Algorithms for polynomial Interpolation

Yushen Hu, Zhijia Xu, Yijian Lian

Abstract

Polynomial interpolation is a fundamental problem in numerical analysis that involves finding a polynomial that passes through a given set of points. In this paper, we will describe various algorithms for polynomial interpolation, including the Lagrange Form Method and the Methods of Divided Differences and their extended transformations, such as Barycentric Lagrange interpolation, Aitken's Method, Neville's Method, and Krogh's Method.

1 Introduction

Polynomial interpolation is a widely used mathematical technique that involves constructing a polynomial function that fits a given set of points exactly. This technique has numerous applications in a variety of fields such as numerical analysis, computer-aided design, signal processing, and data fitting, among others. The accuracy and efficiency of polynomial interpolation algorithms play a critical role in determining the quality of solutions derived from these applications.

Over the years, numerous algorithms have been proposed to perform polynomial interpolation, each with its own strengths and limitations. Some of the well-known methods include Lagrange interpolation, Newton interpolation, and the Neville method. Furthermore, various optimization techniques and hybrid approaches have been introduced to improve the performance of existing algorithms.

This paper provides a comprehensive survey of different algorithms for polynomial interpolation, involving the Lagrange Form Method, Barycentric Lagrange Method, and several extended transformation based Divide Differences Method. We will discuss the theoretical foundations of these algorithms, including their convergence properties and error bounds, and present a comparison of their computational efficiency and accuracy on a range of benchmark problems for different algorithms.

Through error evaluation for each algorithm, we can find the best method with least error and time complexity. Theoretically, every algorithm for polynomial interpolation should output the same value and pass through same data points.

2 Models

In this section, we will introduce several algorithm models for polynomial interpolation. We pick up 5 different algorithms from A comparison of algorithms for polynomial interpolation by Macleod, A. J. (1982), and with an additional Barycentric Lagrange Interpolation algorithm.

2.1 Lagrange Form Method

2.1.1 Lagrange Interpolation

Lagrange interpolation is a method for fitting a polynomial function to a given set of data points. By using n+1 data points (x_0,y_0),(x_1,y_1),...,(x_n,y_n), one can obtain a polynomial function f(x) that passes through all the data points using Lagrange interpolation. The polynomial obtained through Lagrange interpolation is unique for any given set of data points and can be expressed as a weighted sum of individual Lagrange basis functions. These basis functions are constructed so that each function takes a value of 1 at its corresponding data point and takes a value of 0 at all other data points. The Lagrange interpolation polynomial is then a weighted sum of these basis functions.

The Lagrange Form method is defined by:

$$l_j(x) = \frac{(x-x_0)(x-x_1)...(x-x_{j-1})(x-x_{j+1})...(x-x_k)}{(x_j-x_0)(x_j-x_1)...(x_j-x_{j-1})(x_j-x_{j+1})...(x_j-x_k)} = \prod_{m=1,m\neq j}^{k} \frac{x-x_m}{x_j-x_m} \tag{1}$$

The linear combination of corresponding values through the Lagrange interpolating polynomial yields the polynomial for those nodes:

$$p(x) = \sum_{j=0}^{k} y_j l_j(x) \tag{2}$$

Each Lagrange basis polynomial $l_j(x)$ can also be written as the following two forms:

$$l_j(x) = l(x)\frac{w_j}{x-x_j} = \prod_{0<m<k,j\neq m,j=0}^{k} \frac{x-x_m}{x_j-x_m} \tag{3}$$

In this case, the node polynomial is

$$l(x) = \prod_m (x-x_m) \tag{4}$$

The Lagrange polynomial can be expressed in the first barycentric form by factoring out a from the sum:

$$p(x) = l(x) \sum_{j=0}^{k} \frac{w_j}{x - x_j} y_j \tag{5}$$

2.1.2 Barycentric Interpolation

Equation (5) is not the end of the story. Suppose we interpolate, nearby the data l_j, the constant function 1, whose interpolation is exactly itself, then we can get:

$$1 = \sum_{j=0}^{k} l_j(x) \ = l(x) \sum_{j=0}^{k} \frac{w_j}{x - x_j} \tag{6}$$

Dividing (5) by this equation, we can get:

$$p(x) = \frac{l(x) \sum_{j=0}^{k} \frac{w_j}{x - x_j} y_j}{l(x) \sum_{j=0}^{k} \frac{w_j}{x - x_j}} = \frac{\sum_{j=0}^{k} \frac{w_j}{x - x_j} y_j}{\sum_{j=0}^{k} \frac{w_j}{x - x_j}} \tag{7}$$

This is called the second form of the true form of barycentric formula.

2.2 Divide Differences Method

2.2.1 Aitken's

Aitken's algorithm is a numerical method used to speed up the convergence of a sequence of approximations to a root or solution of a function. Specifically, Aitken's algorithm is often used in the context of polynomial interpolation, where it is used to estimate the value of a function at a given point based on a set of data points.
To use Aitken's algorithm for polynomial interpolation, we start with a set of data points and construct a set of polynomials of increasing degree that pass through the data points. We then use Aitken's algorithm to recursively evaluate these polynomials at the given point, gradually converging towards a more accurate estimate of the function value.
Consider

$$p_{k,0} = f_k, k = 0, ..., n \tag{8}$$

We define

$$p_{k,d+1} = \frac{(x_k - t)p_{d,d} - (x_d - t)p_{k,d}}{x_k - x_d} \tag{9}$$

for $k = d + 1, ..., n$ and $d = 0, ..., n - 1$.
Then

$$p_n(t) = p_(n, n) \tag{10}$$

2.2.2 Neville's

Neville's algorithm is another numerical method used for polynomial interpolation. Comparing to Aitken's, it has been more widely used. To use Neville's algorithm for polynomial interpolation, we again start with a set of data points and construct a set of polynomials of increasing degree that pass through the data points. However, unlike Aitken's algorithm, Neville's algorithm does not require recursive evaluation of the polynomials. Instead, it constructs a single polynomial of the same degree that passes through all of the data points and evaluates that polynomial at the given point to estimate the function value.
Similar to Aitken's, we consider

$$p_{k,0} = f_k, k = 0, ..., n \tag{11}$$

We define

$$p_{k,d+1} = \frac{(x_k - t)p_{k-1,d} - (x_{k-d-1} - t)p_{k,d}}{x_k - x_{k-d-1}} \tag{12}$$

for $k = d + 1, ..., n$ and $d = 0, ..., n - 1$.
Then

$$p_n(t) = p_(n, n) \tag{13}$$

2.3 Krogh's Methods

This method is based on the paper written by Fred T. Krogh. Newton's interpolation formula serves as the foundation for various algorithms, such as simple polynomial interpolation, interpolation with supplied derivatives at certain data points, interpolation using piecewise polynomials with a continuous first derivative, and numerical differentiation. These algorithms possess all the benefits of their Aitken-Neville interpolation counterparts, but are more efficient.

