Chapter 1

Interpolation & Numerical Calculus

Suppose we know some quantity q (e.g. temperature, density, concentration, ...) only at certain given locations

j	0	1	2	3	4	...	n
x_j	0.00	0.51	1.06	1.47	2.01	...	x_n
q_j	0.00	0.22	0.25	0.28	0.30	...	q_n

and that for our application we have to compute an approximation of q between the tabulated locations. Or we may need an approximation of the derivative q' or an approximation to the definite integral $\int_{x_1}^{x_3} q \, \mathrm{d}x$. The data points may originate from some physical measurements or from sampling a complex function.

Hence, we want to find a "reasonable" function $q(x)$ that "suits" the given data $\{(x_j, q_j)\}_{j=0}^{n}$. Once we have found this "reasonable" function $q(x)$, we can simply evaluate this function, its derivative and definite integrals wherever we need to.

If one knows from the application that $q(x)$ has a certain functional form, then one can fit $q(x)$ to the data by determining its parameters in the *least squares sense*, for example. Then $q(x_j) \approx q_j$ as illustrated in the left panel for Figure 1.1. We will treat curve fitting from a numerical[1] perspective in Chapter 4 of the (handwritten) lecture notes.

A special case of curve fitting asks that the function $q(x)$ passes through the data exactly, i.e. $q(x_j) = q_j$. This case is known as *interpolation* and one says that $q(x)$ *interpolates* the data. This case is illustrated in the right panel of Figure 1.1. If one evaluates $q(x)$ outside of the range of given locations, one speaks of *extrapolation*.

In the present chapter we deal with interpolation and its applications to numerical calculus. We will base our interpolating function $q(x)$ on the following *linear combination*

$$q(x) = \sum_{j=0}^{n} c_j \phi_j(x) = c_0 \phi_0(x) + \cdots + c_n \phi_n(x), \tag{1.1}$$

where the c_j's are unknown[2] *coefficients* or *parameters* and the ϕ_j's are predetermined *basis functions*. We assume that the basis functions are *linearly independent*, which

[1] The statistical perspective will be treated in the second part of the course

[2] We have to determine them!

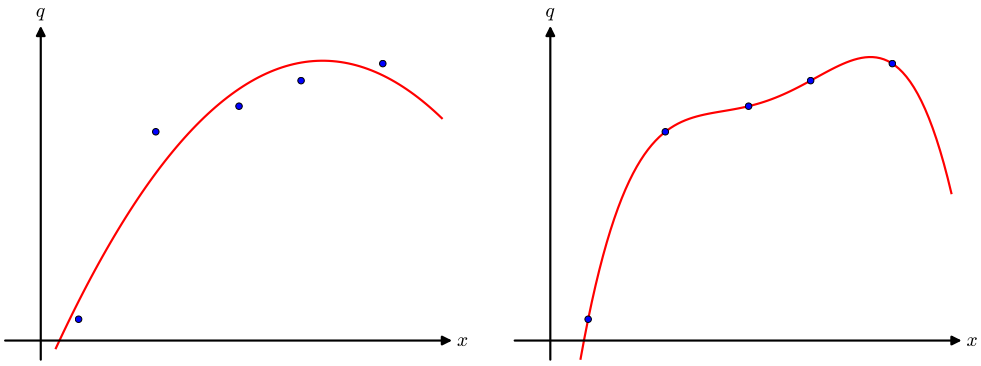

Figure 1.1: Left panel : curve fitting for given data $q(x_i) \approx q_i$. Right panel: interpolation for given data $q(x_i) = q_i$.

means that $q(x) = 0$ for all x is only possible when all coefficients vanish, i.e. $c_j = 0$ for $j = 0, ..., n$. Note that although $q(x)$ is a linear combination of the basis functions $\phi_j(x)$, this does not imply that $\phi_j(x)$, and thereby also $q(x)$, have to be linear functions of x.

In order to determine the coefficients c_j, we ask them to satisfy $n + 1$ *interpolation conditions*

$$q(x_j) = q_j \quad \text{for} \quad j = 0, ..., n. \tag{1.2}$$

This results in a linear system of $n + 1$ in equations in $n + 1$ unknowns, the c_j's, and it can be written in the following compact matrix form

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \cdots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ \vdots \\ q_n \end{bmatrix}. \tag{1.3}$$

Once we have determined the coefficients, we can simply evaluate $q(x)$ at any location we need to. Moreover, derivatives and integrals can easily be computed by differentiating and integrating the basis functions:

$$\begin{aligned} \frac{dq}{dx}(x) &= \sum_{j=0}^{n} c_j \frac{d\phi_j}{dx}(x) \\ \int_a^b q(x)\mathrm{d}x &= \sum_{j=0}^{n} c_j \int_a^b \phi_j(x)\mathrm{d}x. \end{aligned}$$

In this chapter we deal with a particular kind of interpolation known as *polynomial interpolation*. As the name suggests, in this case the basis functions $\phi_j(x)$ are polynomials. In subsection 1.1.1 and 1.1.2 we will see two particular bases for polynomial interpolation. In subsection 1.1.3 we will analyse how good polynomials do at approximating functions by looking at the so-called interpolation error. There we will see that interpolation with high degree polynomials is in general not a good idea and this will lead us to piecewise

interpolation in subsection 1.1.4. The rest of the chapter will then apply (piecewise) polynomial interpolation to compute approximation of derivatives (section 1.2) and (definite) integrals (section 1.3).

Before we proceed, we mention that there are other forms of interpolation. One important example is *trigonometric interpolation* where the basis functions are trigonometric functions $\phi_j(x) = \cos(jx)$, or $\phi_j(x) = \sin(jx)$, or even $\phi_j(x) = e^{jix}$ (where i is the imaginary unit, i.e. $i^2 = -1$). Trigonometric interpolation is extremely useful in signal processing and the description of wave and other periodic phenomena. However, this type of interpolation is beyond the scope of this course and we refer to e.g. [1, Chap. 13] for a good introduction.

> Note that the present chapter may not seem to be the most exciting and enlightening one. However, interpolation is used as a building block in many complex numerical algorithms for differentiation, integration and the solution of differential equations.

1.1 Polynomial interpolation

In this section we consider polynomial interpolation where the basis functions $\phi_j(x)$ are polynomials. We will see two kinds of basis function: the simple monomial basis and the Lagrange polynomials basis.

1.1.1 Monomial interpolation

Given a set of $n+1$ distinct *nodes*, $x_0 < x_1 < ... < x_n$, and corresponding data points, $y_0, y_1, ..., y_n$, we want to find the polynomial of n-th degree

$$p_n(x) = c_0 + c_1 x + c_2 x^2 + ... + c_n x^n \tag{1.4}$$

satisfying the $n+1$ *interpolation conditions*

$$p_n(x_i) = y_i \ , \ i = 0, 1, ..., n. \tag{1.5}$$

Note that the above suits the framework Eq. (1.1) with as basis functions the so-called *monomials* $\phi_j(x) = x^j$.

The set of nodes $x_0 < x_1 < ... < x_n$ is sometimes referred to as a *grid* or *mesh* and by distinct nodes we mean that they are all different, i.e. $x_i \neq x_j$ for $i \neq j$.

