Vol 1, Issue 6, pages 1-12 ***High School Research Journal***

COMPARISON OF LAGRANGIAN AND MULTIVARIATE INTERPOLATION

Vinny Pagano

Long Beach Senior High School, 322 Lagoon Drive West, Long Beach, New York, 11561, 516-325-5078, wilddrummer@optonline.net

ABSTRACT

This project compared the "efficiency" between Lagrangian Polynomial Interpolation and Multivariate Polynomial Interpolation. The results determined what one should use to attain an equation that most resembles a set of points. Additionally, an autonomous process of interpolation was constructed for both methods using Python. The modules Numpy, Sympy, and Scipy were integrated into some of the mathematical processes that had to be implemented into the code. These included the following: Rectification, the Vandermonde Matrix, intricate integration, substitution of variables for both univariate and bivariate instances, and the determination of the roots of a polynomial. For every equation that received the Lagrangian Interpolation treatment, the same initial equation was used and received the Multivariate Interpolation treatment. The variable "t," which denotes the amount of times the function had to be further partitioned and interpolated, was compared amongst both procedures. There were many interpolated equations that did not equate to the original function. To represent these equations, the y-values of the functions at an arbitrarily high x-value were acquired because continuity and expectedness of each equation at this value can be assumed. Additionally, the percent error of the most *accurate* y-values for each equation type for both methods and the y-values of every original equation were analyzed through a matched pairs t-test. For the purpose of this investigation, the research question should be constructed as follows: *Which Method of Polynomial Interpolation is More Efficient/Accurate: Lagrangian Interpolation or Multivariate Interpolation?*

Keywords: Polynomial, Lagrangian, Multivariate

INTRODUCTION

Polynomial Interpolation has multiple applications in mathematics, physics, engineering, and architecture. For example, one would use it for the basis of algorithms used to compute the trajectory that a self-driving car might travel when rounding a curve. The concept of interpolation allows people to represent the relationship of a given set points on a plane with a single, continuous equation. The more points that are being interpolated, the more accurate the equation will be. In two dimensional space, the method of interpolation is Lagrangian Polynomial Interpolation. In other dimensions (three dimensional space was only needed in this study), there is the Multivariate Interpolation Method, which utilizes the Vandermonde Matrix (Harder, 2016). In this study, two-dimensional multivariate interpolation was used because the results would have a larger impact on the science and mathematics community when determining which method to use for interpolating data; two methods corresponding to different dimensions cannot be compared for significant analysis, as they are not both utilized for the same practice. The application of polynomial interpolation is immense. For example, computers or robots of all kinds could use it as part of AI navigational functions. It also has the potential reliability for accurate estimation of geodesic/relativistic physics in the space-time continuum. There are even more abstract usages of the procedure, like in the recognition of patterns in animal movement (Tremblay, 2006). It is thus quite superfluous to epitomize that calculating the efficiencies of these two methods of interpolation will allow engineers and mathematicians in the real world to quickly and easily choose a methodology that will best suit their needs.

Lagrangian Interpolation

Lagrangian (or Lagrange) Interpolation was first discovered by Edward Waring in 1779, but it is named after Joseph Louis Lagrange. The formula for Lagrange Interpolation is the following:

$$p(x) = \sum_{i=0}^{n} \left(\prod_{0 \le j \le n, j \ne i}^{n} \frac{x - x_j}{x_i - x_j} \right) y_i \quad \text{, where}$$

$p(x)$ is the interpolated equation that has been derived from a set of points,

For each of the n equations:

- x_i is the “reference frame” of the equation
- y_i is the “reference frame” of the equation
- x_j contains all the points except for each x_i that is being viewed

When each x_i is substituted for x, the numerator and denominator cancel out, leaving us with the output value y_i. The summation signifies that we are essentially adding n equations together to arrive at $p(x)$, an important feature to understanding the method.

Multivariate Interpolation

Multivariate Interpolation is named after Alexandre-Théophile Vandermonde, and the process utilizes the Vandermonde Matrix. The equation takes the form:

$$p(x) = \sum_{i=0}^{n} c_i x^i, \quad \alpha_m^n = \sum_{i=0}^{n} x^i, \qquad V = \begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \cdots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \cdots & \alpha_m^{n-1} \end{bmatrix}, p(\alpha_i) = y_i \text{ for } i = 1, \ldots, m \,^{**}$$

Each column of the Vandermonde Matrix is c_n. Hence, Vc = y, and V is a system of linear equations set to equal y_m (Harder).[1]

HYPOTHESES

A matched pairs t-test was used since both methods came from the same equations.

$$H_0 : \mu_L - \mu_M = 0$$

There is no significant difference between the means of the data*

$$H_A : \mu_L - \mu_M \neq 0$$

There is a significant difference between the means of the data*

Allow a significance level $\alpha = 0.05$

We took 30 Trials, with 3 randomly generated values per trial

*Percent errors from the methods' *best* y-value to each true y-value (further discussed in "Methodology")

METHODOLOGY

M0. CONSTRUCTION OF GENERAL INTERPOLATION PROCESS TO BE CODED IN PYTHON

A program will be constructed using Python, which abides the following format:

- Lagrangian Polynomial Interpolation
  - Generate an equation that is a function
  - Take all of the relative extrema of the function (relative minima, relative maxima) and roots

[1]** Image from https://en.wikipedia.org/wiki/Vandermonde_matrix

    - Label these points as coordinates, as they will be used for the interpolation
    - Perform the method of interpolation on the set of points (this will be done using Python)
    - If the original equation is not achieved
        - Add 1 to a value “t” (this will be used for the data analysis)
        - Take the coordinates that are the “midpoints” between each of the points
        - Repeat the process of interpolation again
    - Otherwise
        - List the value “t” (this will be used for the data analysis)
        - Repeat process with a new function
- Multivariate Polynomial Interpolation
    - Every equation that is used for Lagrangian Interpolation will also be used for this method
    - Take all of the relative extrema of the function (relative minima, relative maxima) and roots
    - Perform the method of interpolation on the set of points (this will be done using Python)
        - The interpolation equation will be that for the closest perfect square less than that number, and the next n terms such that n is the difference between the perfect square and the number
    - If the original equation is not achieved
        - Add 1 to a value “t2” (this will be used for the data analysis)
        - Take the coordinates that are the “midpoints” between each of the points
        - Repeat the process of interpolation again
    - Otherwise
        - List the value “t2” (this will be used for the data analysis)
        - Repeat process with a new function following the previous guidelines

Furthermore, there are specific guidelines that must be followed in order to be able to significantly continue with the process of interpolation:

1. Every starting equation is analytic, meaning that it is continuous and differentiable
2. Every starting equation has at least three *defining points* (I denote this to include relative extrema and x-intercepts of equation)
3. Every starting equation has at least one real root

A process of polynomial rectification has been constructed which can be applied to all interpolated equations.

M1. STURM'S THEOREM AND ITS POTENTIAL APPLICATIONS

We first determine whether a randomly generated function has real roots using Sturm's Theorem. Let $p_0, \ldots, p_n$ be the Sturm chain of a square-free polynomial p, and $\sigma(\xi)$ denote the number of sign changes (ignoring zeros) in the sequence $p_0(\xi), p_1(\xi), p_2(\xi), \ldots, p_n(\xi)$

We define a Sturm chain of a polynomial to be:

$$p_0(x) = p(x),$$

$$p_1(x) = p'(x),$$

$$p_k(x) = 0,$$

$$p_n(x) = \; -[p_{n-2}(x)\, mod\, p_{n-1}(x)] \text{ for } 2 \le n < k,\ n \in Z$$

Square-free means that the polynomial does not have a factor of the form v^2 (multiplicities do not repeat). The number of distinct roots of p is $\sigma(-\infty)$ - $\sigma(\infty)$. Non-square-free polynomials are irrelevant because $-\infty$ and $+\infty$ will not be multiple roots of p. Next, we must define the procedure of conditions at which the equations will be held to undergo the extraction of curvilinear midpoints between each of the coordinates. To first approach this, polynomial rectification, which is the curvilinear (canonical) distance between two points, is formulated.

M2. DEFINITION OF POLYNOMIAL RECTIFICATION AND ITS IMPLEMENTATION

Given two generally distinct points $(x_1, f(x_1))$ and $(x_2, f(x_2))$, and curve $y = f(x)$,

$$L(x_1, x_2) = \int_{x_1}^{x_2} \sqrt{(dx)^2 + (d(f(x))^2} = \int_{x_1}^{x_2} \sqrt{1 + (\tfrac{df(x)}{dx})^2} dx\,.$$

This is the only mathematical process that can be used to approximate the curvilinear midpoint between two points. With respect to each method of interpolation, terminate the procedure if, at an arbitrarily high x-value,

$$|y_o - y_i| > |y_o - y_{i-1}|,$$

where y_o = Original function's y-value, y_i = Current interpolated equation's y-value, and y_{i-1} = Previous interpolated equation's y-value.