2.3.1 Krogh One

For $k = 1, ..., n$, Let:

$$V_0 = f_0$$
$$V = f_k$$
$$V = \frac{V_i - V}{x_i - x_k} \; for \; i = 0, ..., k - 1$$
$$V_k = V$$

Let $Q_0 = 1, P_0 = f_0$.

$$Q_k = (t - x_{k-1})Q_{k-1}$$
$$P_k = P_{k-1} + Q_k V_k$$

Then $P_n(t) = P_n$.

2.3.2 Krogh Two

Define $V_k,\ k = 0, ..., n$, the same as in Krogh One.
Let $V_{k-1} = V_{k-1} + (t - x_{k-1})V_k$ for $k = n, n-1, ..., 1$.
Then $P_n(t) = V_0$.

Krogh Two, which uses nested multiplication, is expected to be faster than Krogh One as it involves evaluating a polynomial using its power form. In other words, the difference between these two methods can be likened to the difference between evaluating a polynomial using nested multiplication and evaluating it in its power form.

3 Code Implementation

In this section, we introduce the pseudo code implementation of the algorithms mentioned above. All of the code will take 3 inputs, namely the data points to be interpolated with two sets of coordinated x and y given respectively, and the x value of the point to be interpolated.

3.1 Lagrange Form Method

To code the Lagrange Form Method, the first thing is to analyze each component of the formula.

The general equation for Lagrange Form Method is

$$p(x) = \sum_{j=0}^{k} y_j l_j(x) \tag{14}$$

Then, expand each term of this polynomial, and then get:

$$l_j(x) = \prod_{0<m<k, j\neq m, j=0}^{k} \frac{x - x_m}{x_j - x_m} \tag{15}$$

So, rewritten form of polynomial function is:

$$p(x) = \sum_{j=0}^{k} \left(\prod_{0<m<k, j\neq m, j=0}^{k} \frac{x - x_m}{x_j - x_m} \right) y_j \tag{16}$$

Then, formula logic can be represented by pseudocode:

Algorithm 1 Algorithm for Lagrange Polynomial Interpolation

```
1: function LAGRANGE(x,y,x_eval)
2:     n = length(x)
3:     result = 0
4:
5:     for i from 0 to n - 1 do
6:         p = 1
7:         for j from 0 to n - 1 do
8:             if i ≠ j then p = p * (x - x[j]) / (x[i] - x[j])
9:             end if
10:        end for
11:        result = result + p * y[i]
12:    end for
13:
14:    return result
```

3.2 Barycentric Lagrange Interpolation

To fully understand the Barycentric form of the equation, it is important to analyze each component individually, then get:

$$p(x) = \frac{l(x)\sum_{j=0}^{k}\frac{w_j}{x-x_j}y_j}{l(x)\sum_{j=0}^{k}\frac{w_j}{x-x_j}} = \frac{\sum_{j=0}^{k}\frac{w_j}{x-x_j}y_j}{\sum_{j=0}^{k}\frac{w_j}{x-x_j}} \tag{17}$$

w_j can be represented by the following equation:

$$w_j = \frac{1}{\prod_{k\neq j} x_j - x_k} \tag{18}$$

Below is our pseudocode for the Barycentric Lagrange Interpolation:

Algorithm 2 Algorithm for Barycentric Lagrange Interpolation

```
function BARYCENTRIC(x,y,x_eval)
    n = length(x)
    w = array of n ones

    for j from 0 to n - 1 do
        for k from 0 to n - 1 do
            if k ≠ j then return w[j] = w[j] / (x[j] - x[k])
            end if
        end for
    end for

    numerator = 0
    denominator = 0

    for j from 0 to n - 1 do
        difference = c - x[j]
        big_component = w[j] / difference
        numerator = numerator + big_component * y[j]
        denominator = denominator + big_component
    end for
return numerator / denominator
```

3.3 Aitken's

Aitken's method and Neville's method are two popular techniques for polynomial interpolation. Both methods use the idea of constructing a table of divided differences to approximate the interpolating polynomial. Aitken's method is a variant of Neville's method that is more efficient for computing the interpolating polynomial at a specific point.

Since Aitken's method and Neville's method share a similar idea, it is appropriate to choose one method to discuss. In this case, we will focus on Neville's method, which is a widely used technique for polynomial interpolation.

3.4 Neville's

Algorithm 3 Algorithm for Neville's Method

```
1: function NEVILLE(x,y,x_eval)
2:     n = length(x)
3:     p = array of n zeros
4:
5:     for k = 0 to n - 1 do
6:         for i = 0 to n - k - 1 do
7:             if k = 0 then
8:                 p[i] = y[i]
9:             else
10:                p[i] = ((x_eval - x[i+k]) * p[i] + (x[i] - x_eval * p[i+1]) / (x[i] - x[i+k])
11:            end if
12:        end for
13:    end for
14:
15:    return p[0]
```

3.5 Krogh's

Algorithm 4 Algorithm for Krogh's Method

```
1: function KROGH(x,y,x_eval)
2:     dydx = array of length(x) zeros
3:     dydx[0] = (y[1] - y[0]) / (x[1] - x[0])
4:     dydx[-1] = (y[-1] - y[-2]) / (x[-1] - x[-2])
5:
6:     for  i in range between 1 and length(x)-1 do
7:         dydx[i] = ((y[i+1] - y[i]) / (x[i+1] - x[i]) + (y[i] - y[i-1]) / (x[i] - x[i-1])) / 2
8:     end for
9:
10:    for i = 0 to length(x) do
11:        if x_eval > x[i] then
12:            idx = i
13:            break
14:        end if
15:    end for
16:
17:    h = x[idx] - x[idx-1]
18:    a = y[idx-1]
19:    b = dydx[idx-1]
20:    c = (3*(y[idx] - y[idx-1])/h**2) - (2*dydx[idx-1]/h) - (dydx[idx]/h)
21:    d = (2*(y[idx-1] - y[idx])/h**3) + (dydx[idx-1]/h**2) + (dydx[idx]/h**2)
22:
23:    dx = x_eval - x[idx-1]
24:    return a + b*dx + c*dx**2 + d*dx**3
```

