University of Wisconsin-Milwaukee
Department of Mathematics

NUMERICAL ANALYSIS

Comparing different Methods of Interpolation

Jan Kretschmann

1 Abstract

In this work, different methods of interpolation are examined regarding their stability, accuracy, implementation and their behavior on specific problem settings. For this examination, a small dataset as well as a number of pre-defined functions were used. To compare the error, cross validation was applied to the small dataset, whereas the approximated results were directly compared to the actual results of the pre-defined functions.

It is found that the interpolation methods that yield algebraic approximation polynomials (such as Lagrange Interpolation and the related methods Newton's Divided Differences and Neville's Method) are similarly accurate for small to medium sized data sets, but accumulate a huge error as the size of the dataset increases. This phenomena is called numerical instability.

In contrast, it can be seen that interpolation methods which yield non-algebraic approximation polynomials, such as the Natural Cubic Spline Interpolation, are numerically stable, and only increase in accuracy with increasing number of given data points.

2 Related Work

Comparing methods of interpolation is not a new area of interest, and numerous papers have been published that are set to compare different methods of interpolation in certain categories.

For instance, the work "Comparison of spline and Lagrangian interpolation" [McN86], published in the *Journal of Computational and Applied Mathematics* in 1986, compares the errors in Lagrange Interpolation polynomials to those in Cubic Spline Interpolation Polynomials. The author uses Neville's Method to compute Lagrange Polynomials, and compares those to Cubic Spline Polynomials on several functions. He generates a dataset by computing function values in different step sizes starting from an initial point. It is found in this paper, that with large step-sizes, Lagrange Polynomials achieve an approximation that is much closer to the result than Cubic Spline Polynomials, but what is especially interesting is that it is also stated, that with small step sizes, both methods are "about equally good".
This is an interesting result, because in this work, it will be found that for increasing number of data points, Lagrange Polynomials will accumulate a huge error due to their instability. However, the reason for these conflicting results lies in the different application of Lagrange polynomials: even though the author uses step sizes as low as 0.01, only 3 data points are used for every evaluation, which is not enough to see the numerical instability of the Lagrange method.

[McN86] is examining the interpolation methods from a different viewpoint, especially compared to this work. It is to be remembered, that in the time the paper was published (the 1980s), computational power was much more expensive and way more restricted than nowadays. An example of a more recent work was published by Berrut in 2004: "Barycentric lagrange interpolation" [BT04], which introduces a stable version of the Lagrange Interpolation Method. In the comparisons to the so called *Barycentric* Lagrange Interpolation, the authors also find the error of the standard Lagrange approach to explode as the dataset increases in size, which is more similar to the focus of this work.

3 Description of Methods

3.1 Piecewise Linear Interpolation

Linear Interpolation describes the Method of taking two points from a dataset, and fitting a straight line through those points. This becomes increasingly inaccurate, the larger the distance between those two points is. Furthermore, if there are more than two points in the given dataset, it is impossible for a straight line to fit all the data points, unless they were taken from a straight line to begin with.
For these reasons, with larger datasets, the option that is mostly opted for is what is called *Piecewise Linear Interpolation*. With Piecewise Linear Interpolation, one straight line is fitted between every two consecutive points of the dataset. This results in a continuous curve, with a non-continuous derivative.

For example, Figure 1 depicts the piecewise linear function that interpolates the points

$$X = \{(0,0), (3,3), (5,2), (6,0)\}$$

The function is defined as:

$$P(x) = \begin{cases} x & 0 \leq x \leq 3 \\ -0.5x + 4.5 & 3 \leq x \leq 5 \\ -2x + 12 & 5 \leq x \leq 6 \end{cases}$$

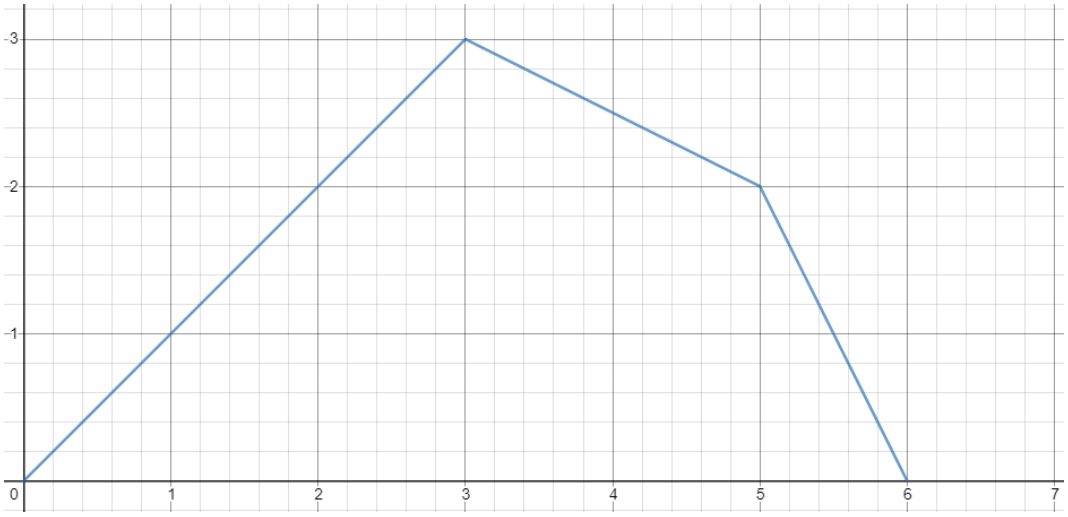

Figure 1: Example of a Piecewise Linear Function.

