Exercise Sheet № 6 Interpolation and Splines

Before any tasks are solved, I want to introduce some conventions and common notations used throughout this submission. The main topic of this exercise sheet is interpolation. For an interpolation problem, we are given a dataset $\boldsymbol{y} \in \mathbb{R}^{n+1}$ supported by the $n+1$ nodes $\mathcal{X}$. Note that $\mathcal{X}$ denotes an ordered $n+1$-tuple of the form $(x_0, x_1, \dots, x_n) \in \mathbb{R}^{n+1}$, where $\forall i = 1, \dots, n\colon x_{i-1} < x_i$. The data-vector $\boldsymbol{y}$ can be any arbitrary vector. Generally speaking, for interpolation we are given $n+1$ data-points $\boldsymbol{y}$ from an unknown function $f\colon D \to \mathbb{R}$, where $D \subseteq \mathbb{R}$. Our goal is to find a function $p\colon D \to \mathbb{R}$, such that $\forall i = 0, \dots, n\colon f(x_i) = p(x_i)$. We call p the *interpolating function*, or *interpolant*, of f.

Within this submission we will mostly assume, that $p \in \mathbb{R}_n[x]$, where

$$\mathbb{R}_n[x] = \left\{ f\colon \mathbb{R} \to \mathbb{R}, f(x) = \sum_{i=0}^{n} a_i x^i \; a_i \in \mathbb{R} \right\}$$

The set $\mathbb{R}_n[x]$ forms a commutative algebra with the point-wise multiplication and addition of functions. The canonical basis for $\mathbb{R}_n[x]$ is chosen as:

$$\{1, x, x^2, \dots, x^n\}$$

Lemma 6.1

Let $p(x) = \sum_{i=0}^{n} \alpha_i x^i$ and $x_0, \dots, x_n$ be $n+1$ distinct values, such that $p(x_i) = 0$, then it follows that $\forall i = 0, \dots, n\colon \alpha_i = 0$.

Proof. Since $p(x_i) = 0$, we get that x_i is a root of p, thus:

$$p(x) = K \prod_{i=0}^{n} (x - x_i)$$

If $K \neq 0$, then $\deg p = n+1$, which is a contradiction to $\deg p \leq n$, therefore $K = 0$, which in return means $\alpha_i = 0$, since:

$$p(x) = \sum_{i=0}^{n} \alpha_i x^i = K \prod_{i=0}^{n} (x - x_i) = 0$$

$\square$

***Definition 6.1:* lerp**

Let $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^2$, then we call $\operatorname{lerp}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ the affine space spanned by the following map:

$$k = \frac{y_2 - y_1}{x_2 - x_1} \qquad d = y_1 - k x_1$$
$$\operatorname{lerp}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \operatorname{im}(kx + d)$$

If $d = 0$, lerp is a subspace of $\mathbb{R}^2$. Alternatively, $\operatorname{lerp}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ may also be represented as a convex-combination of the following form:

$$\operatorname{lerp}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \{(1-t)\boldsymbol{x}_1 + t\boldsymbol{x}_2 | t \in [0,1]\}$$

A third way of defining lerp is as a function $\operatorname{lerp}\colon \mathbb{R} \to \mathbb{R}^2$ with:

$$\operatorname{lerp}[\boldsymbol{x}_1, \boldsymbol{x}_2](x) = (1-x)\boldsymbol{x}_1 + x\boldsymbol{x}_2$$

Definition 6.2: Average of a function

Let $f \in \mathcal{C}(I)$ where $I \subseteq \mathbb{R}$ is an interval and $f\colon I \to \mathbb{R}$. Then the average value of f on I, denoted here $\operatorname{avg}_I f$ is defined as

$$\operatorname{avg}_I f = \frac{1}{|I|} \int_I f(x)\,\mathrm{d}x$$

Lemma 6.2: Aitken

Let $\mathcal{X}$ be a set of nodal points, $n \in \mathbb{N}$ and $x \in \mathbb{R}$, then the interpolating polynomials for Neville's method satisfy:

$$p_n(x) = p_{n,n}(x) = \frac{(x - x_0)p_{n,n-1}(x) + (x_n - x)p_{n-1,n-1}(x)}{x_n - x_0} \tag{1}$$

where $p_{i,k}$ is a polynomial with $\deg p_{i,k} \leq k$, defined on the nodes $x_{i-k}, \ldots, x_i$.

6.1 Lagrange-Interpolation

Task 6.1: Polynomial Interpolation via Lagrange's Method

1. Prove that given distinct nodal points $x_0 < x_1 < \cdots < x_n$, the Lagrange-polynomials ℓ_{jn} defined by

$$\ell_{jn}(x) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k} \qquad x \in \mathbb{R} \qquad 0 \leq j \leq n$$

   form a basis for the vector space $\mathbb{R}_n[x]$ of polynomials of degree n or less.
2. Given the following nodal points and function values:

$\mathcal{X}$	0	0.5	1	1.5
$\boldsymbol{y}$	1	2	3	4

Table 1: Data for task 6.1

   compute the Lagrange polynomials ℓ_{jn} for $0 \leq j \leq n$ and $n = 1, 2, 3$ and the corresponding Lagrange interpolating polynomials p_n.
3. Given distinct nodal points $x_0 < x_1 < \cdots < x_n$ stored in the array write a python script containing the function `LagrangeInterpPoly(x_data)` which returns a plot (in a single frame) of the Lagrange basis functions ℓ_{jn} and a plot of the interpolating polynomial $p_n(x)$. Test your script using the data given in table 1.

Subtask 1:

We want to prove, that $\operatorname{span}_{j=0}^{n}(\ell_{jn}) = \mathbb{R}_n[x]$. Notice the following:

$$\ell_{jn}(x_i) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x_i - x_k}{x_j - x_k} = \delta_{ij}$$

Let $p \in \operatorname{span}_{j=0}^{n}(\ell_{jn})$ and $f \in \mathbb{R}_n[x]$:

$$f(x_i) \overset{!}{=} p(x_i) = \sum_{j=0}^{n} \alpha_j \ell_{jn}(x_i) = \sum_{j=0}^{n} \alpha_j \delta_{ij} = \alpha_i$$
$$\Rightarrow p(x) = \sum_{j=0}^{n} f(x_j) \ell_{jn}(x)$$

Let $h(x) = f(x) - p(x)$, then $h(x_i) = 0$ and $\deg h \leq n$, by lemma 6.1 we get $h(x) = 0$, thus $0 = f(x) - p(x) \Leftrightarrow f(x) = p(x)$. To show that p is unique, let $q \in \operatorname{span}_{j=0}^{n}(\ell_{jn})$ with $\forall i = 0, \ldots, n\colon q(x_i) = p(x_i)$, then it follows that $g(x) = q(x) - p(x) = 0$, thereby $q(x) = p(x)$. Hence $\operatorname{span}_{j=0}^{n}(\ell_{jn}) = \mathbb{R}_n[x]$.

