Exploring Runge's phenomenon

Supervisor: Thierry Meyrath

Aylin Cosgun
Jules Nies

Experimental Mathematics Lab

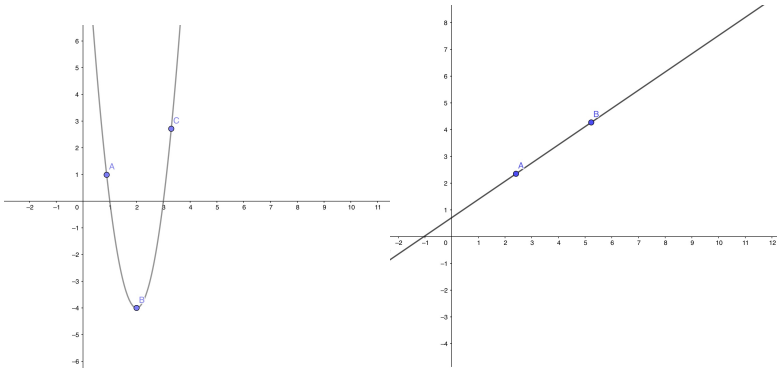
BMATH
Université du Luxembourg
May 31^{st}, 2022

Contents

1 Interpolation

The interpolation of function values or measured values is about expressing pairs of values by a formula, in other words to determine a curve that passes through given pairs of values. Specifically expressed we are looking for a continuous function $f(x)$ for $n+1$ points $(x_k; y_k)$ so that $y_k = f(x_k)$ for $k = 0, \ldots, n$, so that the given points lie on the graph of f. The interpolation allows us to get information in between for measurement points that are far apart, by connecting the given values by a line/curve through the interpolation. As we shall see later, if we have n+1 points $(x_k; y_k)$ with $k = 0, \ldots, n$ and x_k pairwise different then we can always determine a polynomial of degree at most n degree so that $p_n(x) = \sum_{k=0}^{n} c_k x^k$ is true, as we will see later, and we can find c_k so that $y_k = p_n(x_k)$ for all $k = 0, .., n$. To determine this polynomial, which is continuous and arbitrarily often differentiable, one must solve a system of equations.

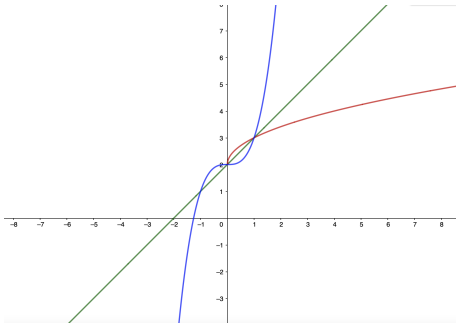
Example 1.1. 2 points lay on a straight line (polynomial of the first degree) and 3 points on a parabola (polynomial of the second degree). These are trivial cases.



To summarize we have the following definition of the interpolation problem:

Definition 1.1. Given n+1 pairs of values $(x_k; y_k)$ with k=0,...,n and x_k pairwise different, which are also called interpolation points. We look for continuous function f with the property: $f(x_k) = y_k \forall k = 0, ..., n$.

Example 1.2. We have the following points: $(0, 2)$ and $(1, 3)$. A possible function for the pairs of values is $f(x) = x + 2$. In this case $f(x)$ is the interpolating one. However, $f(x) = \sqrt{x} + 2$ and $f(x) = x^3 + 2$ are also interpolants of these pairs of values. So the conclusion we draw from this is that there is more than one interpolant for these pairs of values.



So the general interpolation problem is not uniquely solvable. Therefore, we will limit the class of possible functions f.

2 Approximation and Weierstrass Theorem

The approximation by an interpolation function is often the basis of an effective approximate problem solution for complicated and complex functions. First of all, we will start with the Weierstrass theorem.

Theorem 1. Let $f : [a; b] \to \mathbb{R}$ be a continuous function. For any $\epsilon > 0, \exists$p, p being a polynomial, such that $\max_{x \in [a;b]} |f(x) - p(x)| < \epsilon$.

In other words, if we have a function f that is continuous on a certain closed interval, we can find a polynomial p that approximates this function on the interval, and the difference between our approximation polynomial p and the function f is as small as ϵ.

3 Polynomial Interpolation

We now consider polynomial interpolation, that is the interpolation problem with the interpolating function. Being a polynomial, the interpolation polynomial is a polynomial which passes through exactly given points. So we want to construct a polynomial that passes through known points. You could also connect the points by so called linear splines, but polynomials are better suited because they have special properties. For example, you can differentiate them infinitely often, for polynomials up to degree 4 you can calculate exact zero-points and evaluate the function with the Horner Scheme. We have already discussed the case of 3 given interpolation points in the general interpolation, in the following we will interpolate for n+1 interpolation points (x_k, y_k) by polynomials of degree at most n. A polynomial of degree n has n+1 degrees of freedom and coefficients, we will define the coefficients by n+1 conditions. We will do this using a system of equations from the interpolation problem. So we get the following linear system of equations with n+1 equations and as many unknowns

$a_0, ..., a_n$ to solve, with the condition that the points x_k are pairwise different:

$$\begin{pmatrix} x_0^0 & \dots & x_0^n \\ \vdots & \ddots & \vdots \\ x_n^0 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$$

The determinant of the coefficient matrix is called Vandermonde determinant. So now we come to the definition of the existence and uniqueness of the interpolation polynomial.

