomial of $P_5 = 0.0019231x^4 - 0.069231x^2 + 0.567308$. **When n=10, it take 0.9429 seconds to calculate the polynomial of** $P_{10} = -.000023x^{10} - 1.694066e^{-5}x^9 + 0.001267x^8 - 0.024412x^6 - 4.33681e^{-19}x^5 + 0.197376x^4 + 2.151057e^{-16}x^3 - 0.674208x^2 + 1.994932e^{-17}x + 1.0$.**A Lagrange polynomial of order 20 it takes over 5.1059 seconds to calculate. The resulting polynomial is** $P_{20} = 2.72817e^{-19}x^{20} - 2.65314e^{-7}x^{18} - 4.23516e^{-22}x^{17} + 1.07425e^{-5}x^{16} - 4.74338e^{-20}x^{15} - 0.000236x^{14} + 7.21862e^{-16}x^{13} + 0.003102x^{12} + 5.57367e^{-15}x^{11} - 0.025114x^{10} + 1.40582e^{-14}x^9 + 0.126253x^8 - 7.21645e^{-16}x^7 - 0.391630x^6 - 3.33067e^{-16}x^5 + 0.753354x^4 + 2.94209e^{-15}x^3 - 0.965739x^2 - 8.32667e^{-17}x + 1.0$. **There are two main things to be taken from these three Lagrange polynomials. First is that as expected a higher degree polynomial leads to a much better fit at the data points. However as can be seen from the graph at the end points the variation becomes extreme. Second the time taken to calculate the nth degree polynomial is exponential. To calculate a 10th degree polynomial takes just under 1 seconds, but a 20th degree takes over 5 seconds. The usefulness of Lagrange polynomials comes into question with larger values of n. Later in the Optimization section methods to improve on the speed of Lagrange interpolation will be examined.**

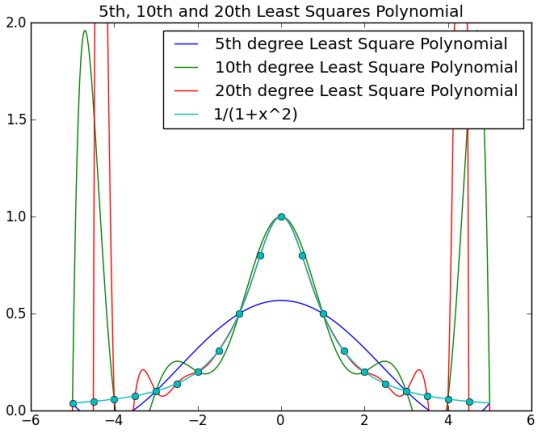

Figure 2:

3.2 Piecewise Interpolation

Linear or Piecewise Interpolation is the simplest form of interpolation. It is linear and does not create a polynomial. Instead the data points are simply "connected". It has the benefit of being a simple linear calculation between only two points. Speed there for is very high and accuracy can be increased by using additional points. Unlike Lagrange polynomials there is no significant disadvantages to more points, such as increased variation. The main issue with linear interpolation such as this, is in the case of data which is a polynomial such as here, it simply doesn't provide a good

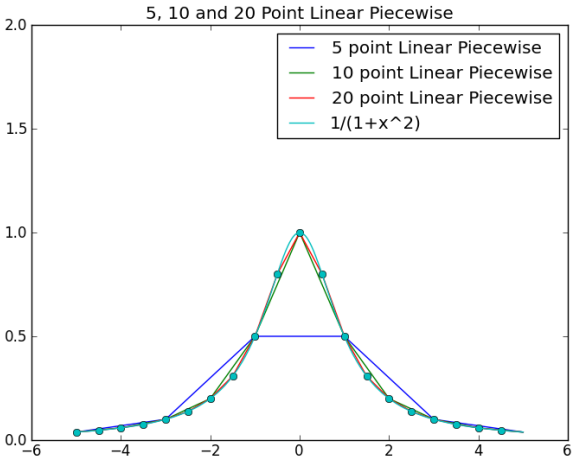

Figure 3:

fit. Thus a larger data set is essential for a more accurate fit.