3.2 Lagrange Interpolation

A standard Lagrange interpolating polynomial passes through every point in a given dataset while at the same time remaining algebraic: if there are $n+1$ data points, the polynomial will be of degree n [BF85].
Lagrange polynomials are constructed as follows:

$$P(x) = \sum_{k=0}^{n} f(x_k) L_{n,k}(x)$$

with $Ln, k(x) = \prod_{i=0, i\neq k}^{n} \frac{x-x_i}{x_k-x_i}$. It can be seen, that the Lagrange polynomial is constructed to fit every point in the dataset: for every $i \neq k$, $L_{n,k}(x_i)$ will be equal to 0, and $L_{n,k}(x_k) = 1$, so that $P(x_k) = f(x_k)$ holds for every k [BF85].
In Figure 2, an example of a Lagrange interpolating polynomial can be seen. It was generated through the points

$$X = (-1, 1), (0, 0), (1, 1)$$

so the resulting polynomial was computed to be

$$P(x) = \frac{(x-1)(x-0)}{(-1-1)(-1-0)} + \frac{(x+1)(x-0)}{(1+1)(1-0)} = x^2$$

which fits all the points in X.

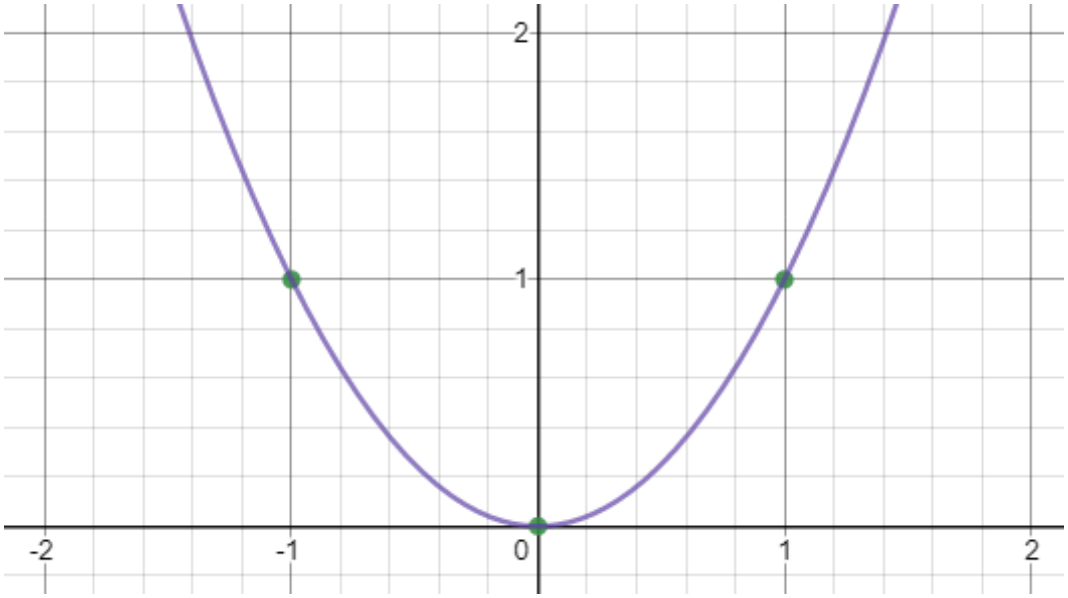

Figure 2: Example of a Lagrange interpolating polynomial.

This form of Lagrange Interpolation brings two problems. Firstly, due to the Lagrange polynomial being algebraic, it loses numerical stability. This means, it becomes increasingly imprecise and accumulates a huge error, when n gets large. Secondly, Lagrange polynomials have an error term of $\frac{f^{(n+1)(\xi(x))}}{(n+1)!}\prod_{i=0}^{n}(x - x_i)$, which is generally not known when the function f is not known, like in the example given above. Therefore, it makes sense to assume that the most precise approximation is the one that was made using every value of the dataset, even though sometimes that might be false (compare Illustration, Ch. 3.2 [BF85]). Generating more than one Lagrange interpolation polynomial is not very efficient, because it contains the same amount of work every time.

3.2.1 Neville's Method

Neville's Method tries to address the problem of generating more than one Lagrange polynomial efficiently, by constructing them in an iterative way [BF85]. It gives an algorithm, that generates a higher order Lagrange polynomial using already generated, lower order Lagrange Polynomials.
The algorithm for Neville's method, implemented in Python, can be seen in Figure 3[1].

[1]It can also be found in the separately uploaded Jupyter Notebook, in case the font is to small.

```python
def neville_interpolation(data, x):
    Q = np.empty((len(data), len(data)))
    Q[:, 0] = data[:, 1]
    for i in range(1, len(data)):
        for j in range(1, i+1):
            Q[i, j] = ((x - data[i-j, 0])*Q[i, j-1] - (x - data[i, 0])*Q[i-1, j-1])/(data[i, 0] - data[i-j, 0])
    return Q[-1, -1]
```

Figure 3: Python Implementation of Neville's Algorithm.

3.2.2 Newton's Divided Differences

Using Newton's Divided Differences is another way to address the issues the standard Lagrange Interpolation brings: instead of iteratively generating polynomials of higher order, Newton's Method successively generates polynomials of degree $< n$, that sum up to the Lagrange polynomial or order n.
The evaluation of an interpolation polynomial constructed with Newton's Divided Differences can be computed reasonably fast, using an algorithm called *Horner's Scheme*.
The implementation of this algorithm can be found in Figure 4[2].

```python
def newton_divided_differences(data, x):
    F = divided_differences(data)
    res = F[0]
    for i in range(1, len(F)):
        mult = reduce(lambda a, b: a*b, [x - data[j, 0] for j in range(i)])
        res += F[i]*mult
    return res

def divided_differences(data):
    coefficients = np.empty((len(data), len(data)))
    coefficients[:, 0] = data[:, 1]
    for i in range(1, len(data)):
        for j in range(1, i+1):
            coefficients[i, j] = (coefficients[i, j-1] - coefficients[i-1, j-1])/(data[i, 0] - data[i-j, 0])
    return np.diag(coefficients)
```

Figure 4: Python Implementation of Newton's Divided Differences Algorithm.