Finally, to take the curvilinear midpoint between two distinct points, we shall elucidate a case of "Computational Limits." This method was applied to all pairs of x-values (from left to right) in each method of interpolation. There were thus 2n-1 more points created every time.

M3. MATHEMATICAL REPRESENTATION AND DEFINITION OF COMPUTATIONAL LIMITS

Consider a set S containing p distinct pairs of points, $[x_1, x_2] \in S$, $x_M = (x_1 < x_2) \in \mathbb{R}$, and let 0 < n < 1. $S = \bigcup_{f=1}^{p} P_f$, where the process P_f is denoted by two incontrovertibly successive methods ("computational limits"):

$$C_i : x_M \left| \left(x_M = x_M + \sum_{j=1}^{a} \left(\tfrac{n_j}{10^{(f-1)}} \right) \wedge \int_{x_1}^{x_M} \sqrt{1 + (\tfrac{df(x)}{dx})^2} dx > \int_{x_M}^{x_2} \sqrt{1 + (\tfrac{df(x)}{dx})^2} dx \right) \right., \forall i \in \bigcup_{m=1}^{\infty} 2m - 1$$

$$C_i : x_M \left| \left(x_M = x_M - \sum_{j=1}^{a} \left(\tfrac{n_j}{10^{f}} \right) \wedge \int_{x_1}^{x_M} \sqrt{1 + (\tfrac{df(x)}{dx})^2} dx < \int_{x_M}^{x_2} \sqrt{1 + (\tfrac{df(x)}{dx})^2} dx \right) \right., \forall i \in \bigcup_{m=1}^{\infty} 2m$$

Repeat first clause ad infinitum, creating a singularity at the value for the midpoint. It should thus follow that

$$\lim_{x \to M} \int_{x_1}^{x_M} \sqrt{1 + (\tfrac{df(x)}{dx})^2} dx - \int_{x_M}^{x_2} \sqrt{1 + (\tfrac{df(x)}{dx})^2} dx = 0,$$

which is the desired outcome; there is a point M where it is exactly half the length of the curve between two other points. This concludes all mathematical procedures implemented in the code.

Many equations, when interpolated, *almost* emulated the original function. Result is due to the method's failure to make terms written in e-scientific notation negligible, since they weren't included in the original equation. Fig(1a) demonstrates this outcome with the equation $y = 5x^3 + 4x^2 - .2$.

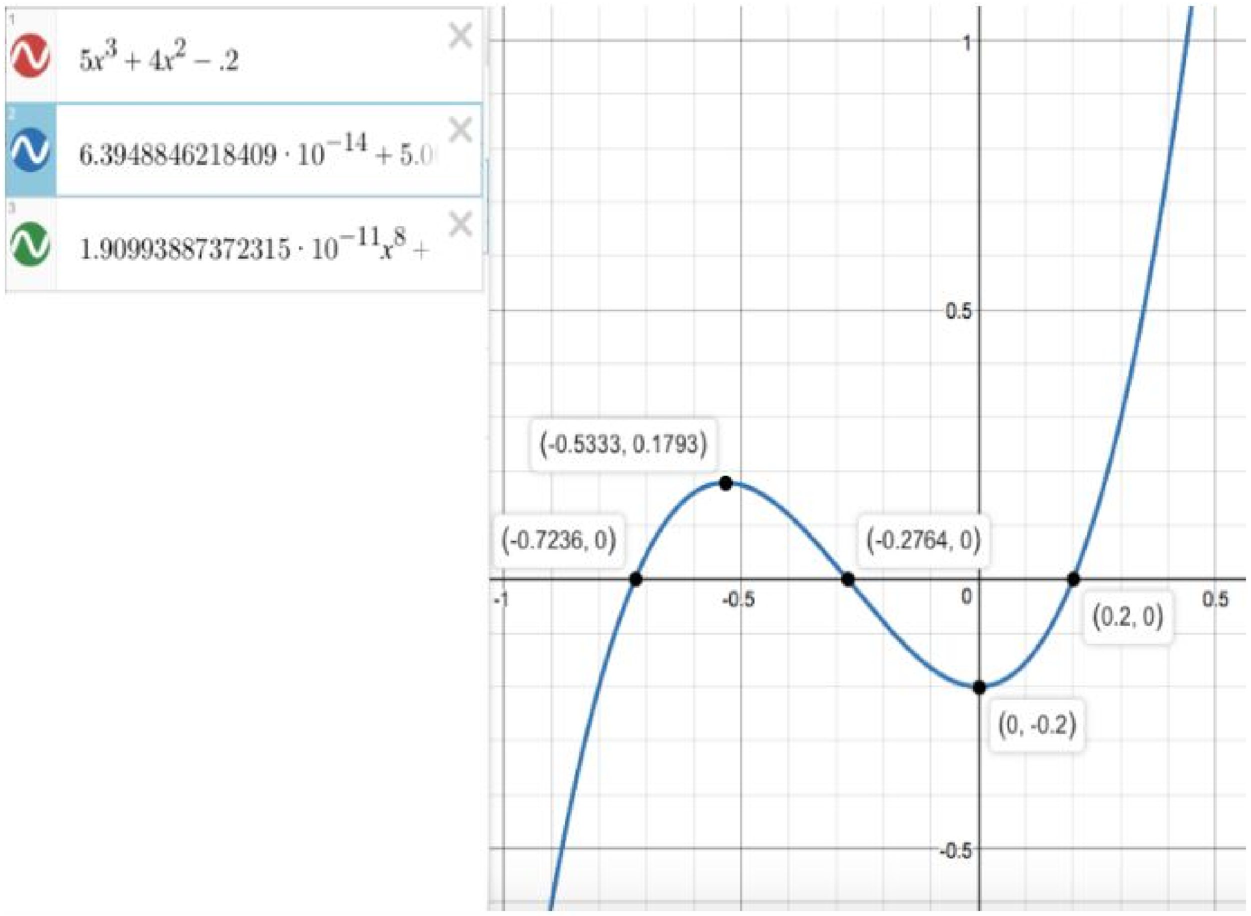
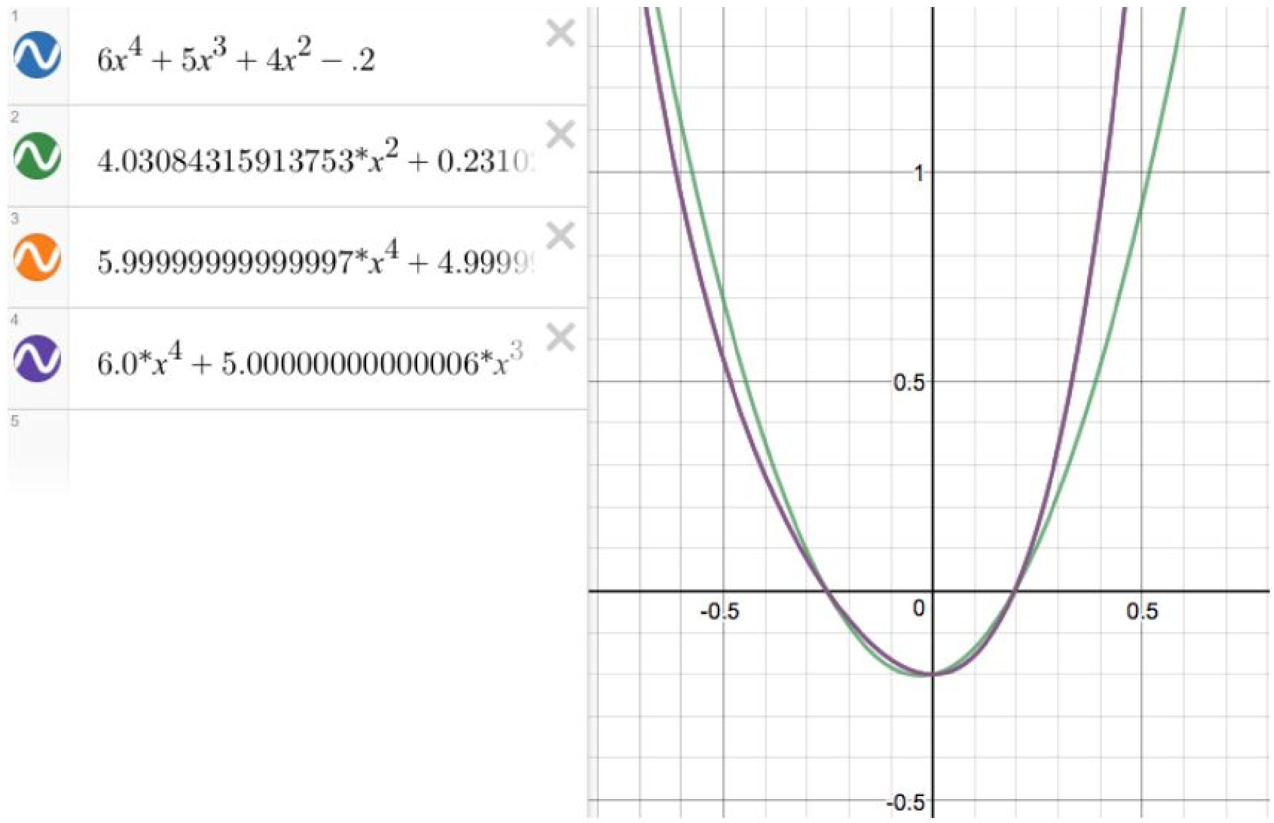


