PHYS 222 Chapter 1: Approximation of a Function

Fahed Abu Shaer

September 2023

In numerical analysis, the results obtained from computations are always approximations of the desired quantities and in most cases are within some uncertainties.

Interpolation is the process of estimating values between two known data points or within a set of discrete data points. The primary goal of interpolation is to construct a continuous function that passes through the given data points or approximates them. This allows for predictions or obtaining values at points that were not originally part of the data set.

1 Linear Interpolation

Consider a discrete data set given from a discrete function $f_i = f(x_i)$, the simplest way to obtain the approximation of $f(x)$ for $x\epsilon[x_i, x_{i+1}]$ is the construct a straight line between x_i and x_{i+1}. The equation of a line is

$$y = ax + b$$

where

$$a = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

and

$$y_i = ax_i + b \Rightarrow y_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \cdot x_i + b \Rightarrow$$

$$b = y_i - \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \cdot x_i$$

Translating into code and applying it to data generated by $y = x^2$,

```
x = np.linspace(-10, 10, 100)
noise = np.random.randint(0, 10)
y = x ** 2 + noise
plt.plot(x,y, color = 'k') #shows the generated function in black
x = np.linspace(-10, 10, 10)
```

```
y = x ** 2 + noise
plt.scatter(x, y)
for i in range(len(x) - 1):
    x1 = x[i]
    x2 = x[i+1]
    y1 = x1 ** 2 + noise
    y2 = x2 ** 2 + noise
    a = (y2 - y1)/(x2 - x1)
    b = y1 - a * x1
    y1 = a * x1 + b
    y2 = a * x2 +b
    plt.plot([x1, x2], [y1, y2])
plt.ylim(-1, 115)
plt.show()
```

This result of the code,

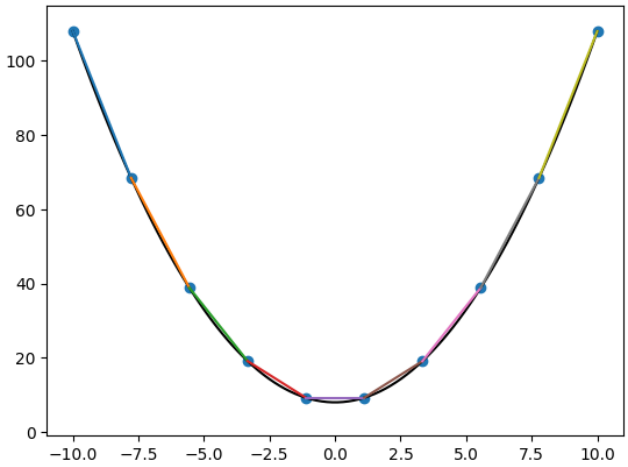

Figure 1: Linear Interpolation in Colors and the Generating Function in Black

The lines are a decent approximate of the function, however it is very sharp, but it doesn't fit the function that much which results in higher errors in the approximated values.

2 Lagrange Polynomials

Theorem 2.1 *Westress Theorem*
Consider a continuous function $f(x)$ *defined over the interval* $[c, d]$*, then for* $\varepsilon > 0$ $\exists$ $P(x)$ *such that* $|f(x) - P(x)| < \varepsilon$ $\forall$ $x \in [c, d]$*, where* $P_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0 x^0$

Starting from the equation that was obtained in linear interpolation,

$$y = \frac{y_1 - y_0}{x_1 - x_0} \cdot x + y_0 - \frac{y_1 - y_0}{x_1 - x_0} \cdot x_0 \Rightarrow y = \frac{(y_1 - y_0)x + (x_1 - x_0)y_0 - (y_1 - y_0)x_0}{(x_1 - x_0)} \Rightarrow$$

$$y = y_0 \left(\frac{x_1 - x_0 - x - x_0}{x_1 - x_0} \right) + y_1 \left(\frac{x - x_0}{x_1 - x_0} \right) \Rightarrow$$

$$y = \frac{x - x_1}{x_0 - x_1} y_0 + \frac{x - x_0}{x_1 - x_0} y_1 \Rightarrow$$