3.2.3 First Comparison

The first three interpolation methods described all result in the same Lagrange polynomial, they merely describe different algorithms to compute it.
The effect of computing the Lagrange polynomial using Newton's or Neville's Algorithms can already be seen in small examples: trying to interpolate the function $f(x) = x^4$ on the interval $[-10, 10]$ with 200 equally spaced data points, the Lagrange polynomial can only be computed[3] using Neville's or Newton's Algorithm,

[2] It can also be found in the mentioned Jupyter Notebook.
[3] Machine: Windows PC, 16GB DDR3 RAM, Intel i7 3770k CPU, Python running on a single Thread. Implementation of Lagrange method can be found in Jupyter Notebook.

see Figure 5.
However, when trying to interpolate the same dataset using the standard Lagrange method, the machine prints out an error, and the resulting polynomial can be seen in Figure 6[4]. The error message can be seen in Figure 7.

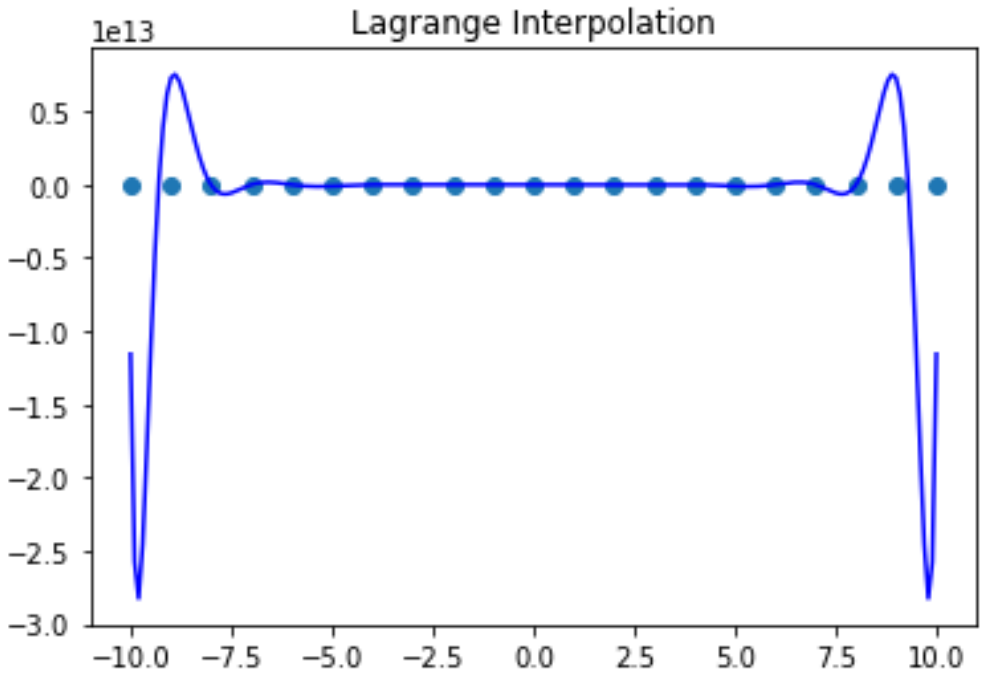

Figure 6: Interpolating $f(x) = x^4$ using the standard Lagrange method.

```
H:\python\lib\site-packages\ipykernel_launcher.py:9: RuntimeWarning: overflow encountered in long_scalars
  if __name__ == '__main__':
```

Figure 7: Error message when trying to interpolate $f(x) = x^4$ using the standard Lagrange method.

3.3 Natural Cubic Spline Interpolation

In contrast to the previously described methods of interpolation, the Cubic Spline approach results in a piecewise-defined function, and not in a single polynomial [BF85]. In theory, the advantage of this approach lies in its stability for large

[4]Attention: the data points look like they are on a straight line here rather than on $f(x) = x^4$, this is due to the scaling of the Y-Axis.