The interpolation problem gives rise to a linear system of $n+1$ equations in the $n+1$ unknown coefficients $c_0, c_1, ..., c_n$. The interpolation conditions Eq. (1.5) can be written in matrix form as follows

$$\boldsymbol{X}\boldsymbol{c} = \boldsymbol{y}, \tag{1.6}$$

where $\boldsymbol{c} = [c_0, c_1, ..., c_n]^T$ and $\boldsymbol{y} = [y_0, y_1, ..., y_n]^T$ are vectors containing the polynomial's coefficients and the data points, respectively, and

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \tag{1.7}$$

is known as a *Vandermonde* matrix. From introductory linear algebra courses you may recall that the determinant of a Vandermonde matrix is[3]

$$\det(\boldsymbol{X}) = \prod_{i=0}^{n-1} \left(\prod_{j=i+1}^{n} (x_j - x_i) \right). \tag{1.8}$$

Also from your linear algebra course, you surely know that a linear system is uniquely solvable if its determinant is nonzero. Now, Eq. (1.8) is clearly not zero if (as we assumed from the beginning!) the interpolation nodes are all distinct. Therefore a solution to the interpolation problem *exists* and is *unique*. So much regarding the usual mathematical concerns on existence and uniqueness.

Actually, existence and uniqueness is just a generalization of facts you already know. Through two (distinct!) points there is a unique linear polynomial (line). Through three (distinct!) points there is a unique quadratic polynomial (parabola). And so on...

After so much theory, let's make a couple of simple examples.

Example 1.1. Let's find the interpolating polynomial through the two data points $(x_0, y_0) = (1, 2)$ and $(x_1, y_1) = (3, 5)$. In this case $n = 1$ and we are simply looking for the linear polynomial $p_1(x)$ (straight line!) through both points:

$$\begin{aligned} p_1(x_0) &= c_0 + c_1 x_0 = c_0 + 1c_1 = 2 \\ p_1(x_1) &= c_0 + c_1 x_1 = c_0 + 3c_1 = 5. \end{aligned}$$

This linear system of two equations can be rewritten in matrix form as

$$\begin{bmatrix} 1 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix},$$

which can be solved (very easily!) to yield $c_0 = 1/2$ and $c_1 = 3/2$. The wanted linear polynomial interpolating $p_1(x)$ is

$$p_1(x) = \frac{1}{2} + \frac{3}{2}x.$$

The linear interpolant together with the data points are shown in Figure 1.2. ▲

[3] Don't worry if you don't! Anyway, nobody is going to ask you such a thing at the exam...

Example 1.2. We add another data point to the previous example: find the interpolating polynomial through the three data points $(x_0, y_0) = (1, 2)$, $(x_1, y_1) = (3, 5)$ and $(x_2, y_2) = (4, 4)$. Now $n = 2$ and we look for the quadratic polynomial $p_2(x)$ interpolating the data points. The interpolation conditions read

$$\begin{aligned} p_2(x_0) &= c_0 + c_1 x_0 + c_2 x_0^2 = c_0 + 1c_1 + 1c_2 = 2 \\ p_2(x_1) &= c_0 + c_1 x_1 + c_2 x_1^2 = c_0 + 3c_1 + 9c_2 = 5 \\ p_2(x_2) &= c_0 + c_1 x_2 + c_2 x_2^2 = c_0 + 4c_1 + 16c_2 = 4, \end{aligned}$$

or, equivalently, in matrix form

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 4 \end{bmatrix}.$$

This can (relatively!) easily be solved again to yield $c_0 = -2$, $c_1 = 29/6$ and $c_2 = -5/6$. The wanted quadratic polynomial interpolating $p_2(x)$ is

$$p_2(x) = -2 + \frac{29}{6}x - \frac{5}{6}x^2.$$

The quadratic interpolant together with the data points are shown in Figure 1.2.

Solving the above 3 by 3 linear system by hand can be tedious. In MATLAB©, you could simply do

```
x = [1;3;4]; % nodes
y = [2;5;4]; % data
A = [1,x(1)^1,x(1)^2;    ...
     1,x(2)^1,x(2)^2;    ...
     1,x(3)^1,x(3)^2;]; % system matrix
c = A \ y % solve for the coefficients
```

yielding the same result as expected (up to rounding errors, of course!). ▲

For convenience, MATLAB© provides the function `polyfit` for (monomial basis) polynomial interpolation. So you don't have to set up the linear system and solve it yourself. The function takes as input the nodes `x`, data `y` and the degree of the interpolating polynomial `n`

```
p = polyfit(x,y,n)
```

and returns a vector containing the coefficients of the polynomial $p_n(x)$ as

$$p_n(x) = \texttt{p(1)}x^n + \texttt{p(2)}x^{n-1} + \cdots + \texttt{p(n)}x + \texttt{p(n+1)}.$$

Note that the returned vector contains the coefficients of the polynomial in descending powers of x. An interpolation polynomial computed by `polyfit` can be conveniently evaluated with the function `polyval`.

For instance, the following commands reproduce Example 1.2 with `polyfit` and `polyval`:

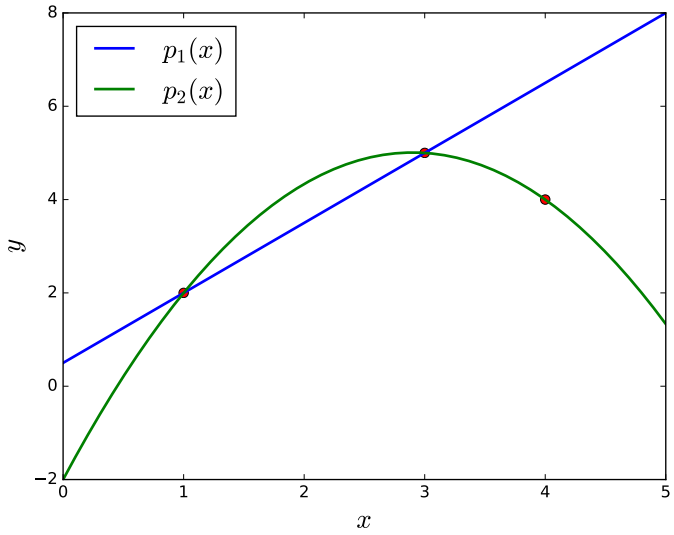

Figure 1.2: Example of linear and quadratic polynomial interpolation.

```
x = [1;3;4];              % nodes
y = [2;5;4];              % data
p = polyfit(x,y,2);       % get quadratic interpolation polynomial
                          % coefficients
xx  = linspace(0.,5.);    % row vector of 100 equally spaced point between 0
                          % and 5
pxx = polyval(p,xx);      % evaluate the quadratic interpolation polynomial
                          % at xx
plot(xx,pxx)              % plot it!
```

For further information please consult the MATLAB© documentation.

1.1.2 Lagrange interpolation

In the monomial basis, the expansion coefficients c_j do not relate to the data points y_j in a straightforward manner. Indeed, the c_j are the solution of a linear system of equations (the interpolation conditions), and are therefore only implicitly given. For what follows, it will be convenient for us to have a basis in which the coefficients can be directly inferred from the data points, i.e. $c_j = y_j$, giving

$$p_n(x) = \sum_{j=0}^{n} c_j \phi_j(x) = \sum_{j=0}^{n} y_j \phi_j(x).$$

Such a basis is provided by the so-called *Lagrange polynomials* $L_j^n(x)$ defined by

$$L_j^n(x) = \prod_{\substack{i=0 \\ i \neq j}}^{n} \frac{x - x_i}{x_j - x_i} \quad \text{for} \quad j = 0, 1, ..., n. \tag{1.9}$$

Then, the interpolation polynomial is given by

$$p_n(x) = \sum_{j=0}^{n} y_j L_j^n(x). \tag{1.10}$$

The Lagrange polynomials $L_j^n(x)$ have all degree n and therefore $p_n(x)$ in Eq. (1.10) is indeed a polynomial of degree n (because it's a linear combination of polynomials of degree n). Moreover, the Lagrange polynomials satisfy

$$L_j^n(x_k) = \delta_{jk} = \begin{cases} 1, & \text{if } j = k, \\ 0, & \text{if } j \neq k. \end{cases} \tag{1.11}$$

Try to convince yourself of that[4]! With the latter property, it is easy to see that $p_n(x)$ satisfies the interpolation conditions:

$$p_n(x_i) = \sum_{j=0}^{n} y_j L_j^n(x_i) = 0 + \cdots + 0 + y_i L_i^n(x_i) + 0 + \cdots + 0 = y_i.$$

Let's digest Lagrange interpolation with the following quick example.

