# Computational Physics: Interpolation

Richard Anderson

November 2024

## 1 Lagrange Interpolation

In order to derive the formula for Lagrange Interpolation, we must consider:
Given $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$ , such that all $x_i$ are distinct,

$$\exists p(x_i) \in \mathbb{R}[x] \mid p(x_i) = y_i \qquad \forall i \in \{0, 1, \ldots, n\}$$

$$p(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} + a_n x^n$$

For this polynomial, we can consider an expansion in terms of some basis polynomials $L_i(x)$ :

$$p_n(x) = \sum_{i=0}^{n} c_i L_i(x)$$

So, $\forall x_i \in \{x_0, x_1, \ldots, x_n\}$ we have the condition,

$$p_n(x_i) = y_i$$

Based on this condition, we can write:

$$p_n(x) = \sum_{i=0}^{n} y_i L_i(x)$$

We can write some basis polynomial as a product of its roots :

$$L_i(x) = C_i \prod_{\substack{0 \le j \le n \\ j \ne i}} (x - x_j)$$

And, from the condition $p_n(x_i) = y_i$ , we can deduce that $L_i(x_i) = 1$

$$L_i(x_i) = C_i \prod_{\substack{0 \le j \le n \\ j \ne i}} (x_i - x_j) = 1$$

$$C_i = \frac{1}{\prod_{\substack{0 \leq j \leq n \\ j \neq i}} (x_i - x_j)}$$

Therefore, we can write $L_i(x)$ as:

$$L_i(x) = \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

Therefore, the interpolating polynomial can be written as:

$$p_n(x) = \sum_{i=0}^{n} y_i \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

```
function LagrangeInterpolationPolynomial($x$, $y$)
    $n \leftarrow$ length($x$)
    $p(x) \leftarrow 0$
    for $i \leftarrow 0$ to $n - 1$ do
        $L_i(x) \leftarrow 1$
        for $j \leftarrow 0$ to $n - 1$ do
            if $j \neq i$ then
                $L_i(x) \leftarrow L_i(x) \times \frac{x - x[j]}{x[i] - x[j]}$
            end if
        end for
        $p(x) \leftarrow p(x) + y[i] \times L_i(x)$
    end for
    return $p(x)$
end function
```

In this algorithm, we perform $n$ operations for each loop, therefore each loop has time complexity $O(n)$, and since there are two nested loops, we have a total time complexity of $O(n^2)$ for the Lagrange Interpolation Algorithm.

Now, here is an example implementation of this algorithm in Python:

```python
def lagrange_interpolation(x, y, z):
    n = len(x)
    p = 0.0
    for i in range(n):
        L = 1.0
        for j in range(n):
            if j != i:
                L *= (z - x[j]) / (x[i] - x[j])
        p += y[i] * L
    return p
```

In our case, the points we are considering are :

| $x$ | $y$ |
|---|---|
| $-3.5$ | $-0.028$ |
| $1.4$ | $0.689$ |
| $2.7$ | $-13.452$ |
| $3.3$ | $-26.773$ |
| $5.3$ | $111.062$ |

Table 1: Data Points

Putting this into the program yields:

$$p_4(x) = 1.26469327953347x^4 - 5.54729187108612x^3 - 12.2727899772144x^2 + 63.7584670528321x - 54.1548623270202$$