Subtask 2:

j	0	1	2	3
		$n=1$		
$\ell_{j1}(x)$	$1-2x$	$2x$		
$p_1(x)$	$2x+1$			
		$n=2$		
$\ell_{j2}(x)$	$(x-1)(2x-1)$	$4x(1-x)$	$x(2x-1)$	
$p_2(x)$	$2x+1$			
		$n=3$		
$\ell_{j3}(x)$	$-\frac{4}{3}x^3+4x^2-\frac{11}{3}x+1$	$4x(1-x)\left(x-\frac{3}{2}\right)$	$x(-4x^2+8x-3)$	$\frac{4}{3}x(x-1)\left(x-\frac{1}{2}\right)$
$p_3(x)$	$2x+1$			

Table 2: Lagrange Polynomials and interpolating functions for various n

Since this is quite a repetitive task, I opted to utilize the sympy package in python. The following snippet provides a simple method of computing the Lagrange polynomials symbolically:

```python
import numpy as np
import sympy as sp
x = sp.symbols('x')

def ljn(xv, j, n):
    xt = np.delete(xv, j)[:n]
    return np.prod((x-xt) / (xv[j] - xt))

def pn(xv, y, n):
    pn = 0
    for j in range(n+1):
        pn += y[j] * ljn(xv, j, n)
    return sp.simplify(pn)

x_data = [0, 1/2, 1, 3/2]
y_data = [1,2,3,4]

for n in [1,2,3]:
    print(n)
    for j in range(n+1):
        display(ljn(x_data, j, n))
    display(pn(x_data, y_data, n))
```

The code above was run in a jupyter-notebook, hence the `display()` calls. To use the snippet as a regular script, replace `display()` with regular `print()` calls.

Subtask 3:

Since we generally do not need a symbolic representation of p_n, we are only interested in it's values along a given interval. Therefore the supplied algorithm computes the values of ℓ_{jn} and p_n respectively on an additional parameter $\boldsymbol{x}$, which may be any 1-dimensional ndarray from the numpy package. The process for a particular x_j is detailed below. First we store the current x_j in a corresponding variable and create a temporary node-vector $\boldsymbol{x}_t = \begin{bmatrix} x_0 & \cdots & x_{j-1} & x_{j+1} & \cdots & x_n \end{bmatrix}$. Given the x-axis $\boldsymbol{t}$ we want to evaluate ℓ_{jn} over, we now compute the „outer difference“ of the two vectors $\boldsymbol{t}$ and $\boldsymbol{x}_t$, which looks like the following:

$$\boldsymbol{t} \ominus \boldsymbol{x}_t = \begin{bmatrix} t_1 \mathbf{1}_{n-1}^T - \boldsymbol{x}_t^T \\ t_2 \mathbf{1}_{n-1}^T - \boldsymbol{x}_t^T \\ \vdots \\ t_m \mathbf{1}_{n-1}^T - \boldsymbol{x}_t^T \end{bmatrix} = \begin{bmatrix} t_1 - x_0 & t_1 - x_1 & \cdots & t_1 - x_{j-1} & t_1 - x_{j+1} & \cdots & t_1 - x_n \\ t_2 - x_0 & t_2 - x_1 & \cdots & t_2 - x_{j-1} & t_2 - x_{j+1} & \cdots & t_2 - x_n \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ t_m - x_0 & t_m - x_1 & \cdots & t_m - x_{j-1} & t_m - x_{j+1} & \cdots & t_m - x_n \end{bmatrix}$$

$$\mathbf{1}_n \in \mathbb{R}^n \qquad \mathbf{1}_n = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^T$$

Notice that each row of $\boldsymbol{t} \ominus \boldsymbol{x}_t$ now has the numerators of the factors. Next we perform an element-wise division:

$$(\boldsymbol{t} \ominus \boldsymbol{x}_t) \oslash (x_j \mathbf{1}_n - \boldsymbol{x}_t) = \begin{bmatrix} \frac{t_1 - x_0}{x_j - x_0} & \frac{t_1 - x_1}{x_j - x_1} & \dots & \frac{t_1 - x_{j-1}}{x_j - x_{j-1}} & \frac{t_1 - x_{j+1}}{x_j - x_{j+1}} & \dots & \frac{t_1 - x_n}{x_j - x_n} \\ \frac{t_2 - x_0}{x_j - x_0} & \frac{t_2 - x_1}{x_j - x_1} & \dots & \frac{t_2 - x_{j-1}}{x_j - x_{j-1}} & \frac{t_2 - x_{j+1}}{x_j - x_{j+1}} & \dots & \frac{t_2 - x_n}{x_j - x_n} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \frac{t_m - x_0}{x_j - x_0} & \frac{t_m - x_1}{x_j - x_1} & \dots & \frac{t_m - x_{j-1}}{x_j - x_{j-1}} & \frac{t_m - x_{j+1}}{x_j - x_{j+1}} & \dots & \frac{t_m - x_n}{x_j - x_n} \end{bmatrix}$$

Computing the element-product of the k-th row yields the evaluation of the Lagrange polynomial ℓ_{jn} in t_k. The python package numpy provides vectorized functions for all the described operations, thus the algorithm should be sufficiently fast.

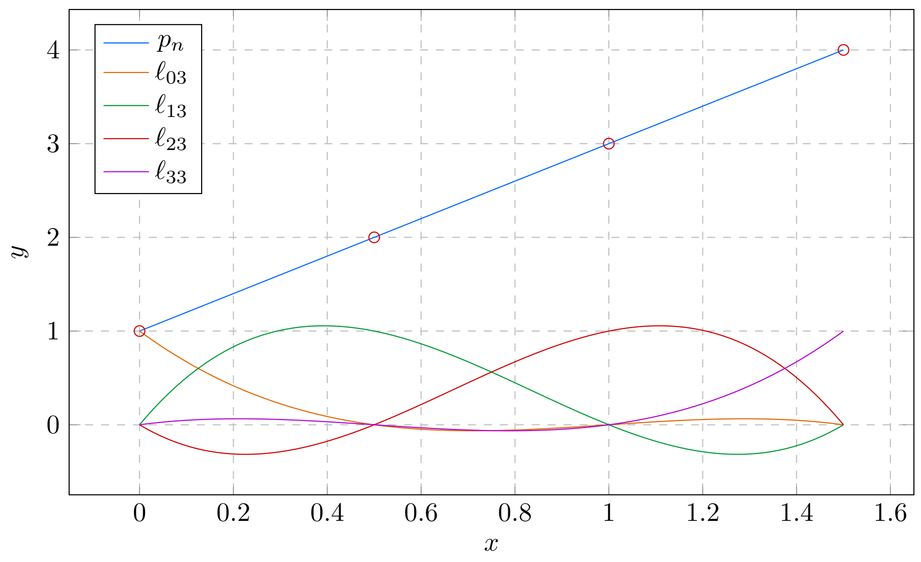

Figure 1: Interpolating Polynomial $p_n(x)$ and the corresponding Lagrange polynomials for table 1

The data generation for the figure above can be found in the submitted jupyter notebook `lagrange_test.ipynb`. The implementation of the lagrange interpolation can be found in the submitted python-file `Lagrange.py`.

Addendum: Although the provided algorithm does work with multiple datasets for $\boldsymbol{y}$, changing the nodes $\boldsymbol{x}$ requires a recomputation of all basis-polynomials, which can become quite computationally intensive if m is large. A more fitting approach is the *Barycentric form*, [Jea04], which decomposes each basis-polynomial into three parts:

$$\ell_j(x) = \ell(x) \frac{w_j}{x - x_j}$$
$$w_j = \prod_{\substack{i=1 \\ i \neq j}}^{n} \frac{1}{x_j - x_i} \qquad \ell(x) = \prod_{i=1}^{n} (x - x_i)$$

We call w_j the *barycentric weight* of ℓ_j. Factoring $\ell(x)$ from the linear combination of f yields the *first barycentric form*

$$p(x) = \ell(x) \sum_{j=0}^{n} \frac{w_j}{x - x_j} y_j \tag{2}$$

Notice that eq. (2) has no analytical problems, however evaluating the sum close to any x_i in an implementation may produce grave numerical errors, since x values very close[1] to any x_i will cause the multiplication of „very small“ values with „very large“ values, relatively speaking. To avoid such errors, any implementation needs to check wether p is evaluated in x_i and simply return y_i instead of computing the sum.