Fig (1a): The results of Lagrangian Interpolation of parent equation with original points labelled
(All images and charts in this paper were created by the researcher except as otherwise noted)

Further attempts to try to refine the code's accuracy, like eliminating "negligible" terms were made. Here is the following is a graph of the parent equation, $y = 6x^4 + 5x^3 + 4x^2 - .2$, which is used to further exemplify the unsound/unanticipated outcome due to the variation in the program.



1 $6x^4 + 5x^3 + 4x^2 - .2$

2 $4.03084315913753 * x^2 + 0.2310$

3 $5.99999999999997 * x^4 + 4.999$

4 $6.0 * x^4 + 5.00000000000006 * x^3$

5

Fig(1b): Interpolated Equation ignoring negligible values

Each equation after the first is the interpolated equation, each with 2n-1 more coordinates than the last equation (excluding the first interpolated equation). The first interpolated equation is clearly distinct from the original equation; this is because the input only consisted of 3 points, and the fewer the number of points, the less accurate the interpolated equation. However, that is not what was found from the data: After the interpolated equation's degree exceeds that of the starting function, the following equations become even more dissimilar to the parent function than from each preceding equation. Hence, in Fig(1b), the third interpolated equation is extremely similar to the parent function, and is the most accurate form of the equation because its degrees are the same. The following chart consists of the equations for each respective parent equation: (Refer to Appendix 1 [Fig(1c)]). It is clear that for some equations, excluding the negligible values can lead to the correct interpolation of the equation.

The following chart represents the values of the equations at x = 10,000:

	Including negligible values	Not including negligible values	Including negligible values	Not including negligible values
Parent Equation	$y = 5x^3 + 4x^2 - .2$	$y = 5x^3 + 4x^2 - .2$	$y = 6x^4 + 5x^3 + 4x^2 - .2$	$y = 6x^4 + 5x^3 + 4x^2 - .2$
Original Equation Value	5000399999999.8	5000399999999.8	6.00050004e+16	6.00050004e+16
Int. Eq 1	5000400000639.36	5000400000639.36	403086625.957482	403086625.957482
Int. Eq. 2	1.91035727675515 e+21	1.91035727675515 e+21	6.00050003999997e+ 16	6.00050003999997e+ 16
Int. Eq. 3	8.88485973841620 e+58	8.88485973841620 e+58	2.32839974270945e+ 22	6.00050004000000e+ 16
Int. Eq. 4	2.92088483636095 e+134	2.92088483636095 e+134	-5.85962888867173e +62	~
Int. Eq. 5	5.80409358261600 e+285	5.80409358261600 e+285	2.66594550170735e+ 143	~

Fig(2) Chart of values of Lagrangian Interpolated Equations at x = 10,000

From the randomly generated sample of 30 different polynomials that were used in the interpolation procedure, the p-value ≈ 0.16 > α. We fail to reject the null hypothesis.

•It can be stated with 84% confidence that the means of the percent errors of the y-values for the two methods are significantly different

•There is insufficient evidence to say that there is a significant difference in the means of the two percent errors of the y-values for Lagrange and Multivariate interpolation

DISCUSSION

A process for Lagrangian Interpolation and Multivariate Interpolation has been successfully constructed in Python 3.0, and has been modified such that it can easily input an expression in 2D space. Regardless of the initial equation, the first interpolated equation will *not* always resemble the parent function the most in spite of the methods' lack of eliminating anomalies such as negligibility (rounding may also need to be used [Fig(1b)]). The errors due to the method are "small," but affect the appearance of the equation to a very large extent when a high x-value is substituted into the equation. After the best interpolated equation, the error increases over the "time" variable. It *can* be possible to definitively have an equation equal the original, because of the error resulting from the method. It seems that Multivariate Interpolation is more accurate than Lagrangian Interpolation, however, a much larger sample size must be taken to confidently confirm findings. It seems to be the case that the t-values for both methods will always be the same for each method, meaning that one method is truly better than the other. When these errors from the interpolation are ignored, the first interpolated equation does not have to be the most accurate (Fig(1b)), where the equation with the same degree as the initial equation is the most accurate, and depending on the equation, it can equal the original equation.

Workarounds were created, with each one bypassing a restriction that would need to be transcended and could not be ignored inasmuch as the calculation in itself was too outlandish and massive in terms of code length and complexity. Hence, it would have to be altered through manipulation of the code itself; this task is something that is very daunting because it is almost certain that a file from a module is linked with another in the same directory, and changing one would disrupt the other, making revisions of said modules nearly impossible.

Each equation overlaps one another because they all become indistinguishable when the range for both x and y coordinates are relatively small. This identifies the key issue with interpolation; the negligible values lose their negligibility when exaggerated. As of yet, a fully autonomous method of interpolation, with randomly generated values (not necessarily integers), has not been successfully created.

CONCLUSION

A process for Lagrangian Interpolation has been successfully constructed, and has been modified such that it can easily input an expression in 2D space. Regardless of the initial equation, the first interpolated equation will always most resemble the parent function because of the Lagrangian Interpolation's lack of eliminating anomalies such as negligibility. The errors due to the method are "small," but affect the appearance of the equation to a very large extent when a high x-value is substituted into the equation. After the "best" interpolated equation, the error increases over the time variable that was previously defined. It can be impossible to definitively have an equation equal the original, because of the error resulting from the method itself. However, as shown in Fig(2), there are equations that, when interpolated a certain number of times, and ignoring the negligible values, can equal the original equation. When these errors from the interpolation are ignored, the first interpolated equation does not have to be the most accurate, where the equation with the same degree as the initial equation is the most accurate. Also, Fig(2) shows that for equations that do not converge, the error in the data becomes almost exponentially greater, and the error becomes relatively unavoidable because of the very definition of what it means to be negligible (as stated by my code).

Once again, we fail to reject the null hypothesis. Reasons for why conclusion might have been reached: the fact that the sample size was small ($n = 30$) means that the results may clash with the Law of Large Numbers; the differences in the y-values of the methods for both equations were miniscule, meaning that error was thus not represented accurately; and the precision of Python 3.0 (15 decimal places) prevented the differences in the equations and the curvilinear midpoints to be magnified.

ACKNOWLEDGEMENTS

I would like to sincerely thank Mr. Cody Onufrock for his assistance with this project. Without his expertise and encouragement, I don't think I would have been able to conduct such research given the overwhelming mathematical and technological obstacles that were initially present in this project.

REFERENCES

Harder, D. W. (n.d.). Numerical Analysis for Engineering. Retrieved December 02, 2016, from https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/05Interpolation/vandermonde/

Harder, D. W. (n.d.). Numerical Analysis for Engineering. Retrieved December 02, 2016, from https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/05Interpolation/multi/

Samiee, K. (2008, July 2). A Simple Expression for Multivariate Lagrange Interpolation. *SIAM Undergraduate Research Online (SIURO) Volume 1, Issue 1, 1*(1). Retrieved from https://www.siam.org/students/siuro/vol1issue1/.

Lambers, J. (n.d.). Lagrange Interpolation. *Lecture Notes*. Retrieved from http://www.math.usm.edu/lambers/mat772/fall10/lecture5

Polynomial Interpolation. (n.d.). *Lawrence Education*. Retrieved from http://www2.lawrence.edu/fast/GREGGJ/Math420/Section_3_1.pdf

Olver, P. J. (April 6, 2016). *On Multivariate Interpolation* (Unpublished doctoral dissertation). University of Minnesota.

Tremblay, Y. (2006). Interpolation of animal tracking data in a fluid environment. Journal of Experimental Biology, 209(1), 128-140. doi:10.1242/jeb.01970

Altered code found from http://www.eddaardvark.co.uk/python_patterns/polynomials.html, which can be found in Appendix 3.

