Methods of Polynomial Interpolation and their Applications in Compression and Encryption

Ibrahim Shanqiti

1/8/2020

Abstract

In this paper, I began with deriving and researching several methods of polynomial interpolation, mainly using matrices, as well as Lagrange Interpolation. After exploring methods of polynomial interpolation, I transcribed the mathematical formulas into Python, so I could investigate the applications of polynomial interpolation into two different topics: Compression and Encryption. Encrypting text was simple, by turning all the characters into their corresponding ASCII decimals, however with images, their RGB values were used instead.

Word Count: 4173

Contents

1 Introduction and Deriving Methods of Polynomial Interpolation

1.1 Sequences and Sets

The idea of polynomial interpolation first intrigued me in ninth grade Algebra, when the class was first covering sequences. The teacher would first put up a list of numbers, such as 1, 3, 5, 7..., and ask the class to find a function that outputs those of numbers given, as well as continuing the sequence. So with the function $f(x)$, $f(n)$ would represent the n^{th} number in the sequence. Eventually, we were told why. In a linear sequence $ax + b$, the first terms would be:

$$a + b,\ 2a + b,\ 3a + b$$

The first difference of any consecutive terms in the sequence would be a, and so the function that represents that set of numbers can be found quite easily. Note that, since the first difference is constant, only two consecutive values need to be given to find the function that models the set of numbers. Afterwards, the teacher moved onto quadratic sequences. Following the same procedure, the first few terms of a quadratic function in form $ax^2 + bx + c$ are:

$$a + b + c,\ 4a + 2b + c,\ 9a + 3b + c$$

It is then clear to see that the first differences of a normal quadratic sequences become a linear sequence:

$$3a + b,\ 5a + b$$

From there, the second difference is now a constant, which is $2a$. To clarify further, in a cubic sequence, the first terms are:

$$a + b + c + d,\ 8a + 4b + 2c + d,\ 27a + 9b + 3c + d,\ 64a + 16b + 4c + d$$

The first difference become a quadratic sequence:

$$7a + 3b + c,\ 19a + 5b + c,\ 37a + 7b + c$$

The second difference is linear sequence:

$$12a + 2b,\ 18a + 2b$$

Hence the third difference is $6a$. Although I personally have not researched this any further, an interesting pattern that I discovered while expanding polynomials of a higher degree, is that the coefficient of a in the final (constant) difference of a single variable polynomial sequence of degree n is $n!$.

Here is an example: A sequence has first terms 1, 3. Find a function that represents the sequence. From the examples above, the first difference between the two terms is equal to a in $ax + b$, and so $a = 2$. Knowing that the first term is equal to $a + b$, the function is $f(x) = 2x - 1$.

Theorem 1. *If polynomial $P(x)$ represents a sequence, the first $n + 1$ terms are needed to find $P(x)$, where n is the degree of the polynomial.*

Proof. To find the value of every coefficient, there need to be enough terms to calculate the final difference, which only represents the coefficient of a. In a sequence with one term, the polynomial that represents the sequence has degree 0. Hence with two terms, it is a linear function with degree 1. And so, with n terms, the degree of the polynomial is $n - 1$. Alternatively, with a polynomial of degree n, there need to be $n + 1$ terms. □

Remark. *The difference between a sequence and a set might seem quite trivial, but I will define them nonetheless to avoid confusion.*

Definition 1.1. Sequence - A Sequence is a list of terms that are connected through a formula or a function, and using that function can be continued indefinitely.

Definition 1.2. Set - A set is a collection that has a finite number of objects or values.

The important thing to note is that any n amount of numbers in a sequence, no matter how unrelated, they can be modeled using a polynomial with degree $n - 1$. The modeled polynomial only represents the items given (and values in between) which is why this method is applicable to sets.

Definition 1.3. Interpolation - Interpolation is the process of finding unknown data points from a given data set. For example, if the points (x_0, y_0) and (x_1, y_1) are given, and $x_0 < x_n < x_1$, the process of finding the y_n value would be interpolation. Some methods involve finding specific ordered pairs, others find a function f(x) that represents the ordered pairs.