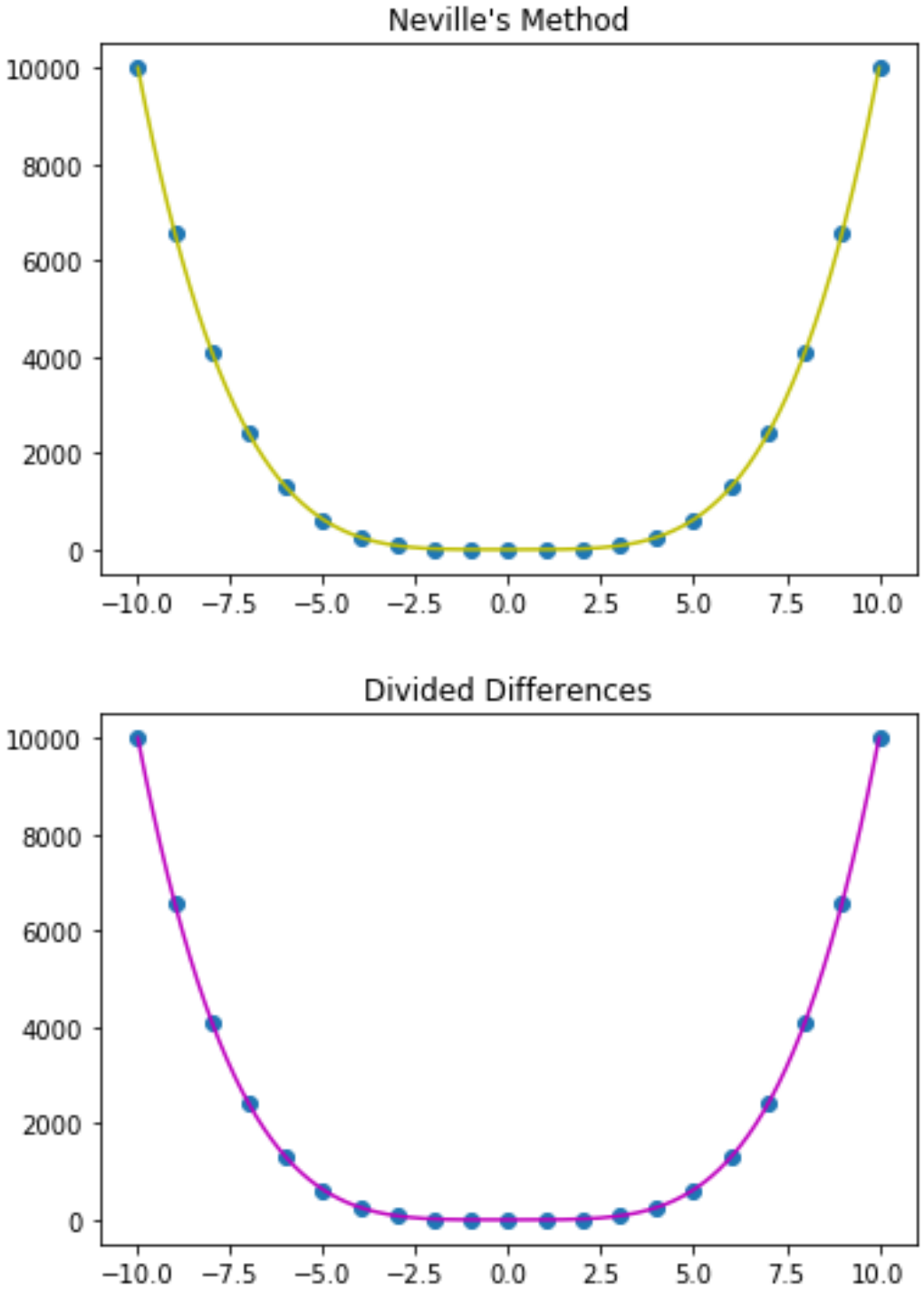

Figure 5: Interpolating $f(x) = x^4$ using Neville's Method and Newton's Divided Differences.

datasets: trying to compute a Lagrange polynomial for a large number of data points results in a high degree polynomial. Those, however, tend to "oscillate erratically" [BF85]. The Cubic Spline approach only contains functions of degree 3, and results in one cubic function between every interval of two consecutive points. This eliminates the instability that causes oscillation and results in higher accuracy, the more data points are given.
Cubic Spline polynomials are constructed as

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

with parameters, a_j, b_j, c_j, d_j for $j = 0, ..., n-1$. A cubic spline satisfies the boundary conditions:

1. $S_j(x)$ is cubic and defined on $[x_j, x_{j+1}]$
2. $S_j(x_j) = f(x_j)$ and $S_j(x_{j+1}) = f(x_{j+1})$
3. $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$
4. $S''_{j+1} = S''_j(x_{j+1})$

Natural cubic splines also satisfy the boundary condition $S''(x_0) = S''(x_n) = 0$ [BF85].
The construction of the natural cubic spline shows that $S_j(x_j) = a_j = f(x_j)$, and the remaining parameters can be solved by the equation system $Ax = b$, with

$$A = \begin{pmatrix} 1 & 0 & 0 & ... & & 0 \\ h_0 & 2(h_0 + h_1) & h_1 & ... & & 0 \\ 0 & h_1 & 2(h_1 + h_2) & h_2 & ... & 0 \\ ... & ... & ... & ... & ... & 0 \\ & ... & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & ... & & & & 1 \end{pmatrix}$$

with $h_i = x_{i+1} - x_i$,

$$x = \begin{pmatrix} c_0 \\ ... \\ c_n \end{pmatrix}$$

and $d_j = \frac{c_{j+1} - c_j}{3h_j}$, using the given boundary conditions [BF85].
It can be seen above, that the construction of a cubic spline is more complex than

that of a Lagrange polynomial, especially using Neville's or Newton's Method. An implementation to find a Natural Cubic Spline polynomial according to Algorithm 3.4 in [BF85] can be found in the uploaded Jupyter Notebook.

4 Comparison of Interpolation Methods and Main Result

4.1 Comparison on dataset

Each method was implemented in Python according to the algorithms given in [BF85]. Then, to compare the different approaches, they were used to interpolate a dataset from p. 141 of [BF85], which can be found in Table 1.

x	0	3	5	8	13
y	75	77	80	74	72

Table 1: dataset used for the comparison of interpolation methods.