Theorem 2. Given n+1 pairs of values (x_k, y_k) with $k = 0, .., n$ and $y_k \in \mathbb{R}$ and x_k pairwise different $\in \mathbb{R}$, there exists a unique polynomial p with degree at most n such that $p(x_k) = y_k \forall k \in 0, \dots, n$. [3]

Proof. It can be shown that the above Vandermonde determinant is non-zero. Hence, the above linear system has a unique solution.
We now give an alternative proof for the unicity, we will do a proof by contradiction. So let's suppose there exists two interpolating polynomials $P(x)$ and $Q(x)$, of degree at most n for the same pairs of values (x_k, y_k), with $P(x) \neq Q(x)$.
Let's define $D(x) = P(x) - Q(x)$ of degree most n. Then, $D(x_k) = 0$, because $P(x_k) = Q(x_k) = y_k$. Then $D(x)$ must have n+1 roots.
But from the Fundamental Theorem of Algebra we know that $D(x)$ cannot have more than n roots. Contradiction!
It must be $P(x) = Q(x)$ and $D(x) = 0$. □

Since the above method to determine the polynomial interpolation requires us to set up long and difficult systems of equations and then solve them, it is very time-consuming. Therefore, we will be interested in other methods to determine the interpolation polynomial. Now that we know that such an interpolation polynomial exists. Therefore, we have multiple choices, among which the Lagrange and the Newton polynomials are the most common.

4 Lagrange interpolation

Lagrange interpolation allows us to find a polynomial that passes exactly through given points. Let's start by saying that we need a function that satisfies the following:

$$l_i(x) = \begin{cases} 1 & x = x_i, \\ 0 & x = x_j, j \neq i \end{cases}$$

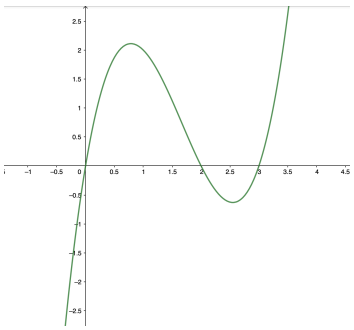
If we have the following polynomial: $y(x) = (x - a)(x - b)(x - c)$ we find the zeros when x=a, x=c and x=b. So we can produce polynomials which are equal to zero at different points. For example, if we have nodes at x=0,1,2 and 3. We would like to have the following fulfilled:

$$l_1(1) = 1 \; and \; l_1(0) = l_1(2) = l_1(3) = 0$$

So to satisfy this we could choose the following polynomial:

$$l_1^*(x) = (x-0)(x-2)(x-3)$$

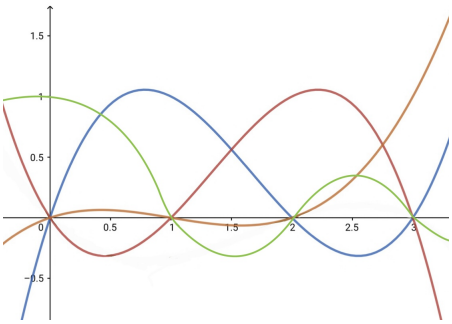
So we get the following graph:



We notice that it goes through the right nodes but at x=1 we do not get the right number. When x=1 we have (1-0)(1-2)(1-3)=2 and not 1 as we wanted. So this formula does not work. However, if we take this calculation on the denominator and take the polynomial on the numerator. We get:

$$l_1(x) = \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)}$$

We now have $l_1 = 1$ and at all the other nodes we have it equal to zero and if we divide this by the same polynomial evaluated at the same node where we want it to equal one, then we get the so-called Lagrange polynomial. So we get the following Lagrange polynomials for the nodes 0,1,2 and 3:



Definition 4.1. The Lagrange form of the interpolation polynomial is given as the sum of the function values times the Lagrange polynomials:

$$P_n(x) = \sum_{i=0}^{n} y_i \cdot l_i(x)$$

$$\text{with } l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

For a set of nodes $x_i = 1, .., n$, we get $n+1$ Lagrange polynomials, for x_i pairwise different.
The condition $j = 0, j \neq i$ is necessary, because otherwise we could get 0 on the denominator and this is impossible.

First, polynomials are created that pass through a certain point. For example, we have $l_0 = 1$ at the first point and 0 at the second point, $l_1 = 1$ at the first node and zero at the second and so on. These are then the Lagrange polynomials, which each contribute to its own node. Now each individual polynomial is multiplied by a y-value that it should have and then all are added together. To calculate the Lagrange polynomial, you only need the x-values of the given points. The polynomial that comes out of this then interpolates all the given points.

Example 4.1. We have the following points: $A = (2, 1), B = (1, 3)$ and $C = (5, 4)$. We first have to calculate the Lagrange polynomials:

$$l_0 = \frac{x-1}{2-1} \cdot \frac{x-5}{2-5} = -\frac{x^2 - 6x + 5}{3}$$
$$l_1 = \frac{x-2}{1-2} \cdot \frac{x-5}{1-5} = \frac{x^2 - 7x + 10}{4}$$
$$l_2 = \frac{x-2}{5-2} \cdot \frac{x-1}{5-1} = \frac{x^2 - 3x + 2}{12}$$

So now we have to multiply them by the y-values and add everything together:

$$P_n(x) = 1 \cdot -\frac{x^2 - 6x + 5}{3} + 3 \cdot \frac{x^2 - 7x + 10}{4} + 4 \cdot \frac{x^2 - 3x + 2}{12} = \frac{3x^2 - 17x + 26}{4}$$

The degree of the interpolation polynomial is at most as large as the number of given points minus one and there can be no other polynomial of the same degree that also passes through the same points. This calculated polynomial is therefore unique. The advantage of Lagrange interpolation is that the Lagrange polynomial depends solely on the x-values and not on the measured y-values. This means that in an experimental set-up where you always take a measurement at the same time and the same place, you can always renew this by multiplying the new y-values with the Lagrange base polynomials. The disadvantage of Lagrange interpolation is that if only one x-value is added you have to calculate all the base values again and this is very time consuming. In addition, one is inclined to make running errors in these calculations, so the Lagrange interpolation becomes numerically unstable. Therefore, we will talk about a better, faster and easier method to calculate interpolation polynomials.