$$y = L_0(x) f(x_0) + L_1(x) f(x_1)$$

So, the general form of the **Lagrange polynomials** is

$$\prod \frac{x - x_m}{x_j - x_m}$$

The code that implement this on data generated by the function $y = x^2$,

```
noise = np.random.randint(0, 10)
x = np.linspace(-10, 10, 10)
y = x ** 2 + noise
plt.scatter(x, y)
for i in range(0, len(x) - 2):
    x1 = x[i]
    x2 = x[i+1]
    x3 = x[i+2]
    y1 = x1 ** 2 + noise
    y2 = x2 ** 2 + noise
    y3 = x3 ** 2 + noise
    xs = np.linspace(x[i], x[i + 2], 10)
    L1 = ((xs - x2)/(x1 - x2)) * ((xs - x3)/(x1 - x3))
    L2 = ((xs - x1)/(x2 - x1)) * ((xs - x3)/(x2 - x3))
    L3 = ((xs - x1)/(x3 - x1)) * ((xs - x2)/(x3 - x2))
    y = L1 * y1 + L2 * y2 + L3 * y3
    plt.plot(xs, y, color = 'm')
plt.ylim(-10, 115)
plt.show()
```

The resulting graph,

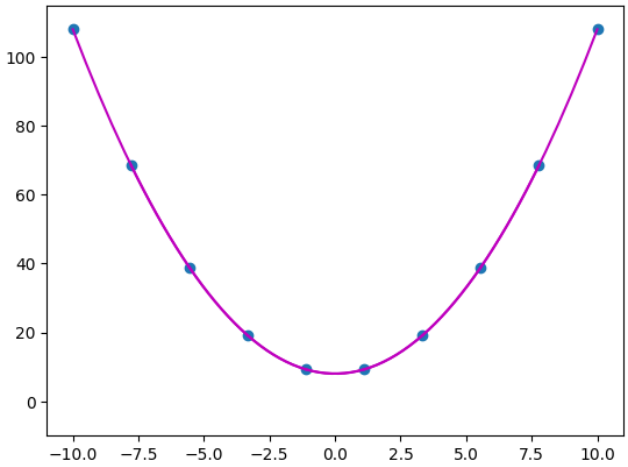

Figure 2: 2nd Order Lagrange Interpolation

This method yielded a smoother graph that fit the data point, but the margin of error is still big.

3 The Aitken's Method

One way to achieve the Lagrange interpolation efficiently is by performing a sequence of linear interpolation. Consider four data point with their respective coordinates,

$$(x_0, y_0),\ (x_1, y_1),\ (x_2, y_2),\ (x_3, y_3)$$

$$P_{01} = \frac{x - x_1}{x_0 - x_1} P_0 + \frac{x - x_0}{x_1 - x_0} P_1$$

$$P_{12} = \frac{x - x_2}{x_1 - x_2} P_1 + \frac{x - x_1}{x_2 - x_1} P_2$$

$$P_{23} = \frac{x - x_3}{x_2 - x_3} P_2 + \frac{x - x_2}{x_3 - x_2} P_3$$

$$P_{012} = \frac{x - x_2}{x_0 - x_2} P_{01} + \frac{x - x_0}{x_2 - x_0} P_{12}$$

$$P_{123} = \frac{x - x_1}{x_3 - x_1} P_{12} + \frac{x - x_1}{x_3 - x_1} P_{23}$$

$$P_{0123} = \frac{x - x_3}{x_0 - x_3} P_{012} + \frac{x - x_0}{x_3 - x_0} P_{123}$$

A recursive pattern can be noticed, so the general form of the polynomial is,

$$P_{i..j} = \frac{x - x_j}{x_i - x_j} P_{i...j-1} + \frac{x - x_i}{x_j - x_i} P_{i+1...j}$$