[1]depending on the given architecture

Equation (2) can be further improved. Notice that the constant function 1 has the following representation in the Lagrange basis:

$$1 = \ell(x) \sum_{j=0}^{n} \frac{w_j}{x - x_j}$$

Thus dividing $p(x)$ from eq. (2) yields:

$$p(x) = \frac{p(x)}{1} = \frac{\ell(x) \sum_{j=0}^{n} \frac{w_j}{x-x_j} y_j}{\ell(x) \sum_{j=0}^{n} \frac{w_j}{x-x_j}} = \frac{\sum_{j=0}^{n} \frac{w_j}{x-x_j} y_j}{\sum_{j=0}^{n} \frac{w_j}{x-x_j}} \tag{3}$$

The formula for $p(x)$ from eq. (3) reduces the evaluation effort to $\mathcal{O}(n)$ floating-point operations, as the weights w_j can be computed beforehand.

6.2 Newton-Interpolation

Task 6.2: Polynomial Interpolation via Newton's Method

1. Given the data points `x_data` and `y_data`, write a python script containing
   a the function `compute_coeffs(x_data, y_data)` which returns the coefficient array `coeffs` in the divided difference table for Newton's Interpolation method
   b the function `eval_poly(coeffs, x_data, x)` which evaluates the interpolant p at any point x vie Horner's method.
2. The data points in table 3 lie on the graph of the function $f(x) = 4.8 \cdot \cos \frac{\pi x}{20}$. Using your python script from Subtask 1, interpolate this data by Newton's method at $x = 0, 0.5, 1, \dots, 8$ and compare the results with the „exact“ values $y_i = f(x_i)$.

$\mathcal{X}$	0.15	2.30	3.15	4.85	6.25	7.95
$\boldsymbol{y}$	4.79867	4.49013	4.2243	3.47313	2.66674	1.51909

Table 3: Data for task 6.2

Subtask 1: [Han11, p. 97] introduces the following algorithm for computing the coefficients c_i:

Algorithm 1: Coefficients for divided differences

```
input: $\boldsymbol{y} \in \mathbb{R}^{n+1}$, nodes $\mathcal{X}$
output: $\boldsymbol{c} \in \mathbb{R}^{n+1}$

compute_coeffs($\boldsymbol{y}$, $\mathcal{X}$):
    $\boldsymbol{c} = \boldsymbol{y}$
    for $k = 1, \dots, n$ do
        for $i = n, n-1, \dots, k$ do
            $\boldsymbol{c}[i] = \frac{\boldsymbol{c}[i] - \boldsymbol{c}[i-1]}{\mathcal{X}[i] - \mathcal{X}[i-k]}$
        end
    end
    return $\boldsymbol{c}$
```

Note, that evaluating the interpolant requires a slightly modified version of horner's method, [Han11]:

Algorithm 2: Modified Horner's method for newton interpolation

```
input: $\boldsymbol{c} \in \mathbb{R}^{n+1}$, nodes $\mathcal{X}$, $x \in \mathbb{R}$
output: $p(x) \in \mathbb{R}$

eval_poly($\boldsymbol{c}$, $\mathcal{X}$, $x$):
    $p = \boldsymbol{c}[n]$
    for $k = n-1, \dots, n$ do
        p = $\boldsymbol{c}[k] + (x - \mathcal{X}[k])p$
    end
    return $p$
```

Using the python package numpy, the code from algorithm 2 can utilize numpy's vectorized operations and thus runs rather quickly if we want to evaluate $p(x)$ over a large x-axis $\boldsymbol{x}$.

Subtask 2:

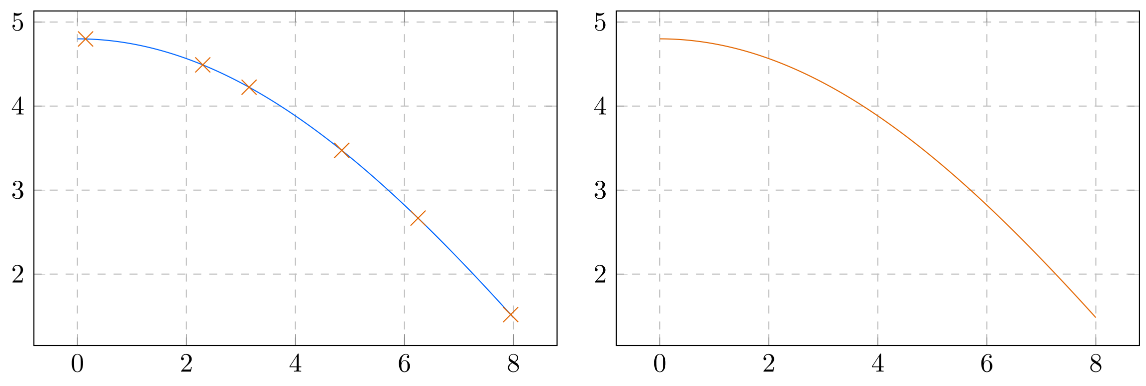

Figure 2: Plot of $f(x)$ (left) and $p(x)$ (right) over $[0, 8]$ and the sample points