5 Newton interpolation

Definition 5.1. Given n+1 pairs of points $(x_k; y_k)$ $\forall k \in 0, \ldots, n$. The Newton interpolation polynomial is the sum of constant coefficients multiplied by the Newton base polynomials.

$$P_n(x) = \sum_{i=0}^{n} C_i \cdot N_i(x)$$
$$with\ N_i(x) = \prod_{j=0}^{i-1}(x - x_j)$$

C_i being the so-called divided differences.

The problem with Newton interpolation is the same as with Lagrange interpolation, finding a polynomial which passes though given points (nodes). The difference between the two interpolation methods is the calculation of the basis polynomial. Compared to the Lagrange basis polynomials, the Newton basis is relatively easy to determine. With the Newton basis polynomials, the difficulty lies in calculating the constant coefficients, but there is a method that makes it easier, it is called divided differences. By the definition, interpolating polynomials must agree with the values at the nodes, this means: $p_n(x_i) = y_i$ and for example with (x_0, y_0), (x_1, y_1) and (x_2, y_2) as nodes we have the following:

- For 1 node: $p_0(x) = y_0 \Leftrightarrow c_0 = y_0$.

- For 2 nodes we have: $p_1(x) = c_0 + c_1(x - x_0) = a_0$. If we substitute x with x_0, we get: $p_1(x_0) = c_0 + c_1(x_0 - x_0) = a_0$ and $p_1(x_0) = y_0$. So c_0 is still y_0 like we had with only one node. So we see that when we add a new node we can keep the old calculation and only need to add the new one. If we rearrange: $p_1(x_0) = y_0 + c_1(x_1 - x_0) = y_1 \Leftrightarrow c_1 = \frac{y_1 - y_0}{x_1 - x_0} =: [y_0, y_1]$, which is the divided difference of first order.

- For 3 nodes: $p_2(x) = y_0 + [y_0, y_1](x - x_0) + c_2(x - x_0)(x - x_1)$. If we substitute x with x_2, we get: $p_2(x_2) = y_0 + [y_0, y_1](x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) = y_2$. If we rearrange this we get to: $c_2 = \frac{[y_0, y_2] - [y_0, y_1]}{x_2 - x_1}$

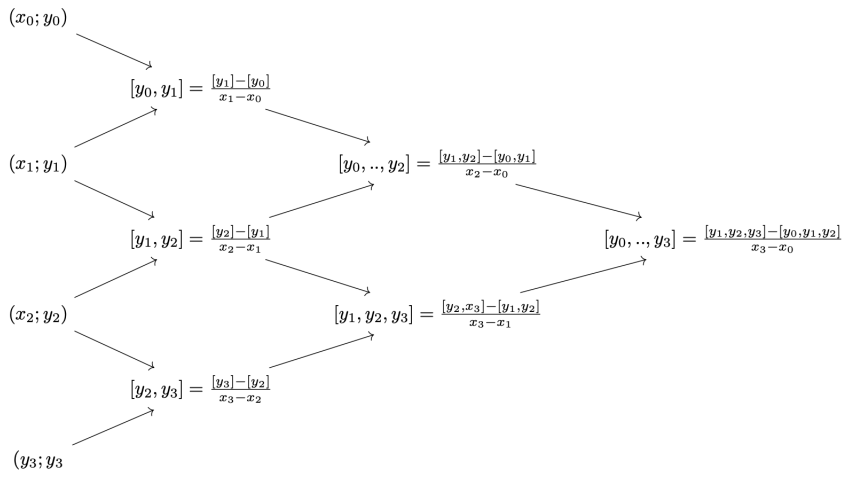
So these calculations of the divides differences leads us to a definition:

Definition 5.2. Given n+1 pairs of points $(x_k; y_k)$, k = 0, ..., n. The divided differences are defined recursively by $[y_k] = y_k$, k = 0, ..., n:

$$[y_k, y_{k+1}, ..., y_{k+(i-1)}, y_{k+i}] = \frac{[y_{k+1}, ..., y_{k+(i-1)}, y_{k+i}] - [y_k, y_{k+1}, ..., y_{k+(i-1)}]}{x_{k+i} - x_k}$$

for i = 1, ..., n and k = 0, ..., n-1
This equation allows us to calculate the $n+1^{th}$ order of divided differences from an n^{th} order of divided differences.

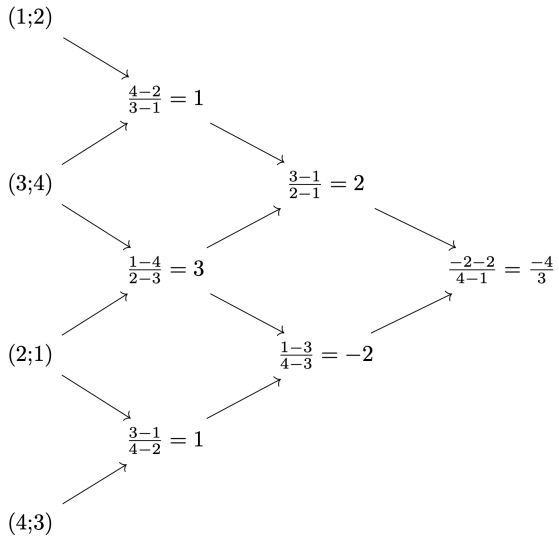
So now we can use the board of divided differences :



For the scheme of divided differences, we think of the difference quotient. That means we divide the difference of two y-values by the difference of the corresponding x-values. We do this with all adjacent pairs of points. We can also swap the rows, the scheme will then be slightly different, but the end result remains the same.

Example 5.1.

We want to determine the polynomial with the minimum degree that goes through the following points.