If $Set\ A = 19, 21.3, 12$, it can still be modeled as any polynomial $f(x)$, where $f(n)$ represents the n^{th} value in the set. To demonstrate:

$$2a = (12 - 21.3) - (21.3 - 19),\ 3a + b = (21.3 - 19),\ a + b + c = 19$$

$$a = -5.8,\ b = 19.7,\ c = 5.1.$$

Hence the function $f(x)$ that represents the values in the set is $-5.8x^2 + 19.7x + 5.1$, and the domain is $x \in (1, 2, 3)$. Therefore:

Corollary 1.1. *The polynomial $P(x)$ that represents the Set A, where $A = a_1,\ a_2,\ a_3,\ ...a_n$ and $P(n) = a_n$, has degree $n - 1$ and the domain $x \in (1,\ 2,\ 3...\ n)$, where n is the cardinality of A.*

1.2 Matrices

Where *Set A* $= 1,\ 4,\ 8$, from the previous corollary, the polynomial $P(x)$ representing Set A will have degree 2. Hence:

$$a + b + c = 1$$

$$4a + 2b + c = 4$$

$$9a + 3b + c = 8$$

This can also be written as a matrix in the form $Ax = B$:

$$\begin{bmatrix} 1 & 1 & 1 \\ 4 & 2 & 1 \\ 9 & 3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 8 \end{bmatrix}$$

Using Gaussian Elimination to solve the augmented matrix:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 4 & 2 & 1 & 4 \\ 9 & 3 & 1 & 8 \end{array}\right] \xrightarrow{R_2-4(R_1)\to R_2} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & -2 & -3 & 0 \\ 9 & 3 & 1 & 8 \end{array}\right] \xrightarrow{R_3-9(R_1)\to R_3} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & -2 & -3 & 0 \\ 0 & -6 & -8 & -1 \end{array}\right] \xrightarrow{R_3-3(R_2)\to R_3}$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & -2 & -3 & 0 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

From here, it's clear that $c = -1$, and $-2b = -3(-1)$, and so $b = 3/2$. Finally, since $a + b + c = 1$, and so $a = 1/2$. The polynomial $P(x)$ is $x^2/2 + 3x/2 - 1$.

This matrix method can be applied to any set of numbers with cardinality n. Hence:

Theorem 2. *If Set* $A = a_0,\ a_1 ... , a_n$ *the polynomial* $P(x)$ *that represents Set* A *will be*

$$P(x) \ = \sum_{y=0}^{n} c_{(n-y)} x^{(n-y)} \tag{1}$$

The matrix that represents the solutions to the coefficients of the polynomial is:

$$\begin{bmatrix} 1^n & 1^{n-1} & \dots & 1^{n-n} \\ 2^n & 2^{n-1} & \dots & 2^{n-n} \\ \vdots & \vdots & \ddots & \vdots \\ n^n & n^{n-1} & \dots & n^{n-n} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \tag{2}$$

This is the first method of polynomial interpolation, and it is practically an extended version of solving linear equations.

1.3 Lagrange Interpolation and Polynomials

Obvious from the name, this method is eponymous of Joseph-Louis Lagrange, the Italian mathematician who discovered it. Lagrange Interpolation is often used as to actually interpolate numbers within a set that aren't included in the set. For example, consider the following set of numbers from a function $f(x)$.

x	0.3	1.2	2
y	0.9	6	4

The process of Lagrange interpolation would find an approximation of $f(n)$, where n is any number within the range of x values given, such as f(1).

Definition 1.4. Lagrange Interpolation - In any Set with ordered pairs $(x_0, y_0)...(x_n, y_n)$ such as the one below:

x	x_0	$\dots$	x_n
y	y_0	$\dots$	y_n

The polynomial $P(x)$, representing these ordered pairs as a function is:

$$P(x) = F_0(x)y_1 + F_1(x)y_2 ... F_n(x)y_n$$

$$F_n(x) = \prod_{j=1, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}, k \in 0,\ 1 ... n$$

This can be written as one function:

$$P(x) = \sum_{k=0}^{n} \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j} y_k$$

The functions $F_n(x)$ are the Lagrange polynomials.