Example 1.3. We want to find the interpolating polynomial through the same data points as in the previous Example 1.2 but this time with Lagrange's interpolation formula. We have $(x_0, y_0) = (1, 2)$, $(x_1, y_1) = (3, 5)$ and $(x_2, y_2) = (4, 4)$. Here $n = 2$ and we look for the quadratic polynomial $p_2(x)$ that goes through these data points. The Lagrange polynomials can be computed from formula Eq. (1.9):

$$\begin{aligned}
L_0^2(x) &= \frac{(x - x_1)}{(x_0 - x_1)} \frac{(x - x_2)}{(x_0 - x_2)} = \frac{(x - 3)}{-2} \frac{(x - 4)}{-3} = \frac{1}{6}(x^2 - 7x + 12) \\
L_1^2(x) &= \frac{(x - x_0)}{(x_1 - x_0)} \frac{(x - x_2)}{(x_1 - x_2)} = \frac{(x - 1)}{2} \frac{(x - 4)}{-1} = -\frac{1}{2}(x^2 - 5x + 4) \\
L_2^2(x) &= \frac{(x - x_0)}{(x_2 - x_0)} \frac{(x - x_1)}{(x_2 - x_1)} = \frac{(x - 1)}{3} \frac{(x - 3)}{1} = \frac{1}{3}(x^2 - 4x + 3)
\end{aligned}$$

Note that here it should be trivial to see that Eq. (1.11) is valid. By plugging this into Eq. (1.10) we get

$$\begin{aligned}
p_2(x) &= y_0 L_0^2(x) + y_1 L_1^2(x) + y_2 L_2^2(x) \\
&= 2\frac{1}{6}(x^2 - 7x + 12) - 5\frac{1}{2}(x^2 - 5x + 4) + 4\frac{1}{3}(x^2 - 4x + 3) \\
&= -2 + \frac{29}{6}x - \frac{5}{6}x^2.
\end{aligned}$$

The quadratic interpolating polynomial $p_2(x)$ and the data points are shown in the left panel of Figure 1.3. The right panel of the same figure shows the three Lagrange polynomials. ▲

[4] Sometimes certain things are not so obvious for general j, k and n... In Example 1.3 a concrete example is given for $n = 2$.

You probably have noted that the interpolating polynomials in Example 1.2 and Example 1.3 are the same. In the former example we have used the monomial basis while in the latter the Lagrange one. Is it surprising that both give the same interpolating polynomial? Of course not, since through three points there exist a unique quadratic polynomial.

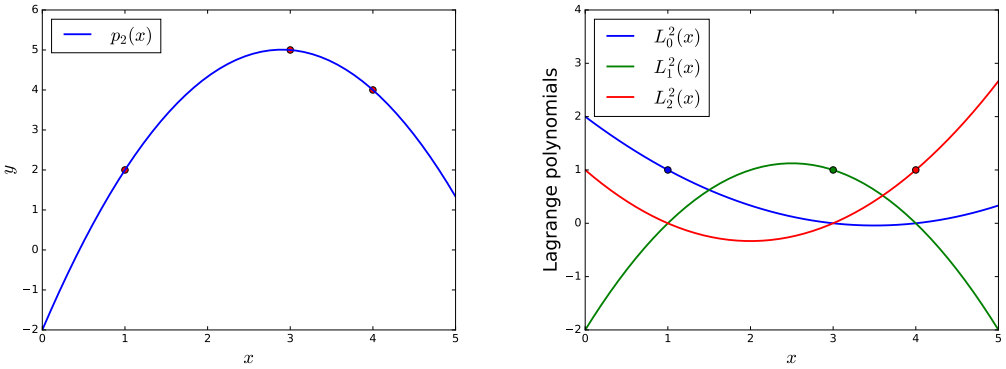

Figure 1.3: Example of Lagrange quadratic polynomial interpolation. The left panel shows the polynomial together with the underlying data points and the right panel shows the individual Lagrange polynomials. Moreover, the right panel gives also a visual confirmation that $L_j^n(x_k) = \delta_{jk}$.

1.1.3 The interpolation error

So far we have considered polynomial interpolation at some given nodes x_j and data points y_j. We have not stipulated any further how or from where the y_j's have been obtained. In the following, we shall assume that they are generated by some function $f(x)$, i.e. $y_j = f(x_j)$ for $j = 0, 1, ..., n$. Moreover, we will assume that this function possesses a certain number of derivatives.

So let's assume we are given a function $f(x)$ from some (real) interval $[a, b]$ to $\mathbb{R}$, in short $f : [a, b] \to \mathbb{R}$, and let it be $(n + 1)$-times differentiable. Let $p_n(x)$ be the n-th degree polynomial that interpolates the function at the $n+1$ (distinct) nodes $x_j \in [a, b]$, i.e. $p_n(x_j) = f(x_j)$ for $j = 0, 1, ..., n$.

We then define the *interpolation error function* $e_n(x)$ of our interpolation polynomial $p_n(x)$ as

$$e_n(x) = f(x) - p_n(x). \tag{1.12}$$

So the interpolation error function measures the difference between the function $f(x)$ and the interpolating polynomial $p_n(x)$. In other words, it measures how well $p_n(x)$ approximates $f(x)$. Then one can show[5] that for each $x \in [a, b]$ there is a point $\xi =$

[5] It's pretty easy: few applications of Rolle's theorem and you have it... You can find the proof in any numerical analysis textbooks, e.g. [3, Chapter 3] or [1, Chapter 10].

$\xi(x) \in [a, b]$ such that

$$e_n(x) = f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n} (x - x_j). \tag{1.13}$$

Note that ξ depends on x, hence the notation $\xi(x)$. In particular, we have the following bound on the *maximum interpolation error*

$$\max_{x\in[a,b]} |e_n(x)| = \max_{x\in[a,b]} |f(x) - p_n(x)| \leq \frac{\max\limits_{x\in[a,b]} |f^{(n+1)}(x)|}{(n+1)!} \max_{x\in[a,b]} \left| \prod_{j=0}^{n} (x - x_j) \right|. \tag{1.14}$$

Let us rewrite the above in a slightly less cluttered manner as

$$\|e_n\|_\infty \leq \frac{\|f^{(n+1)}\|_\infty}{(n+1)!} \left\| \prod_{j=0}^{n} (x - x_j) \right\|_\infty . \tag{1.15}$$

In case you are not familiar with the notation, $\|\cdot\|_\infty$ denotes a function norm known as *maximum* (or *sup* or L_∞) norm and it is defined by

$$\|g\|_\infty = \max_{x\in[a,b]} |g(x)| \tag{1.16}$$

for some function $g : [a, b] \to \mathbb{R}$.

From Eq. (1.14), or equivalently Eq. (1.15), we see that the interpolation error depends on two things:

(i) the function itself, or more precisely, on the $(n + 1)$-th derivative of the function f

(ii) on the distribution of the nodes x_j

It turns out that both can play an important role.

Point (i) above is often referred to as a smoothness requirement on the function $f(x)$. Generally, a *smooth function* is a function that has continuous derivatives up to sufficiently high order over some domain, that is the function is sufficiently many times continuously differentiable. One says that a function is smooth over some restricted domain such as an interval, e.g. $[a, b]$. What exactly "sufficiently many times" means depends on the context. But in the present context, it means at least $n + 1$!

From the above discussion one may get the impression that by cranking up the number of nodes, and therewith the degree of the interpolating polynomial, one can obtain an arbitrary good approximation to any function f by interpolation. This is in general not the case and the following famous example due to Runge[6] (1901) demonstrates this.