The recursive function that is constructed to reflect that relation is,

```
def Aitkin (x, P, first, last):
    if first == last:
        return P[first]
    elif first != last:
        return ((x - x[last])/(x[first] - x[last])) * Aitkin(x, P, first, last - 1)
             + ((x - x[first])/(x[last] - x[first])) * Aitkin(x, P, first + 1, last)
```

and that function is applied on a data set generated from the function $y = \frac{1}{1+x^2}$

```
x = np.linspace(-50, 50, 200)
y = 1/(1 + x ** 2)
plt.plot(x, y, color = 'k') #shows the generated function
x = np.linspace(-50, 50, 20)
y = 1/(1 + x ** 2)
plt.scatter(x, y)
for i in range(len(x) - 1):
    xs = np.linspace(x[i], x[i+1], 10)
    ys = 1/(1 + xs ** 2)
    Ps = Aitkin(xs, ys, 0, len(xs) - 1)
    plt.plot(xs, Ps, color = 'r')
plt.show()
```

and the result of the code is,

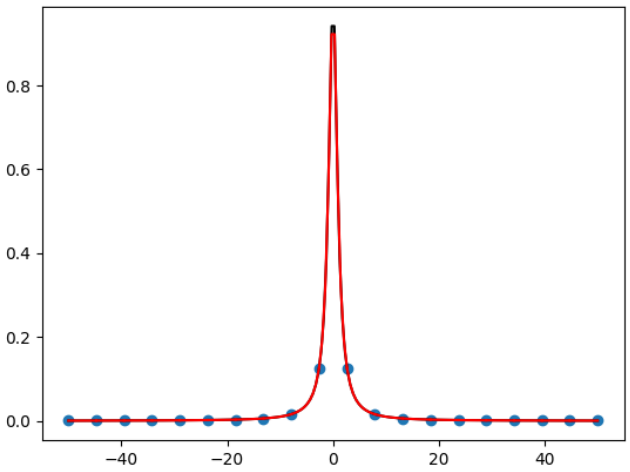

Figure 3: Aitken's Method for Approximating in Red and the Generating Function in Black

The interpolation is almost exact, but there are some limitation and uncertainties.

4 Splines

In many instances, we have a set of data that varies rapidly over the range of interest, then there is no such things as a global polynomial behavior, so we want to fit the function locally and to connect each piece of the function smoothly.

A **spline** is a tool that interpolates the data locally through a polynomial and fits the data overall by connecting each segment of the interpolation polynomial by matching the function and it's derivatives at the data points.

Consider a discrete data set $f_i = f(x_i)$ for $i = 0, 1, ..., n$, we can use mth-order polynomial to approximate $f(x)$ for $x \epsilon [x_i, x_{i+1}]$, $P_i(x) = \sum_{k=0}^{m} c_{ik} x^k$. The coefficients c_{ik} are determined from the smoothness conditions at the non-boundary data points with the lth-order derivative there satisfying the condition $P_i^{(l)}(x_{i+1}) = P_{i+1}^{(l)}(x_{i+1})$.
Hence, the equations to find the polynomial that fit between the two points x_i and x_{i+1} are,

$$a_n x_i^n + a_{n-1} x_i^{n-1} + ... + a_0 = y_i$$

$$a_n x_{i+1}^n + a_{n-1} x_{i+1}^{n-1} + ... + a_0 = y_{i+1}$$

$$P_{i-1}^{(n)}(x_i) = P_i^{(n)}(x_i)$$

Let the column matrix containing the coefficients $(a_n, a_{n-1}, ...a_0)$ be C, the matrix containing what is multiply the coefficients be X, and the column matrix containing what is on the right side of the equations be Y, so the system reduces to $XC = Y$, and to solve this system you just multiply to the left by the inverse of X, $X^{-1}XC = X^{-1}Y \Rightarrow C = X^{-1}Y$

4.1 Quadratic Splines

The interpolation of data using 2nd order spline is called **quadratic splines**. To interpolate, a matrix is needed to be constructed based on the following system of equation where the interval is $[x_i, x_{i+1}]$