4 Data Analysis

We implement our pseudo code of the four algorithms respectively using python and Google Colab notebook. To analyze the performance of these four algorithms, we test their runtime and error on different degrees and functions. There are four functions that are involved in testing the performance of the interpolation methods: Exponential, Runge Function, Absolute Value Function, and Trigonometric function. The functions mentioned above are defined as follows:
The exponential function is defined as:

$$f(x) = e^{-x} \tag{19}$$

The Runge function is defined as:

$$f(x) = \frac{1}{1 + 16x^2} \tag{20}$$

The absolute function is defined as:

$$f(x) = |x| \tag{21}$$

The trigonometric function is defined as:

$$f(x) = cos(x) \tag{22}$$

We generated two sets of data points around these functions: The first set of data points is generated by a set of Chebyshev nodes of different degrees scattering on x axis, and the second set of data points is generated by 500 evenly spaced nodes on x axis using numpy's *linspace* function.
The Chebyshev nodes is defined as:

$$x_k = cos(\frac{2k-1}{m}\frac{\pi}{2}) \tag{23}$$

The first set of data points, which is generated by Chebyshev nodes, is used as the input of the four different polynomial interpolation functions. The second set of data points is then used to test and compare the error value for the four different algorithms. For each of the 500 evenly spaced nodes, we generate the different interpolated value corresponding to four algorithms, which will produce 4 numerically different vectors, each with the degree of 500. Then, these vectors will be subtracted by another vector which records the true value of the corresponding function, and the norm of their subtraction will be used as the metric of error. The runtime will be recorded and stored as well during the computation.

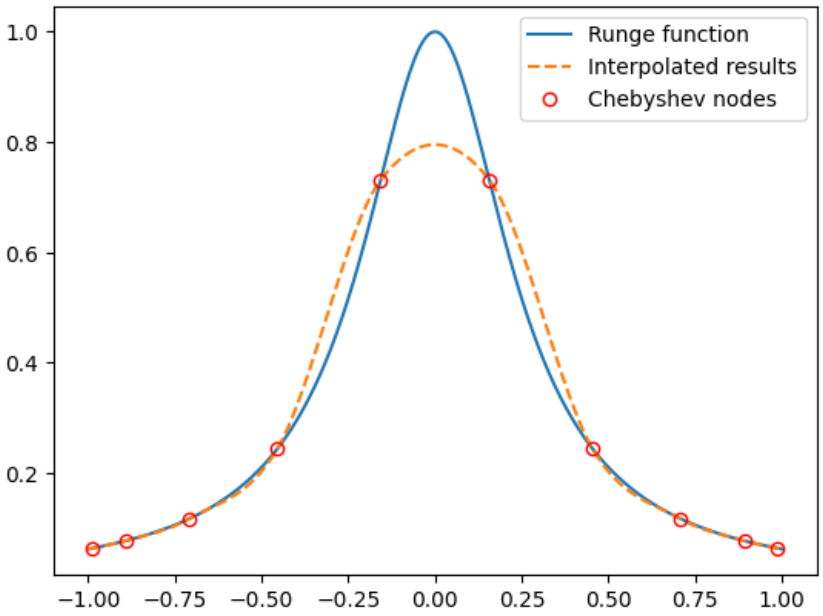


Figure 1: Plot with Runge function using Krogh's method

*Figure*1 visualizes the result of Krogh's method interpolating data points scattering by a set of Chebyshev nodes of degree 9.

4.1 Time complexity

The theoretical runtime is computed forehead of the testing with Big-O notation. Based on the code we have, the Lagrange method gives a runtime of $O(n^2)$, given we have to do 2 nested loops with each of them iterates n times, thus a total of n^2 iterations; The Barycentric Lagrange method is similar to Lagrange, but with one more loop with n iterations, giving it a total of $n^2 + n$ iterations and Big-O of $O(n^2)$ as well (though it may run a little bit slower than Lagrange); The Neville's algorithm has a slightly better efficiency, with a same Big-O of $O(n^2)$, but we only have to do roughly $\frac{(n+1)n}{2}$ iterations, namely $1+2+3+...+n$; The Krogh's algorithm has the best overall efficiency, with only $O(n)$ runtime, which only does 2 iterations, each with n loops.

Figure 2 demonstrates the runtime analysis of the four methods when interpolating the exponential function changing with the increasing degrees of Chebyshev nodes. The x axis is the number of degrees of Chebyshev nodes, also makes it the size of the set of coordinate input taken by the interpolation function. The y axis is the runtime of each of the four polynomial interpolation algorithms.

Based on the plotted graph, we observe that the runtime costs are slightly different than that we have anticipated. By the Big-O of what we have analyzed, Neville's method's runtime should only be half as the Lagrange's and Barycentric's, however