Example 1.4 (Runge)**.** Let's try to approximate the function

$$f(x) = \frac{1}{1 + 25x^2}$$

[6] Runge, Carl David Tolmé (1856 - 1927)

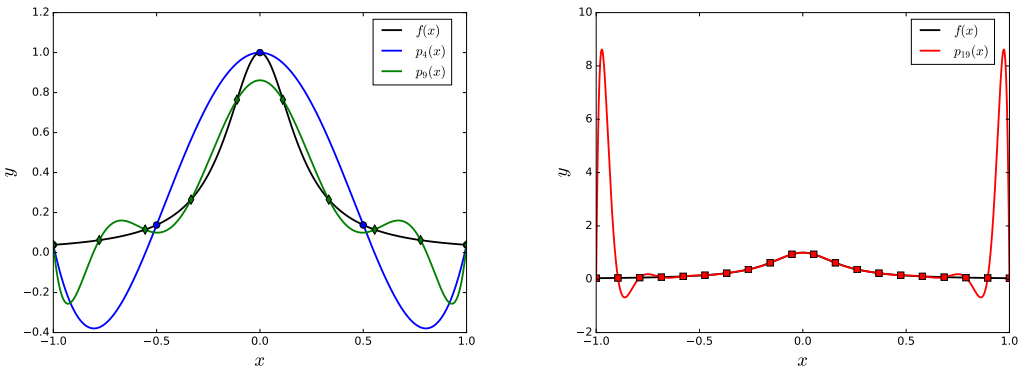

Figure 1.4: Interpolation of Runge function with equidistant nodes. In the left panel $n = 4, 9$ and $n = 19$ in the right.

in the interval $I = [-1, 1]$ by polynomial interpolation of increasingly high degree n.

As probably the most obvious choice, we choose the following set of nodes

$$x_j = -1 + \frac{2}{n} j \quad \text{for} \quad j = 0, 1, ..., n$$

equidistantly distributed in the interval $[-1, 1]$ and the corresponding data points $y_j = f(x_j)$.

The result of the interpolation is shown in Figure 1.4 for 5, 10 and 20 nodes. As you can see, the interpolation with 20 nodes has large errors at the interval ends. As a matter of fact, the interpolation error grows with the degree of the interpolating polynomial n.

However, choosing a different set of nodes can improve the situation. The so-called Chebyshev[7] nodes are given by

$$x_j = \cos\left(\frac{2j+1}{2(n+1)}\pi\right) \quad \text{for} \quad j = 0, 1, ..., n.$$

Note that these points are not distributed equidistantly. They tend to concentrate more near the interval ends.

The result of the interpolation with the Chebyshev nodes is shown in Figure 1.5. As you can see, the result is much better. The reason behind this remarkable improvement is due to the special node distribution. Actually, one can show that the Chebyshev nodes minimize the error contribution due to the node distribution, i.e. the part involving the product in Eq. (1.13). ▲

Let us make the results from the previous Example 1.4 more quantitative. Instead of measuring the error in the so-called *"eyeball" norm*[8], we measure the interpolation error

[7] Chebyshev, Pafnuty Lvovich Chebyshev (1821 - 1894). A variety of transliterations of his name are used in the literature: Chebychev, Chebysheff, Chebyshov (English); or Tchebychev, Tchebycheff (French); or Tschebyschev, Tschebyschef, Tschebyscheff (German).

[8] To be taken literally!

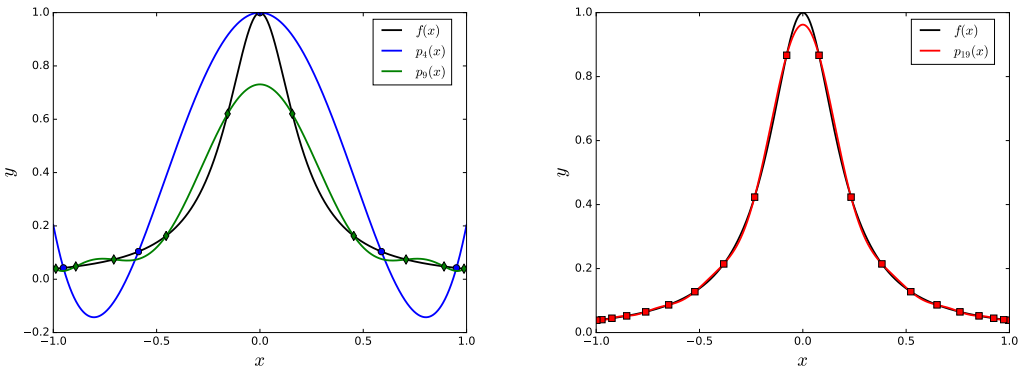

Figure 1.5: Interpolation of Runge function with Chebyshev nodes. In the left panel $n = 4, 9$ and $n = 19$ in the right.

$\|e_n\|_\infty$. Practically, one estimates the latter by sampling the error function $e_n(x)$ at a sufficiently large number of points ξ_k, say N, and simply pick out the maximum

$$\|e_n\|_\infty \approx \max_{k=1,...,N} |e_n(\xi_k)| = \max_{k=1,...,N} |f(\xi_k) - p_n(\xi_k)|. \tag{1.17}$$

For the sampling points ξ_k, we choose N evenly spaced points given by

$$\xi_k = a + k\frac{b-a}{N+1} \text{ for } k = 1, ..., N.$$

Be careful to distinguish between lower case n (the degree of the interpolating polynomial $p_n(x)$) and upper case N (the number of sampling points to compute $\|e_n\|_\infty$)!

In Figure 1.6 is shown the maximum interpolation error $\|e_n\|_\infty$ for polynomial interpolation as a function of the polynomial degree n. In the left panel of the figure is shown the case with equidistant nodes while in the right panel the one with Chebyshev nodes. As you observe, the left panel confirms quantitatively what we have already observed in Figure 1.4: the error grows with increasing polynomial degree n. Likewise, the right panel confirms our observations from Figure 1.5 that the error gets smaller with increasing polynomial order.

What does "a sufficiently large number of points" mentioned above mean? Well this is difficult to answer in general, but often one chooses N to be two or three orders of magnitude larger than the number of interpolation nodes.

The message to take home from Example 1.4, is that the numerical approximation of functions by polynomial interpolation with equidistant nodes is generally only recommendable for small polynomial degrees n, i.e. up to $n = 4, 5$ maybe. The reason for this is that at high degrees strong oscillations can appear and the interpolation becomes useless.

Before we end this section, let us derive another estimate for Eq. (1.13). By realizing that $(x - x_j) \leq (b - a)$ for all nodes x_j $(j = 0, ..., n)$ and all $x \in [a, b]$, we obtain a (very) crude upper bound for the interpolation error as

$$\|e_n\|_\infty \leq (b-a)^{n+1} \frac{\left\|f^{(n+1)}\right\|_\infty}{(n+1)!}. \tag{1.18}$$

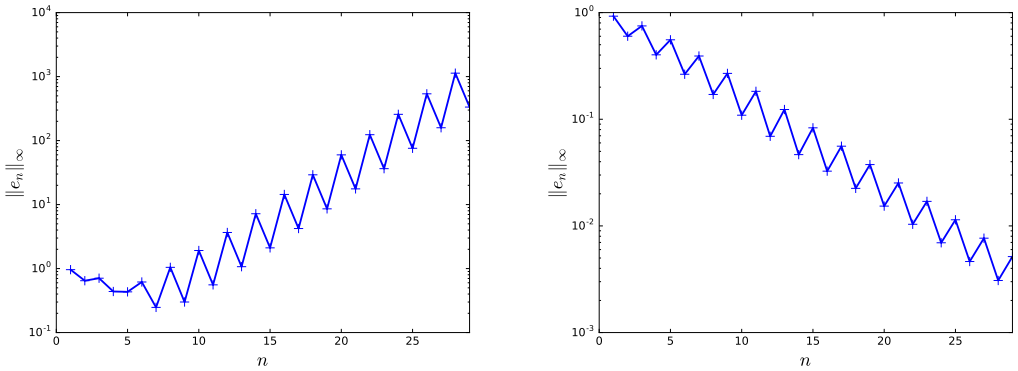

Figure 1.6: Maximum interpolation error $\|e_n\|_\infty$ for Runge's example as a function of the polynomial degree n: left panel for equidistant nodes and right panel for Chebyshev nodes.