APPENDICES

APPENDIX 1:

	Including negligible values	Not including negligible values	Including negligible values	Not including negligible values
Parent Equation	$y = 5x^3 + 4x^2 - .2$	$y = 5x^3 + 4x^2 - .2$	$y = 6x^4 + 5x^3 + 4x^2 - .2$	$y = 6x^4 + 5x^3 + 4x^2 - .2$
Int. Eq. 1	6.3948846218409e-14*x**4 + 5.00000000000007*x**3 + 4.00000000000001*x**2 - 2.44249065417534e-15*x - 0.2	5.00000000000007*x**3 + 4.00000000000001*x**2 - 0.2	4.03084315913753*x**2 + 0.23102437288166*x - 0.2	4.03084315913753*x**2 + 0.23102437288166*x - 0.2
Int. Eq. 2	1.90993887372315e-11*x**8 + 4.18367562815547e-11*x**7 + 3.04680725093931e-11*x**6 + 7.16227077646181e-12*x**5 - 1.10844666778576e-12*x**4 + 4.99999999999948*x**3 + 4.00000000000002*x**2 + 6.55031584528842e-15*x - 0.2	4.99999999999948*x**3 + 4.00000000000002*x**2 - 0.2	5.99999999999997*x**4 + 4.99999999999999*x**3 + 4.0*x**2 + 5.55111512312578e-17*x - 0.2	5.99999999999997*x**4 + 4.99999999999999*x**3 + 4.0*x**2 - 0.2

Int. Eq. 3	8.8810920715332e-6*x**16 + 3.76701354980469e-5*x**15 + 6.53266906738281e-5*x**14 + 6.62803649902344e-5*x**13 + 3.74317169189453e-5*x**12 + 1.02519989013672e-5*x**11 - 3.57627868652344e-7*x**10 - 9.61124897003174e-7*x**9 - 2.36555933952332e-7*x**8 - 4.19095158576965e-9*x**7 + 6.57746568322182e-9*x**6 + 5.96628524363041e-10*x**5 - 9.18589648790658e-11*x**4 + 4.99999999998886*x**3 + 4.00000000000031*x**2 + 4.70734562441066e-14*x - 0.2	8.8810920715332e-6*x**16 + 3.76701354980469e-5*x**15 + 6.53266906738281e-5*x**14 + 6.62803649902344e-5*x**13 + 3.74317169189453e-5*x**12 + 1.02519989013672e-5*x**11 + 4.99999999998886*x**3 + 4.00000000000031*x**2 - 0.2	2.3283064365387e-10*x**8 + 8.73114913702011e-11*x**7 - 5.82076609134674e-11*x**6 + 1.09139364212751e-11*x**5 + 6.0*x**4 + 5.00000000000006*x**3 + 3.99999999999998*x**2 - 1.55431223447522e-15*x - 0.2	6.0*x**4 + 5.00000000000006*x**3 + 3.99999999999998*x**2 - 0.2

Int. Eq. 4	2918400.0*x**32 + 24838144.0*x**31 + 102170624.0*x**30 + 254672896.0*x**29 + 410517504.0*x**28 + 529793024.0*x**27 + 465567744.0*x**26 + 308674560.0*x**25 + 157220864.0*x**24 + 53346304.0*x**23 + 12113920.0*x**22 + 1009664.0*x**21 - 249344.0*x**20 + 116480.0*x**19 + 173048.0*x**18 + 63008.0*x**17 + 13128.0*x**16 + 1261.9375*x**15 - 17.75*x**14 - 15.53125*x**13 + 0.15625*x**12 + 0.390869140625*x**11 -	2918400.0*x**32 + 24838144.0*x**31 + 102170624.0*x**30 + 254672896.0*x**29 + 410517504.0*x**28 + 529793024.0*x**27 + 465567744.0*x**26 + 308674560.0*x**25 + 157220864.0*x**24 + 53346304.0*x**23 + 12113920.0*x**22 + 1009664.0*x**21 - 249344.0*x**20 + 116480.0*x**19 + 173048.0*x**18 + 63008.0*x**17 + 13128.0*x**16 + 1261.9375*x**15 - 17.75*x**14 - 15.53125*x**13 + 0.15625*x**12 + 0.390869140625*x**11 -	-0.05859375*x**16 - 0.025390625*x**15 + 0.017578125*x**14 + 0.00146484375*x**13 + 0.004150390625*x**12 - 0.00029754638671875*x**11 + 7.2479248046875e-5*x**10 + 4.75645065307617e-5*x**9 + 1.43051147460938e-6*x**8 - 1.51339918375015e-7*x**7 + 1.30385160446167e-8*x**6 + 4.86033968627453e-9*x**5 + 6.00000000012733*x**4 + 5.00000000000421*x**3 + 4.00000000000008*x**2 - 2.22044604925031e-15*x - 0.2	Equal

	0.0099029541015625*x**10 - 0.00939559936523438*x**9 - 0.00081634521484375*x**8 + 3.51890921592712e-5*x**7 + 1.25132501125336e-5*x**6 + 5.72996214032173e-7*x**5 - 1.58470356836915e-8*x**4 + 4.99999999880629*x**3 + 4.00000000001887*x**2 + 1.31628041799559e-12*x - 0.2	0.0099029541015625*x**10 - 0.00939559936523438*x**9 - 0.00081634521484375*x**8 + 3.51890921592712e-5*x**7 + 1.25132501125336e-5*x**6 + 4.99999999880629*x**3 + 4.00000000001887*x**2 - 0.2		

Int. Eq. 5	5.79394472896058e+29*x**64 + 1.01407381564595e+31*x**63 + 8.11078970926667e+31*x**62 + 4.7073691294359e+32*x**61 + 1.87960397572825e+33*x**60 + 5.16890969531265e+33*x**59 + 1.21893221274618e+34*x**58 + 2.30077742142197e+34*x**57 + 3.89118622457276e+34*x**56 + 5.30508607069013e+34*x**55 + 6.23287360288499e+34*x**54 + 6.4717088794883e+34*x**53 + 5.9031160869803e+34*x**52 + 4.54689434317332e+34*x**51 + 2.98268777889404e+34*x**50 + 1.84179629657644e+34*x**49 + 9.13337800880298e+33*x**48 + 3.46629957773121e+33*x**47 + 6.98036785724333e+32*x**46 - 3.50820843230772e+32*x**45 - 4.71048796462237e+32*x**44 - 2.962099888242	5.79394472896058e+29*x**64 + 1.01407381564595e+31*x**63 + 8.11078970926667e+31*x**62 + 4.7073691294359e+32*x**61 + 1.87960397572825e+33*x**60 + 5.16890969531265e+33*x**59 + 1.21893221274618e+34*x**58 + 2.30077742142197e+34*x**57 + 3.89118622457276e+34*x**56 + 5.30508607069013e+34*x**55 + 6.23287360288499e+34*x**54 + 6.4717088794883e+34*x**53 + 5.9031160869803e+34*x**52 + 4.54689434317332e+34*x**51 + 2.9826877788	2.66586902075802e+15*x**32 + 764839186137088.0*x**31 - 296911089172480.0*x**30 - 170010911703040.0*x**29 - 148239722479616.0*x**28 + 1.3380826997719e+15*x**27 - 58264653922304.0*x**26 + 81425927766016.0*x**25 + 46480190603264.0*x**24 + 32425132425216.0*x**23 + 1147081850880.0*x**22 + 2231121379328.0*x**21 - 235851784192.0*x**20 + 135119376640.0*x**19 - 7509918336.0*x**18 + 621264016.0*x**17 + 11394492.0*x**16 - 31388263.5*x**15 - 79479.9375*x**14 - 62127.3465270996*x**13 + 1068.11059570313*x**12 - 92.8738861083984	Equal