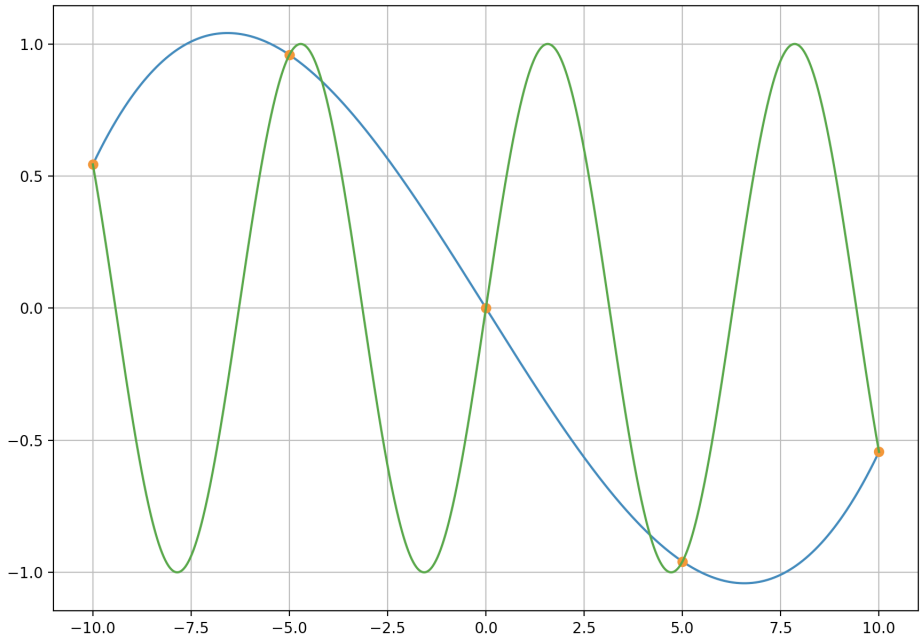


Now we have calculated the coefficients and just the following calculation left to do:

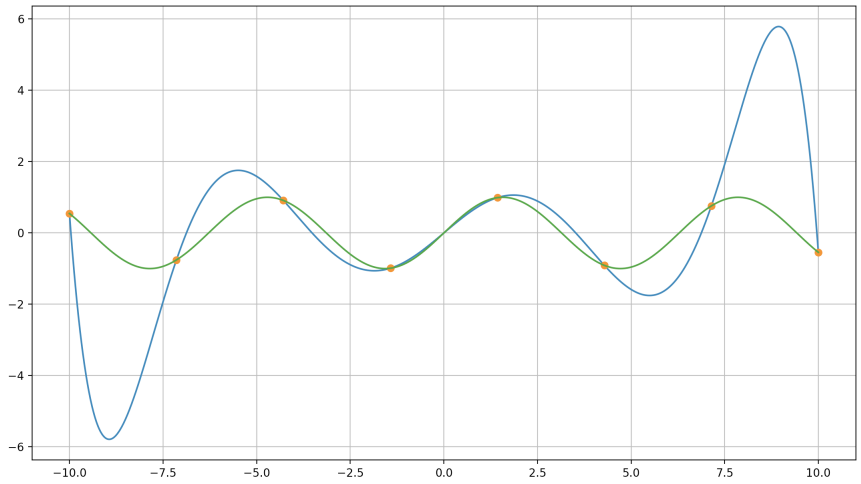
$$\begin{aligned} P_3(x) &= 2 + 1 \cdot (x-1) + 2(x-1)(x-3) - \frac{4}{3}(x-1)(x-3)(x-2) \\ &= \frac{4}{3}x^3 + 10x^2 - \frac{65}{3}x + 15 \end{aligned}$$

So we see that this calculation is much shorter to calculate the interpolation polynomial compared to the Lagrange method. If we want to add another point through which this interpolation polynomial should pass, we do not have to start the calculation from the beginning, but we can simply add the point at the bottom of the table and then calculate the scheme of divided differences again and let the pyramid point to a value again, then we can use the same summands, we only have to add one summand at the end. In addition, the last point is always missing from the sum, so you can reduce numerical rounding errors by cleverly swapping the rows. So you can make the problem numerically more stable.

To show how Newton interpolation works, we take a look at another example. In particular, we will also investigate if interpolation polynomials can be used to approximate functions (Recall that by Weierstrass theorem, continuous functions on closed intervals can be approximated by interpolation polynomials). We consider the function f(x) = sin(x) on the interval $[-10; 10]$. We know that this function is oscillating between -1 and 1. First, we start by interpolating sin(x) by 5 equidistant points. We get the following graph:

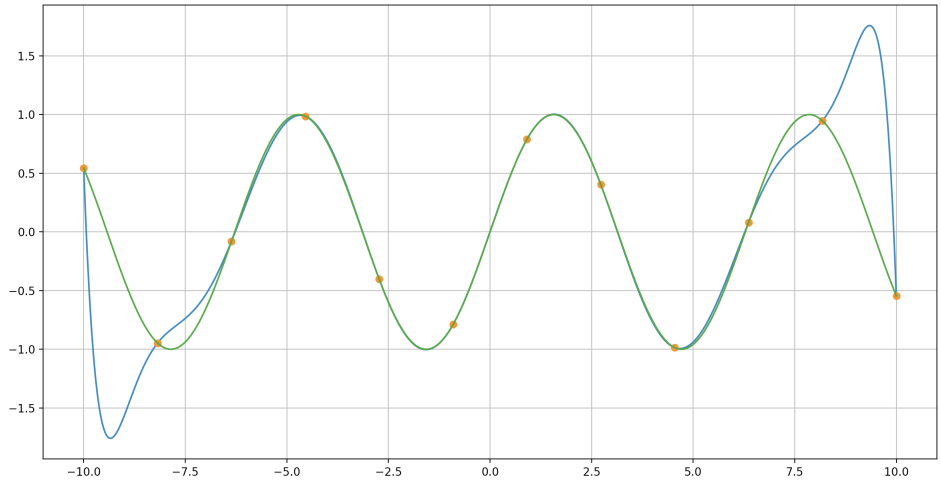


Here, the green line represents the original sin(x) function, and the blue line is the interpolation polynomial at 5 equidistant points of sin(x). We conclude that the maximal error between sin(x) and the polynomial is still very high because the interpolation cannot take into account the oscillation of sin(x) yet.
Therefore, we try by interpolating at 8 points. We get the following graph:

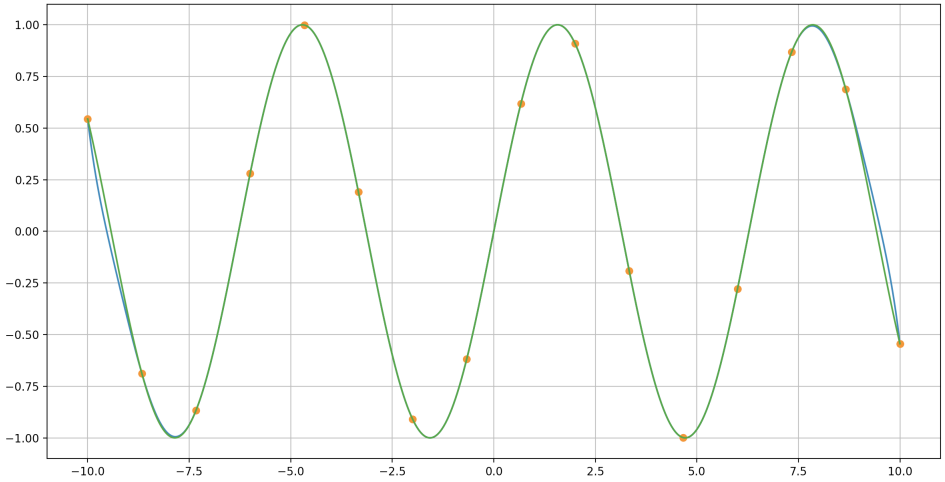


We can see an amelioration in the approximation because more interpolation points can increase the accuracy of the approximation, but it is still not perfect.

We now take 12 equidistant points:



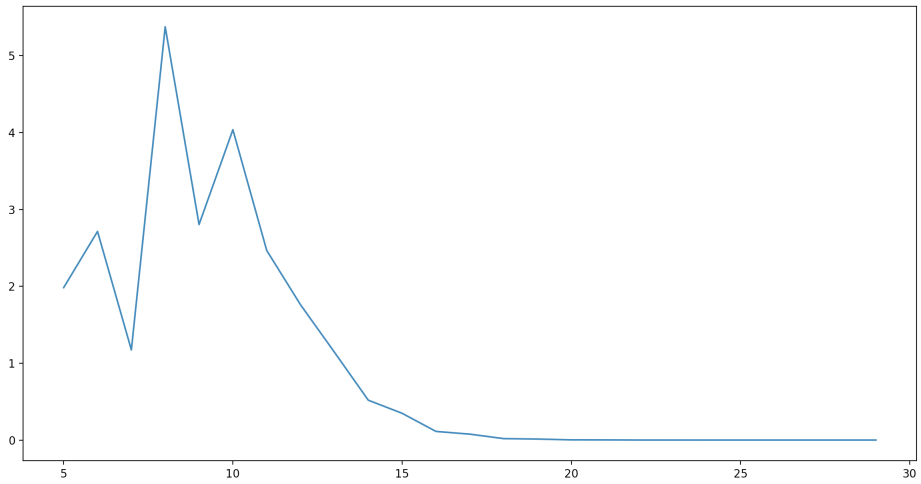
We see that the interpolation error minimizes whilst approaching 0, but still expands towards both extremities of the graph. Therefore, we now take 16 equidistant points to interpolate at and we get the following graph:



This interpolation polynomial is already very close to the real sin(x) function. We now have a look at the evolution of the maximum error whilst increasing the number of interpolation points. This table represents the number of interpolation points and the absolute value of the maximum error between the interpolation polynomial P(x) and sin(x).

Number of points	$\max_{[-10;10]} \lvert \sin(x) - P(x) \rvert$
5	1.98
6	2.71
7	1.17
8	5.37
9	2.80
10	4.03
11	2.46
12	1.75
13	1.14
14	0.51
15	0.34
16	0.11
17	0.07
18	0.02
19	0.01
20	0.003
55	0.0003

We can represent this table in a graph, the x-axis gives the number of points where we interpolate in, and the y-axis gives the maximum error.



We see that the maximum error tends towards 0 as the number of interpolation points increases. This is the case for a lot of functions.

6 Polynomial interpolation error

We take individual points of a function $f(x)$ in order to calculate an interpolation polynomial $p(x)$ for these points. Now we would like to find out the interpolation

error that appears when calculating the interpolation polynomial, i.e. how large the error is between the polynomial $f(x)$ and the polynomial $p(x)$. So in other words, we want to know how good $p(x)$ approximates $f(x)$ on a certain interval. The error varies from point to point, at some points it is zero but at other points it could be very high. This is because we only have the data from the given points and know nothing about the intervals between the points. So we want to find out the maximum error that the interpolation polynomial has. We have the following theorem and definition:

Theorem 3. If we have n+1 pairwise different points on an Interval I=[a,b] and f(x) has n+1 derivatives.The error at $x \in I$ is defined as: $E_n(x, f) = f(x) - p(x)$ where $p(x)$ denotes the interpolation polynomial that interpolates f. Then $\forall x \in I$, there exists a number $\xi(x)$ in the interval I such that:

$$f(x) = p(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)...(x - x_n)$$

So we obtain the following expression for the maximum error:

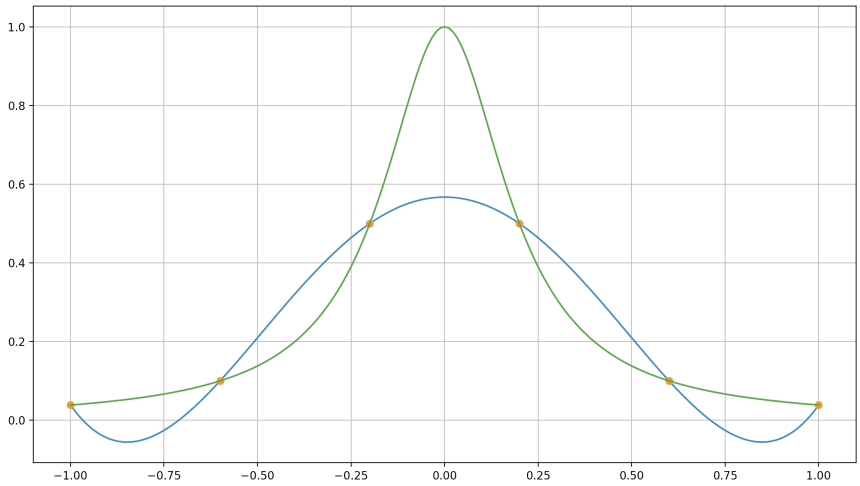
$$\max_{[a,b]}|f(x) - p(x)| \leq \frac{1}{(n+1)!} \max_{[a,b]}|f^{(n+1)}(x)| \max_{[a,b]}|\prod_{i=0}^{n}(x - x_i)|$$

The first term depends of the function f(x) and the second term depends on the choice of points.

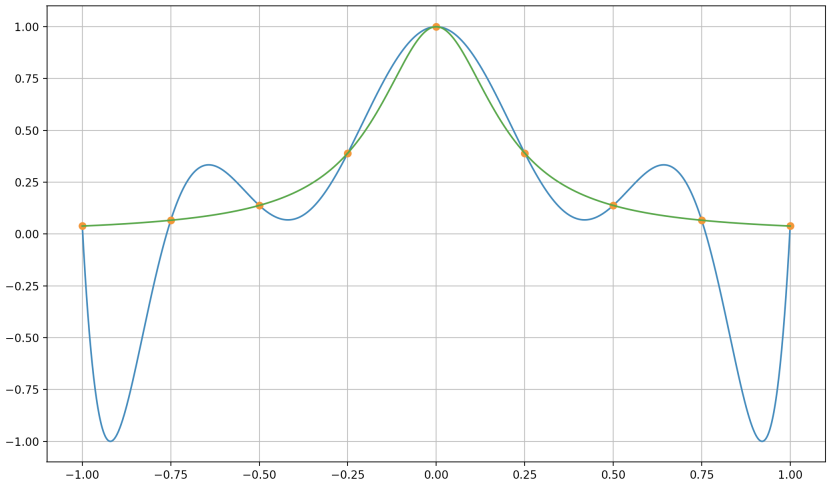
If we increase the number of points, one would think that the interpolation error should go towards 0 and that approximation by interpolation polynomials could be possible. However, as we shall now see, this is not always the case.

7 Runge's phenomenon

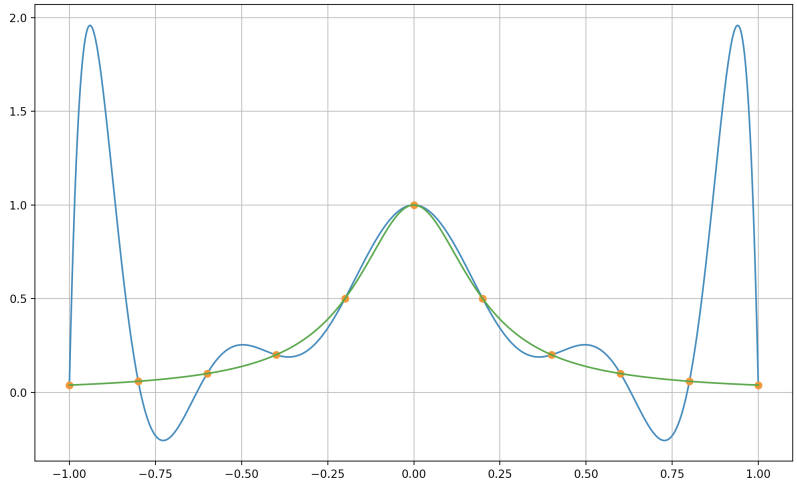
We now try to interpolate the so-called Runge function $f(x) = \frac{1}{25x^2+1}$. We start by interpolating in 5 equidistant points in [-1;1].



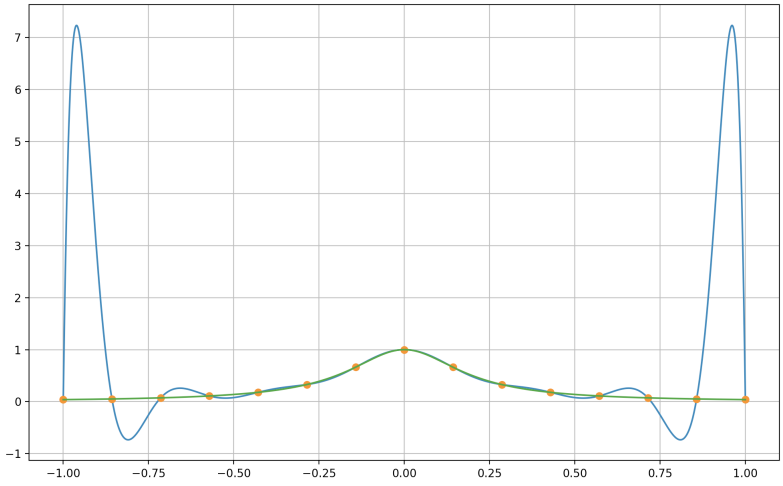
Recall that the green line represents f(x) and the blue line represents the interpolation polynomial of f(x). We see that five points are not enough to make a perfect approximation, so we try, by what we know so far, to increase the number of interpolation points in order to increase approximation accuracy.



With 8 interpolation points, the accuracy increases in the vicinity of 0. Now we increase the number of points again, and with 11 points, we can observe a strange behavior of the Newton interpolation polynomial.