1.1.4 Piecewise polynomial interpolation

As we have just seen in Example 1.4, interpolation with high degree polynomials, that is many nodes and data points, is in general not recommendable. To circumvent these problems, one resorts to so-called *piecewise polynomial interpolation*. As the name suggests, the idea is to divide an interval of interest $I = [a, b]$ into a number, say M, of smaller subintervals. This is achieved by partitioning I as

$$a = \tau_0 < \tau_1 < ... < \tau_{M-1} < \tau_M = b, \tag{1.19}$$

where the τ_0, ..., τ_M are referred to as the partition's *break points*. On each subinterval $I_j = [\tau_{j-1}, \tau_j]$, $j = 1, ..., M$, one builds a relatively[9] low order interpolation polynomial $s_j(x)$. These polynomial pieces $s_j(x)$ are then patched together to form an interpolating curve $v(x)$ over the whole interval I as

$$v(x) = s_j(x) \quad \text{for} \quad x \in I_j \quad \text{and} \quad j = 1, ..., M. \tag{1.20}$$

Piecewise polynomial interpolation is very general as we could play on

(i) the degree n of the polynomial pieces $s_j(x)$ on each subinterval,

(ii) the number of subintervals M and their sizes (which need not to be uniform),

(iii) the way we patch together the polynomial pieces to form $v(x)$.

Concerning (iii), one could for example impose the condition that $v(x)$ is continuous over the interval I, i.e. this imposes that neighboring polynomial pieces match at touching break points. Or one could require that $v(x)$ is once (or twice, ...) times continuously differentiable over I. Otherwise, one could also ask that $v(x)$ is discontinuous (at the break points) but that it features some non-oscillatory properties.

[9] Relatively low with respect to M. Why? Well because the most extreme choice $M = 1$ and $n \gg M$ would lead to the situation we try to circumvent here!

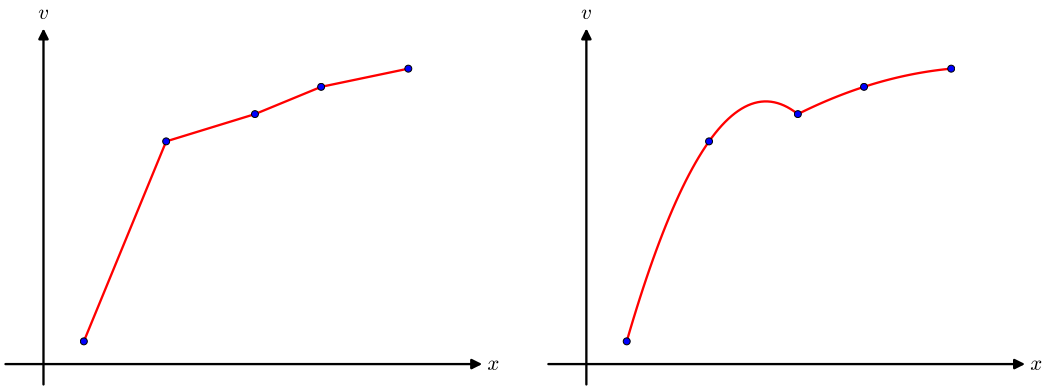

Figure 1.7: Piecewise linear (left panel) and quadratic (right panel) of some data.

However, in the following we will only discuss the cases where $v(x)$ is continuous. The other cases lead to Hermite, spline and piecewise discontinuous interpolation which are beyond the scope of this introductory course. We refer to [1, 2] for further information and references.

The simplest possible continuous piecewise interpolation is constructed by patching together piecewise linear polynomials. It simply consists of connecting data points by straight lines and it is illustrated in the left panel of Figure 1.7. Due to its appearance, piecewise linear interpolation is also known as *broken line interpolation*.

The next possible choice would be continuous piecewise quadratic interpolation. This is illustrated in the right panel of Figure 1.7.

Please note that without any further information on what the data represents and on how it has been obtained, it is absolutely not clear which choice of interpolation is better in Figure 1.7. So if you lack this crucial information, the safest possible choice is probably the simplest: broken line interpolation.

Next we have a look at the piecewise interpolation error. For this to make sense, we assume that we approximate a function $f(x)$ by interpolation on a certain interval $I = [a, b]$. Moreover, we also assume that $f(x)$ is sufficiently smooth, that is the function is sufficiently many times continuously differentiable over I. We define the *piecewise interpolation error* as

$$e_{\text{pw}}(x) = f(x) - v(x). \tag{1.21}$$

Then by applying Eq. (1.18) on each subinterval, one easily gets the following estimate for the piecewise interpolation error

$$|e_{\text{pw}}(x)| = |f(x) - v(x)| \leq \frac{h^{n+1}}{(n+1)!} \left\| f^{(n+1)} \right\|_\infty \quad \text{for} \quad x \in I, \tag{1.22}$$

where

$$h = \max_{1 \leq j \leq M} (\tau_i - \tau_{i-1}) \tag{1.23}$$

is the maximum subinterval length. Because $|e_{\text{pw}}(x)|$ is bound from above by $\|e_{\text{pw}}\|_\infty$, we can also simply write

$$\|e_{\text{pw}}\|_\infty \leq \frac{h^{n+1}}{(n+1)!} \left\|f^{(n+1)}\right\|_\infty . \tag{1.24}$$

Estimates of the sort Eq. (1.22) and (1.24) arise very frequently in numerical analysis. Actually so often, that one introduces a convenient notation known as *big-O* or *big-oh notation*. For some error function $e(h)$ depending on some discretization step size or length h one writes

$$e = O(h^r) \tag{1.25}$$

if there are two positive constants r and C, independent of h, such that

$$|e| \leq Ch^r \tag{1.26}$$

for all $h > 0$ small enough. Moreover, the exponent r is known as the *order of accuracy*. Such expressions tell you how uch the error decreases if you decrease h.

Using the just introduced notation, we have for the piecewise polynomial interpolation error

$$|e_{\text{pw}}(x)| = O(h^{n+1}) \quad \text{for} \quad x \in I \tag{1.27}$$

and likewise

$$\|e_{\text{pw}}(x)\|_\infty = O(h^{n+1}). \tag{1.28}$$

Here, the constant C in Eq. (1.26) is simply $C = \left\|f^{(n+1)}\right\|_\infty /(n+1)!$ and the order of accuracy of piecewise polynomial interpolation is $r = n+1$, which are both independent of h. Hence, C depends on the function one wants to approximate and r on the degree of the polynomial pieces patched together.

After so many definitions, let's make an example.

Example 1.5. Let's try again to approximate the function

$$f(x) = \frac{1}{1+25x^2}$$

in the interval $I = [-1, 1]$, but this time by piecewise polynomial interpolation.

The first thing to do then is to partition the interval of interest I. We shall use an equidistant placement of the break points as

$$\tau_j = -1 + jh \quad \text{for} \quad j = 0, ..., M,$$

where $h = 2/M$. Therefore all subintervals $I_j = [\tau_i, \tau_{j+1}]$ have the same size h.

First, we choose the degree n of the polynomial pieces we wish to use on each subinterval I_j. First we use $n = 1$, that is piecewise linear polynomials on each subinterval. The construction of the piecewise linear polynomials is simply achieved by connecting by straight lines the data at the break points. This is illustrated in the left panel of Figure 1.8 for $M = 5, 11$ subintervals.