As with Lagrange once again the interpolation is $f(x) = \frac{1}{1+x^2}$ over the range $[-5, 5]$, with 5, 10 and 20 data points. See figure 3 for a graph of all three functions. The time taken to calculate n=5 was 0.00224 seconds, while n=10 was 0.00835 seconds, and n=20 was 0.01703 seconds. Unsurprisingly this was significantly faster than Lagrange interpolation. Below is the piecewise functions for these three interpolations.

$$\textbf{When n = 5, } I_5(x) = \begin{cases} 0.030769x + 0.115385 & for\ -5 < x \leqslant -3 \\ 0.2x + 0.5 & for\ -3 < x \leqslant -1 \\ -0.5 & for\ -1 < x \leqslant 1 \\ -0.2x - 0.3 & for\, 1 < x \leqslant 3 \\ -0.030769x - 0.007692 & for\, 3 < x \leqslant 5 \end{cases}$$

$$\textbf{When n = 10, } I_{10}(x) = \begin{cases} 0.020362x + 0.063349 & for\ -5 < x \leqslant -4 \\ 0.041176x + 0.105882 & for\ -4 < x \leqslant -3 \\ 0.1x + 0.2 & for\ -3 < x \leqslant -2 \\ 0.3x + 0.4 & for\ -2 < x \leqslant -1 \\ 0.5x & for\ -1 < x \leqslant 0 \\ -0.5x - 1 & for\, 0 < x \leqslant 1 \\ -0.3x - 0.2 & for\, 1 < x \leqslant 2 \\ -0.1x & for\, 2 < x \leqslant 3 \\ -0.041176x + 0.023529 & for\, 3 < x \leqslant 4 \\ -0.020362x + 0.022624 & for\, 4 < x \leqslant 5 \end{cases}$$

When n = 20, $I_{20}(x) =$

$$\begin{cases} 0.017195x + 0.047511 & for\ -5 < x \leqslant -4.5 \\ 0.023529x + 0.058824 & for\ -4.5 < x \leqslant -4 \\ 0.033296x + 0.074362 & for\ -4 < x \leqslant -3.5 \\ 0.049056x + 0.096226 & for\ -3.5 < x \leqslant -3 \\ 0.075862x + 0.127586 & for\ -3 < x \leqslant -2.5 \\ 0.124138x + 0.172414 & for\ -2.5 < x \leqslant -2 \\ 0.215385x + 0.230769 & for\ -2 < x \leqslant -1.5 \\ 0.384615x + 0.269231 & for\ -1.5 < x \leqslant -1 \\ 0.6x + .01 & for\ -1 < x \leqslant -0.5 \\ 0.4x - 0.6 & for\ -0.5 < x \leqslant 0 \\ -0.4x - 1 & for\ 0 < x \leqslant 0.5 \\ -0.6x - 0.5 & for\ 0.5 < x \leqslant 1 \\ -0.384615x - 0.115385 & for\ 1 < x \leqslant 1.5 \\ -0.215385x + 0.015385 & for\ 1.5 < x \leqslant 2 \\ -0.124138x + 0.048276 & for\ 2 < x \leqslant 2.5 \\ -0.075862x + 0.051724 & for\ 2.5 < x \leqslant 3 \\ -0.490566x + 0.047170 & for\ 3 < x \leqslant 3.5 \\ -0.033296x + 0.041065 & for\ 3.5 < x \leqslant 4 \\ -0.023529x + 0.035294 & for\ 4 < x \leqslant 4.5 \\ -0.017195x + 0.030317 & for\ 4.5 < x \leqslant 5 \end{cases}$$

3.3 Raised Cosine Interpolation

Raised cosine interpolation stems from attempting to find a better fit than linear interpolation and Lagrange. Using cosine provides a "nicer", that is smoother, fit between points than linear and higher order Lagrange. Since cosine is a continuously smooth curve, interpolating between points yields a continuous function.

The basic method for interpolating between two points with raised cosine, is to fit a cosine function to the two points. The formula is given by $C_n(x) = y_0 + (\frac{-cos(\pi t)}{2} + \frac{1}{2})(y_1 - y_0)$ **for each point** $(x_1, y_1), (x_2, y_2)$.

Overall this method is quicker in the time took to calculate the piecewise function. For n = 5, it took only 0.0311 seconds, while for n=10 0.00501, and n=20 0.01048 . As can be seen the