The graphs of the different interpolation polynomials can be seen in Figures

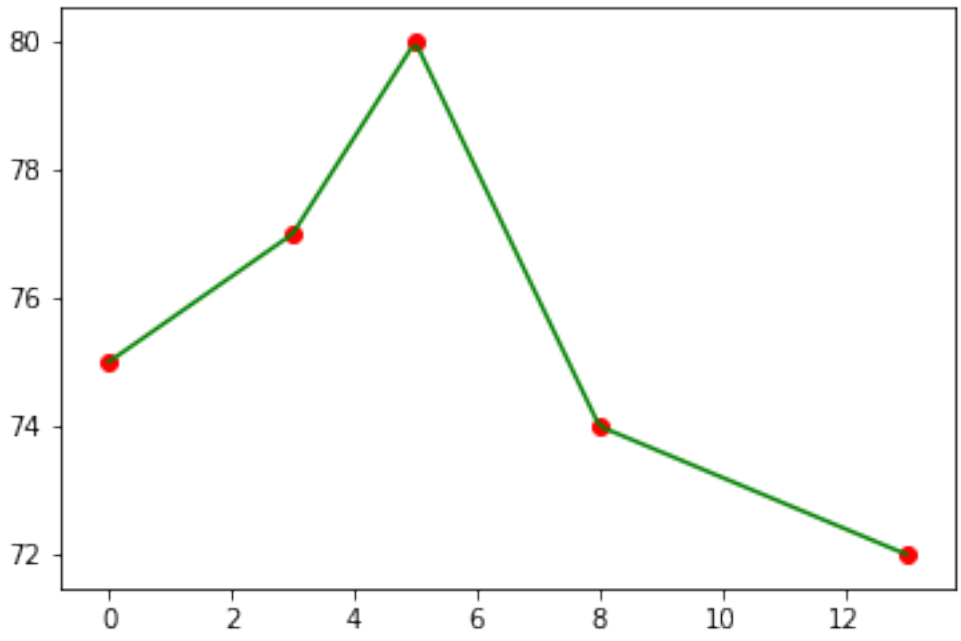

Figure 8: Piecewise Linear Interpolation of the given dataset.

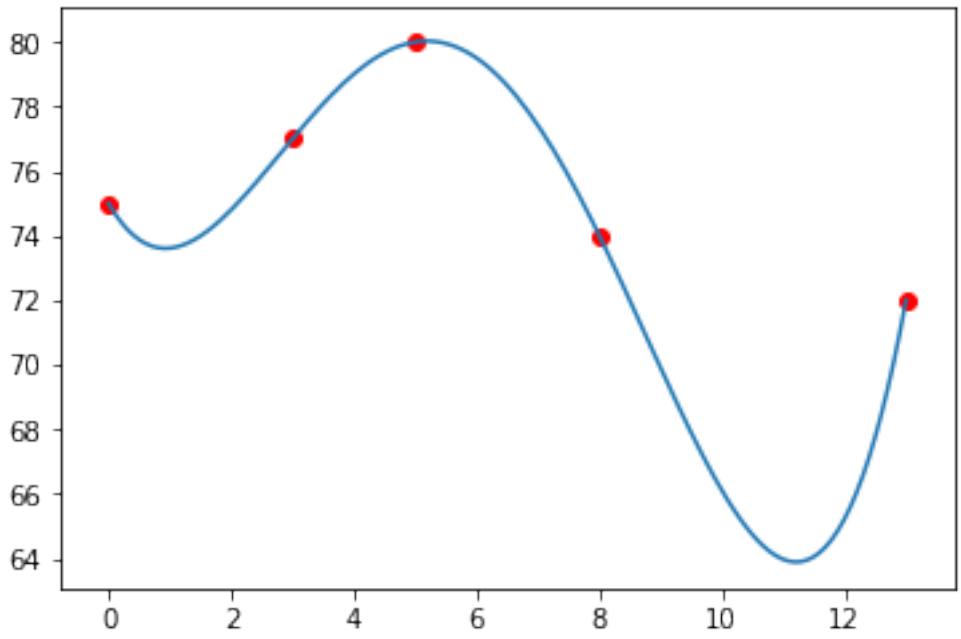

Figure 9: Lagrange Interpolation Polynomial of the given dataset.

As expected, the graphs produced by Neville's Method, Newton's Divided Differences and the standard Lagrange polynomial produce the same output, so only one is shown in Figure 9.

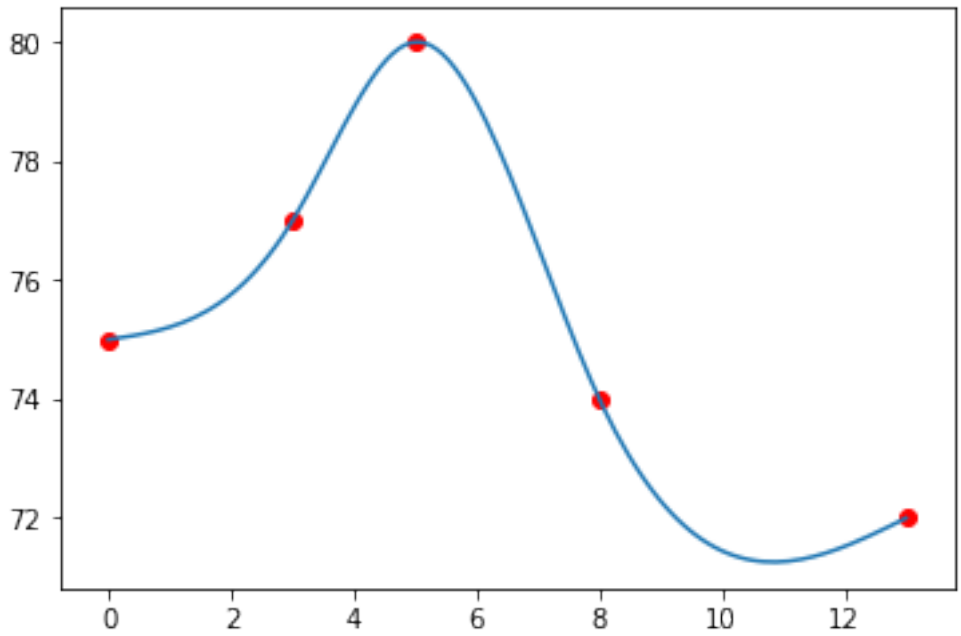

Figure 10: Cubic Spline Interpolation Polynomial of the given dataset.