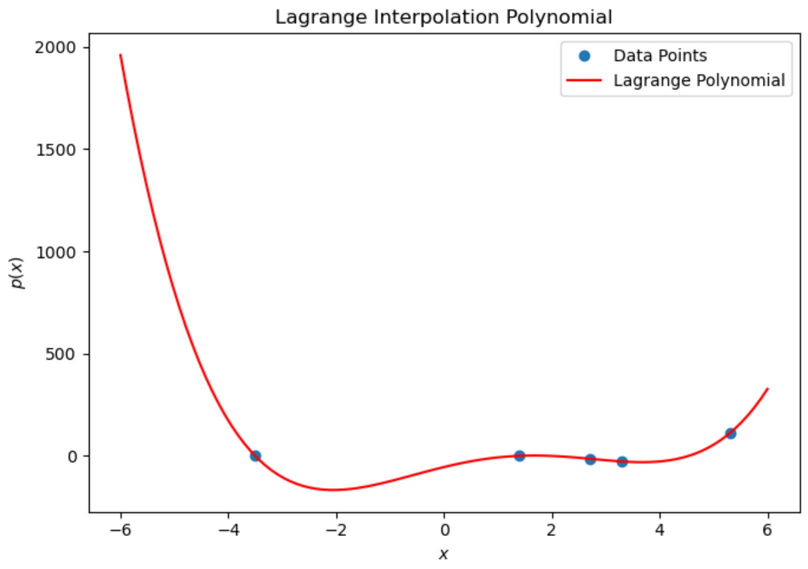

Figure 1: Lagrange Interpolation Output

## 2 Newton Interpolation

Now, we will take a different approach to interpolation. We can start from :

$$p_0(x_0) = y_0$$

And of course, this goes for all $x_i$, so now we must construct a polynomial which goes through $y_0$ and $y_1$, this can be given by a linear equation:

$$p_1(x) = x\left(\frac{y_1 - y_0}{x_1 - x_0}\right) + a$$

Which is just point slope form of a line going through $(x_0, y_0)$ and $(x_1, y_1)$. Using the condition that $p_1(x_0) = y_0$ :

$$x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right) + a = y_0 \implies a = y_0 - x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$$

$$p_1(x) = x\left(\frac{y_1 - y_0}{x_1 - x_0}\right) + y_0 - x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right) = y_0 + (x - x_0)\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$$

We can see that not only does $p_1(x_0) = y_0$, but $p_1(x_1) = y_1$ as well. Now, to fit through $x_2$ and $y_2$, we will need a quadratic equation. First, we can note that:

$$p_1(x) = p_0(x) + (x - x_0)\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$$

This implies that we can define the polynomials recursively, so we will define a quadratic equation :

$$p_2(x) = p_1(x) + a(x - x_0)(x - x_1)$$

Now, asserting that $p_2(x_2) = y_2$ :

$$y_0 + (x_2 - x_0)\left(\frac{y_1 - y_0}{x_1 - x_0}\right) + a(x_2 - x_0)(x_2 - x_1) = y_2$$

$$a = \frac{y_2 - y_0 - (x_2 - x_0)\left(\frac{y_1 - y_0}{x_1 - x_0}\right)}{(x_2 - x_0)(x_2 - x_1)}$$

$$a = \frac{y_2 - y_1 + y_1 - y_0 - (x_2 - x_0)\left(\frac{y_1 - y_0}{x_1 - x_0}\right)}{(x_2 - x_0)(x_2 - x_1)}$$

$$a = \frac{y_2 - y_1 + (y_1 - y_0)\left[1 - \frac{x_2 - x_0}{x_1 - x_0}\right]}{(x_2 - x_0)(x_2 - x_1)}$$

$$a = \frac{y_2 - y_1 + (y_1 - y_0)\left[\frac{x_1 - x_2}{x_1 - x_0}\right]}{(x_2 - x_0)(x_2 - x_1)}$$

$$a = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}$$

This can be referred to as the *divided difference* :

$$a = f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

this would mean that we can neatly write $p_2(x)$ as:

$$p_2(x) = y_0 + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1)$$

Indeed, for $p_n(x)$, that is, a dataset with $n + 1$ points, we can write the interpolating polynomial as:

$$p_n(x) = y_0 + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \\ \ldots + f[x_0, x_1, \ldots, x_n](x - x_0)(x - x_1)(x - x_2)\ldots(x - x_{n-1})$$

In a more concise format:

$$p_n(x) = \sum_{i=0}^{n} f[x_0, x_1, \ldots, x_i] \prod_{j=0}^{i-1}(x - x_j)$$

So, the procedure for implementing this numerically is:

```
function DividedDifference($x$, $y$)
    $n \leftarrow$ length($x$)
                        ▷ Initialize a 2D array $F$ of size $n \times n$ with $F[i][0] = y[i]$ for
$i \in \{0, \dots, n-1\}$
    for $i = 0$ to $n - 1$ do
        $F[i][0] \leftarrow y[i]$
    end for
    for $j = 1$ to $n - 1$ do
        for $i = 0$ to $n - j - 1$ do
            $F[i][j] \leftarrow \frac{F[i+1][j-1] - F[i][j-1]}{x[i+j] - x[i]}$
        end for
    end for
    return $F[0]$                    ▷ Coefficients of the Newton polynomial
end function
function NewtonPolynomial($x$, $y$, $z$)
    $a \leftarrow$ DividedDifference($x$, $y$)
    $n \leftarrow$ length($a$)
    $p \leftarrow a[0]$              ▷ Initialize the polynomial with the first coefficient
    for $i = 1$ to $n - 1$ do
        $term \leftarrow a[i]$
        for $j = 0$ to $i - 1$ do
            $term \leftarrow term \times (z - x[j])$
        end for
        $p \leftarrow p + term$
    end for
    return $p$
end function
```