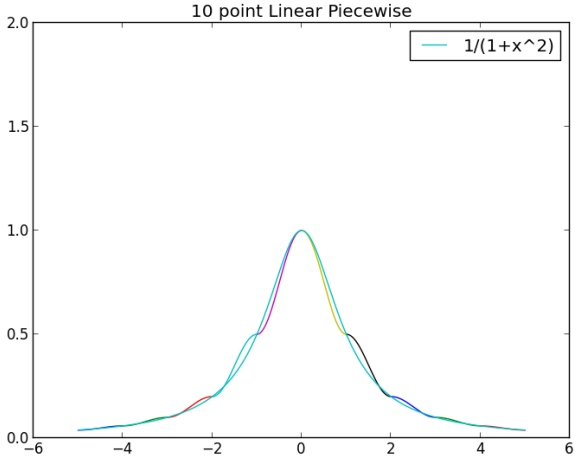

Figure 4:

function is so quick that there is no statistical significance to the change in time taken. The main resulting disadvantage of raised cosine over Lagrange is that while the function is continuous it is not differentiable. As seen in figure 4, when using n=10, the function is very smooth at all points and fits the graph reasonably well.

$$\textbf{When n = 5, } I_5(x) = \begin{cases} -0.021757\cos(\pi x) + 0.047473 & for\ -5 < x \leqslant -3 \\ -0.2\cos(\frac{\pi x}{2} - 4.71239) + 0.3 & for\ -3 < x \leqslant -1 \\ -0.5 & for\ -1 < x \leqslant 1 \\ 0.2\cos(\frac{\pi x}{2} + \frac{\pi}{2}) + 0.3 & for\, 1 < x \leqslant 3 \\ 0.021757\cos(\pi x) + 0.909879 & for\, 3 < x \leqslant 5 \end{cases}$$

$$\textbf{When n = 10, } I_{10}(x) = \begin{cases} 0.010181\cos(\pi x) + 0.048643 & for\ -5 < x \leqslant -4 \\ -0.020588\cos(\pi x) + 0.079412 & for\ -4 < x \leqslant -3 \\ 0.05\cos(\pi x) + 0.15 & for\ -3 < x \leqslant -2 \\ -0.15\cos(\pi x) + 0.35 & for\ -2 < x \leqslant -1 \\ 0.25\cos(\pi x) + 0.75 & for\ -1 < x \leqslant 0 \\ 0.25\cos(\pi x) + 0.75 & for\, 0 < x \leqslant 1 \\ -0.15\cos(\pi x) + 0.35 & for\, 1 < x \leqslant 2 \\ 0.05\cos(\pi x) + 0.15 & for\, 2 < x \leqslant 3 \\ -0.020588\cos(\pi x) + 0.079412 & for\, 3 < x \leqslant 4 \\ 0.010181\cos(\pi x) + 0.048643 & for\, 4 < x \leqslant 5 \end{cases}$$

$$\textbf{When n} = \textbf{20},\ I_{20}(x) = \begin{cases} -0.008597\cos(\pi x) + 0.038462 & for\ -5 < x \leqslant -4.5 \\ -0.005882\cos(\pi x + 14.1377) + 0.052941 & for\ -4.5 < x \leqslant -4 \\ 0.016648\cos(\pi x) + 0.075472 & for\ -4 < x \leqslant -3.5 \\ -0.012264\cos(2\pi x + 10.9956) + 0.087734 & for\ -3.5 < x \leqslant -3 \\ -0.037931\cos(\pi x) + 0.09999 & for\ -3 < x \leqslant -2.5 \\ -0.031034\cos(2\pi x + 7.853982) + 0.168966 & for\ -2.5 < x \leqslant -2 \\ 0.107692\cos(\pi x) + 0.307692 & for\ -2 < x \leqslant -1.5 \\ -0.096154\cos(2\pi x + 4.71238) + 0.403846 & for\ -1.5 < x \leqslant -1 \\ -0.3\cos(\pi x) + 0.5 & for\ -1 < x \leqslant -0.5 \\ -0.1\cos(2\pi x + 1.57080) + & for\ -0.5 < x \leqslant 0 \\ -0.2\cos(\pi x) + 0.8 & for\ 0 < x \leqslant 0.5 \\ 0.15\cos(2\pi x - 1.57079) + 0.65 & for\ 0.5 < x \leqslant 1 \\ 0.192308\cos(\pi x) + 0.5 & for\ 1 < x \leqslant 1.5 \\ 0.053846\cos(2\pi x - 4.71239) + 0.253846 & for\ 1.5 < x \leqslant 2 \\ -0.062068\cos(\pi x) + 0.137931 & for\ 2 < x \leqslant 2.5 \\ 0.018966\cos(2\pi x - 7.853981) + 0.118966 & for\ 2.5 < x \leqslant 3 \\ 0.024528\cos(\pi x) + 0.1 & for\ 3 < x \leqslant 3.5 \\ 0.008324\cos(2\pi x) + 0.067148 & for\ 3.5 < x \leqslant 4 \\ -0.011765\cos(\pi x) + 0.047059 & for\ 4 < x \leqslant 4.5 \\ 0.004299\cos(2\pi x - 14.1372) + 0.042760 & for\ 4.5 < x \leqslant 5 \end{cases}$$