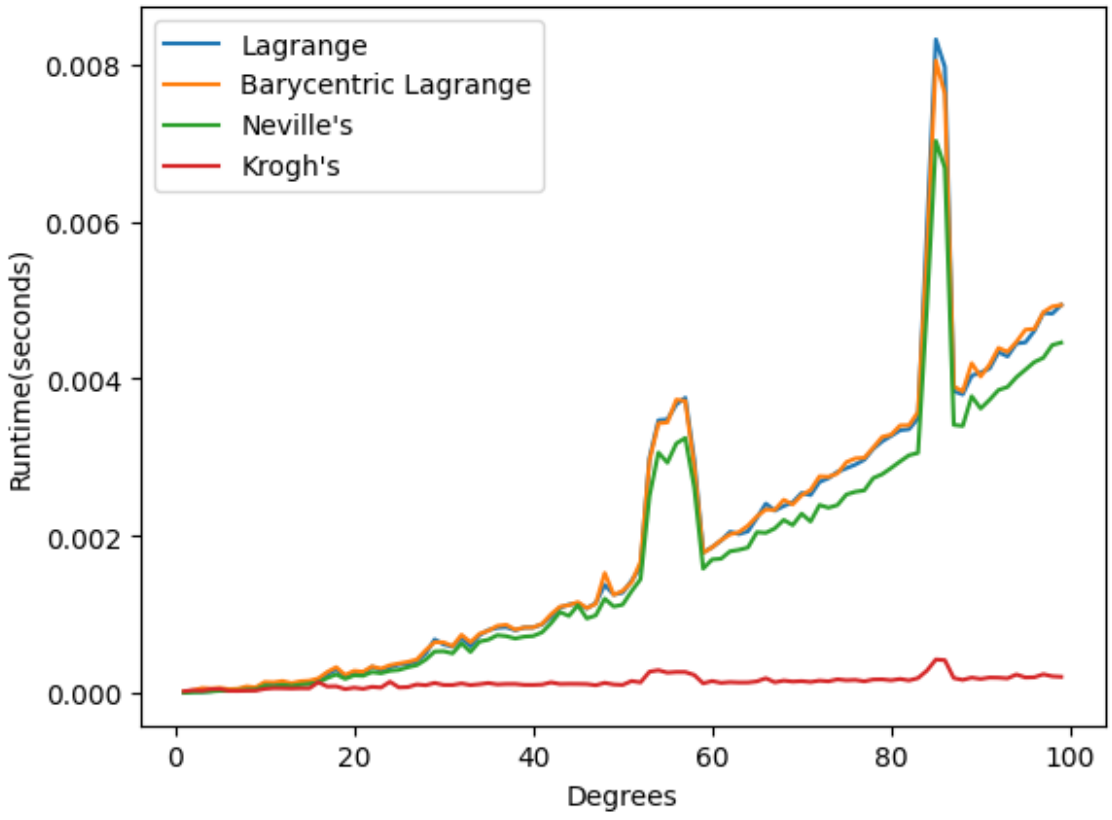


Figure 2: Plot of the runtime

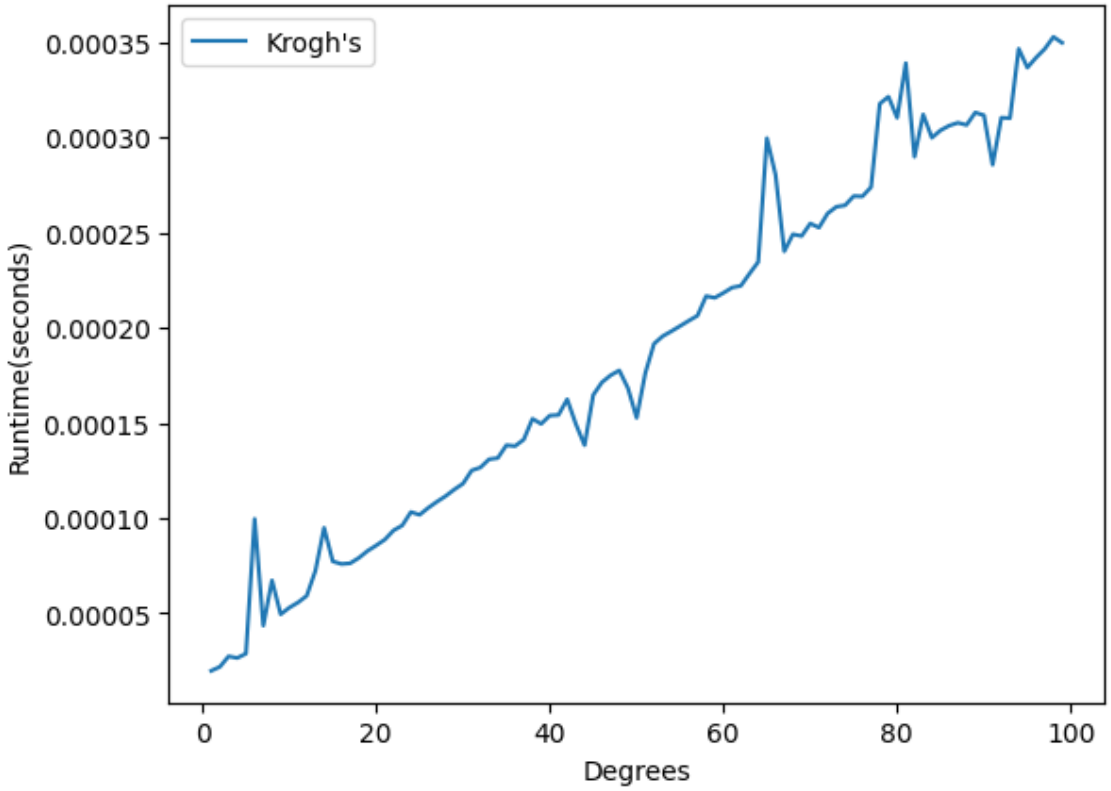


Figure 3: Plot of the runtime on Krogh's only

in the graph, their runtimes are nearly identical, with only a slight margin between Neville's and Lagrange/Barycentric.

Also, we observe nearly no slope on the Krogh's graph, which has the runtime of $O(n)$, due to the runtime of a single iteration is extremely fast to make the slope to be observed by the plot and human eye with small degrees. If we plot Krogh's solely, as Figure 3 has shown, we can rather easily observe the slope. The other three algorithms, however, presents a slight "curve" on their graph, which we anticipate given their $O(n^2)$ runtime.

4.2 Error Analysis

For the error analysis of our 4 methods, the vectors containing the results from the polynomial interpolation will be subtracted by a corresponding vector containing the actual values. The resulting discrepancy will then be quantified by computing the norm of their difference, which serves as a reliable measure of error.

In order to assess the relative efficacy of our four methods, we adopt a two-pronged approach that measures their performance along two distinct dimensions. Firstly, we evaluate their ability to accurately approximate a diverse range of functions, thereby