	88e+32*x**43 - 1.23812498075726e+32*x**42 - 3.40870972700565e+31*x**41 + 1.2372278308619e+30*x**40 + 8.30180270884888e+30*x**39 + 6.17263477288363e+30*x**38 + 3.13471172680706e+30*x**37 + 1.25355292783106e+30*x**36 + 4.22554904045007e+29*x**35 + 1.23606773718941e+29*x**34 + 3.16772655521119e+28*x**33 + 7.2159234790729e+27*x**32 + 1.48522861924678e+27*x**31 + 2.8323256299442e+26*x**30 + 5.10012652990308e+25*x**29 + 8.71398108120337e+24*x**28 + 1.42386161884791e+24*x**27 + 2.15959421407425e+23*x**26 + 3.08211442874338e+22*x**25 + 4.1750048148692e+21*x**24 + 5.559607703204e+20*x**23 + 7.33331542792229e+19*x**22 +	9404e+34*x**50 + 1.84179629657644e+34*x**49 + 9.13337800880298e+33*x**48 + 3.46629957773121e+33*x**47 + 6.98036785724333e+32*x**46 - 3.50820843230772e+32*x**45 - 4.71048796462237e+32*x**44 - 2.96209988824288e+32*x**43 - 1.23812498075726e+32*x**42 - 3.40870972700565e+31*x**41 + 1.2372278308619e+30*x**40 + 8.30180270884888e+30*x**39 + 6.17263477288363e+30*x**38 + 3.13471172680706e+30*x**37 + 1.25355292783106e+30*x*	*x**11 + 6.81631088256836*x**10 - 0.309356898069382*x**9 - 0.00298209860920906*x**8 - 0.000981289194896817*x**7 - 3.89424603781663e-5*x**6 - 2.72643092102953e-7*x**5 + 5.99999999765083*x**4 + 5.00000000019248*x**3 + 3.99999999999951*x**2 - 1.01649374681378e-15*x - 0.2	

	9.52918908001793e+18*x**21 + 1.12291720774812e+18*x**20 + 1.15357083754299e+17*x**19 + 1.00517371382333e+16*x**18 + 702888351367168.0*x**17 + 38521848594432.0*x**16 + 1614588264448.0*x**15 + 33149427968.0*x**14 - 41595968.0*x**13 + 38290976.0*x**12 + 6568977.75*x**11 + 595267.1875*x**10 + 27507.3676757813*x**9 + 844.391967773438*x**8 + 19.9400253295898*x**7 + 0.466943144798279*x**6 + 0.00682185590267181*x**5 - 0.000112030655145645*x**4 + 4.99999647587538*x**3 + 4.00000002934303*x**2 + 1.06013331446775e-9*x - 0.2	*36 + 4.22554904045007e+29*x**35 + 1.23606773718941e+29*x**34 + 3.16772655521119e+28*x**33 + 7.2159234790729e+27*x**32 + 1.48522861924678e+27*x**31 + 2.8323256299442e+26*x**30 + 5.10012652990308e+25*x**29 + 8.71398108120337e+24*x**28 + 1.42386161884791e+24*x**27 + 2.15959421407425e+23*x**26 + 3.08211442874338e+22*x**25 + 4.1750048148692e+21*x**24 + 5.559607703204e+20*x**23 + 7.33331542792229e+19*x**22 +		

		9.52918908001793e+18*x**21 + 1.12291720774812e+18*x**20 + 1.15357083754299e+17*x**19 + 1.00517371382333e+16*x**18 + 702888351367168.0*x**17 + 38521848594432.0*x**16 + 1614588264448.0*x**15 + 33149427968.0*x**14 - 41595968.0*x**13 + 38290976.0*x**12 + 6568977.75*x**11 + 595267.1875*x**10 + 27507.3676757813*x**9 + 844.391967773438*x**8 + 19.9400253295898*x**7 + 0.466943144798279*x**6 + 0.00682185590267181*x**5 - 0.000112030655145645*x*		