$$ax_i^2 + bx_i + c = y_i$$

$$ax_{i+1}^2 + bx_{i+1} + c = y_{i+1}$$

$$P_{i-1}^{'}(x_i) = P_i^{'}(x_i)$$

In Python, I constructed a function that construct that matrix, solves it, and gives me the coefficients.

```
def quad_splines(x, y):
    n = len(x)
    A = np.zeros([3 * n - 4, 3 * n - 4])
    Y = np.zeros(3 * n - 4)
    for i in range(2):
        A[i, 0] = x[i]
        A[i, 1] = 1
    j = 0
    for i in range(2, 2 * n - 2, 2):
        k = int(i/2)
        A[i, j + 2] = x[k] ** 2
        A[i + 1, j + 2] = x[k + 1] ** 2
        A[i, j + 3] = x[k]
        A[i + 1, j + 3] = x[k + 1]
        A[i, j + 4] = 1
        A[i + 1, j + 4] = 1
        j = j + 3
    A[2 * n - 2, 0] = 1
    A[2 * n - 2, 1] = -2 * x[1]
    A[2 * n - 2, 3] = -1
    m, l = 0, 2
    for i in range(2 * n - 1, 3 * n - 4):
        A[i, m + 2] = 2 * x[l]
        A[i, m + 3] = 1
        A[i, m + 5] = -2 * x[l]
        A[i, m + 6] = -1
        m = m + 3
        l = l + 1
    Y[0] = y[0]
    Y[1] = y[1]
    for i in range(2, 2 * n - 2):
        if i % 2 == 0:
            k = int(i/2)
        else:
            k = int(i/2) + 1
        Y[i] = y[k]
    C = np.linalg.solve(A, Y)
    return C
```

I applied the function on a data set generated by the function $\frac{1}{1+x^2}$

```
x = np.linspace(-20, 20, 40)
y = 1 / (x**2 + 1)
plt.scatter(x, y)
C = quad_splines(x, y)
x0 = np.linspace(x[0], x[1], 20)
p0 = C[0] * x0 + C[1]
plt.plot(x0, p0)
j = 1
for i in range(2, len(C), 3):
    xs = np.linspace(x[j], x[j+1], 10)
    ps = C[i] * xs ** 2 + C[i + 1] * xs + C[i + 2]
    plt.plot(xs, ps)
    j = j + 1
plt.show()
```

After running the code, this is the resulting graph,

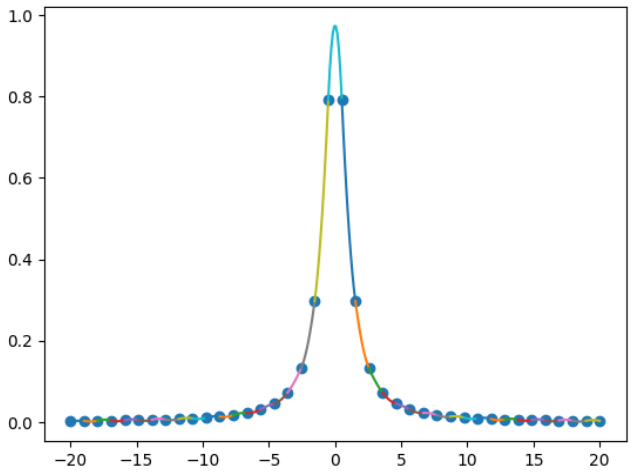

Figure 4: Approximation of Data Set Using Quadratic Splines

The curve seems to perfectly fit the data points and the space between them. It is better than using the Aitken's method to interpolate the same function.

4.2 Cubic Splines

I imported the package scipy (scp) from the Python library that implements approximation using cubic splines, 3rd order interpolation, and this is the code I ran,

```
x = np.linspace(-20, 20, 40)
y = 1 / (x**2 + 1)
plt.scatter(x, y)
for i in range(len(x) - 1):
    xs = np.linspace(x[i], x[i + 1], 10)
    ys = 1 / (xs ** 2 + 1)
    P = scp.interpolate.CubicSpline(xs, ys)
    plt.plot(xs, P(xs))
plt.show()
```

This is the results I got:

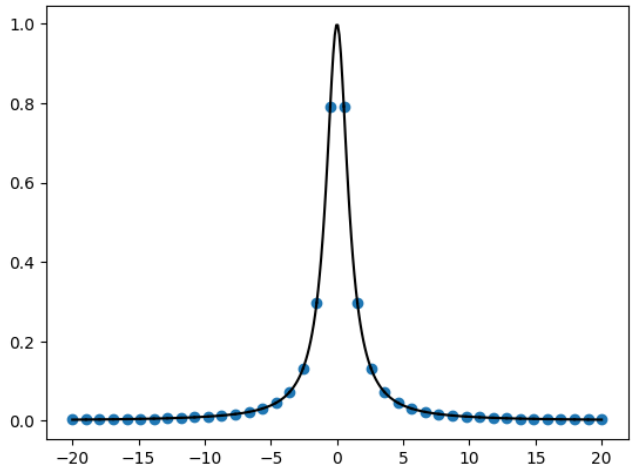

Figure 5: Approximation of a Data Set Using Cubic Splines

This yielded similar results to the one I ran to get the approximation using quadratic splines. This raised the question of how different orders of the polynomials used looked like?