When interpolating this small dataset, the oscillation that occurs in algebraic polynomials with high degrees can already be seen in the Lagrange polynomial. On the interval $[10, 12]$, the polynomial reaches y values as low as 64, which is not indicated to happen anywhere in the dataset. The Cubic Spline polynomial, however, has a minimum that is barely below $y = 72$. The difference is clearly visible in Figure 11, where the Lagrange Polynomial is shown in black and the Cubic Spline in blue.

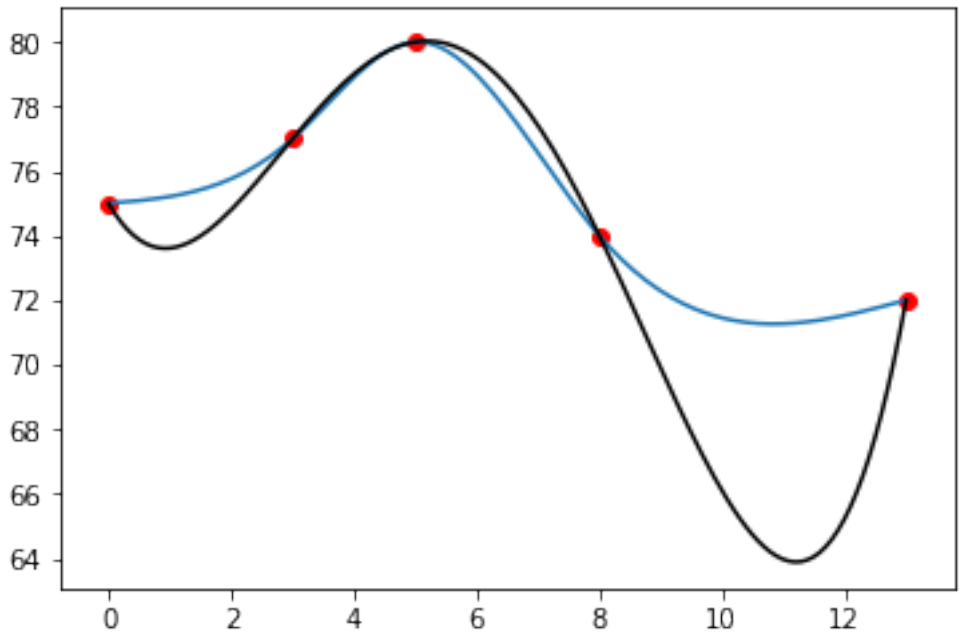

Figure 11: Comparison of Cubic Spline and Lagrange Polynomial.

To evaluate the accuracy of the different interpolation techniques on this dataset, where the exact function f is unknown, cross validation was used. This is a common approach in machine learning, and it means the dataset was partitioned in "training"(in this case interpolating) data and testing data. The algorithm is then evaluated on the training data, and knowing the result of the testing data, it can then be used to measure the accuracy of the interpolation polynomial.

In this particular case, let the dataset be $D = \{p_1, p_2, p_3, p_4, p_5\}$, with $p_i = (x_i, y_i)$. In each step, one p_j was taken from the dataset to be the testing point. The interpolation methods were used with $D^* = \{p_i | i \neq j\}$, to compute an interpolation polynomial $P(x)$. At the end of each step, the difference $|P(x_j) - y_j|$ was accounted for as error.
Using this method, the errors for the examined methods can be found in Table 2.
Obviously, all the Lagrange polynomials generate the same absolute error. As expected, the Natural Cubic Spline has less absolute error than the Lagrange polynomials. Surprisingly, the lowest error in this case was accumulated by the piecewise linear interpolation, which seems to be the most accurate for the given dataset.

Method	PW-Linear	Lagrange	Neville	Newton	Spline
Error	8.2	17.39	17.39	17.39	12.98

Table 2: Comparison of errors of the different Interpolation Methods.

4.2 Comparison on Predefined Function

To get a more accurate comparison of the accuracy of the interpolating methods, they were also used to interpolate the functions

- $f_1(x) = e^x$
- $f_2(x) = \sin(x)$
- $f_3(x) = x$
- $f_4(x) = x^4$

For this test, the respective functions were first evaluated on 40 equally spaced points on the interval $[0, 6]$, in order to get a dataset for the interpolation. All the methods were used to generate interpolation polynomials P_j from this dataset. Then, to measure the accuracy, the points $\{(0.5 + i, f_j(0.5 + i))|i = 0, ..., 5, j = 1, ..., 4\}$ were computed. The absolute error for the function used in this work is summed up as follows:

$$e_j = \sum_{i=0}^{5} |f_j(0.5 + i) - P_j(0.5 + i)|$$

The resulting absolute errors can be found in Table 3. It can be seen, that the Lagrange polynomials seem to be about as accurate as the spline polynomials under these circumstances, as also found in [McN86], see Section 2.
The differences between the different ways to compute Lagrange polynomials can be attributed to machine imprecision: *NumPy* was used in programming the methods, and as seen in the Jupyter Notebook at the very beginning, the default datatype in *NumPy* has a precision of roughly 1.19×10^{-7}.