Second, let's use piecewise quadratic polynomials to approximate Runge's function. In order to define a quadratic polynomial on each subinterval I_j, we need one more node and data point on I_j and we choose simply the point in the middle of the break points $(\tau_{i-1} + \tau_i)/2$. So we can patch together piecewise quadratic polynomials and this is illustrated in the right panel of Figure 1.8 for for $M = 5, 11$ subintervals.

Next let's have a look at the piecewise interpolation error as the number of subintervals M increases, that is we want to investigate how well we are approximating Runge's function with piecewise polynomial interpolation and how this error depends on the number of subintervals. In Figure 1.9 is shown in a $\log - \log$ scale the maximum error as function of the number of subintervals M. The blue and red solid lines correspond to the error for picewise linear and quadratic polynomial interpolation, respectively. Firs we observe that the error get's small as the number of subintervals increases, which was already visible in the "eye-ball" norm from Figure 1.8.

However, we observe that the error decreases faster for the piecewise quadratic interpolation than for the piecewise linear one. From Eq. (1.28) we know that the error scales with the suninterval size/length as $O(h^2)$ for piecewise linear and $O(h^3)$ for piecewise qadratic interpolation. The number of subintervals M determines the size of the subintervals $h = 2/M$. Hence, we can express how the interpolation error in terms of the number of subintervals M as $O(M^{-2})$ for piecewise linear and $O(M^{-3})$ for piecewise quadratic polynomials.

In a $\log - \log$ we can thus directly read of the order of accuracy[10] and we indeed confirm from Figure 1.9 that piecewise linear and quadratic polynomial interpolation have order of accuracy two and three, respectively. Can you guess what is the polynomial degree underlying the piecewise interpolation for which the error is the green solid line in Figure 1.9?[11]

As a final comment, note that M has to be bigger that roughly 10 in order to observe the from theory predicted orders of accuracy. Indeed, the error estimates are only valid in an asymptotic sense and this is what is meant by "for h small enough", or equivalently, "for M large enough". ▲

1.2 Numerical differentiation

The interpolation polynomials to discretely ("in table form") given functions is the foundation to compute their derivatives approximately:

$$\begin{aligned} f'(x) &\approx p_n'(x) \\ f''(x) &\approx p_n''(x) \\ &\dots \end{aligned}$$

[10] Recall that $O(h^r)$ stands for $e \leq Ch^r$. Then by taking the log on each side gives: $\log(e) = r\log(h) + \log(C)$. Therefore the order of accuracy r is the slope of the error in a $\log - \log$ plot.

[11] The green line has a slope of -6 and hence the degree of the polynomial pieces is $n = 5$. (The slope is negative because the error is plotted as a function of $M = 2/h$.)

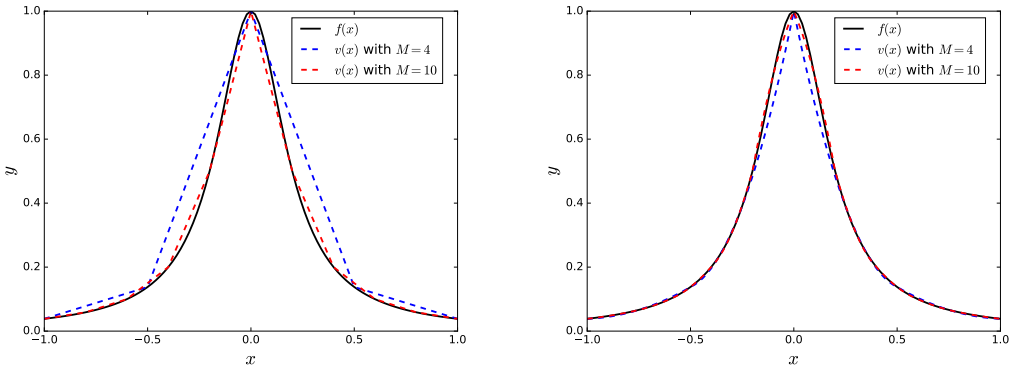

Figure 1.8: Example of Runge with piecewise linear (left panel) and quadratic (right panel) interpolation.

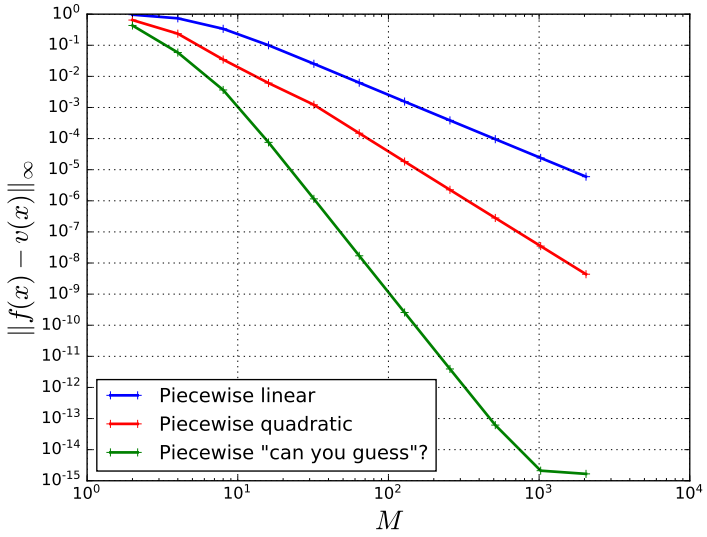

Figure 1.9: Piecewise interpolation error for Runge's example as a function of the number of equally sized subintervals M. The blue/red solid line represent the piecewise linear/quadratic interpolation errors, that is $n = 1/n = 2$. Can you guess the degree of the polynomial pieces n for the green curve?

This procedure can be used to derive the numerical differentiation a.k.a. *finite difference* formulas below. Indeed, these formulas can also be used to approximately compute derivatives of analytically computable functions.

We now consider a continuously differentiable (up to adequate order) function $f : I = [a, b] \mapsto \mathbb{R}$ and want to compute approximate derivatives numerically. Let $p_n(x)$ be the polynomial interpolating the function at the $n + 1$ nodes $x_0 < x_1 < ... < x_n$. Then we can write a possible approximation of the k-th derivative $f^{(k)}(x)$ simply as

$$\frac{d^k f(x)}{dx^k}(x) \approx \frac{\mathrm{d}^k p_n(x)}{\mathrm{d}x^k} = \sum_{j=0}^{n} \frac{\mathrm{d}^k L_j^n(x)}{\mathrm{d}x^k} f(x_j). \tag{1.29}$$

Note that to compute reasonably the k-th derivative, your interpolation polynomial has to be based at least on $k+1$ nodes. Otherwise, your approximation will give zero all the time!

Usually, these approximations are used for equidistantly spaced nodes

$$x_j = x_0 + jh \ , \ j \in \mathbb{Z}, \tag{1.30}$$

where h is the constant spacing between the nodes. To approximate the first derivative of the function $f(x)$ let's take 2 nodes. The interpolating polynomial using one node to the right of x_0, i.e. x_1, is then simply

$$p_1(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1) \tag{1.31}$$

Analogously, the interpolating polynomial using one node to the left of x_0, i.e. x_{-1}, is

$$p_1(x) = \frac{x - x_{-1}}{x_0 - x_{-1}} f(x_0) + \frac{x - x_0}{x_{-1} - x_0} f(x_{-1}) \tag{1.32}$$

By differentiating these expressions we obtain the following approximations

$$\begin{aligned} f'(x) \approx \quad & \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \quad \frac{f(x_0 + h) - f(x_0)}{h} \ , \qquad x \in [x_0, x_1] \\ f'(x) \approx \quad & \frac{f(x_0) - f(x_{-1})}{x_0 - x_{-1}} = \quad \frac{f(x_0) - f(x_0 - h)}{h} \ , \quad x \in [x_{-1}, x_0] \end{aligned} \tag{1.33}$$

The first/second equation is called *forward/backward finite difference* for obvious reasons.