5 Least Square Approximation

Interpolation is mainly used to find the local approximation of a given discrete set of data, like the splines method, but in some situations we need to know the global behavior of a set of data in order to understand the general trend of our observation. So, the most common approximation scheme is based on the **least**

squares of the differences between the approximation, $p_m(x) = \sum_{k=0}^{m} a_k x^k$, and the data.

Consider the first-order case where $m = 1$, the approximation is $p_1 = a_0 + a_1 x$, and the function is $f(x)$, so the least square error is $\chi^2 = \sum_{i=0}^{n}(p_1(x_i) - f(x_i))^2$. To get the least square approximation we have to minimize χ^2.

$$\frac{\partial \chi^2}{\partial a_0} = 0 \Rightarrow \sum_{i=0}^{n}(a_0 + a_1 x_i - f(x_i))^2 = 0 \Rightarrow (n+1)a_0 + \sum_{i=0}^{n} a_1 x_i - \sum_{i=0}^{n} f(x_i) = 0$$

$$\text{let } C_1 = \sum_{i=0}^{n} x_i \text{ and } c_2 = \sum_{i=0}^{n} f(x_i)$$
$$\Rightarrow (n+1)a_0 + c_1 a_1 - c_2 = 0$$

$$\frac{\partial \chi^2}{\partial a_1} = 0 \Rightarrow \sum_{i=0}^{n}(a_0 + a_1 x_i - f(x_i))^2 = 0 \Rightarrow \sum_{i=0}^{n} x_i a_0 + \sum_{i=0}^{n} a_1 x_i x_i - \sum_{i=0}^{n} f(x_i) x_i = 0$$

$$\text{let } C_3 = \sum_{i=0}^{n} x_i^2 \text{ and } c_4 = \sum_{i=0}^{n} x_i f(x_i)$$
$$\Rightarrow c_1 a_0 + c_3 a_1 - c_4 = 0$$

Therefore, the system of equation to find the coefficients is,

$$\begin{cases} (n+1)a_0 + c_1 a_1 - c_2 = 0 \\ c_1 a_0 + c_3 a_1 - c_4 = 0 \end{cases}$$

We construct the following matrices to solve the system,

$$C = \begin{pmatrix} n+1 & c_1 \\ c_1 & c_3 \end{pmatrix}$$

$$a = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

$$C = \begin{pmatrix} c_2 \\ c_4 \end{pmatrix}$$

and we solve the following equation by multiplying on the left by the inverse of C,

$$Ca = d \Rightarrow C^{-1}Ca = C^{-1}d \Rightarrow a = C^{-1}d$$

One way to construct the matrices is to use a Vondermond matrix, a matrix that contains all powers of x_i up to a nth order.

$$V = \begin{pmatrix} x_0^n & x_0^{n-1} & ... & x_0 & 1 & \\ ... & ... & ... & ... & ... & ... \\ x_i^n & x_i^{n-1} & ... & x_i & 1 & \end{pmatrix}$$

it was noticed that $C = V^T V$, $d = V^T y$ where y is the matrix containing the ordinates of the data points $\Rightarrow$ $a = C^{-1}d = (V^T V)^{-1} V^T y$
In my code, I constructed a function that would construct the Vandermond matrix,

```
def Vandermond(x, y, n):
    V = np.empty([len(x), n + 1])
    for i in range(n + 1):
        V[:, i] = x ** i
    return V
```

I also wrote a function that would solve the system of equations and give me the coefficients of the polynomials,

```
def LeastSquare(V, y):
    Vt = np.transpose(V)
    A = Vt @ V
    Y = Vt @ y
    b = np.linalg.solve(A, Y)
    return b
```

In addition to that, I constructed another function that would fit the polynomial given the data, coefficients, and order,

```
def fitpolynomial(x, b, n):
    Ps = np.zeros(len(x))
    for j in range(len(x)):
        p = 0
        for i in range(n + 1):
            p += b[i] * x[j] ** i
        Ps[j] = p
    return Ps
```