Proof. Consider the polynomial $f(x) = y$, which takes x_0, x_1... x_n, and outputs y_0, y_1... y_n, hence creating $n + 1$ ordered pairs, such as (x_n, y_n). $f(x)$ is in the form:

$$f(x) = a_0(x - x_1)(x - x_2)...(x - x_n) + a_1(x - x_0)(x - x_2)...(x - x_n)... + a_n(x - x_0)...(x - x_{n-1})$$

Remark. *Notice how there are no terms where $a_n(x - x_n)$. Where a_n is the coefficient, there is every $(x - x_i)$ term that follows it, except where $i = n$.*

Plugging x_0 into $f(x)$:

$$f(x_0) = a_0(x_0 - x_1)...(x_0 - x_n) + a_1(x_0 - x_0)...(x_0 - x_n)... + a_n(x_0 - x_0)...(x_0 - x_n)$$

$$y_0 = a_0(x_0 - x_1)...(x_0 - x_n) + a_1(0)...(x_0 - x_n) + a_n(0)...(x_0 - x_n)$$

$$y_0 = a_0(x_0 - x_1)(x_0 - x_2)...(x_0 - x_n)$$

Plugging x_1 into $f(x)$:

$$f(x_1) = a_1(x_1 - x_1)...(x_1 - x_n) + a_1(x_1 - x_0)...(x_1 - x_n)... + a_n(x_1 - x_0)...(x_0 - x_n)$$

$$y_1 = a_1(x_1 - x_0)(x_1 - x_2)...(x_1 - x_n)$$

Hence:

$$y_n = a_n(x_n - x_0)(x_n - x_i)...(x_n - x_{n-1}),\ i \in N, 0 < i < n, i \neq n$$

And so, finding the value of a_n:

$$a_n = \frac{y_n}{(x_n - x_0)(x_n - x_1)(x_n - x_2)...(x_n - x_{n-1})}$$

Now knowing the value of any a_n, it can be replaced in the original function $f(x)$:

$$f(x)=\frac{(x-x_1)(x-x_2)...(x-x_n)}{(x_0-x_1)(x_0-x_2)...(x_0-x_n)}y_0+\frac{(x-x_0)(x-x_2)...(x-x_n)}{(x_1-x_0)(x_1-x_2)...(x_1-x_n)}y_1...+\frac{(x-x_0)...(x-x_{n-1})}{(x_n-x_0)...(x_n-x_{n-1})}y_n$$

This can be written as:

$$f(x)=\sum_{k=0}^{n}\prod_{j=0,j\neq k}^{n}\frac{x-x_j}{x_k-x_j}y_k$$

Thus concludes the proof of the Lagrange Interpolation formula. □

An example of Lagrange interpolation, consider the following Set A:

x	1	3	4.5
y	14	43	83.125

Applying Lagrange Interpolation,

$$f(x)=\frac{(x-3)(x-4.5)}{(1-3)(1-4.5)}14+\frac{(x-1)(x-4.5)}{(3-1)(3-4.5)}43+\frac{(x-1)(x-3)}{(4.5-1)(4.5-3)}83.125$$

$$f(x)=3.5x^2+0.5x+10$$

From here, $f(x)$ represents the set of ordered pairs. $f(1)=14$, $f(3)=43$, and $f(4.5)=83.125$.