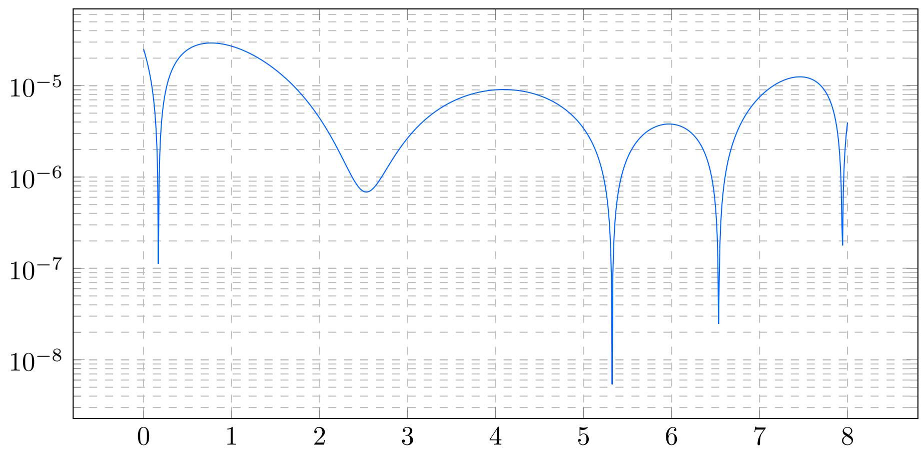

Figure 3: Absolute error $|f(x) - p(x)|$ over $[0, 8]$

x_i	$f(x_i)$	$p(x_i)$	$\lvert f(x_i) - p(x_i)\rvert$
0	4.800000	4.800025	$2.509448 \cdot 10^{-5}$
0.5	4.785203	4.785178	$2.471042 \cdot 10^{-5}$
1	4.740904	4.740877	$2.706328 \cdot 10^{-5}$
1.5	4.667376	4.667361	$1.491978 \cdot 10^{-5}$
2	4.565071	4.565067	$4.415098 \cdot 10^{-6}$
2.5	4.434622	4.434621	$6.970328 \cdot 10^{-7}$
3	4.276831	4.276829	$2.665362 \cdot 10^{-6}$
3.5	4.092673	4.092666	$6.641327 \cdot 10^{-6}$
4	3.883282	3.883273	$8.997872 \cdot 10^{-6}$
4.5	3.649949	3.649941	$7.787585 \cdot 10^{-6}$
5	3.394113	3.394109	$3.411524 \cdot 10^{-6}$
5.5	3.117351	3.117352	$1.622021 \cdot 10^{-6}$
6	2.821369	2.821373	$3.794143 \cdot 10^{-6}$
6.5	2.507993	2.507994	$4.673448 \cdot 10^{-7}$
7	2.179154	2.179147	$7.491871 \cdot 10^{-6}$
7.5	1.836880	1.836868	$1.242922 \cdot 10^{-5}$
8	1.483282	1.483286	$3.968906 \cdot 10^{-6}$

Table 4: Numerical evaluations of f and p at various x-values and their absolute error

The data-generation for figs. 2 and 3 as well as table 4 can be found in the submitted jupyter-notebook `newton_testing.ipynb`. The implementation of algorithms 1 and 2 can be found in the submitted python-file `Newton.py`.

6.3 Neville Interpolation

Task 6.3: Polynomial Interpolation via Neville's Method

1. Write a python script containing the function `Neville(x_data, y_data, x)` which evaluates the interpolant p_n using Neville's method at x, where p_n passes through the data-points specified by $\boldsymbol{y}$ and $\mathcal{X}$.
2. This subtask is an example of *inverse interpolation*. Test your script from Subtask 1 by determining the root of $f(x) = 0$ via Neville's method given the data from table 5

$\mathcal{X}$	4	3.9	3.8	3.7
$\boldsymbol{y}$	−0.06604	−0.02724	0.01282	0.05383

Table 5: Data for task 6.3

Neville Interpolation is particularly useful for evaluating $p(x)$ only for a few values, as stated in the lecture notes. By lemma 6.2, we get the following representation $p_{i,k}$:

$$p_{i,k} = \begin{cases} \frac{(x-x_{i-k})p_{i,k-1}+(x_i-x)p_{i-1,k-1}}{x_i-x_{i-k}} & k=1,\dots,n \quad i=k,\dots,n \\ y_i & k=0 \quad i=k,\dots,n \end{cases} \tag{4}$$

Using eq. (4), we arrive at the following table of helper functions to evaluate $p(x)$:

		$p_{i,0}$	$p_{i,1}$	$p_{i,2}$	$\cdots$	$p_{i,n}$
p_0	x_0	y_0				
p_1	x_1	y_1	$\frac{(x-x_0)y_1+(x_1-x)y_0}{x_1-x_0}$			
p_2	x_2	y_2	$\frac{(x-x_1)y_2+(x_2-x)y_1}{x_2-x_1}$	$\frac{(x-x_0)p_{2,1}+(x_2-x)p_{1,1}}{x_2-x_0}$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	
p_n	x_n	y_n	$\frac{(x-x_{n-1})y_n+(x_n-x)y_{n-1}}{x_n-x_{n-1}}$	$\frac{(x-x_{n-2})p_{n,1}+(x_n-x)p_{n-1,1}}{x_n-x_{n-2}}$	$\cdots$	$\frac{(x-x_0)p_{n,n-1}+(x_n-x)p_{n-1,n-1}}{x_n-x_0}$

Table 6: The various helper functions for Neville Interpolation produced by eq. (4)

Notice that $p_{n,n} = p$. Using table 6, we can find the following iterative method for computing $p(x)$:

Algorithm 3: Iterative Evaluation for Neville Interpolation

```
input: nodes $\mathcal{X}$, $\boldsymbol{y} \in \mathbb{R}^{n+1}$, $x \in \mathbb{R}$
output: $p(x) \in \mathbb{R}$

Neville($\mathcal{X}$, $\boldsymbol{y}$, $x$):
    $\boldsymbol{p} = \boldsymbol{y}$
    for $k = 1, \dots, n$ do
        for $i = 0, \dots, n-k$ do
            $\boldsymbol{p}[i] = \frac{(x-\mathcal{X}[i+k])\boldsymbol{p}[i]+(\mathcal{X}[i]-x)\boldsymbol{p}[i+1]}{\mathcal{X}[i]-\mathcal{X}[i+k]}$
        end
    end
    return $\boldsymbol{p}[0]$
```

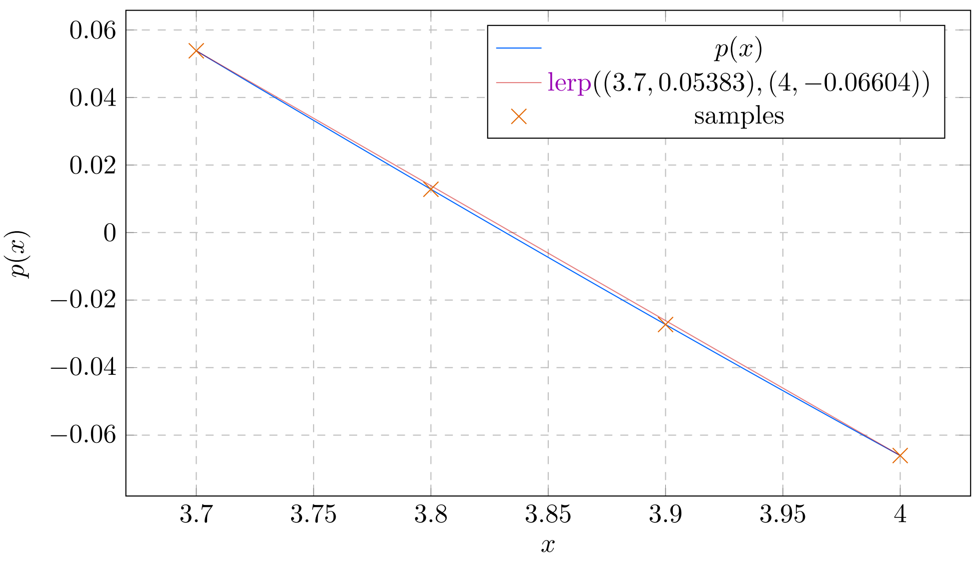

Figure 4: Plot of the interpolant p over $[3.7, 4]$

Subtask 2:

Assuming the dataset $\boldsymbol{y}$ is monotonous around a possible zero of f, like e.g. in table 5, we can linearly interpolate the function f using Neville's method. Let $\widetilde{p}$ be a first-oder polynomial, then by table 6, $\widetilde{p}$ is of the following form

$$\widetilde{p}(x) = \frac{(x - x_{i-1})y_i + (x_i - x)y_{i-1}}{x_i - x_{i-1}} \tag{5}$$

where, without loss of generality $f(x_{i-1}) > 0$ and $f(x_i) < 0$, then by request, $\widetilde{p}(x_{i-1}) = f(x_{i-1}) > 0$ and $\widetilde{p}(x_i) = f(x_i) < 0$. Since $\widetilde{p}$ is continuous, it admits a unique zero x_z in the interval (x_{i-1}, x_i). Solving eq. (5) yields:

$$\begin{aligned}
&\widetilde{p}(x_z) = 0 \Leftrightarrow (x_z - x_{i-1})y_i + (x_i - x_z)y_{i-1} = 0 \\
&\Leftrightarrow (x_z - x_{i-1})y_i = -(x_i - x_z)y_{i-1} \Leftrightarrow (x_z - x_{i-1})\frac{y_i}{y_{i-1}} = x_z - x_i \\
&x_i - x_{i-1}\frac{y_i}{y_{i-1}} = x_z\left(1 - \frac{y_i}{y_{i-1}}\right) \\
&\Rightarrow x_z = \frac{x_i - x_{i-1}\frac{y_i}{y_{i-1}}}{1 - \frac{y_i}{y_{i-1}}} = \frac{\frac{x_i y_{i-1} - x_{i-1} y_i}{y_{i-1}}}{\frac{y_{i-1} - y_i}{y_{i-1}}} = \frac{x_i y_{i-1} - x_{i-1} y_i}{y_{i-1} - y_i}
\end{aligned} \tag{6}$$

If $|x_i - x_{i-1}|$ is sufficiently small, we can reasonably take the first estimate for x_z produced by eq. (6). However, one might want to introduce iteration to further improve the estimate. Given $x_{z,0}$, we can check wether $p(x_{z,0}) > 0$ or $p(x_{z,0}) < 0$ and then resolve $\widetilde{p}_k(x_{z,k}) = 0$ with modified y_i and x_i. This leads us to the following algorithm, based on the bisection method:

Algorithm 4: Iterative Inverse Interpolation using Neville's method

```
input: nodes $\mathcal{X}$, $\boldsymbol{y} \in \mathbb{R}^{n+1}$, $i, k \in \mathbb{N}$
output: $x_{z,k} \in \mathbb{R}$

IInvNeville($\mathcal{X}, \boldsymbol{y}, i, k$):
    $y_l = \boldsymbol{y}[i]$
    $y_r = \boldsymbol{y}[i+1]$
    $x_l = \mathcal{X}[i]$
    $x_r = \mathcal{X}[i+1]$

    for $l = 1, \ldots, k$ do
        $x_z = \frac{x_l y_r - x_r y_l}{y_r - y_l}$
        $y_z =$ Neville($\mathcal{X}, \boldsymbol{y}, x_z$)
        if $y_z > 0$ thenx^
            if $y_l > 0$ then
                $y_l = y_z$
                $x_l = x_z$
            else if $y_r > 0$ then
```

```
                    $y_r = y_z$
                    $x_r = x_z$
                end
            else if $y_z < 0$ then
                if $y_l < 0$ then
                    $y_l = y_z$
                    $x_l = x_z$
                else if $y_r < 0$ then
                    $y_r = y_z$
                    $x_r = x_z$
                end
            end
        end
        return $x_z$
```

Adaptions or extensions of algorithm 4 may lead to *inverse quadratic interpolation*[2] or *Muller's method*, which use Lagrange-Interpolation and Newton's method respectively.

One problem of algorithm 4 is the fact, that for large k, we expect $y_z \to 0$, as we want to find a root of f. Therefore, any „real" machine will produce divisions by zero eventually, and the algorithm will fail. Thus the implementation should provide checks wether or not y_z is approximately zero[3].

Running an implementation of algorithm 4 produces $x_z \approx 3.8317084549112$ after just 6 iterations, where $p(x_z) = -1.9821 \cdot 10^{-16}$, see fig. 5 below.

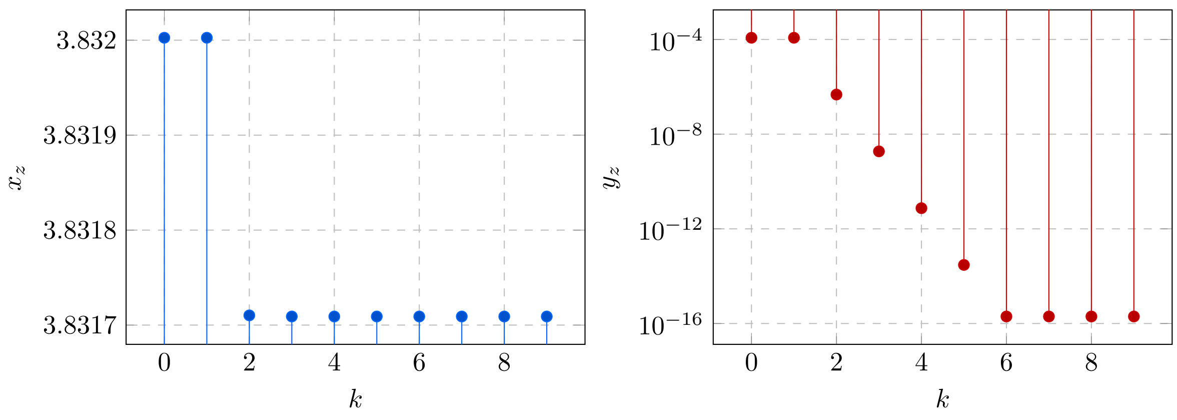

Figure 5: Evolution of x_z (left) and y_z (right) after k iterations

The data-generation for figs. 4 and 5 can be found in the submitted jupyter-notebook `neville_testing.ipynb`. The implementation of algorithms 3 and 4 can be found in the submitted python-file `Neville.py`.

[2]Epp07, see p.182-185.
[3]this is very dependent on the underlying architecture the implementation is used on

6.4 Hermite-Interpolation

Task 6.4: Polynomial Interpolation via Hermite's Method

1. Compute the Hermite Polynomial that agrees with the data from table 7 by hand.

$\mathcal{X}$	1.3	1.6	1.9
$f(x)$	0.620086	0.4554022	0.2818186
$f'(x)$	-0.5220232	-0.5698959	-0.5811571

Table 7: Data for task 6.4

2. Given $n+1$ nodes $\mathcal{X}$ stored in the array `x_data`, and function values $f(x_i)$ and $f'(x_i)$, stored in the arrays `y_data` and `y_prime` respectively, write a python script containing the function `HermiteInterp(x_data, y_data, y_prime, x)` which returns the plot of the Hermite interpolant $H(x)$ and prints it's polynomial value at the given point x. Test your script using the data from table 7 and determine the Hermite polynomial approximation at $x = 1.5$.