Lastly, I wrote a function that would calculate the sum of square errors SSE, and give me the overall error of the fit,

```
def SSE(P, y):
    dif = np.subtract(P, y)
    squared = np.zeros(len(dif))
    for i in range(len(dif)):
        squared[i] = dif[i] ** 2
    error = np.sum(squared)
    return error
```

First, I generated data points from a function that is in the orders of x^3. Second, I took random 80% of the points and constructed the model of the best fit for the data for different orders. Third, I took the other 20% of points, and I plugged them into the model to approximate them. Then, I compared the approximation to the actual values generated. Lastly, I compared the error of the training data

and the error of the testing data over different orders. Translating all of this into code,

```
x1 = np.linspace(-20, 20, 20)
x2 = np.random.choice(x1, int(0.8*len(x1)), replace = False)
y = np.random.uniform(0, 10, int(0.8*len(x1))) * x2 ** 3 + np.random.uniform(0, 10, int(0.8*
x_test = np.setdiff1d(x1, x2)
y_test = np.random.uniform(0, 10, int(0.2*len(x1))) * x_test ** 3 + np.random.uniform(0, 10,

k = 5 # order

order = np.arange(1, k+1)
error1 = np.zeros(k)
error2 = np.zeros(k)

for n in range(1, k + 1):
    V = Vandermond(x2, y, n)
    b = LeastSquare(V, y)
    Ps = fitpolynomial(x2, b, n)
    for i in range(len(x2) - 1):
        xs = np.linspace(x2[i], x2[i+1], 50)
        p = fitpolynomial(xs, b, n)
        plt.plot(xs,p)

    error = SSE(Ps, y)
    error1[n - 1] = error

    Ps2 = fitpolynomial(x_test, b, n)
    plt.scatter(x2, y)
    plt.show()

    error = SSE(Ps2, y_test)
    error2[n - 1] = error

plt.plot(order, error1, label = "Train")
plt.plot(order, error2, label = "Test")
plt.xticks(np.arange(1, k+1))
plt.legend()
plt.show()
```