To approximate the second derivative we take the 3 nodes x_{-1}, x_0 and x_1. Then we have

$$\begin{aligned} p_2(x) = & \frac{x - x_0}{x_{-1} - x_0} \frac{x - x_1}{x_{-1} - x_1} f(x_{-1}) \\ & + \frac{x - x_{-1}}{x_0 - x_{-1}} \frac{x - x_1}{x_0 - x_1} f(x_0) \\ & + \frac{x - x_{-1}}{x_1 - x_{-1}} \frac{x - x_0}{x_1 - x_0} f(x_1) \end{aligned} \tag{1.34}$$

and obtain (more or less[12]) immediately

$$\begin{aligned} f''(x) \approx p_2''(x) &= \frac{f(x_1) - 2f(x_0) + f(x_{-1})}{(x_0 - x_{-1})(x1 - x_0)} \\ &= \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} \ , \ x \in [x_{-1}, x_1]. \end{aligned} \tag{1.35}$$

This expression is known as *second-order central/centered finite difference.*

We can also use Eq. (1.34) to obtain an approximation for the first derivative

$$\begin{aligned} f'(x) \approx p_2'(x) = &\left(\frac{1}{x_{-1} - x_1} \frac{x - x_0}{x_{-1} - x_0} + \frac{1}{x_{-1} - x_0} \frac{x - x_1}{x_{-1} - x_1} \right) f(x_{-1}) \\ &+ \left(\frac{1}{x_0 - x_1} \frac{x - x_{-1}}{x_0 - x_{-1}} + \frac{1}{x_0 - x_{-1}} \frac{x - x_1}{x_0 - x_1} \right) f(x_0) \\ &+ \left(\frac{1}{x_1 - x_0} \frac{x - x_{-1}}{x_1 - x_{-1}} + \frac{1}{x_1 - x_{-1}} \frac{x - x_0}{x_1 - x_0} \right) f(x_1) \ , \ x \in [x_{-1}, x_1] \end{aligned} \tag{1.36}$$

Note that the approximation of $f'(x)$ now explicitly depends on x! In practice, the latter expression is almost exclusively evaluated at x_0

$$f'(x_0) \approx p_2'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h}, \tag{1.37}$$

which is called the *first-order central/centered finite difference.*

Time for an example.

Example 1.6. Let us try out the expressions for the first derivative at a concrete example with $f(x) = \sin(x)$ at $x = 1.2$. Obviously the exact answer is $f'(x = 1.2) = \cos(1.2)$. In figure 1.10 we show in a loglog plot the absolute (computational) error of the forward and centered finite difference formulas for various sizes of h. We first observe that the error does get smaller with decreasing h, i.e. the formulas converge. However, the convergence "speed" seems to be different for the two formulas: the error for the forward difference formula has a slope of one, while the centered difference has a slope of two. Moreover, we observe that the error starts to grow after a certain value for h has been reached: this is the manifestation of the fact that computers have a finite precision! ▲

How can we explain the difference between these two formulas for the first derivative? For this, we go back to the good old Taylor expansion formula:

$$f(x + h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + ... + \frac{f^{(k)}(x)}{k!}h^k + \frac{f^{(k+1)}(\xi)}{(k+1)!}h^{k+1} \tag{1.38}$$

for some $\xi \in]x, x + h[$.

Let's plug that into the forward finite difference formula

$$\frac{f(x + h) - f(x)}{h} = \frac{f(x) + f'(x)h + \frac{f''(\xi)}{2}h^2 - f(x)}{h} = f'(x) + \frac{h}{2}f''(\xi) = f'(x) + O(h).$$

[12] Don't worry, nobody is going to ask you to do this!

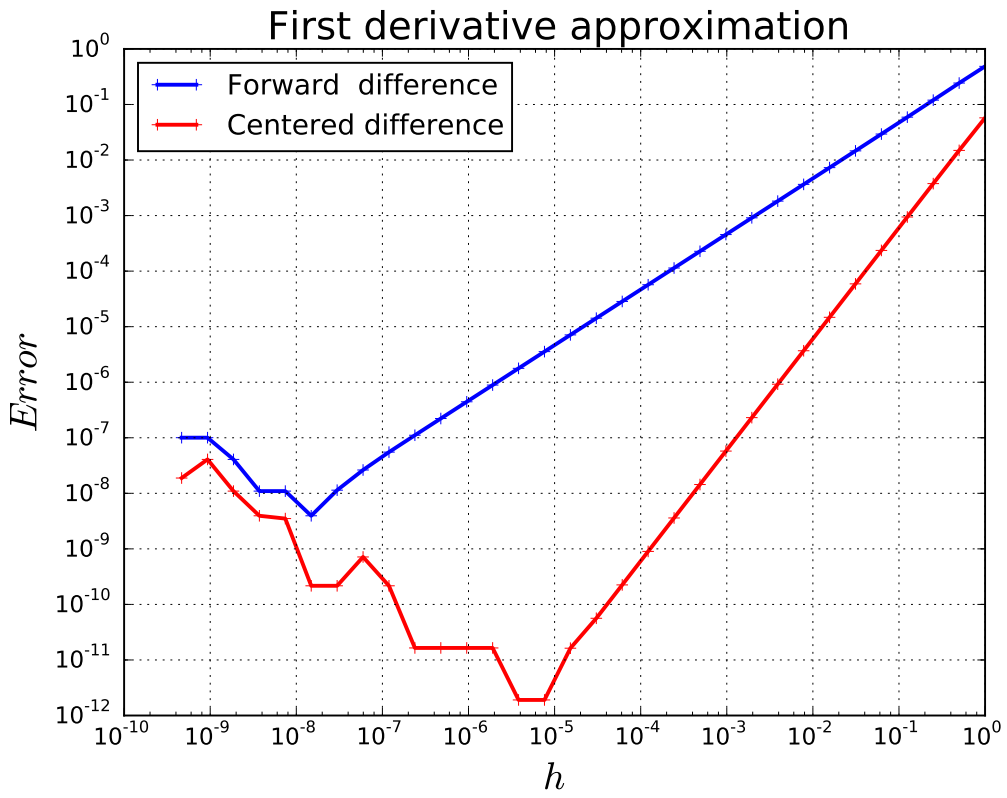

Figure 1.10: Finite difference (forward and centered) approximation of the first derivative of $f(x) = \sin(x)$ at $x = 1.2$. Note the slopes: they indicate that the forward and centered difference have order of convergence $p = 1$ and $p = 2$, respectively. Also note that these formulas become inaccurate for too small h because of the finite precision of computers!

This shows that the error of the forward difference formula is proportional to h: if we make h ten times smaller, the error becomes ten times smaller. That's exactly what we observe from figure 1.10. Note that we silently assumed, that the function $f(x)$ is at least twice continuously differentiable.

In exactly the same manner, one can show that the central differences,

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \\ f''(x) &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2), \end{aligned}$$

are second-order accurate approximation to the first and second derivatives, respectively. Note that showing the latter are popular exam questions.

We will come across these finite difference formulas several times during the course of the lecture...