Subtask 1:

1.3	0.620086					
		-0.5220232				
1.3	0.620086		-0.08974267			
		-0.548946		0.066365567		
1.6	0.4554022		-0.069833		0.002649887	
		-0.5698959		0.06796555		-0.002746533
1.6	0.4554022		-0.02905367		0.001001967	
		-0.578612		0.06856667		
1.9	0.2818186		-0.00848367			
		-0.5811571				
1.9	0.2818186					

Table 8: Divided differences for Hermite interpolation using data from table 7

Subtask 2: Since we are only concerned with first derivatives of f, we can setup the first two columns of the divided differences for Hermite interpolation very easily:

$$\begin{aligned}
\boldsymbol{x} &= \begin{bmatrix} x_0 & x_0 & x_1 & x_1 & \cdots & x_n & x_n \end{bmatrix}^T \\
\boldsymbol{c}_1 &= \begin{bmatrix} y_0 & y_0 & y_1 & y_1 & \cdots & y_n & y_n \end{bmatrix}^T \\
\boldsymbol{c}_2 &= \begin{bmatrix} f'(x_0) & \frac{y_1-y_0}{x_1-x_0} & f'(x_1) & \cdots & \frac{y_n-y_{n-1}}{x_n-x_{n-1}} & f'(x_n) \end{bmatrix}
\end{aligned}$$

Then we simply apply divided differences using $\boldsymbol{x}$ and $\boldsymbol{c}_2$ to construct our coefficient vector $\boldsymbol{c}$. Given $\boldsymbol{c}$, we can compute $H_n(x)$ using the recurrence relation found in [Cla22, p. 56]:

$$H_{n+1}(x) = H_n(x) + [x_0, \dots, x_n, x]f \cdot \prod_{i=0}^{n}(x - x_i)$$

This leads us to the following algorithm for evaluating $H_n(x)$.

Algorithm 5: Evalution for Hermite Interpolation

```
input:  nodes $\mathcal{X}$, $\boldsymbol{y}, \boldsymbol{y}_p \in \mathbb{R}^{n+1}$, $x \in \mathbb{R}$
output: $p_n(x) \in \mathbb{R}$

Hermiteval($\mathcal{X}$, $\boldsymbol{y}$, $\boldsymbol{y}_p$, $x$):
    $\boldsymbol{c}$ = Hermite($\mathcal{X}$, $\boldsymbol{y}$, $\boldsymbol{y}_p$)
    $H = \mathbf{c}[0]$
    $\mu = x - \boldsymbol{x}[0]$
    for $c_k, x_k$ in $\boldsymbol{c}, \boldsymbol{x}$ do
        $H = H + c_k \cdot \mu$
        $\mu = \mu \cdot (x - x_k)$
    end
    return $p$
```

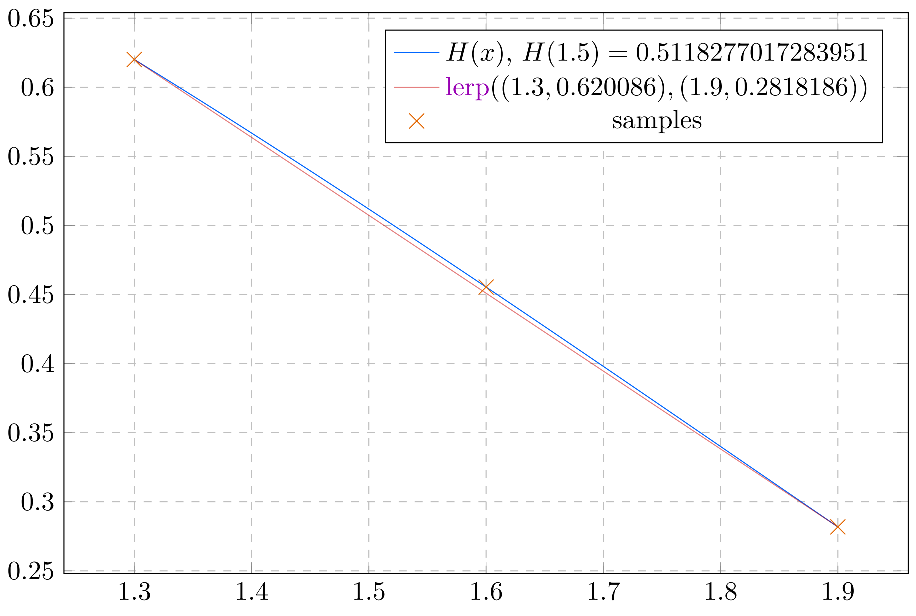

Figure 6: Plot of H from table 7 over $[1.3, 1.9]$

The data-generation for fig. 6 can be found in the submitted jupyter-notebook `hermite_testing.ipynb`. The implementation of divided-differences for hermite interpolation and algorithm 5 can be found in the submitted python-file `Hermite.py`.

6.5 Runge's Phenomenon

Task 6.5: Polynomial Wiggle and Runge's Phenomenon

In this task, we use data-points, that lie on the graph of $f(x) = \frac{1}{1+8x^2}$, gathered in tables 9 to 11.

$\mathcal{X}$	-1	-0.5	0	0.5	1
$\boldsymbol{y}$	$\frac{1}{9}$	$\frac{1}{3}$	1	$\frac{1}{3}$	$\frac{1}{9}$

Table 9: First dataset for task 6.5

$\mathcal{X}$	-1	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1
$\boldsymbol{y}$	$\frac{1}{9}$	$\frac{2}{11}$	$\frac{1}{3}$	$\frac{2}{3}$	1	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{2}{11}$	$\frac{1}{9}$

Table 10: Second dataset for task 6.5

$\mathcal{X}$	-1	-0.8	-0.6	-0.4	-0.2	0	0.2	0.4	0.6	0.8	1
$\boldsymbol{y}$	$\frac{1}{9}$	$\frac{25}{153}$	$\frac{25}{97}$	$\frac{25}{57}$	$\frac{25}{33}$	1	$\frac{25}{33}$	$\frac{25}{57}$	$\frac{25}{97}$	$\frac{25}{153}$	$\frac{1}{9}$

Table 11: Third dataset for task 6.5

1. Write a python script that plots the function $f(x)$, the newton interpolants $p_{N,i}$ and lagrange interpolants $p_{L,i}$ for $i = 4, 8, 10$, using the the three sets of samples points tables 9 to 11.
2. Discuss what happens to the approximation error $p_n(x) - f(x)$ as the degree of the interpolating polynomial increases.

Subtask 1: Since the newton-interpolant $p_{N,i} \in \mathbb{R}_i[x]$ is a unique polynomial with $i+1$ known values (x_k, y_k), the lagrange interpolant $p_{L,i}$ is identical[4] to $p_{N,i}$. Due to this reasoning, the following plots will only contain one method[5].

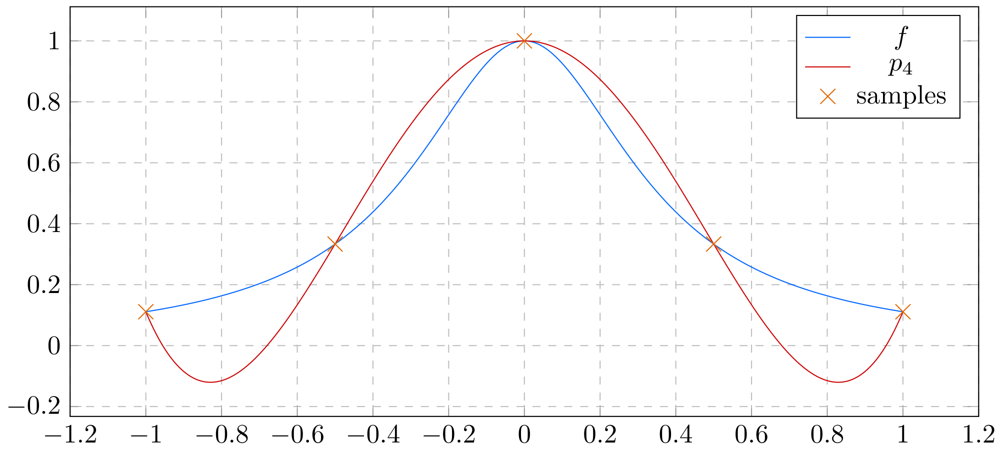

Figure 7: Lagrange-Interpolation with 5 samples

[4]Since any polynomial of degree n is uniquely defined by $n+1$ points, every „classical“ polynomial interpolation method, i.e. Lagrange, Newton and Neville should all produce identical interpolants. If we lessen the constraints for Hermite Interpolation, i.e. use it's degeneration into Newton's method, Hermite-Interpolation will also produce the same interpolant.