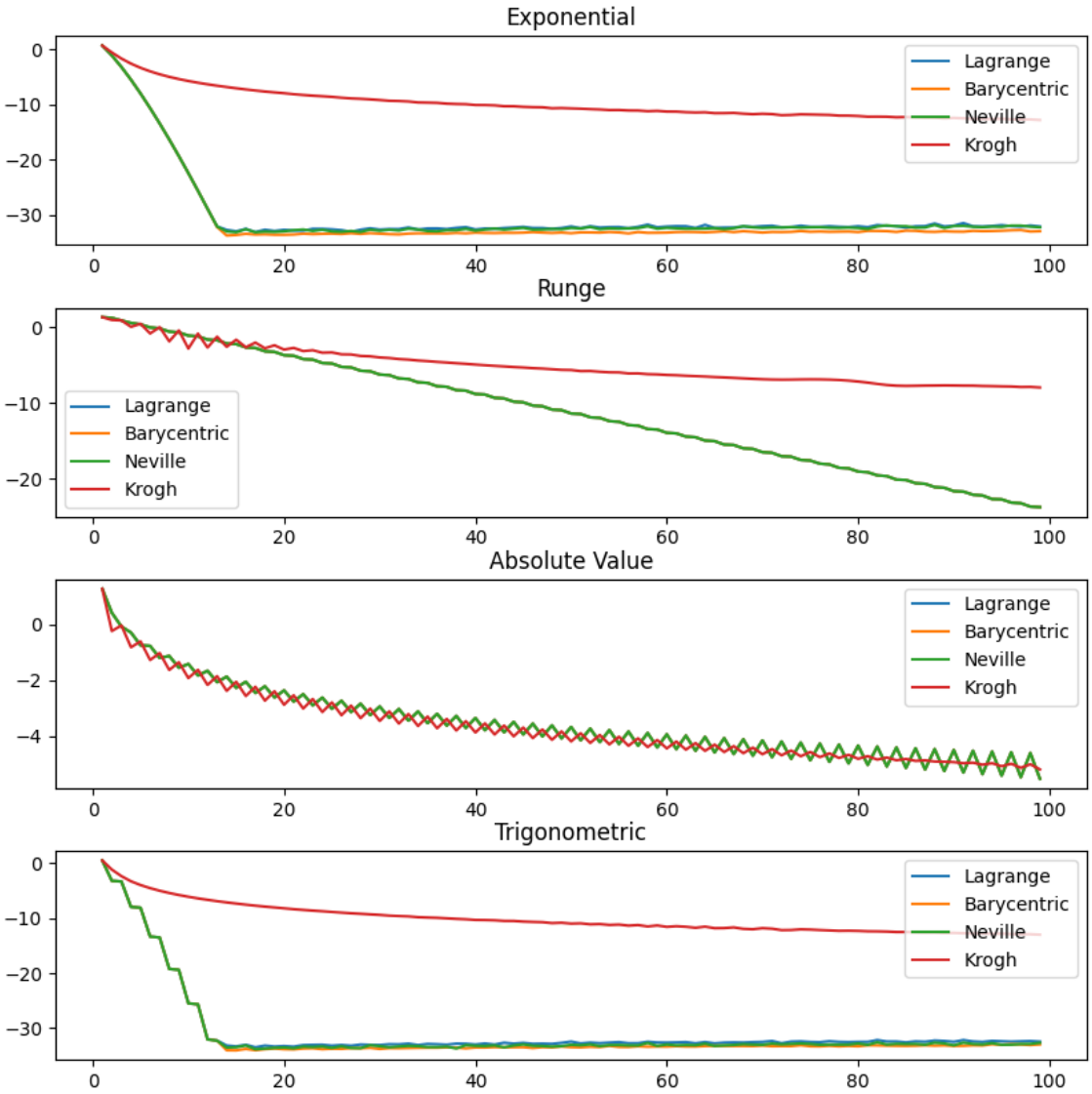


Figure 4: Error vs Degree across Different Functions

providing insight into their overall generalizability and robustness. Secondly, we analyze their performance with an increasing degree of interpolation, enabling us to gauge their effectiveness in handling increasingly complex and nuanced data sets. By systematically measuring and comparing these two critical dimensions, we aim to provide a comprehensive and rigorous evaluation of the methods' relative strengths and weaknesses.

For this part of analysis, we visualized our result in Figure 4. The graph displays four distinct subplots, each of which measures the error incurred by the four methods under consideration on a distinct category of functions. The horizontal axis of the graph represents the degree of interpolation used, while the vertical axis represents the logarithmic scale of the computed error matrices. By presenting this data in a clear and concise manner, we aim to facilitate a more detailed understanding of the performance of each method and the specific functions on which they excel or struggle.

Upon examination of Figure 4, we observe that, on the whole, Krogh's method exhibits a consistently higher error matrix compared to the other methods under consideration. Additionally, we note that the three alternative methods exhibit a remarkably similar behavior across all degrees of interpolation, indicating a relatively comparable level of accuracy and consistency in their performance.

When interpolating an exponential function (represented by $f(x) = e^{-x}$) and a trigonometric function (represented by $f(x) = cos(x)$), we observe a similar trend. Specifically, we note that the difference in error between Krogh's method and the other three methods increases at lower degrees of interpolation and peaks when the degree of interpolation reaches approximately 15. These findings provide further evidence of the relative strengths and weaknesses of each method, particularly with regard to their effectiveness in handling specific types of functions.

The exceptional results occur when interpolating absolute value function (represented by $f(x) = |x|$). Based on our observation, Krogh's has a better performance in absolute value function. We observe an oscillation in Runge function and absolute value function, which can be explained by the parity of the degree of Chebyshev nodes. When the degree is odd, there is a node $x = 0$, makes the error of that region extremely small, but when the degree is even, the interpolation algorithms will not consider the maximum/minimum of the function, thus increasing the error. An interesting observation is that the oscillation nearly disappear for Krogh's algorithm when the degree increases in both Runge function and absolute value function, even though the other three functions have a better performance than Krogh's in Runge function.

We pick up a extreme condition, where the degree is extremely high, as $deg = 500$, far beyond the normal iteration range, and see how these four methods perform. From Figure 4, we expect the errors from Krogh's in Exponential and Trigonometric to be slowly converged into the errors of other three algorithms. As Figure 5 demonstrates, there are still considerable margins between the error of Krogh's and

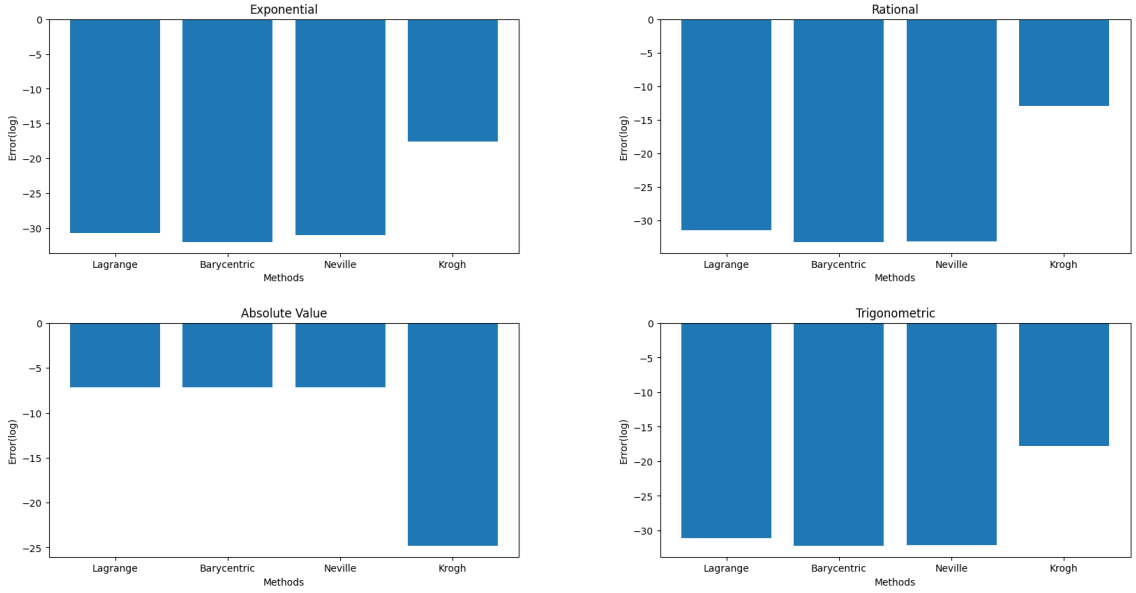


Figure 5: Bar chart of errors when degree is 500

others' in Exponential and Trigonometric. We thus make the assumption that the error of Krogh's algorithm will hardly ever converge to other three's errors till the algorithms are impratical to take such large amount of inputs.