	PW-Linear	Lagrange	Neville	Newton	Spline
e^x	0.25024	1.335×10^{-6}	3.853×10^{-8}	2.357×10^{-7}	0.00018
$\sin(x)$	0.00357	2.818×10^{-8}	4.021×10^{-9}	7.414×10^{-9}	7.008×10^{-7}
x	0	9.853×10^{-8}	3.409×10^{-8}	0	0
x^4	0.68810	5.014×10^{-6}	1.910×10^{-7}	3.640×10^{-7}	0.00019

Table 3: Main Results regarding Accuracy: Accuracy of Interpolation Methods on predefined Functions.

4.3 Main Result regarding Stability

One thing, that was mentioned in 3.3, is that the Cubic Spline Interpolation is stable, while Lagrange polynomials, due to them being algebraic, generally are not.
And this can be seen when comparing the development of the error of the Lagrange polynomial to that of a Cubic Spline polynomial, as the number of data points increases: working still on the interval $[0, 6]$, the amount of error was examined for everything between 5 and 500 data points.
First we can see, the Piecewise Linear Interpolation polynomial, which gets more and more accurate the larger the dataset gets (see Figure 13), and is almost equal to 0 once the polynomial is evaluated at all 200 nodes.

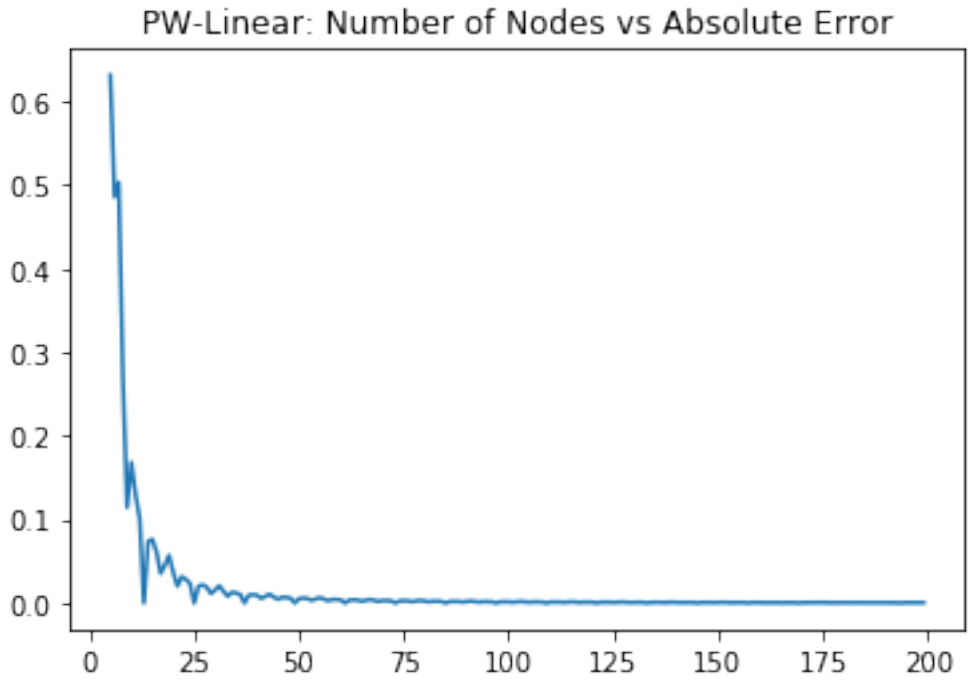

Figure 12: Piecewise Linear Interpolation Polynomial: Error vs Number of Data Points.

We see very similar behavior for the Cubic Spline Polynomial, compare Figure 13, it approaches 0 the larger the dataset gets.

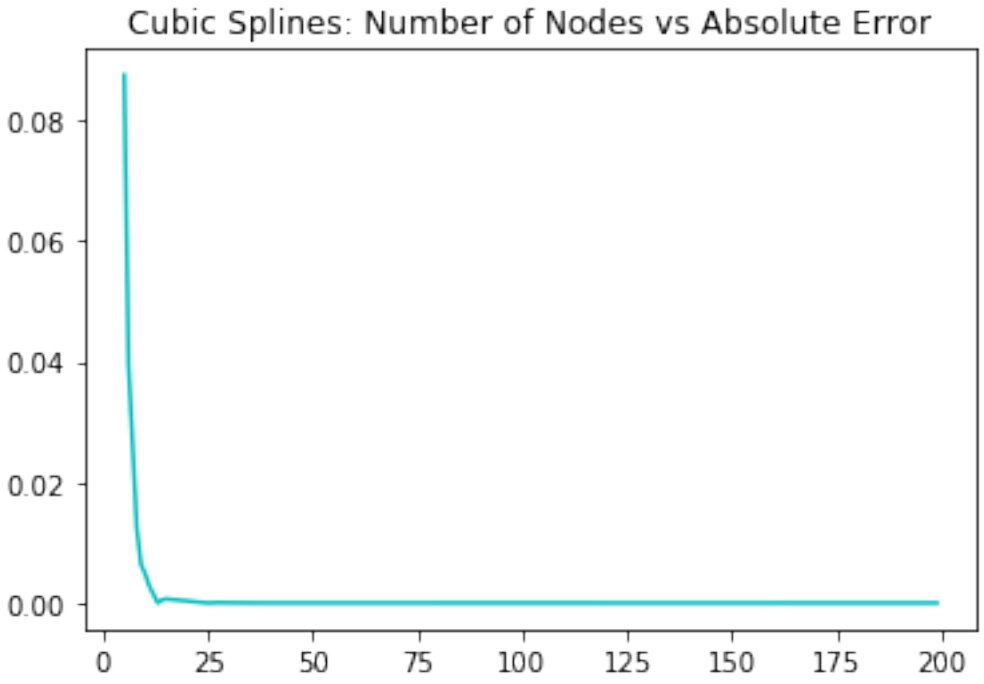

Figure 13: Cubic Spline Polynomial: Error vs Number of Data Points.