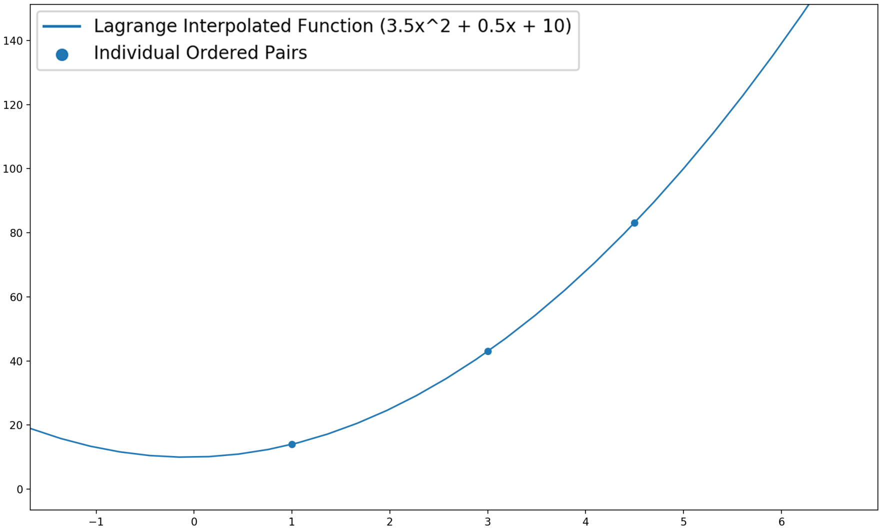

Figure 1: The graph of the points in Set A as well as the graph from its interpolating polynomial.

2 Applications

2.1 Methods in Python

To begin the applications method of this paper, I needed to write each interpolation method into a programming language to actually gauge the usefulness of each. I chose Python because of its versatility with many different modules, which made the process infinitely easier. Initially, transcribing these was difficult, but the more familiar I became with each method, the easier it was. The first method I programmed was the matrix method. I made use of *numpy*, a python package created for scientific computing. This was simple enough, and so I created the following script to take an input and output the A in the matrix in the matrix $Ax = B$ as mentioned earlier.

Listing 1: Initial Matrix Script

```
import numpy as np
n = int(input())
array = []
for original in range(1, n +1):
    in_array = []
    for values in range(n, 0, -1):
        in_array.append(original ** (values - 1))
    array.append(in_array)
print(array(
```

The script above would return:

$$\begin{bmatrix} 1^n & 1^{n-1} & \dots & 1^{n-n} \\ 2^n & 2^{n-1} & \dots & 2^{n-n} \\ \vdots & \vdots & \ddots & \vdots \\ n^n & n^{n-1} & \dots & n^{n-n} \end{bmatrix}$$

Since Numpy solves augmented matries, just an array for B in $Ax = B$ needs to be given. This can simply just be inputted by the user. And so:

Listing 2: Final Matrix Script

```
import numpy as np
B_number = int(input("How many numbers are in this set?"))
```

```
B_array = []
for i in range(1, B_number+1):
    value = int(input())
    B_array.append(value)
array = []
for original in range(1, B_number +1):
    in_array = []
    for values in range(B_number, 0, -1):
        in_array.append(original ** (values - 1))
    array.append(in_array)
solution = np.linalg.solve(array, B_array)
print(solution)
```

If the inputs are 3, and $1, 4, 9$ respectively, the output of the script above will be:

$$[1, 0, 0]$$

Which correctly displays the coefficient of the function $1x^2 + 0x + 0$.

Writing a script for Lagrange Interpolation was admittedly much more difficult. Since Lagrange Interpolation uses actual functions and algebra as opposed to matrices and arrays, I used Sympy, which is a module made for Symbolic Mathematics. I specifically utilized their Symbols in math, like the variable x, as well as their *.expand*, which essentially simplified functions. This specific script was altered from one that was written online [1].

Listing 3: Lagrange Interpolation Script

```
import sympy
x = sympy.symbols(’x’)
x_list = USER-INPUT
y_list = USER-INPUT
z = x - x
o = z + 1
def linterpolation(y, xs=None):
    if xs is None:
        xs = list(range(1, len(y) + 1))
    assert len(y) == len(xs)

    for j, (xj, yj) in enumerate(zip(xs, y)):
        polynomial = o
        for m, xm in enumerate(xs):
            if m != j:
                polynomial *= (x - xm) / (xj - xm)
    result += yj * polynomial
    return sympy.expand(result).evalf()
print(linterpolation(y_list, x_list))
```

Remark. *If no x_list is given, the script will automatically assume that the list is* $1, 2, 3...n$.