Each DividedDifference call is of $O(n^2)$ time complexity, but DividedDifference is called only once each time NewtonPolynomial is, and therefore overall NewtonPolynomial is of $O(n^2)$ time complexity as well.

Now, we will apply the algorithm using the following points:

| $x$ | $y$ |
|---|---|
| $-3.5$ | $-0.028$ |
| $1.4$ | $0.689$ |
| $2.7$ | $-13.452$ |
| $3.3$ | $-26.773$ |
| $5.3$ | $111.062$ |

Table 2: Data Points

Here is an implementation of the Newton Interpolation Algorithm in Python:

```python
def divided_difference(x, y):
    n = len(x)
    F = np.zeros((n, n))
    for i in range(n):
        F[i][0] = y[i]
    for j in range(1, n):
        for i in range(n - j):
            F[i][j] = (F[i + 1][j - 1] - F[i][j -
                1]) / (x[i + j] - x[i])
    return F[0]

def newton_poly(x, y, z):
    a = divided_difference(x, y)
    n = len(a)
    p = a[0]
    for i in range(1, n):
        term = a[i]
        for j in range(i):
            term *= (z - x[j])
        p += term
    return p
```

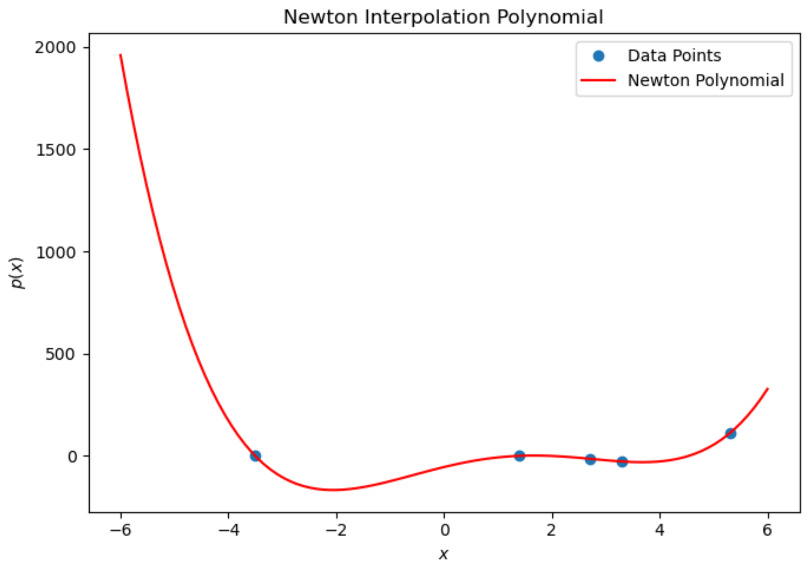

Figure 2: Newton Interpolation Output

The Python script produces the following polynomial, as well as the plot shown:

$$p_4(x) = 1.26469327953347x^4 - 5.54729187108612x^3 - 12.2727899772144x^2 + 63.758467052832x - 54.1548623270202$$

## 3 Comparison and Discussion

This polynomial differs only by $\sim 9.948 \times 10^{-13}$ in the first order term in $x$, meaning that the two algorithms are nearly identical in efficiency and output. The primary distinguishing factor between the two is the ability to include new points. That is, were we to add a data point, in the case of Lagrange Interpolation, we would have to recalculate every $L_i(x)$, whereas we would only need to calculate one extra DividedDifference, since they are calculated recursively. This is a prominent and primary advantage of Newton interpolation over Lagrange interpolation, though one other factor to note is the numerical instability which arises due to floating point precision when evaluating products with large values. It is for these reasons that Newton Interpolation is the preferred method in real world applications of interpolation.