5 Conclusion and Future work

In this paper, we introduce four algorithms for polynomial interpolation and implement them into python code for performance analysis using runtime and error as the metrics. There are a few things that are left for us to discuss in the future.
First, although we evaluate these algorithms under four different functions, there are still a large portion of functions that are left to be discussed. We would like to find a efficient way to generalize the performance of these algorithms in the future.
Secondly, we have discuss the results from the code and plot, but we have yet to find a optimal solution for polynomial interpolation, which involves the metric of time and error. As the results demonstrate, the runtime and error of these algorithms are inversely related, which makes it hard to evaluate the efficiency of these algorithms without a clear metric. We would like to find such metric for evaluation in the future.
Lastly, we implement all these four algorithms, but there are still some rooms left for us to optimize the implementation involving efficiency and precision, we would like to integrate more techniques in the future to optimize the algorithm.

6 Reference

Macleod, A. J. (1982). A comparison of algorithms for polynomial interpolation. Journal of Computational and Applied Mathematics, 8(4), 275-277.

Appendix

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
```

Define parameters

```
#number of the highest degree of the desired polynomial function
NUM_DEG = 100
#number of the inputs
NUM_X = 50
#number of the randomized polynomial functions
LEN = 50
```

Chebyshev Nodes

```
def SetChebyshevNodes(k):
    return np.cos((2*np.arange(k+1)+1)/(2*(k+1))*np.pi)


def GetExpectOutput(input, func_type):
    #exponential
    if func_type == 1:
        return np.exp(-input)
    elif func_type == 2:
        return 1 / (1 + 15 * input**2)
    elif func_type == 3:
        return np.abs(input)
    elif func_type == 4:
        return np.cos(input)
```

Lagrange

```
def Lagrange(x,y,x_eval):
    n = len(x)
    result = 0

    for i in range(n):
        p = 1
        for j in range(n):
            if i != j:
                p *= (x_eval - x[j])/(x[i] - x[j])
        result += p * y[i]

    return result
```

Barycentric Lagrange Algorithm

```
def BarycentricLagrange(x,y,x_eval):
    n = len(x)
    w = np.ones(n)
    for j in range(n):
        for k in range(n):
            if k != j:
                w[j] /= x[j] - x[k]

    x_eval = np.asarray(x_eval)
    n = len(x)
    numerator = 0
    denominator = 0

    for j in range(n):
        difference = x_eval - x[j]
        big_component = w[j]/difference
        numerator += big_component * y[j]
        denominator += big_component

    return numerator/denominator
```

Neville's Algorithm

```
def Neville(x, y, x_eval):
    n = len(x)
    p = n*[0]
    for k in range(n):
        for i in range(n-k):
            if k == 0:
                p[i] = y[i]
            else:
                p[i] = ((x_eval-x[i+k])*p[i]+(x[i]-x_eval)*p[i+1])/(x[i]-x[i+k])
    return p[0]
```

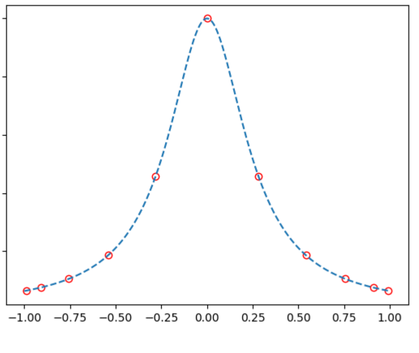
Krogh

```
def Krogh(x, y, x_eval):
    x = np.asarray(x)
    y = np.asarray(y)

    dydx = np.zeros_like(x)
    dydx[0] = (y[1] - y[0]) / (x[1] - x[0])
    dydx[-1] = (y[-1] - y[-2]) / (x[-1] - x[-2])
    for i in range(1, len(x)-1):
        dydx[i] = (((y[i+1] - y[i]) / (x[i+1] - x[i]) +
                   (y[i] - y[i-1]) / (x[i] - x[i-1])) / 2
    idx = 0
    for i in range(len(x)):
        if x_eval > x[i]:
            idx = i
            break
    #print(idx)
    h = x[idx] - x[idx-1]
    a = y[idx-1]
    b = dydx[idx-1]
    c = (3*(y[idx] - y[idx-1])/h**2) - (2*dydx[idx-1]/h) - (dydx[idx]/h)
    d = (2*(y[idx-1] - y[idx])/h**3) + (dydx[idx-1]/h**2) + (dydx[idx]/h**2)

    dx = x_eval - x[idx-1]
    return a + b*dx + c*dx**2 + d*dx**3
```

Plot the example interpolation

```
x1 = SetChebyshevNodes(10)
y1 = GetExpectOutput(x1,2)
x2 = np.linspace(-1,1,500)
y2 = GetExpectOutput(x2,2)

plt.plot(x1,y1,'o',mfc='none',color='r')
plt.plot(x2,y2,'--')
```

```
[<matplotlib.lines.Line2D at 0x7f4c5966c790>]
```



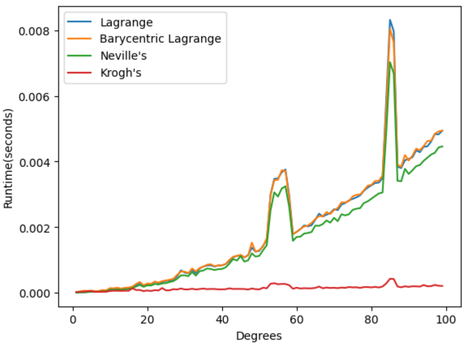
Output three different runtime

```
#When calling runtime, the system will automatically generate a randomized dataset
#and test each method on it.
#The function will return an array of length 4, the first item records Lagrange,
#the second records Barycentric Lagrange, the third records Neville's,
```

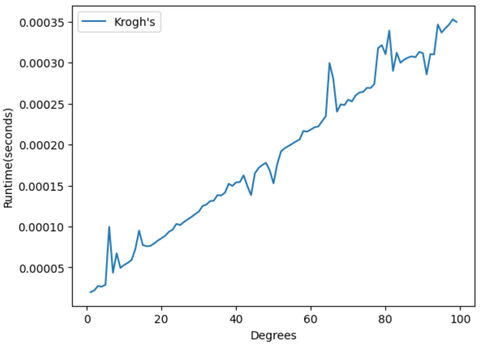
```
#and the third records Krogh.
import time

def Runtime(func_type):
    nt_1 = []
    nt_2 = []
    nt_3 = []
    nt_4 = []
    for i in range(1,NUM_DEG):
        xis = SetChebyshevNodes(i)
        t1 = 0
        t2 = 0
        t3 = 0
        t4 = 0
        for j in range(LEN):
            x_eval = random.uniform(min(xis) + 1,max(xis) - 1)
            y = GetExpectOutput(xis, func_type)