		*4 + 4.99999647587538*x**3 + 4.00000002934303*x**2 - 0.2		

Fig(1c): Chart of Interpolated Equations for Lagrange Interpolation

APPENDIX 2:

```
import sympy
import itertools as itt
from collections import Counter
import sys
import numpy as np
import scipy
import Findroots
import sympy as sp
import itertools
from scipy.integrate import quad
from sympy import *
from sympy import Poly
from sympy import poly
from sympy.abc import x
from sympy import Symbol, cos

def l(xcoo,ycoo):
    x = Symbol('x')
    y = Symbol('y')
    if len(xcoo) != len(ycoo):
        print("Error!")
    else:
        repeat = 0
        oldfunction = 0
        while repeat < len(xcoo):
            n = 0
            oldterm = ycoo[repeat]
            while n < len(xcoo)-1:
                if n < repeat:
```

```
               newterm = x-xcoo[n]
            else:
                newterm = x-xcoo[n+1]
            newfunction = oldterm*newterm
            oldterm = newfunction
            n = n+1

        n = 0

        while n < len(xcoo)-1:
            if n < repeat:
               newterm = xcoo[repeat]-xcoo[n]
            else:
                newterm = xcoo[repeat]-xcoo[n+1]
            newfunction = oldterm/newterm
            oldterm = newfunction
            n = n+1

        newf = newfunction + oldfunction
        oldfunction = newf

        repeat = repeat + 1

    return newf
def d(coef,exp):
    x = Symbol('x')
    n = 0
    newterm = 0
    while n < len(coef):
        oldterm = exp[n]*coef[n]*x**(exp[n]-1)
        newterm = oldterm + newterm
        n = n + 1
    derivative = newterm
    print(derivative)

def function(coef,exp):
        x = Symbol('x')
        n = 0
        newterm = 0
        while n < len(coef):
            oldterm = coef[n]*x**(exp[n])
            newterm = oldterm + newterm
            n = n + 1
        function = newterm
        return function
def roots(coef):
    r = Findroots.FindRoots(coef)
    return r

def rootsofd(coef,exp): #Fixed (Add something that inserts left out zeros in the future instead of
having to type them in)
    n = 0
    print(function(coef,exp))
    print(d(coef,exp))
    while n < len(coef):
            oldterm = [exp[n]*coef[n]]
            n = n + 1
            while n < len(coef):
                a = exp[n]*coef[n]
                if a == 0:
                    a = 0
                oldterm.append(a)
                n = n + 1
    term = oldterm
    print(term)
    newlist = list()
    i = 1
    while i <= len(exp):
        indexes = sorted(set(exp))[-i]
        print(indexes)
        indexterm = exp.index(indexes)
        print(indexterm)
        sortterm = term[indexterm]
        print(sortterm)
        newlist.append(sortterm)
        i = i + 1
        print(newlist)
    term = newlist
    h = 0
    b = 0
    derexp = [x-1 for x in exp]
    for n, i in enumerate(derexp):
        if i < 0:
            derexp[n] = 0
    print(derexp)
    print('value')
    while h < max(derexp):
        print(h)
        o = h in derexp
        if o is False:
            o = h in derexp
            term.insert(len(term)-h-b,0)
            b = b + 1
            print(term)
        h = h + 1
    print(term)
    j = roots(term)
    return j

def f(coef,exp,value): #Outputs y-value (use for rel. min and max) to find points
        n = 0
        newterm = 0
        while n < len(coef):
            oldterm = coef[n]*value**(exp[n])
            print(oldterm)
            newterm = oldterm + newterm
            n = n + 1
        function = newterm
        print(function)
        return function
def fvalue(coef,exp):
    term = coef
    print(term)
    newlist = list()
    i = 1
    while i <= len(exp):
        indexes = sorted(set(exp))[-i]
        print(indexes)
        indexterm = exp.index(indexes)
        print(indexterm)
        sortterm = term[indexterm]
        print(sortterm)
        newlist.append(sortterm)
        i = i + 1
        print(newlist)
    term = newlist
    h = 0
    derexp = [x for x in exp]
    print(derexp)
    while h < max(derexp):
        if not h in derexp:
            term.insert(len(term)-h,0)
            print(h)
            print(term)
        h = h + 1
    return term
def fvalued(coef,exp):
    term = coef
    print(term)
    newlist = list()
    i = 1
    while i <= len(exp):
        indexes = sorted(set(exp))[-i]
        print(indexes)
        indexterm = exp.index(indexes)
        print(indexterm)
        sortterm = term[indexterm]
        print(sortterm)
        newlist.append(sortterm)
        i = i + 1
        print(newlist)
    term = newlist
```

```
  h = 1
  derexp = [x-1 for x in exp]
  print(derexp)
  while h < max(derexp):
    if not h in derexp:
      term.insert(len(coef)+1-h,0)
      print(h)
      h = h + 1
    h = h + 1
  print(term)
  return term
def xcoo(coef,exp):
   xcoo = rootsofd(coef,exp)
   f = fvalue(coef,exp)
   a = roots(f)
   xcoo.extend(a)
   return xcoo
def p(coef,exp):
   xcoo = rootsofd(coef,exp)
   print(xcoo)
   f = fvalue(coef,exp)
   b = 0
   p = [Findroots.GetValue(f,xcoo[b])]
   b = b + 1
   while b < len(xcoo):
     p.append(Findroots.GetValue(f,xcoo[b]))
     b = b + 1
   a = roots(f)
   xcoo.extend(a)
   r = len(p)
   o = 1
   while o <= len(xcoo)-r:
     p.append(0)
     o = o + 1
   return p
def rectify(smallestterm, nextsmallestterm, a):
  exp = []
  s = len(a)-1
```

```
while s >= 0:
    exp.append(s)
    s = s - 1
print(exp)
s = 0
newterm = 0
while s < len(a):
    oldterm = a[s]*(x)**(exp[s])
    newterm = oldterm + newterm
    s = s + 1
funct = newterm
print(funct)
poly = funct
expr = sqrt(1+(pow(diff(poly),2)))
w = quad(lambda x_: expr.subs(x, x_), smallestterm, nextsmallestterm)
w = w[0]
print(w)
dist = abs(w/2) #EDITED FROM HERE
print(dist)
n = 0
newterm = 0
while n < len(a):
    oldterm = a[n]*(x + smallestterm)**(len(a)-n-1)
    print(oldterm)
    newterm = oldterm + newterm
    n = n + 1
function = newterm
print("function")
print(function)
function = function.expand()
print("expand")
print(function)
function = Poly(function)
function = function.coeffs()
value = Findroots.GetValue(function, dist)
print("value")
print(value)
expr = lambdify(x, expr)
```

```
a = smallestterm + .00001
while quad(expr, smallestterm, a) < quad(expr, a, nextsmallestterm):
   a = a + .00001
print(a)
while quad(expr, smallestterm, a) > quad(expr, a, nextsmallestterm):
   a = a - .000001
print(a)
while quad(expr, smallestterm, a) < quad(expr, a, nextsmallestterm):
   a = a + .0000001
print(a)
while quad(expr, smallestterm, a) > quad(expr, a, nextsmallestterm):
   a = a - .00000001
print(a)
while quad(expr, smallestterm, a) < quad(expr, a, nextsmallestterm):
   a = a + .000000001
print(a)
while quad(expr, smallestterm, a) > quad(expr, a, nextsmallestterm):
   a = a - .0000000001
print(a)
while quad(expr, smallestterm, a) < quad(expr, a, nextsmallestterm):
   a = a + .00000000001
print(a)
while quad(expr, smallestterm, a) > quad(expr, a, nextsmallestterm):
   a = a - .000000000001
print(a)
while quad(expr, smallestterm, a) < quad(expr, a, nextsmallestterm):
   a = a + .0000000000001
print(a)
while quad(expr, smallestterm, a) > quad(expr, a, nextsmallestterm):
   a = a - .00000000000001
print(a)
while quad(expr, smallestterm, a) < quad(expr, a, nextsmallestterm):
   a = a + .000000000000001
print(a)
truth = print(quad(expr, smallestterm, a) > quad(expr, a, nextsmallestterm))
if truth == True:
   while print(quad(expr, smallestterm, a) > quad(expr, a, nextsmallestterm)):
      a = a - .0000000000000001
```

```
    print(a)
    e = quad(expr, smallestterm, a)
    print(e)
    print(quad(expr, a, nextsmallestterm))
    value = a
    print("value")
    print(value)
    return value

def lint(coef,exp):   #Now with x-intercepts and rel mins/maxes
    function(coef,exp)
    d(coef,exp)
    xcoo = rootsofd(coef,exp)
    print(xcoo)
    f = fvalue(coef,exp)
    b = 0
    p = [Findroots.GetValue(f,xcoo[b])]
    b = b + 1
    while b < len(xcoo):
        p.append(Findroots.GetValue(f,xcoo[b]))
        b = b + 1
    a = roots(f)
    print(p)
    xcoo.extend(a)
    print('value1')
    print(xcoo)
    r = len(p)
    o = 1
    while o <= len(xcoo)-r:
        p.append(0)
        o = o + 1
    print("xcoo")
    print(xcoo)
    print("ycoo")
    print(p)
    e = l(xcoo,p)
    print("Interpolation equation equals")
    return e
```

```
def mint(coef,exp):
    d(coef,exp)
    xcoo = rootsofd(coef,exp)
    print(xcoo)
    f = fvalue(coef,exp)
    b = 0
    p = [Findroots.GetValue(f,xcoo[b])]
    b = b + 1
    while b < len(xcoo):
        p.append(Findroots.GetValue(f,xcoo[b]))
        b = b + 1
    a = roots(f)
    print(p)
    xcoo.extend(a)
    print('value1')
    print(xcoo)
    r = len(p)
    o = 1
    while o <= len(xcoo)-r:
        p.append(0)
        o = o + 1
    print("xcoo")
    print(xcoo)
    print("ycoo")
    print(p)
    e = multieq(xcoo,p)
    print("Interpolation equation equals")
    return e
def lagrange(coef,exp,t,e,a,x,pcoo,newexp): #W/o negligible values
  if t == 0:
    a = fvalue(coef,exp)
    u = Poly(lint(coef,exp))
  else:
    u = e
  print(a)
  print(t)
  print(e)
  print(x)
  print(pcoo)
  print('U-value')
  print(u)
  u = u.coeffs()
  copy = u
  counter = 1
  m = []
  while counter < len(u) + 1:
    m.append(counter-1)
    counter = counter + 1
  print("m") #no m.reverse
  print(m)
  u = [x for x in u if abs(x) > 0.000001]
  print(u)
  copy.reverse()
  print(copy)
  exps = [i for i in m if copy[i] in u]
  exps.reverse()
  print("exps")
  print(exps)
  u = function(u,exps)
  u = Poly(u)
  print("new u")
  print(u)
  avalue = Findroots.GetValue(a,10000)
  b = u.all_coeffs()
  print(b)
  bvalue = Findroots.GetValue(b,10000)
  print(avalue)
  print(bvalue)
  if avalue != bvalue:
    print('not equal')
    t = t + 1
    lp = 0
    while lp < 9:
        print(t)
        lp = lp + 1
    if t == 6: #Just for data
```

```
        return
    poly = b
    c = len(poly) - 1
    print(c)
    exponents = []
    while c + 1 > 0:
        exponents.append(c)
        c = c - 1
    print(poly)
    print(exponents)
    poly = function(poly,exponents)
    print(poly)
    if t == 1:
        x = xcoo(coef,exp)
    o = len(x)
    print('value2')
    if t == 1:
        pcoo = p(coef,exp)
        print(pcoo)
    term = x
    rectpts = []
    while o > 1:
        smallestterm = sorted(set(term))[-o]
        print(smallestterm)
        nextsmallestterm = sorted(set(term))[-o+1]
        print(nextsmallestterm)
        mid = rectify(smallestterm, nextsmallestterm, a)
        pcoo.append(Findroots.GetValue(a,mid))
        print(pcoo)
        rectpts.append(mid)
        x.append(mid)
        o = o - 1
    print("RECTIFICATION COORDS")
    print(rectpts)
    print(x)
    print(pcoo)
    e = (l(x,pcoo))
    e = expand(e, multinomial=True)
```

```
        print(e)
        e = Poly(e)
        print(e)
        newexp = []
        newcoef = e.coeffs()
        k = len(e.coeffs()) - 1
        while k >= 0:
            newexp.append(k)
            k = k - 1
        print("newcoef")
        print(newcoef)
        print("newexp")
        print(newexp)
        lagrange(newcoef, exp, t, e, a, x, pcoo, newexp)
    else:
        print('equal')
        t = t + 1
        print(t)
        return
def lagrangee(coef,exp,t,e,a,x,pcoo,newexp): #With negligible values
    if t == 0:
        a = fvalue(coef,exp)
        u = Poly(lint(coef,exp))
    else:
        u = e
    print(a)
    print(t)
    print(e)
    print(x)
    print(pcoo)
    print('U-value')
    print(u)
    u = u.coeffs()
    counter = 1
    m = []
    while counter < len(u) + 1:
        m.append(counter-1)
        counter = counter + 1
```

```
m.reverse()
print("m")
print(m)
u = function(u,m)
u = Poly(u)
print("new u")
print(u)
avalue = Findroots.GetValue(a,10000)
b = u.all_coeffs()
print(b)
bvalue = Findroots.GetValue(b,10000)
print(avalue)
print(bvalue)
if avalue != bvalue:
    print('not equal')
    t = t + 1
    poly = b
    c = len(poly) - 1