If the function *linterpolation* was run with only *y_list* $= [1, 4, 9]$, it would print:

$$x**2$$

2.2 Applications in Image Files

One application that I thought could be useful would be either the compression or encryption of images. The initial idea was to make a script that would initially retrieve the RGB values of each pixel in an image, concatenate them each into strings, to then combine them. From then on, input them into either of the methods. For example, pixel (25, 30) of an image has RGB values 23,123,3. The script would check how the length of each value. If it has one value, it would concatenate two "0"s in front of it. With a length of two, it would add one "0". Eventually, they will be in the form, 023,123,003. Concatenating them together would yield 023123003, and then converting it into an integer would yield 23123003.

Listing 4: Concatenating RGB

```
zero = "0"
def color(x):
    if len(str(x)) == 1:
        x_2 = zero + zero + str(x)
        return str(x_2)
    elif len(str(x)) == 2:
        x_3 = zero + str(x)
        return str(x_3)
    else:
        return x
```

An algorithm can then sort through each integer. If it has a length of 9, the algorithm knows that every 3 values would respectively represents the RGB values. If it has a length of 8, it would take the first 2 values, and then the other 6 would respectively be Green and Blue. Same applies to 7 values.

For the purpose of demonstration (and simplicity), I will only be using images with a 1:1 length to width ratio, because implementing code on them will be much easier. And so, from here on, I can implement the two scripts above and use them on images. From here on, the next step is to extract the RGB values from the image's pixels. Using a python module named "Pillow", this was relatively simple.

Listing 5: Retrieving image Pixels and Dimensions

```
from PIL import Image
im = Image.open('IMAGE_NAME')
rgb_im = im.convert('RGB')
width, height = im.size
```

From here, I decided to use the matrix method. I extracted all the values using a simple for loop, and used them with the matrix found in Listing 1. The full code file can be found here. There was a problem with this method, because it would not compute the solutions of the matrices at 5x5 and above. Because of this, I redid the script using Lagrange Interpolation, which can be found here.

Figure 2: This is the image before its size was changed.

The following graph compares the sizes of the polynomial notes file VS the size of the original image, at sizes n*n.

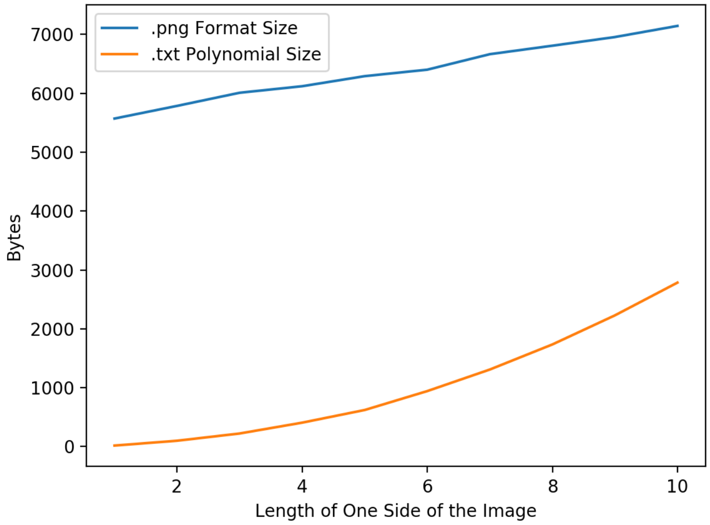

Figure 3: The graph of the size (in bytes) of a polynomial representing an image with dimensions (n*n), as well as the size of the original image as a PNG file.

Unfortunately, despite the fact that the polynomial text files are initially smaller in size, as can be seen from Figure 2, the .png files increase in a seemingly linear relationship, however the polynomial text files increase in what seems to be an exponential relationship. After evaluating both sizes with a 20*20 image, the polynomial is larger in size. Hence, this specific method is not ideal for compression of large images, however I will research this in a future paper.