            start = time.time()
            a = Lagrange(xis,y,x_eval)
            t1 = t1 + (time.time() - start)

            start = time.time()
            b = BarycentricLagrange(xis,y,x_eval)
            t2 = t2 + (time.time() - start)

            start = time.time()
            c = Neville(xis,y,x_eval)
            t3 = t3 + (time.time() - start)

            start = time.time()
            d = Krogh(xis,y,x_eval)
            t4 = t4 + (time.time() - start)

        t1 = t1 / LEN
        nt_1.append(t1)

        t2 = t2 / LEN
        nt_2.append(t2)

        t3 = t3 / LEN
        nt_3.append(t3)

        t4 = t4 / LEN
        nt_4.append(t4)

    return [nt_1,nt_2,nt_3,nt_4]
```

Plot the data

```
def PlotRuntime():
    x = [0] * (NUM_DEG - 1)
    for i in range(NUM_DEG-1):
        x[i] = i + 1
    [nt_1,nt_2,nt_3,nt_4] = Runtime(1)
    plt.plot(x, nt_1, label='Lagrange')
    plt.plot(x, nt_2, label='Barycentric Lagrange')
    plt.plot(x, nt_3, label='Neville\'s')
    plt.plot(x, nt_4, label='Krogh\'s')
    plt.xlabel('Degrees')
    plt.ylabel('Runtime(seconds)')
    plt.legend()
    plt.show()


PlotRuntime()
```

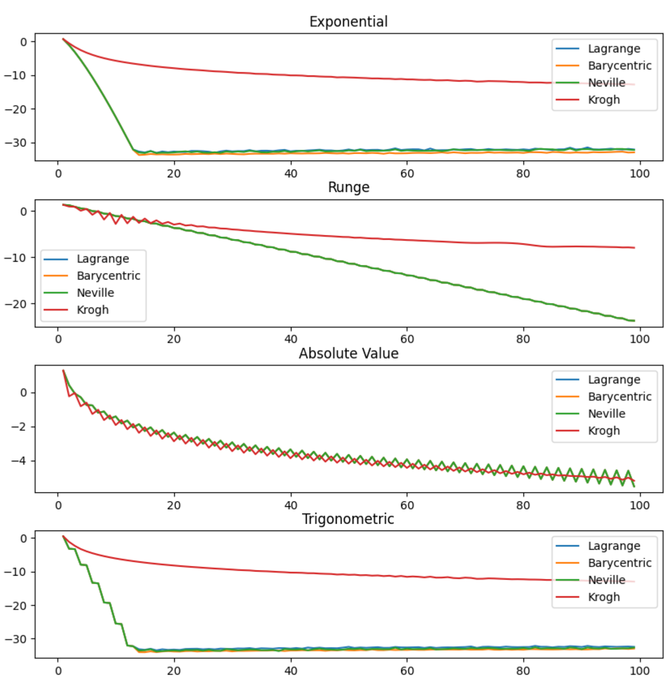



```
x = [0] * (NUM_DEG - 1)
for i in range(NUM_DEG-1):
    x[i] = i + 1
[nt_1,nt_2,nt_3,nt_4] = Runtime(1)
plt.plot(x, nt_4, label='Krogh\'s')
plt.xlabel('Degrees')
plt.ylabel('Runtime(seconds)')
plt.legend()
plt.show()
```



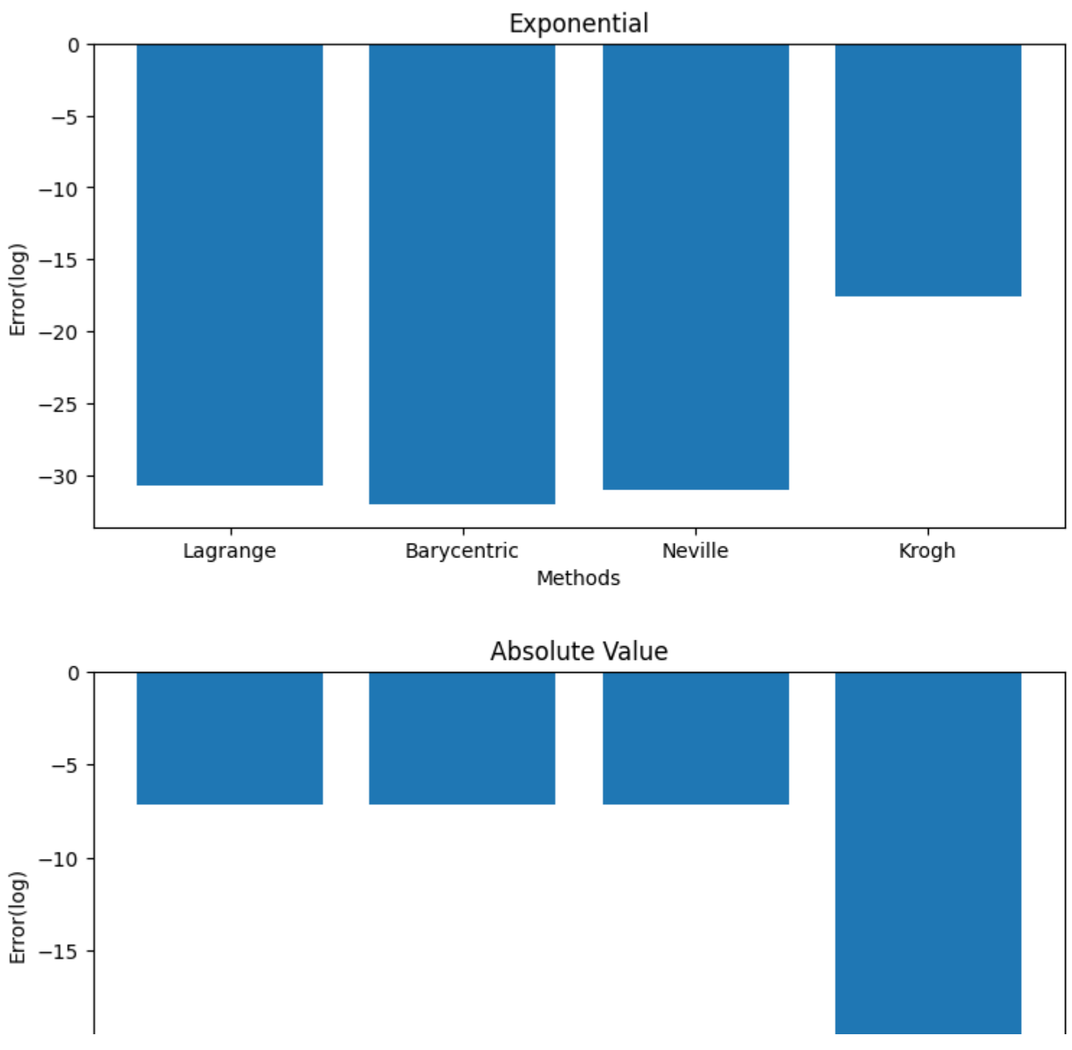