    print(c)
    exponents = []
    while c + 1 > 0:
        exponents.append(c)
        c = c - 1
    print(poly)
    print(exponents)
    poly = function(poly,exponents)
    print(poly)
    if t == 1:
        x = xcoo(coef,exp)
    o = len(x)
    print('value2')
    if t == 1:
        pcoo = p(coef,exp)
        print(pcoo)
    term = x
    rectpts = []
    while o > 1:
        smallestterm = sorted(set(term))[-o]
        print(smallestterm)
        nextsmallestterm = sorted(set(term))[-o+1]
        print(nextsmallestterm)
        mid = rectify(smallestterm, nextsmallestterm, a)
        pcoo.append(Findroots.GetValue(a,mid))
        print(pcoo)
        rectpts.append(mid)
        x.append(mid)
        o = o - 1
    print("RECTIFICATION COORDS")
    print(rectpts)
    print(x)
    print(pcoo)
    e = (l(x,pcoo))
    e = expand(e, multinomial=True)
    print(e)
    e = Poly(e)
    print(e)
    newexp = []
    newcoef = e.coeffs()
    k = len(e.coeffs()) - 1
    while k >= 0:
        newexp.append(k)
        k = k - 1
    print("newcoef")
    print(newcoef)
    print("newexp")
    print(newexp)
    lagrangee(newcoef, exp, t, e, a, x, pcoo, newexp)
  else:
    print('equal')
    t = t + 1
    print(t)
    return

def test(coef1,coef,exp,t,e,a,x,pcoo,newexp,lastvalue,tlist1,ylist,ylist1,tlist2,ylist2,rep): #With
negligible values
  while rep > 0:
```

```
if t == 0:
    coef1 = coef
    a = fvalue(coef,exp)
    u = Poly(lint(coef,exp))
else:
    u = e
print(a)
print(t)
print(e)
print(x)
print(pcoo)
print('U-value')
print(u)
u = u.coeffs()
counter = 1
m = []
while counter < len(u) + 1:
    m.append(counter-1)
    counter = counter + 1
m.reverse()
print("m")
print(m)
u = function(u,m)
u = Poly(u)
print("new u")
print(u)
avalue = Findroots.GetValue(a,10000)
b = u.all_coeffs()
print(b)
bvalue = Findroots.GetValue(b,10000)
if t != 0:
    lastvalue2 = lastvalue
    lastvalue = bvalue
else:
    lastvalue = bvalue
print(avalue)
print(bvalue)
if avalue != bvalue:
        print('not equal')
        if t != 0:
            if abs(avalue-bvalue) > abs(avalue-lastvalue2):
                print("Most accurate value achieved")
                print(t)
                print(lastvalue2)
                tlist1 = []
                ylist1 = []
                tlist1.append(t)
                ylist1.append(lastvalue2)
                func = 0
                t1 = 0
                xcoor = 0
                ycoor = 0
                multivariate(a, coef1, exp, t1, e, x, pcoo, newexp, 0, tlist1, ylist, ylist1, tlist2, ylist2,
rep) #will need *starting* xcoo and ycoo, and function
        t = t + 1
        poly = b
        c = len(poly) - 1
        print(c)
        exponents = []
        while c + 1 > 0:
            exponents.append(c)
            c = c - 1
        print(poly)
        print(exponents)
        poly = function(poly,exponents)
        print(poly)
        if t == 1:
            x = xcoo(coef,exp)
        o = len(x)
        print('value2')
        if t == 1:
            pcoo = p(coef,exp)
            print(pcoo)
        term = x
        rectpts = []
        while o > 1:
            smallestterm = sorted(set(term))[-o]
            print(smallestterm)
            nextsmallestterm = sorted(set(term))[-o+1]
            print(nextsmallestterm)
            mid = rectify(smallestterm, nextsmallestterm, a)
            pcoo.append(Findroots.GetValue(a,mid))
            print(pcoo)
            rectpts.append(mid)
            x.append(mid)
            o = o - 1
        print("RECTIFICATION COORDS")
        print(rectpts)
        print(x)
        print(pcoo)
        e = (l(x,pcoo))
        e = expand(e, multinomial=True)
        print(e)
        e = Poly(e)
        print(e)
        newexp = []
        newcoef = e.coeffs()
        k = len(e.coeffs()) - 1
        while k >= 0:
            newexp.append(k)
            k = k - 1
        print("newcoef")
        print(newcoef)
        print("newexp")
        print(newexp)
        test(coef1, newcoef, exp, t, e, a, x, pcoo, newexp, lastvalue, tlist1, ylist, ylist1, tlist2,
ylist2,rep)
    else:
        print('equal')
        t = t + 1
        print(t)
        return
  print('Lagrangian Interpolation')
  print(tlist1)
```

```
    print(ylist1)
    print('Multivariate Interpolation')
    print(tlist2)
    print(ylist2)
    print('Real y Values')
    print(ylist)
def multivariate(a,coef,exp,t1,e,x,pcoo,newexp,lastvalue,tlist1,ylist,ylist1,tlist2,ylist2,rep):
    if t1 == 0:
        print(coef)
        print(exp)
        u = Poly(mint(coef,exp))
    else:
        u = e
    print(a)
    print(t1)
    print(e)
    print(x)
    print(pcoo)
    print('U-value')
    print(u)
    u = u.coeffs()
    counter = 1
    m = []
    while counter < len(u) + 1:
        m.append(counter-1)
        counter = counter + 1
    m.reverse()
    print("m")
    print(m)
    u = function(u,m)
    u = Poly(u)
    print("new u")
    print(u)
    b = u.all_coeffs()
    avalue = Findroots.GetValue(a,10000)
    bvalue = Findroots.GetValue(b,10000)
    if t1 != 0:
        lastvalue2 = lastvalue
        lastvalue = bvalue
    else:
        lastvalue = bvalue
    if avalue != bvalue:
        print('not equal')
        if t1 != 0:
            if abs(avalue-bvalue) > abs(avalue-lastvalue2):
                print("Most accurate value achieved")
                print(t1)
                print(lastvalue2)
                tlist2 = []
                ylist2 = []
                tlist2.append(t1)
                ylist2.append(lastvalue2)
                print("most recent values")
                print(tlist1)
                print(tlist2)
                print(ylist1)
                print(ylist2)
                print(avalue)
                ylist.append(avalue)
                print("Now do analysis or gen. more random numbers") #ANALYSIS STAGE
                rep = rep + 1
                test(coef1,coef,exp,t,e,a,x,pcoo,newexp,lastvalue,tlist1,ylist,ylist1,tlist2,ylist2,rep)
        t1 = t1 + 1
        poly = b
        c = len(poly) - 1
        print(c)
        exponents = []
        while c + 1 > 0:
            exponents.append(c)
            c = c - 1
        print(poly)
        print(exponents)
        poly = function(poly,exponents)
        print(poly)
        if t1 == 1:
            x = xcoo(coef,exp)
```

```
o = len(x)
print('value2')
if t1 == 1:
    pcoo = p(coef,exp)
    print(pcoo)
term = x
rectpts = []
while o > 1:
    smallestterm = sorted(set(term))[-o]
    print(smallestterm)
    nextsmallestterm = sorted(set(term))[-o+1]
    print(nextsmallestterm)
    mid = rectify(smallestterm, nextsmallestterm, a)
    pcoo.append(Findroots.GetValue(a,mid))
    print(pcoo)
    rectpts.append(mid)
    x.append(mid)
    o = o - 1
print("RECTIFICATION COORDS")
print(rectpts)
print(x)
print(pcoo)
e = (multieq(x,pcoo))
e = expand(e, multinomial=True)
print(e)
e = Poly(e)
print(e)
newexp = []
newcoef = e.coeffs()
k = len(e.coeffs()) - 1
while k >= 0:
    newexp.append(k)
    k = k - 1
print("newcoef")
print(newcoef)
print("newexp")
print(newexp)
```

```
        multivariate(a, newcoef, exp, t1, e, x, pcoo, newexp, lastvalue, tlist1, ylist, ylist1, tlist2,
ylist2, rep) #tlist1,ylist1,tlist2,ylist2
    else:
        print('equal')
        t1 = t1 + 1
        print(t1)
        return
def multieq(xcoo,p):
    s = len(xcoo)-1
    print(s)
    V = np.polynomial.polynomial.polyvander(xcoo,s)
    print(V)
    c = np.linalg.solve(V,p) #.lstsq
    print(c)
    n = 0
    equation = 0
    while n <= s:
        x = Symbol('x')
        equation = c[s-n]*x**(s-n)
        n = n + 1
        while n <= s:
            equation2 = equation + c[s-n]*x**(s-n)
            equation = equation2
            n = n + 1
    print(equation2)
    return equation2

def sturm(poly): #GREATEST DEGREE IS 6
    poly = Poly(poly)
    deg = polys.polytools.degree(poly)
    schangepos = 0
    schangeneg = 0
    if deg > 0:
        p0 = poly
        print(p0)
        sign1a = Findroots.GetValue(p0.all_coeffs(), 100000000000000)
        sign1a = sign1a/abs(sign1a)
        print(sign1a)
```

```
sign1b = Findroots.GetValue(p0.all_coeffs(), -100000000000000)
sign1b = sign1b/abs(sign1b)
print(sign1b)
p1 = diff(p0)
print(p1)
sign2a = Findroots.GetValue(p1.all_coeffs(), 100000000000000)
sign2a = sign2a/abs(sign2a)
print(sign2a)
sign2b = Findroots.GetValue(p1.all_coeffs(), -100000000000000)
sign2b = sign2b/abs(sign2b)
print(sign2b)
if sign1a + sign2a == 0:
    schangepos = schangepos + 1
    print(schangepos)
if sign1b + sign2b == 0:
    schangeneg = schangeneg + 1
    print(schangeneg)
deg = deg - 1
if deg > 0:
    q, r = div(p0, p1)
    p2 = -r
    print(p2)
    sign3a = Findroots.GetValue(p2.all_coeffs(), 100000000000000)
    sign3a = sign3a/abs(sign3a)
    print(sign3a)
    sign3b = Findroots.GetValue(p2.all_coeffs(), -100000000000000)
    sign3b = sign3b/abs(sign3b)
    print(sign3b)
    if sign2a + sign3a == 0:
        schangepos = schangepos + 1
        print(schangepos)
    if sign2b + sign3b == 0:
        schangeneg = schangeneg + 1
        print(schangeneg)
    deg = deg - 1
    if deg > 0:
        q, r = div(p1, p2)
        p3 = -r
```

```
print(p3)
sign4a = Findroots.GetValue(p3.all_coeffs(), 100000000000000)
sign4a = sign4a/abs(sign4a)
print(sign4a)
sign4b = Findroots.GetValue(p3.all_coeffs(), -100000000000000)
sign4b = sign4b/abs(sign4b)
print(sign4b)
if sign3a + sign3a == 0:
    schangepos = schangepos + 1
    print(schangepos)
if sign3b + sign3b == 0:
    schangeneg = schangeneg + 1
    print(schangeneg)
deg = deg - 1
if deg > 0:
    q, r = div(p2, p3)
    p4 = -r
    print(p4)
    sign5a = Findroots.GetValue(p4.all_coeffs(), 100000000000000)
    sign5a = sign5a/abs(sign5a)
    print(sign5a)
    sign5b = Findroots.GetValue(p4.all_coeffs(), -100000000000000)
    sign5b = sign5b/abs(sign5b)
    print(sign5b)
    if sign4a + sign5a == 0:
        schangepos = schangepos + 1
        print(schangepos)
    if sign4b + sign5b == 0:
        schangeneg = schangeneg + 1
        print(schangeneg)
    deg = deg - 1
    if deg > 0:
        q, r = div(p3, p4)
        p5 = -r
        print(p5)
        sign6a = Findroots.GetValue(p5.all_coeffs(), 100000000000000)
        sign6a = sign6a/abs(sign6a)
        print(sign6a)
```

```
          sign6b = Findroots.GetValue(p5.all_coeffs(), -100000000000000)
          sign6b = sign6b/abs(sign6b)
          print(sign6b)
          if sign5a + sign6a == 0:
            schangepos = schangepos + 1
            print(schangepos)
          if sign5b + sign6b == 0:
            schangeneg = schangeneg + 1
            print(schangeneg)
          deg = deg - 1
          if deg > 0:
            q, r = div(p4, p5)
            p6 = -r
            print(p6)
            deg = deg - 1
            if deg > 0:
              q, r = div(p5, p6)
              p7 = -r
              print(p7)
              sign7a = Findroots.GetValue(p6.all_coeffs(), 100000000000000)
              sign7a = sign7a/abs(sign7a)
              print(sign7a)
              sign7b = Findroots.GetValue(p6.all_coeffs(), -100000000000000)
              sign7b = sign7b/abs(sign7b)
              print(sign7b)
              if sign6a + sign7a == 0:
                schangepos = schangepos + 1
                print(schangepos)
              if sign6b + sign7b == 0:
                schangeneg = schangeneg + 1
                print(schangeneg)
  realroots = schangeneg-schangepos
  print('realroots')
  print(realroots)
  return realroots

def check(coef1):
  counter = 1
```

```
    length = np.random.randint(2,6)
    print(length)
    len2 = length
    coef1 = []
    m = []
    while counter < length + 1:
        m.append(counter-1)
        counter = counter + 1
        coef1.append(np.random.randint(-10,10))
    m.reverse()
    print("m")
    print(m)
    u = function(coef1,m)
    print(u)
    u = Poly(u)
    if sturm(u) < 1:
        check(0)
    else:
        print(coef1)
        print(m)
        return coef1
```