[5]namely Lagrange-Interpolation

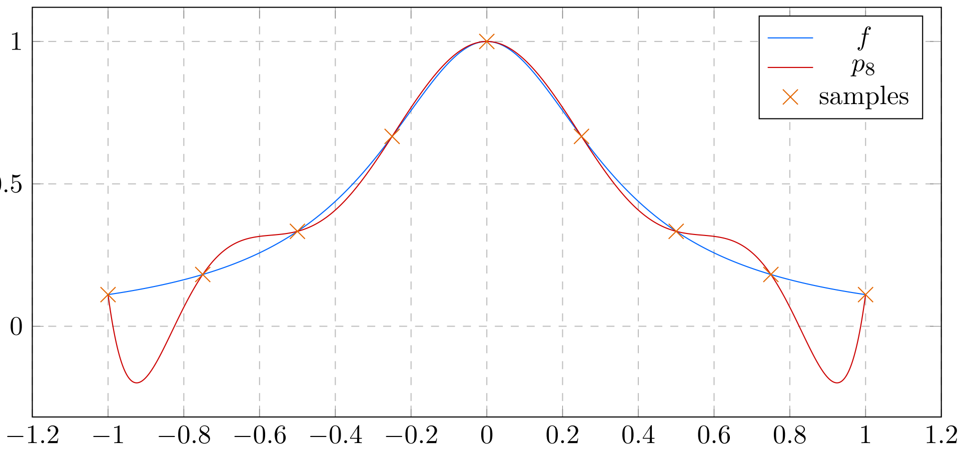

Figure 8: Lagrange-Interpolation with 9 samples

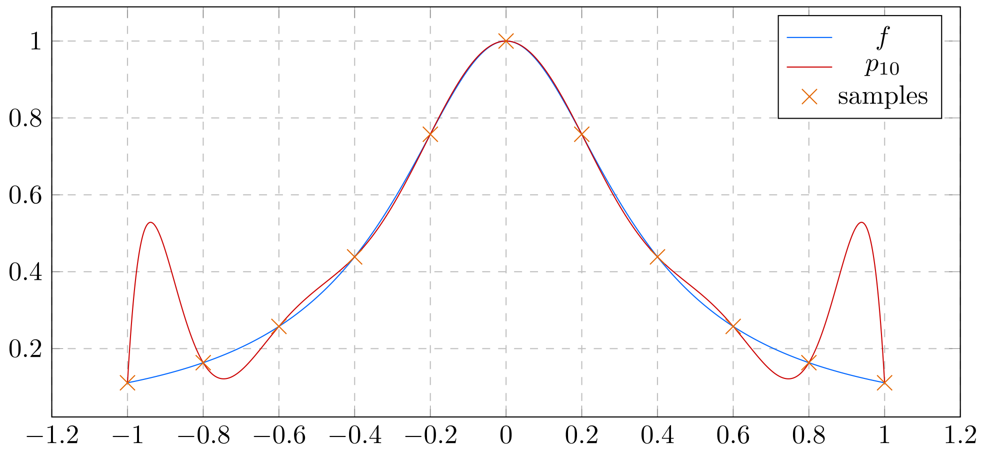

Figure 9: Lagrange-Interpolation with 11 samples

Subtask 2:

As we can clearly see in figs. 7 to 9, the overall approximation of f through p_n becomes better on (x_1, x_{n-1}) with growing n. However as fig. 9 clearly shows, p_n starts to oscillate on the „boundary“ intervals (x_0, x_1) and (x_{n-1}, x_n) quite a lot. This osciallation, or „wiggle“, is labelled ***Runge's phenomenon***. Formally, Runge's phenomenon is the result of the following upper bound of the interpolation error[6]:

$$\|f - p_n\|_\infty \leq \|\omega_n\|_\infty \frac{M}{(n+1)!} \tag{7}$$

where $\|f^{(n+1)}\|_\infty \leq M$ and $\omega_n = \prod_{i=0}^{n}(x - x_i)$. Equation (7) shows us, that even if $\frac{\mathrm{d}^{n+1} f}{\mathrm{d}x^{n+1}}$ is bounded, the interpolation error may still become very large, as $\|\omega_n\|_\infty$ can get very large. As stated in [Cla22, p. 55], determining a basis of $\mathbb{R}_n[x]$, such that $\|\omega_n\|_\infty$ becomes minimal leads us to Chebyshev-Interpolation.

To further show how Runge's phenomenon is relevant, we plot the average and maximum error for $n = 1, \ldots, 99$ in the figure below.

[6]given that $f \in \mathcal{C}^{n+1}([x_0, x_n])$

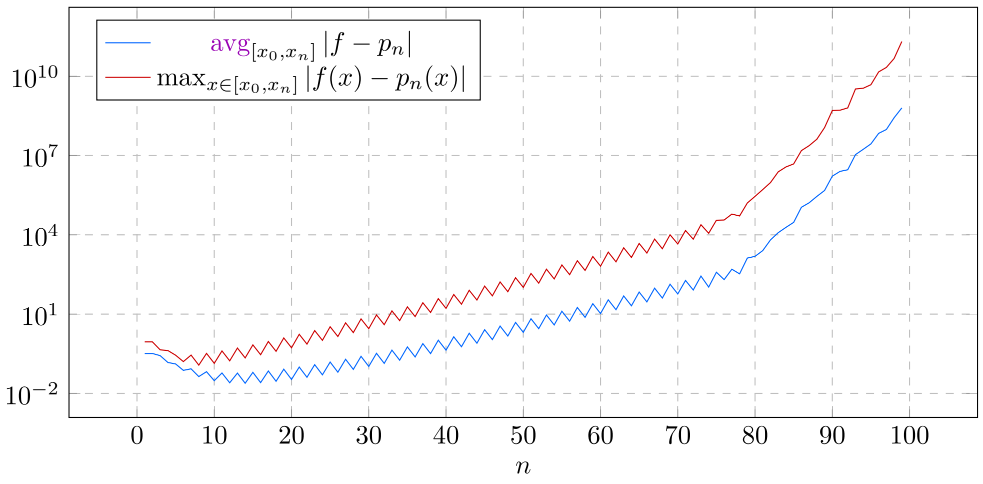

Figure 10: Evolution of interpolation error for rising n, when using Lagrange-interpolation

All data generation for the provided plots is gathered in the submitted files `Runge.py` and `runge_error.py`.