```
def EvaluateError(deg,func_type):
    x = SetChebyshevNodes(deg)
    y = GetExpectOutput(x, func_type)

    xe = np.linspace(-1,1,100)
    ye = GetExpectOutput(xe, func_type)
    ye1 = [0] * len(xe)
    ye2 = [0] * len(xe)
    ye3 = [0] * len(xe)
    ye4 = [0] * len(xe)

    for i in range(len(xe)):
        p1 = Lagrange(x,y,xe[i])
        p2 = BarycentricLagrange(x,y,xe[i])
        p3 = Neville(x,y,xe[i])
        p4 = Krogh(x,y,xe[i])
        ye1[i] = p1
        ye2[i] = p2
        ye3[i] = p3
        ye4[i] = p4

    return [np.linalg.norm(ye-ye1),np.linalg.norm(ye-ye2),np.linalg.norm(ye-ye3),np.linalg.norm(ye-ye4)]


def PlotError():
    x = [0] * (NUM_DEG - 1)
    y1 = []
    y2 = []
    y3 = []
    y4 = []
    for i in range(len(x)):
        x[i] = i + 1
        y1.append(EvaluateError(i + 1,1))
        y2.append(EvaluateError(i + 1,2))
        y3.append(EvaluateError(i + 1,3))
        y4.append(EvaluateError(i + 1,4))

    e11 = [i[0] for i in y1]
    e12 = [i[1] for i in y1]
    e13 = [i[2] for i in y1]
```

```
    e14 = [i[3] for i in y1]

    e21 = [i[0] for i in y2]
    e22 = [i[1] for i in y2]
    e23 = [i[2] for i in y2]
    e24 = [i[3] for i in y2]

    e31 = [i[0] for i in y3]
    e32 = [i[1] for i in y3]
    e33 = [i[2] for i in y3]
    e34 = [i[3] for i in y3]

    e41 = [i[0] for i in y4]
    e42 = [i[1] for i in y4]
    e43 = [i[2] for i in y4]
    e44 = [i[3] for i in y4]

    for i, d in enumerate(e11):
        e11[i] = math.log(d)

    for i, d in enumerate(e12):
        e12[i] = math.log(d)

    for i, d in enumerate(e13):
        e13[i] = math.log(d)

    for i, d in enumerate(e14):
        e14[i] = math.log(d)


    for i, d in enumerate(e21):
        e21[i] = math.log(d)

    for i, d in enumerate(e22):
        e22[i] = math.log(d)

    for i, d in enumerate(e23):
        e23[i] = math.log(d)

    for i, d in enumerate(e24):
        e24[i] = math.log(d)


    for i, d in enumerate(e31):
        e31[i] = math.log(d)

    for i, d in enumerate(e32):
        e32[i] = math.log(d)

    for i, d in enumerate(e33):
        e33[i] = math.log(d)

    for i, d in enumerate(e34):
        e34[i] = math.log(d)


    for i, d in enumerate(e41):
        e41[i] = math.log(d)

    for i, d in enumerate(e42):
        e42[i] = math.log(d)

    for i, d in enumerate(e43):
        e43[i] = math.log(d)

    for i, d in enumerate(e44):
        e44[i] = math.log(d)

    fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(10,10))

    ax1.plot(x,e11,label='Lagrange')
    ax1.plot(x,e12,label='Barycentric')
    ax1.plot(x,e13,label='Neville')
    ax1.plot(x,e14,label='Krogh')
    ax1.legend()
    ax1.set_title('Exponential')

    ax2.plot(x,e21,label='Lagrange')
    ax2.plot(x,e22,label='Barycentric')
    ax2.plot(x,e23,label='Neville')
    ax2.plot(x,e24,label='Krogh')
    ax2.legend()
    ax2.set_title('Runge')

    ax3.plot(x,e31,label='Lagrange')
    ax3.plot(x,e32,label='Barycentric')
    ax3.plot(x,e33,label='Neville')
    ax3.plot(x,e34,label='Krogh')
    ax3.legend()
    ax3.set_title('Absolute Value')

    ax4.plot(x,e41,label='Lagrange')
    ax4.plot(x,e42,label='Barycentric')
    ax4.plot(x,e43,label='Neville')
    ax4.plot(x,e44,label='Krogh')
    ax4.legend()
    ax4.set_title('Trigonometric')

    fig.subplots_adjust(wspace=0.3,hspace=0.3)

PlotError()
```




```
grouped_data=[]
for i in range(1,5):
  grouped_data.append(EvaluateError(500,i))


for i, d in enumerate(grouped_data):
  for j, e in enumerate(d):
    grouped_data[i][j]=math.log(e)


func=['Exponential','Rational', 'Absolute Value', 'Trigonometric']
method=['Lagrange','Barycentric',"Neville",'Krogh']
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(20,10))

ax1.bar(method, grouped_data[0])
ax1.set_xlabel('Methods')
ax1.set_ylabel('Error(log)')
ax1.set_title(func[0])
```

```
ax2.bar(method, grouped_data[1])
ax2.set_xlabel('Methods')
ax2.set_ylabel('Error(log)')
ax2.set_title(func[1])

ax3.bar(method, grouped_data[2])
ax3.set_xlabel('Methods')
ax3.set_ylabel('Error(log)')
ax3.set_title(func[2])

ax4.bar(method, grouped_data[3])
ax4.set_xlabel('Methods')
ax4.set_ylabel('Error(log)')
ax4.set_title(func[3])

fig.subplots_adjust(wspace=0.3,hspace=0.3)
```




```
grouped_data
```

```
[[-30.74997466064677,
  -31.961158201393836,
  -30.98410574875419,
  -17.61989445907397],
 [-31.53669360218438,
  -33.23268044136614,
  -33.15340220903044,
  -12.928307521957501],
 [-7.183766384535591,
  -7.183766384535272,
  -7.1837663845355735,
  -24.792960365993068],
 [-31.078065552064007,
  -32.20218180417801,
  -32.1107236322823,
  -17.795452007112676]]
```

```
x = SetChebyshevNodes(9)
y = GetExpectOutput(x,2)
xe = np.linspace(-1,1,500)
ye = GetExpectOutput(xe,2)

yy = []
for xx in xe:
    yy.append(Krogh(x,y,xx))

plt.plot(xe,ye,'-',label='Runge function')
plt.plot(xe,yy,'--',label='Interpolated results')
plt.plot(x,y,'o',mfc='none',color='r',label='Chebyshev nodes')
plt.legend()
plt.show()
```