APPENDIX 3:

```
import sympy
import sys
from sympy import Poly
from sympy import Symbol, cos

def GetValue (polynomial, z):
    """
    Calculate the value of the polyminal at 'z'
    """

    n = len (polynomial)
    r = 0

    for i in range (n):
        r = r * z + polynomial [i]

    return r

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~
def divide (_poly1, poly2):
    """
    Divide poly1 by poly2. Returns the new polynomial and the remainder
    """

    num = len(_poly1) - len(poly2)

    if num < 0:
        raise Exception("divisor too long")

    poly1 = [i for i in _poly1]
    n2 = len(poly2)

    ret = []
    for i in range (num+1):
        f = float(poly1 [i]) / float(poly2 [0])
        ret.append(f)
        for j in range (n2):
            poly1 [i+j] -= f * poly2[j]
    return ret,poly1
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def Differentiate (polynomial):
    """
    Given an polynomial, returns its derivative
    """
    diff = []
    n = len (polynomial)-1

    for i in range (n):
        diff.append (polynomial [i] * (n-i))

    return diff
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def GetNewtonPolynomials (polynomial):
    """
    given an polynomial, returns the numerator and denominator to
    be used in the iteration x -> numerator(x) / denominator (x)
    """
    denominator = Differentiate(polynomial)
    numerator = []

    xfdash = denominator + [0]

    for i in range (0, len (polynomial)):
        numerator.append (xfdash [i] - polynomial [i])

    return numerator, denominator
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def FindRoots (polynomial):
    """
    Find the roots of a polynomial
    """
    roots = []
    while len(polynomial) > 2:
        root = FindRoot (polynomial)
        if root == None:
            if GetValue(polynomial,0) == 0:
                roots.append(0)
                return roots
            else:
                raise Exception ("Can't solve {0}", polynomial)

    # decide if this is a single real root or a conjugate pair.
    # we do this by dividing the polynomial by the roots and comparing
    # the remainders

        rp1 = [1, -root.real]
        rp2 = [1, -2 * root.real, root.real * root.real]

        p1, rem1 = divide (polynomial, rp1)
```

```
        p2, rem2 = divide (polynomial, rp2)

        m1 = abs (rem1 [-1])
        m2 = max (abs (rem2 [-1]),abs (rem2 [-2]))

        if m1 < m2:
            roots.append(root.real)
            print(roots)
            polynomial = p1
        else:
            return roots

    if len (polynomial) == 2:
        roots.append (-polynomial[1]/polynomial[0])

    return roots
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def GetNewtonOrbit (polynomial, x, y, num):
    """
    Returns the path taken by a starting "guess" as it converges on a
    root (or not)
    """
    num,den = GetNewtonPolynomials (polynomial)

    z = complex (x,y)
    orbit = [z]

    for i in range (num):
        v1 = GetValue (num, z)
        v2 = GetValue (den, z)
        z = v1 / v2
        orbit.append (z)

    return orbit
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def FindRoot (polynomial):
    """
    Find a root of the polymonial
```

```
    """
    xvals = [-1000,1000,0]
    yvals = [0,1000,-1000]

    num,den = GetNewtonPolynomials (polynomial)

    for x in xvals:
        for y in yvals:
            z = complex (x,y)
            try:
                for i in range (1000):
                    v1 = GetValue (num, z)
                    v2 = GetValue (den, z)
                    zm = z
                    z = v1 / v2
                    v3 = GetValue (polynomial, z)

                    if z == zm:
                        z = z.real
                        return z
                v = GetValue (polynomial, z)
                if abs (v) < 1e-10:
                    z = z.real
                    return z
            except:
                pass
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def SortRoots (_roots):
    """
    Sort some roots
    """

    roots = [r for r in _roots]

    while True:
        swapped = 0
        for i in range (0, len(roots)-1):
```

```
            if roots[i].real > roots[i+1].real or (roots[i].real == roots[i+1].real and roots[i].imag >
roots[i+1].imag):
                temp = roots[i]
                roots [i] = roots [i+1]
                roots [i+1] = temp
                swapped += 1
        if swapped == 0:
            break

    return roots

def function(coef,exp):
        x = Symbol('x')
        n = 0
        newterm = 0
        while n < len(coef):
            oldterm = coef[n]*x**(exp[n])
            newterm = oldterm + newterm
            n = n + 1
        function = newterm
        print(function)
def f(coef,exp):
    x = Symbol('x')
    a = function(coef,exp)
```