These are the results of the code,

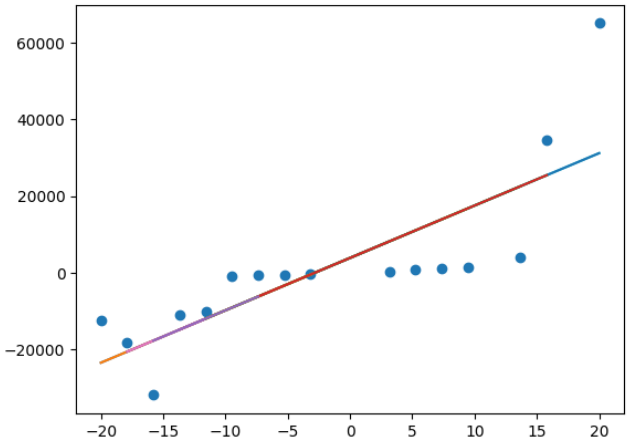

Figure 6: First Order Least Square Approximation

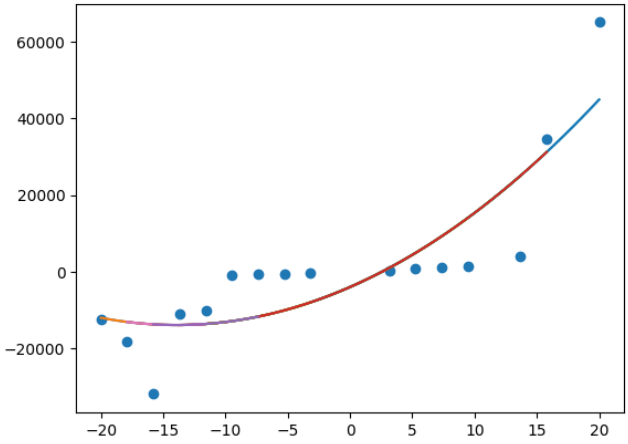

Figure 7: Second Order Least Square Approximation

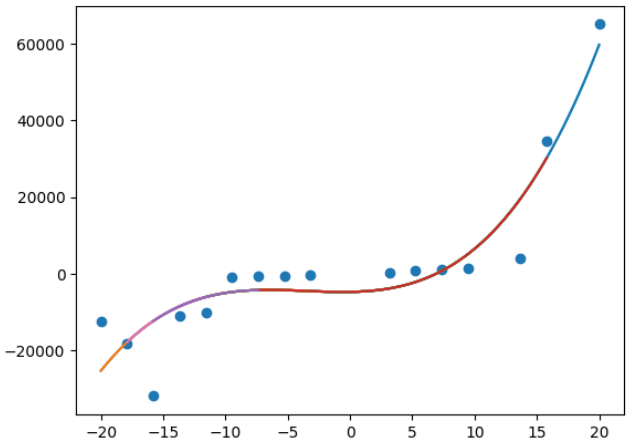

Figure 8: Third Order Least Square Approximation

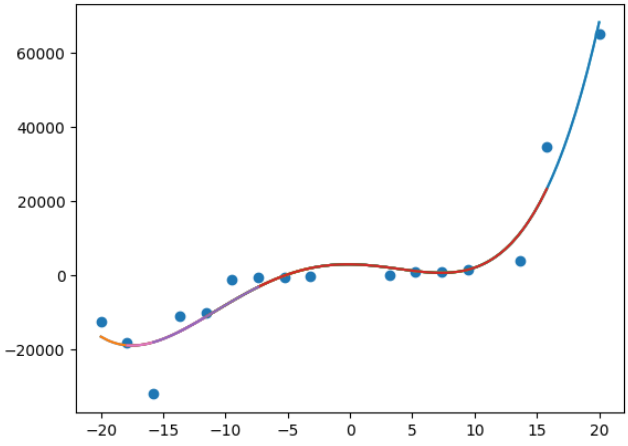

Figure 9: Fourth Order Least Square Approximation

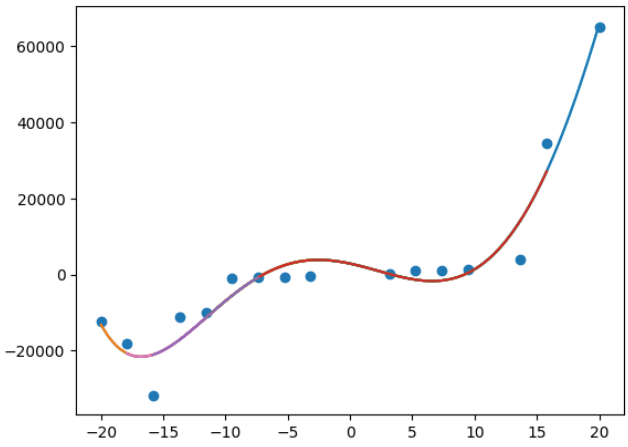

Figure 10: Fifth Order Least Square Approximation

The results above show us the different fits of the same data over different orders. As it is observed there are fit that under fit the data like figure 6 and figure 7 and fits that over fit the data like figure 9 and figure 10. This is apparent in figure 11, where we can see that where the two curves intersect is the optimal order to fit these data, 3rd order.

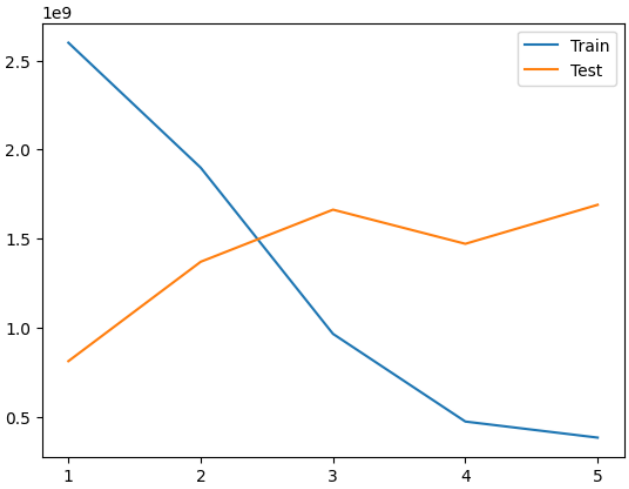

Figure 11: The Errors of the Training Data and Testing Data as Functions of Order