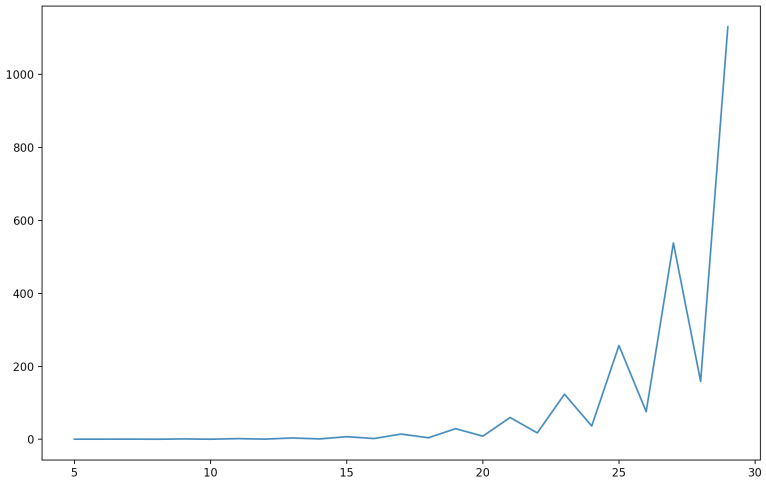
As the approximation accuracy increases around x=0, the maximum error between f(x) and the interpolation polynomial increases as well unlike expected. When interpolating at 15 points, we get the evidence that the approximation does not work as we expected:



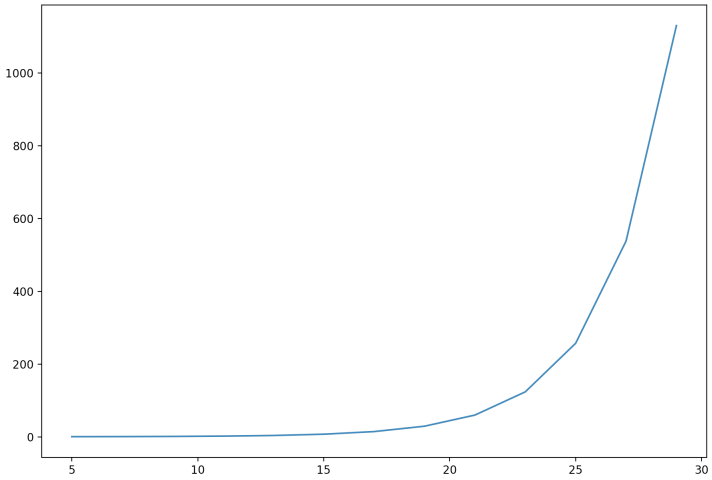
It seems that the interpolation for a function like f(x) is not usable for approximation, because it gets worse while increasing the number of interpolation points. Before looking at the actual issue we commit while interpolating, let us analyze the evolution of the maximum error between our interpolation polynomial P(x) and f(x).

Number of points	$\max_{[-10;10]} \lvert f(x) - P(x) \rvert$
5	0.438
6	0.432
7	0.617
8	0.247
9	1.0
10	0.3
11	1.91
12	0.55
13	3.66
14	1.07
15	7.19
16	2.1
17	14.37
18	4.22
19	29.2
20	8.57
21	59.8
22	17.6
23	123.6
24	36.4
25	257.2
26	75.8
27	538.2
28	158.7
29	1130.7
45	481579.3
75	51960402911.3

We can represent this table in a graph, where we can observe another phenomenon:



First, we see that the maximum error is increasing, as we expected. We can observe as well that for an odd number n of interpolation points, the error is greater than the error when taking the next even n+1 interpolation points. This is due because if we try to have a symmetry both on the left and on the right side of the y-axis and by taking an odd number of interpolation points, one point will lie exactly on the y-axis, which causes this to happen. By taking only odd numbers of points, we get the following maximum error graph:



As we now know that for some functions, the approximation by interpolation can go wrong and we can find interpolation polynomials that interpolate in a lot of points but are far off the given function. We emphasize that so far, we only considered equidistant interpolation points. We suppose that by choosing the interpolation points differently, we can reduce the maximum error.

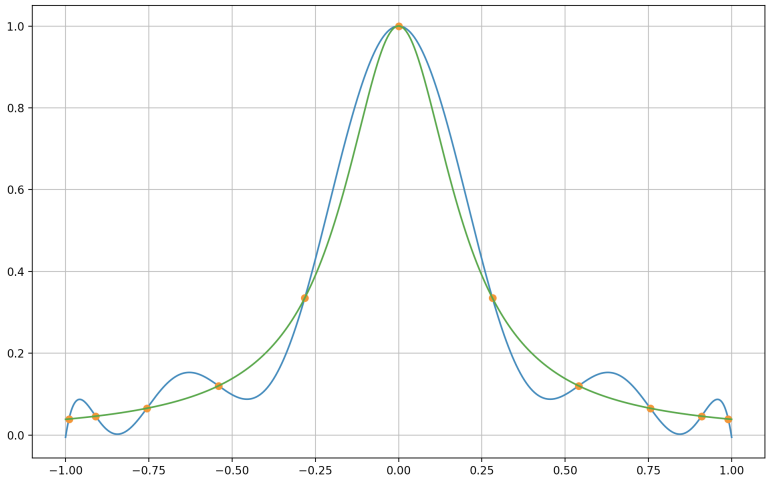
8 Chebyshev interpolation

Chebyshev nodes for polynomial interpolation were designed to optimize the choice of nodes in a way that the error will be distributed equally throughout the interpolating interval.
Consider an interval]a,b[, then the Chebyshev nodes are defined as follows:

$$\bar{x}_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos(\frac{2i+1}{2n+2}\pi) \ \forall i \in (0,..,n)$$

So we have again the shift to the midpoint of interval [a,b] and then the scaling factor with the cosine function which distributes the x_i on this interval.
In conclusion we now know that if we take the Chebyshev nodes instead of equidistant points, the error can be minimized so much such that $|f(x)-P(x)| < \epsilon$ can be achieved $\forall \epsilon > 0$.