In terms of encryption, this does work, however since there is no x_n values being given into the Lagrange Interpolation formula, there is no actual encryption happening. If there were a random set of x_n values given, that would be the answer key, which is what is happening here. All I had to add to the original script is a random non-repeating number generator (since functions cannot output different values from one input):

Listing 6: Random Non-Repeating Number Generator

```
import random
random_numbers=[]
for i in range(int(height)*int(width)):
    r=random.randint(1,100)
    if r not in random_numbers: random_numbers.append(r)
print(random_numbers)
```

From this script, the array that it outputs is used as the corresponding x_n values in the ordered pairs in form (x_n, y_n) in Lagrange Interpolation. Hence, without the password/array, the polynomial would be useless. The entire image encryption script (along with the passkey) can be found here.

2.3 Applications in Text

Because of the fact that text is inherently almost as compressed that it can be, I won't be applying the compression technique to text as well. However, encryption is still an application that would be applicable. Using the same "Polynomial Encryption" discussed earlier, every letter would be made into a number. Luckily, ASCII already does this. And so a script would take in a sentence input, and append each character's ASCII decimal value to a list, for it to be inputted to the Lagrange Interpolation function.

Listing 7: Text to ASCII

```
solved = []

text = input("text: ")

for char in text:
    number = ord(char.lower())
    solved.append(number)

print(solved)
```

Using Listing 8, the final Lagrange Interpolation encryption form is the following:

Listing 8: Text to ASCII Interpolation Algorithm

```
import sympy
import random
solved = []
text = input("text: ")
for char in text:
    number = ord(char.lower())
    solved.append(number)
x = sympy.symbols(’x’)
random_numbers=[]
for i in range(len(solved)):
    r=random.randint(1,100)
    if r not in random_numbers: random_numbers.append(r)
z = x - x
o = z + 1
def linterpolation(y, xs=None):
    if xs is None:
        xs = list(range(1, len(y) + 1))
    assert len(y) == len(xs)

    result = z
    for j, (xj, yj) in enumerate(zip(xs, y)):
        polynomial = o
        for m, xm in enumerate(xs):
            if m != j:
                polynomial *= (x - xm) / (xj - xm)
        result += yj * polynomial
    return sympy.expand(result).evalf()
print(random_numbers)
print(linterpolation(solved, random_numbers))
```

To show how the Algorithm works, say the sentence “i am happy” was inputted. The first step of the process is to convert every letter in the word to ASCII numbers. These numbers are appended to the list “solved”. Next, a small function finds the length of the list “solved” and finds that many random non-repeating integers and appends them to the list “random_numbers”. These two lists are inputted to the linterpolation function. The full script outputs the “random_numbers” list, which is essentially a passcode. The other output, a polynomial in one variable. Let it be clear that the polynomial itself is completely useless without the passcode/cyphertext.

If you wish to download the Encryption Algorithm, click here. For the decrypting script, click here.

Remark. *The more characters in the text you want to encrypt, the larger the range for the random non-repeating integer generator. If the range isn't large enough, the algorithm will not work. In Listing 9, the numbers are 1-100 are arbitrary.*

3 Results and Discussion

3.1 Results

The results were rather disappointing, especially with the matrix method. Since this method uses *numpy*'s ability to solve matrice, as opposed to simple arithmetic, it had many factors. Firstly, the condition number of each matrix increased as n increased, as can be seen in Figure 4 below. The more ill-figured (higher the condition number) was, the larger the change in the output from a change in the input. An easy way to think of this is from the Economic concept of Price Elasticity. Bread, being inelastic, has a small difference of quantity demanded in response to a change in price. This would be well defined, with a low condition number. The higher the condition number is, the more the output changes with a small change of an input. The reason this is important is because it relates to matrices, because if a matrix is ill-defined, any small approximation of B in $Ax = B$ will have a large effect on x.

Not only that, but the program itself refused to calculate matrices where $n \geq 5$. Below is a graph of the condition numbers of the matrices.