Addendum: Since we are dealing with a known function $f(x)\colon \mathbb{R} \to \mathbb{R}$, which is continuously differentiable, we can gather information to further improve the interpolation with Hermite's method. We will however only take $f'(x)$ into account in the following experiment.

$$f'(x) = -\frac{16x}{(1+8x^2)^2}$$

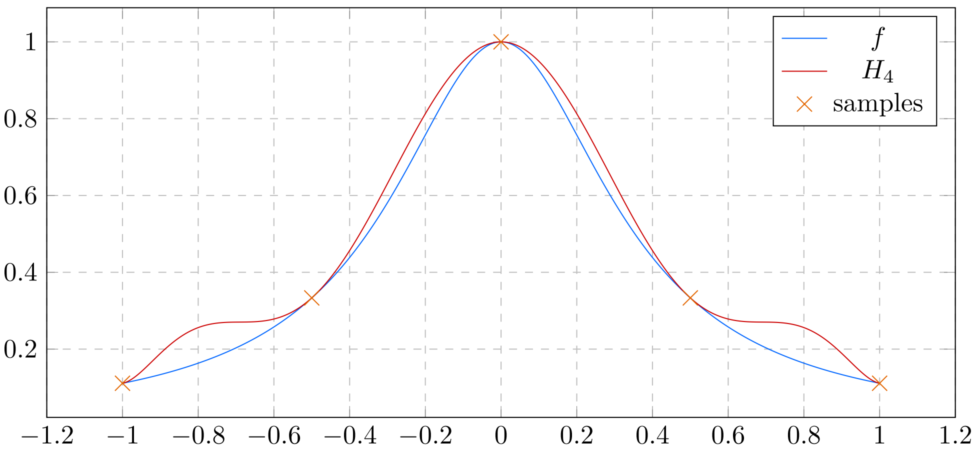

Figure 11: Hermite-Interpolation with 5 samples

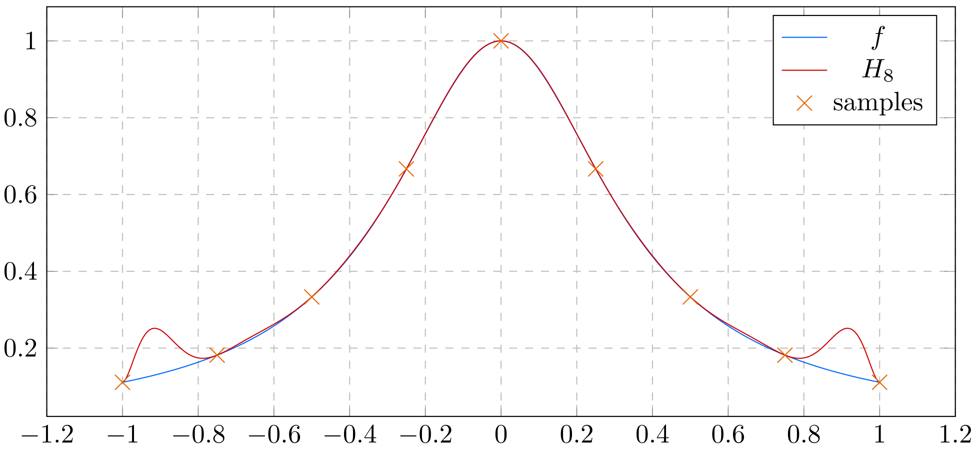

Figure 12: Hermite-Interpolation with 9 samples

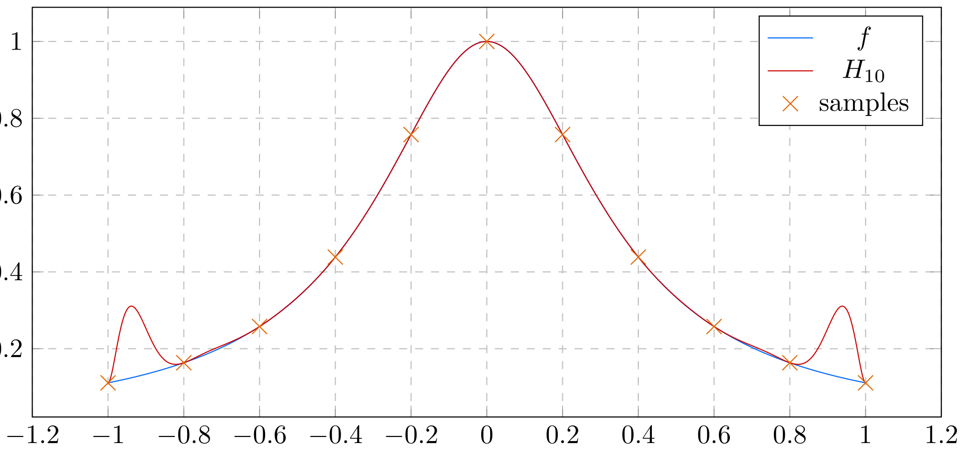

Figure 13: Hermite-Interpolation with 11 samples

Notice that Runge's phenomenon still occurs on the edges of the interval, however compared with fig. 9, the magnitude of the „wiggle“ is comparably smaller.

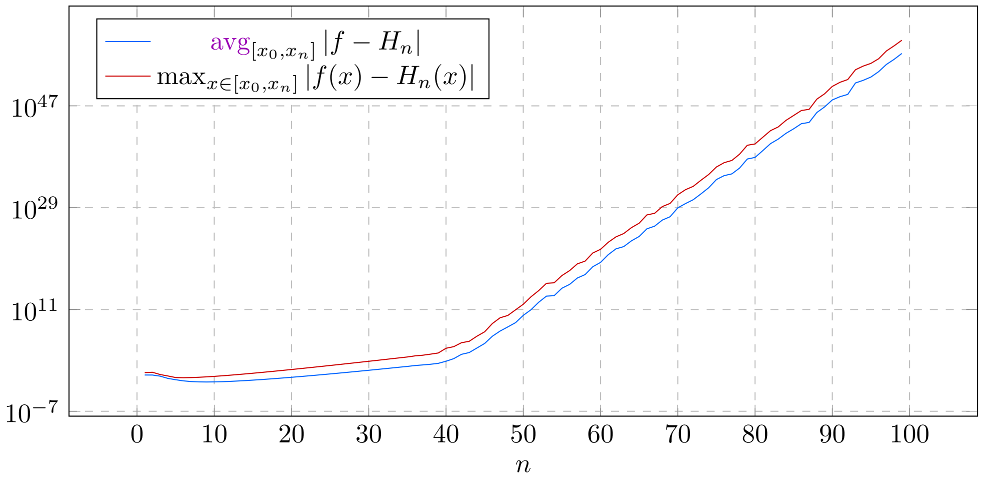

Figure 14: Evolution of interpolation error for rising n, when using Hermite-interpolation

As we can clearly see in fig. 14, even though hermite-interpolation may start out „better“, the interpolation error starts becoming very large for $n \approx 40$ and rises exponentially. This is also the case of Lagrange-interpolation, see fig. 10, but the steep incline only starts at $n \approx 80$.

References

[Cla22] Christian Clason. *Numerische Mathematik 1.* 2022.

[Epp07] James F. Epperson. *An Introduction to Numerical Methods and Analysis, Revised Edition.* John Wiley & Sons, Inc., 2007. ISBN: 978-0-470-04963-1.

[Han11] Norbert Köckler Hans Rudolf Schwarz. *Numerische Mathematik.* Ed. by Barbara Gerlach Ulrike Schmickler-Hirzebruch. 8th ed. Vieweg+Teubner, Springer Fachmedien Wiesbaden GmbH, 2011. ISBN: 978-3-8348-1551-4.

[Jea04] Lloyd Trefethen Jean-Paul Berrut. 'Barycentric Lagrange Interpolation'. In: *Society for Industrial and Applied Mathematics* 46.3 (2004), pp. 501–517.