However, for the Lagrange polynomials, it is a different story: their numerical instability is clearly visible as the number of data points gets bigger: every method of computing Lagrange polynomials has an exploding error when the dataset gets too large[5]. This is explained by the oscillating of polynomials of high degree, as mentioned earlier and in [BF85]. The error figures for all Lagrange polynomials can be seen in Figure 14. Again, slight differences between the different methods are accounted for by machine imprecision.

[5]Notice, the Y-Axes are scaled in 10^{46}, 10^{44} and 10^{45}, so the error is not 0 at the beginning.

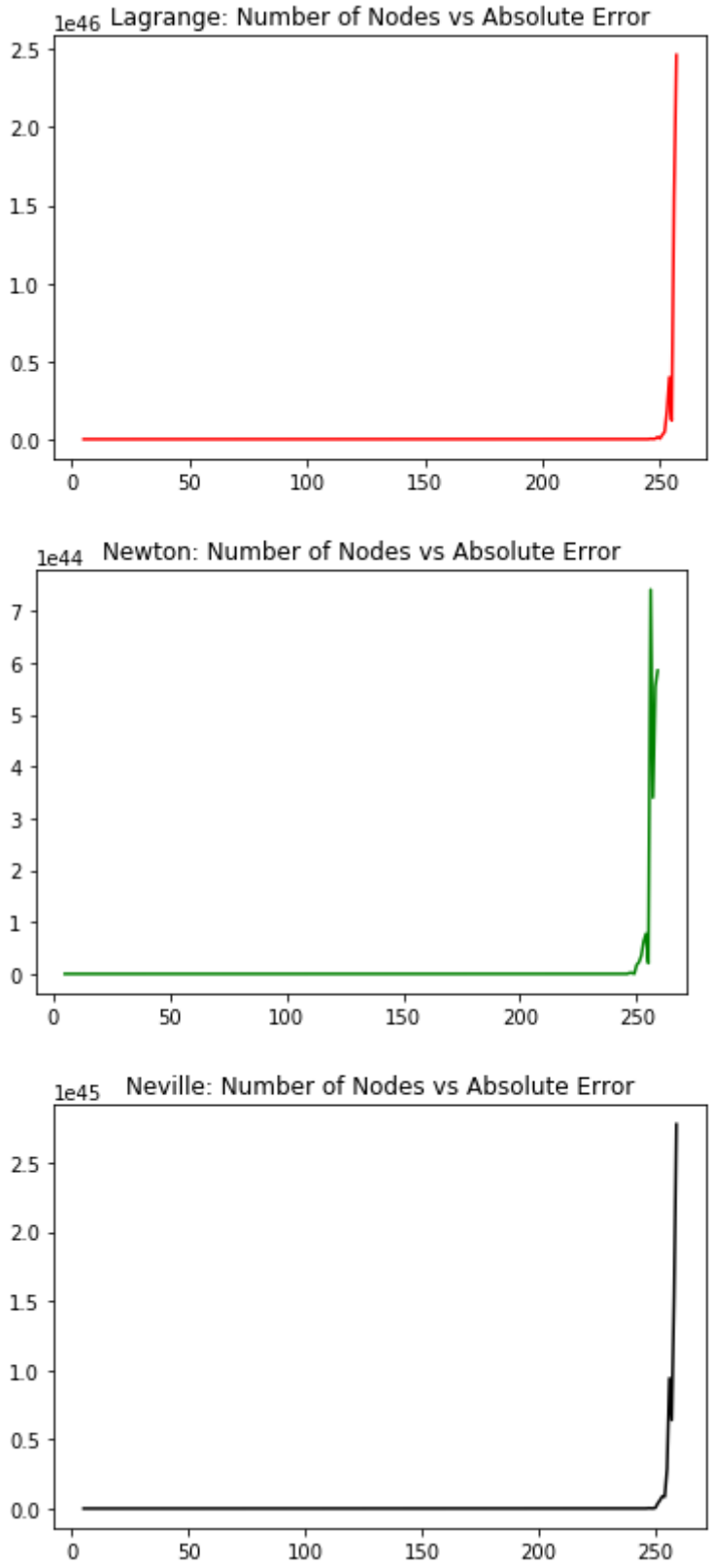

Figure 14: Piecewise Linear Interpolation Polynomial: Error vs Number of Data Points.

5 Conclusion

From the data gathered, there is no one method that is always the right one. It was seen, that Lagrange interpolation is similarly accurate to Natural Cubic Spline Interpolation on smaller datasets.
As the datasets get larger, however, the numerical instability of the Lagrange method is visible, it accumulates a huge error due to oscillation of high degree polynomials, while the spline and piecewise-linear interpolation polynomials got more accurate. This is a striking difference between those polynomials examined that are algebraic, and those that are not.
On the ground of what was observed, it is impossible to make a recommendation for what interpolation method to use on small to medium sized datasets. However, as the number of data points gets larger (and the points are closer together), the non-algebraic interpolation methods seem to be superior.

References

[BF85] Richard L Burden and Douglas J Faires. Numerical analysis. 1985.

[BT04] Jean-Paul Berrut and Lloyd N Trefethen. Barycentric lagrange interpolation. *SIAM review*, 46(3):501–517, 2004.

[McN86] John Michael McNamee. Comparison of spline and lagrangian interpolation. *Journal of computational and applied mathematics*, 16(2):237–240, 1986.