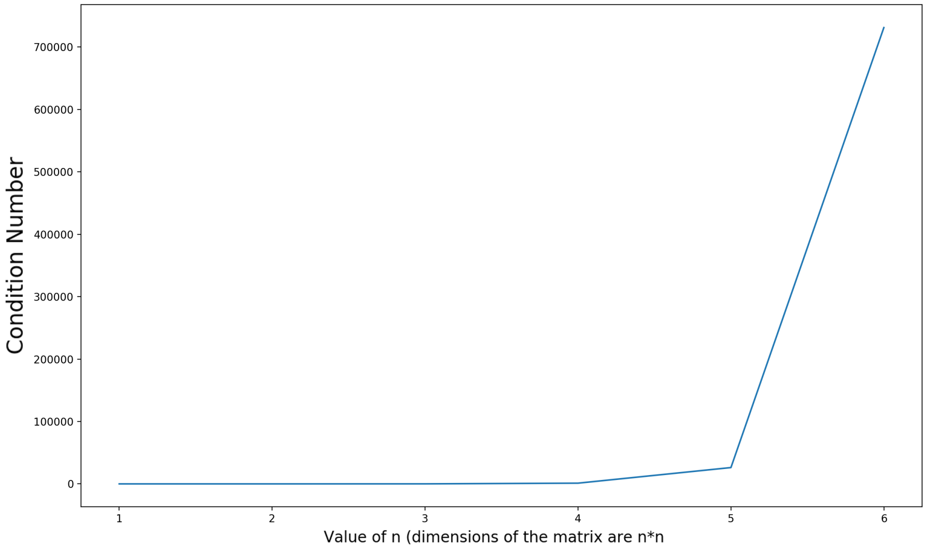

Figure 4: The graph of the condition number against increasing values of n in the matrix, as computed by the $np.linalg.cond()$ function.

Remark. *When inputted into Mathematica, the computational capacity increases, however eventually the program returned error messages that there could be approximations that are effecting the results. Hence, the matrix method becomes inefficient regardless.*

The process of polynomial encryption as detailed in this paper is not limited to Lagrange interpolation; it is essentially applicable to any form of interpolation where the interpolating function takes in ordered pairs. The x_n values would be the password. The main issue with this is the computations that would take place. Especially with text, the ASCII form causes a lot of repeating numbers. For example, the word “Mississippi”, there are many repeated letters. This means that a polynomial with degree 10, must repeatedly go through those corresponding numbers multiple times. The more the function repeats through numbers, the larger (or smaller) the coefficients will be. The larger/smaller they get, approximations become more frequent, and the condition numbers would increase, which is a similar issue faced with the matrices.

3.2 Extensions

There are many things that I want to investigate further, especially in terms of the mathematics behind polynomial interpolation. At the beginning of this paper, I was showing how a polynomial can be interpolated from a set of numbers using the polynomial's term differences. As mentioned before, I noticed that the last difference, which is the coefficient of a, is n!, where the polynomial is $P(x) = a_0x^n\ +\ a_1x^{n-1} + ...a_{n+1}x^0$. In those differences, there were many patterns I noticed, and I want to extend further into that since I cannot find any current research on it. That could possibly be a more effective form of polynomial interpolation. An easier way of interpolation could possibly serve as a better form of encryption, or compression.

Another extension that I will be looking to investigate is the compression aspect, regarding images. Since initial polynomials have a smaller size than the images, a possible solution could be splitting up the image into smaller polynomials that would have a smaller size than the pixels they represent. The same concept can be applied to encrypt large images, or even text. In large text files, such as books, research papers, etc, text can be split into sentences, which should be a suitable size to encrypt. The result would be a collection of polynomials, and a collection of keys, both of which are needed to output the full, original plaintext.

Needless to say, this is an extremely basic form of encryption, and no one should be using this to encrypt any sensitive information.

References

[1] *Finding a polynomial formula for sequence of numbers*. With a comment. by Rory Daulton. URL: https://stackoverflow.com/a/56827866.