If we now take the Chebyshev nodes with our Runge function, we get the following graph for 11 interpolation points:

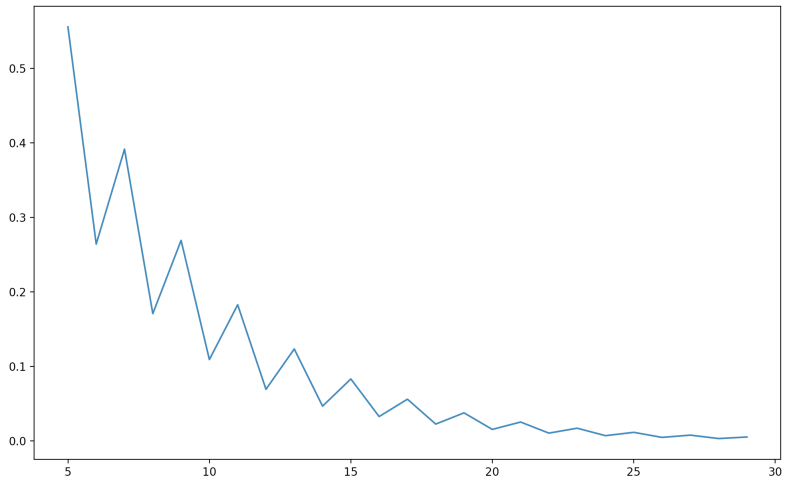


The maximum approximation error is 0.1 for this graph and it was 1.91 for the Runge function interpolated on equidistant points.

We can resume the maximum approximation error for interpolating the Runge function f(x) with equidistant points versus Chebyshev nodes in a table. We call the interpolation polynomial interpolating through Chebyshev nodes C(x).

Number of points	$\max_{[-10;10]} \lvert f(x) - P(x) \rvert$	$\max_{[-10;10]} \lvert f(x) - C(x) \rvert$
7	0.617	0.26
9	1.0	0.17
11	1.91	0.11
13	3.66	0.07
15	7.19	0.05
17	14.37	0.03
19	29.2	0.02
21	59.8	0.02
23	123.6	0.01
25	257.2	0.01
27	538.2	0.004
29	1130.7	0.003
75	51960402911.3	

By interpolating in Chebyshev nodes, we can represent the maximum approximation error in a graph:



9 Python code

In this section, we will briefly look at the Python code for the Newton interpolation of the Runge function.

```
1 import numpy as np
2 from matplotlib import pyplot as plt
```

Therefore, we have the part where we define our functions:

```
def divided_differences(x, y):
    matr = np.zeros([len(x), len(x)])
    matr[:,0] = y
    for k in range(1,len(x)):
        for i in range(len(x)-k):
            matr[i][k] = \
           (matr[i+1][k-1] - matr[i][k-1]) / (x[i+k]-x[i])

    return matr


def newton(matr, x, new_x):

    n = len(x) - 1
    p = matr[n]
    for k in range(1,len(x)):
        p = matr[n-k] + (new_x -x[n-k])*p
    return p

def runge(x):
    y = 1 / (25*x**2 + 1)
    return y
```

We define the `divided_differences`, the `Newton` interpolation as well as the `Runge` function which we will use later on.
Then we proceed by defining in what range of points we want to interpolate, so here, we start by interpolating in 5 points until we end up at 30 excluded.

```
F = []
N = range(5,30,1)
```

Then we create a for loop in which we interpolate for every number of points. Here, we first create the `Runge` function again in order to be displayed as the green line on every graph. Then, we create some points, in this case 1000 new points where we compute the y-coordinate via the Newton formula. This gives us a very rounded approximation polynomial curve. Then, we use this part of the code to compute the maximum error between the interpolation polynomial graph P(x) and f(x) as well, which is stored in a list called F. Thereafter, the code just plots all the graphs with P(x), f(x) and the interpolation points and it outputs the maximum error as well.

```
for n in N:
    x = np.linspace(-1,1,n)
    y = 1 / (25*x**2 + 1)
    d_d0 = divided_differences(x, y)[0]
    new_x = np.linspace(-1, 1, 1000)
    new_y = newton(d_d0, x, new_x)
```

```
    f = max(np.absolute(runge(x_plot)-new_y))
    F = np.append(F, f)
    print("The maximum error is: ", f)
    plt.plot(new_x, new_y)
    plt.plot(x, y, "o")
    plt.plot(x_plot, runge(x_plot))
    plt.grid()
    plt.show()
plt.plot(N, F)
plt.show()
```

These code snippets can be run all together in that order in one .py file. If we now change the definition of x from the np.linspace to

```
x = chebyshev(-1, 1, n)
```

where [a,b] is the interval which the nodes are computed in and n is the number of nodes, we can create the Chebyshev nodes like in the following code:

```
def chebyshev(a, b, n):
    k = np.array(range(n+1))
    c = np.cos((2*k+1)*(np.pi)/(2*(n+1)))
    return 0.5*(a+b)+0.5*(b-a)*c
```

If we add this function to our code, then it will compute the Runge interpolation in the Chebyshev nodes so that the error of approximation decreases by increasing the number of interpolation points.

10 References

[1] Michael T. Heath. *Scientific Computing: An introduction Survey* , International edition 1979

[2] Eugene Isaacson Herbert Bishop Keller. *Analysis of numerical Methods*, Dover Edition 1994

[3] Michael Knorrenschild. *Numerische Mathematik: Eine beispielorientierte Einführung*, September 2012 (5. aktualisierte Auflage)

[4] Günter Bärwolff. *Numerik für Ingenieure, Physiker und Informatiker*, Juni 2006 (3. Auflage)

[5] Michelle Schatzman. *Numerical Analysis: a mathematical introduction*, 2002