3.4 Least Squares Approximation

Least squares approximation is a method of fitting a graph to data so as to minimize the sum of the squares of the differences between the observed values and the estimated values. In comparison to Lagrange, linear and cosine piecewise, it can be see than linear offers a much smoother curve that Lagrange, comparable to cosine at low orders. However as you increase the order, just as with Lagrange the smoothness at towards the end points begins to breakdown and the function goes to extremes. At lower values of n, least squares can be favorable to cosine, as it offers a differentiable polynomial, along with the smoothness between points.

The main disadvantage to least squares approximation is that instead of matching at given data points, it's main focus is for the sum of the squares of observed minus estimated to be zero. In this the overall curve might be a good fit, but at individual points there is no guarantee for the curve to fit exact as with Lagrange.

The formula for least squares is given by $y = Xa \implies X^T y = X^T X a \implies a = (X^T X)^{-1} y$**, where the residual is given by** $R^2 = \sum_{i=1}^{n} [y_i - (a_0 + a_1 x + ... + a_k x_i^k]^2$**. An**

ideal fit would have the residual equal to 1.

As with previous interpolations, the function to be interpolated is $f(x) = \frac{1}{1+x^2}$ over the range $[-5, 5]$, with $n = \{5, 10, 20\}$. The first thing one notices immediately is the speed at which this interpolation runs. With order 5, it takes 0.000562 seconds, order 10 takes 0.000701 seconds and order 20 takes 0.001019 seconds. This is by far the quickest interpolation method implemented. The question that remains is how well does the data fit? As seen in figure 5, with n=5, it matches the five points but lacks overall fit. When n=10 seems to be the best fit, as n=20 results in oscillation at the end points.

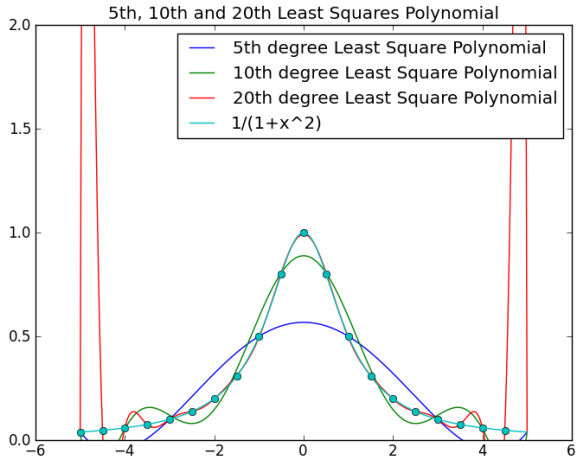

Figure 5:

The resulting formulas and residuals are as follows: When n=5, the residual is $r^2 = 1$ and $L_5(x) = 0.001923x^4 - 6.939e^{-17}x^3 - 0.06923x^2 + 9.929e^{-16} + 0.5673$. When n=10, the residual is $r^2 = 0.9985$ and $L_{10}(x) = 1.027e^{-18}x^9 + 5.239e^{-5}x^8 - 5.239e^{-17}x^7 - 0.00279x^6 + 8.608e^{-16}x^5 + 0.04931x^4 - 5.163e^{-15}x^3 - 0.3413x^2 + 8.642e^{-15}x + 0.888$. When n=20, the residual is $r^2 = 0.9999$ and $L_{20}(x) = -1.838e^{-17}x^{19} - 4.477e^{-09}x^{18} + 1.738e^{-15}x^{17} + 4.303e^{-07}x^{16} - 6.766e^{-14}x^{15} - 1.715e^{-05}x^{14} + 1.409e^{-12}x^{13} + 0.0003693x^{12} - 1.703e^{-11}x^{11} - 0.0047x^{10} + 1.217e^{-10}x^9 + 0.03645x^8 - 5.002e^{-10}x^7 - 0.1723x^6 + 1.093e^{-09}x^5 + 0.4905x^4 - 1.07e^{-09}x^3 - 0.8476x^2 + 3.054e^{-10}x + 0.9915$.

4 Optimizations

The nature of python as described earlier yields many possible optimizations to be done to the algorithms and implementations of the methods used in this paper. Due to the constraints and scope of this project the main focus will be improvements in python code and not implementing inline c, or other such optimizations. Along with this, there will be two main focuses. First on improving newton's method in performance and second in decrease the time taken to calculate Lagrange polynomials. These two method are prime examples where without proper optimization, when data becomes large the time it takes to calculate these functions grows exponentially.

4.1 Newton's Method Optimizations

The most logical computerized optimization for newton's method is to estimate the first derivative which is required. This numerical derivative is where the time is spent in the calculations. The most wide used estimation, the previous two points, yields the secant method.

In the secant method $f'(x)$ is replaced with $\frac{f(x_{n-1})-f(x_{n-2})}{x_{n-1}-x_{n-2}}$. This results in $x_{n+1} = x_n - \frac{f(x_n)(x_n-x_{n-1})}{f(x_n)-f(x_{n-1})}$. When this is implemented in python, a very large speed improvement is found. With two starting points of 0 and 0.5, for the function of $e^{\frac{-x}{5}} - \sin x = 0$ it takes 5 iterations and only 0.006321 seconds to find the zero of $x = 0.96832$. While this is one more iteration than newton's method it is 17 times faster than newton's method. While it might seem like fractions of seconds do not matter. In more complex functions it will take longer to estimate the derivative.

Using this secant method saves the computer from numerical derivation, which as shown here is a much slower process than simple multiplication, and addition. Depending on the architecture and the extension your computer supports, there might be built in support for hardware differentiation. However on the standard x86 only multiplication and addition are built in instructions. This means it takes only one cycle to do multiplication, however derivatives are broken down and take multiple cycles. Here the clear advantage in terms of speed is using the simple algebraic secant method even though it take one additional iteration.

4.2 Lagrange Optimization

As seen in the previous section on Lagrange polynomials, the time it takes to calculate the Lagrange is exponential based on the order. Here optimizations will be taken to decrease this exponential change in order to make higher order Lagrange polynomials more feasible in terms of time taken. The immediate and obvious optimization is to parallelize the calculation of each $L_n(x)$. This can be done by spawning a new thread or process for each calculation. Due to the nature of python spawning new processes is a much more elegant form. Since the host OS this is being run on is Linux, the overhead of an additional process is negligible compared to threads. If a different host OS was used this type of optimization might not be idea based on the overhead of new processes verses spawning new threads.

In the code attached in appendix A, the function Lagrange2() implements this multiprocess polynomial approach. In calculating the same 20 degree polynomial it takes a blistering 0.21240 seconds to calculate the polynomial instead of the over 5 seconds if done linearly. With nearly all modern computers and designs going to parallelization and multicore, multiprocessor, it seems only logical to

find ways to run stages in parallel if possible. Lagrange lends itself to being massively parallel. This parallel implementation of Lagrange is over 24 times faster than the serial implementation.

5 Conclusion

The goal of this paper was to implement various methods of numerical analysis. It sought to study the factor and difficulties in implementing these methods and formulas in the language of python. First root finding methods were studied for both speed and iterations. It was found that while Newton's method had the fewest iterations it took the most wall time to calculate based on the fact that doing numerical derivatives are not as fast as algebraic operations. The midpoint and ultimately the secant method were far quicker with this simple function of $e^{\frac{-x}{5}} - \sin x = 0$. Thus what might take the fewest iterations or be the fastest by hand does not always translate to the faster method on a computer.

The second field examined was interpolation. In the field of interpolation there are various methods that one might use depending on their needs. First Lagrange polynomials were examined and it was noticed that no only is there an issue with smoothness at higher orders but also with the time it takes to calculate these. Piecewise linear interpolation was found to be incredibly fast but lacking in smoothness and accuracy even with a high number of points. Raised cosine interpolation solved the smoothness issue, but results in a piecewise function that is continuous but not differentiable.

Least squares approximation yields a function that is continuous and differentiable as Lagrange is. At lower degrees of order it tends to be more smooth than Lagrange is, but the trade off is that there is no guarantee that at the given data points the function will be equal. This is a result of the methodology which is that the squares of the sum of the difference from the given data points is to be zero.

In an effort to find a interpolation function which is smooth and fast, the Lagrange method was re-examined. It was determined that if the code was parallelized it could result in a net increase in the speed while maintaining the same level of smoothness. The speed up was over 24 times that of linear operations.

When implementing numerical analysis methods of any topic it is highly important to examine and cater to the platform that is being used. One must take into account if parallelization is possible, and where or not calculus based methods would be faster than algebraic approximations. In many cases it will not make a significant difference. In other cases where the data is large enough the exponential speed up would be much welcome.

6 Appendix A