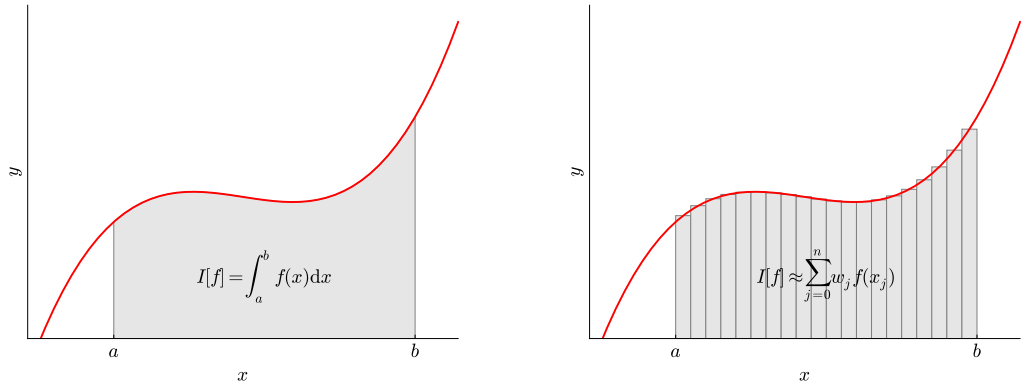

Figure 1.11: Approximation of the integral.

1.3 Numerical integration

Numerical integration, also commonly called *quadrature*, has as goal to compute approximations $\tilde{I}$ of definite integrals

$$I[f] = \int_a^b f(x)\mathrm{d}x \tag{1.39}$$

To achieve this, one divides the interval $[a, b]$ in a number of subintervals and approximates the definite integral on them. This is illustrated in figure 1.11.

A finite calculation rule to compute $\tilde{I}$ is called *quadrature rule*

$$\tilde{I}[f] = \sum_{j=0}^{n} w_j f(x_j), \tag{1.40}$$

where the x_j, $j = 0, 1, ..., n$, are called the *quadrature nodes* and the w_j the *quadrature weights*.

The idea is then to approximate the function $f(x)$ with polynomial interpolation $f(x) \approx p_n(x)$. Quadratures rules $Q_n[f]$ based on this Ansatz are called *Newton-Cotes* quadratures. The interpolating polynomial can then easily be integrated analytically

$$\begin{aligned} I[f] = \int_a^b f(x)\mathrm{d}x \approx \int_a^b p_n(x)\mathrm{d}x &= \int_a^b \sum_{j=0}^{n} L_j^n(x) f(x_j)\mathrm{d}x \\ &= \sum_{j=0}^{n} \int_a^b L_j^n(x)\mathrm{d}x f(x_j) \\ &= (b-a)\sum_{j=0}^{n} w_j f(x_j) = Q_n[f], \end{aligned} \tag{1.41}$$

where we defined the quadrature weights as

$$w_j = \frac{1}{b-a}\int_a^b L_j^n(x)\mathrm{d}x \ , \ j = 0, 1, ..., n. \tag{1.42}$$

The quadrature weights can easily calculated once one settles for a certain nodes distribution. A popular choice is again the equidistant distribution

$$x_j = x_0 + j\frac{b-a}{n}\ ,\ j = 0, 1, ..., n. \tag{1.43}$$

Furthermore, it is advantageous to apply the following variable substitution

$$x = a + (b-a)t \longrightarrow \mathrm{d}x = (b-a)\mathrm{d}t. \tag{1.44}$$

Then $x = a, b$ for $t = 0, 1$, respectively. Plugging this into (1.42), we obtain

$$w_j = \frac{1}{b-a}\int_a^b L_j^n(x)\mathrm{d}x = \frac{1}{b-a}\int_0^1 L_j^n(t)(b-a)\mathrm{d}t = \int_0^1 L_j^n(t)\mathrm{d}t\ ,\ j = 0, 1, ..., n. \tag{1.45}$$

Since the w_j are independent of $f(x)$ and the integration interval $[a, b]$, they can easily be computed once and tabulated for posterity.

For interpolating polynomials with degree $n = 0, 1, 2$ one obtains the following quadrature nodes and weights

Degree n	Nodes x_j	Weights w_j
0	$x_0 = \frac{a+b}{2}$	$w_0 = 1$
1	$x_0 = a$	$w_0 = \frac{1}{2}$
	$x_1 = b$	$w_1 = \frac{1}{2}$
2	$x_0 = a$	$w_0 = \frac{1}{6}$
	$x_1 = \frac{a+b}{2}$	$w_1 = \frac{2}{3}$
	$x_2 = b$	$w_2 = \frac{1}{6}$

Don't worry, you will never be asked to know them by heart. However, you should know how to derive/"draw" them (midpoint and trapezoidal rule are obtained by elementary area considerations!). Directly plugged into the quadrature rule:

$$Q_0[f] = (b-a)f\left(\frac{a+b}{2}\right) \tag{1.46}$$

$$Q_1[f] = (b-a)\left(\frac{1}{2}f(a) + \frac{1}{2}f(b)\right) \tag{1.47}$$

$$Q_2[f] = (b-a)\left(\frac{1}{6}f(a) + \frac{2}{3}f\left(\frac{a+b}{2}\right) + \frac{1}{6}f(b)\right) \tag{1.48}$$

These quadrature rules are visualized in figure 1.12. Q_0 and Q_1 get their names from their appearance: Q_0 *midpoint rule* and Q_1 *trapezoidal rule*. Q_2 is called *Simpson rule* after Thomas Simpson[13].

As a measure of quality of a quadrature, one introduces the notion of degree of exactness. The *degree of exactness* q is defined as the maximum polynomial degree a quadrature rule can integrate exactly.

[13] Simpson, Thomas (1710 - 1761)

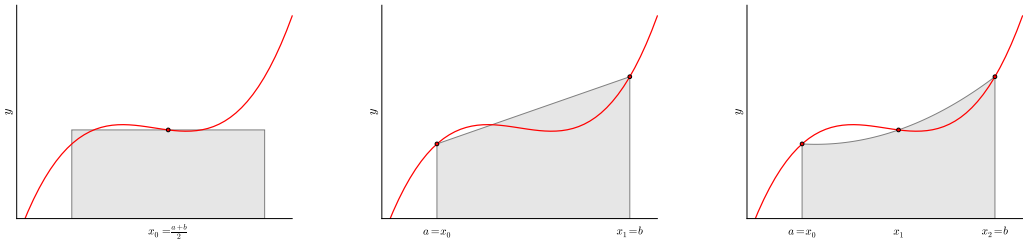

Figure 1.12: Midpoint (left), trapezoidal (middle) and Simpson (right) quadrature rule.

It is rather obvious, that the degree of exactness is at least equal to the degree of the interpolating polynomial which the quadrature rule is based on. So for the midpoint, trapezoidal and Simpson rule this would be $q = 0, 1, 2$, respectively. However, it turns out that quadrature rules based on even degree[14] interpolating polynomials have one degree of exactness more than one would expect. For instance, the midpoint rule has degree of exactness $q = 1$ rather than 0. Likewise, the Simpson rule has $q = 3$ instead of 2. To see that, consider the integration interval $I = [-1, 1]$ and an even degree quadrature rule, i.e. Q_{2r} for $r = 0, 1, ...$ Such a quadrature rule is based on $2r + 1$ nodes, symmetrically distributed in the integration interval I. Moreover, the weights are also symmetric. Then one sees by symmetry that

$$Q_{2r}[x^{2r+1}] = 0 = I[x^{2r+1}] = 0.$$

Therefore, the quadrature rule has degree of exactness $q = 2r + 1$ instead of $2r$. Try it out for the midpoint and the Simpson rule. The midpoint rule integrates linear function exactly. The Simpson rule integrates cubic polynomials exactly.

What can one say for the approximation error of quadrature? Well, one can use some polynomial interpolation error formula from subsection 1.1.3 and try to derive some error estimates, but that would be very tedious. So we just state the following: One says, that a quadrature rule is s-th order accurate if

$$|Q_n[f] - I[f]| = O((b - a)^s) \tag{1.49}$$

for sufficiently smooth functions f and it is possible to show that $s = q + 1$, that is the order of acccuracy of a quadrature is it's degree of exactness plus one.

As we have seen for interpolation, by increasing the degree n of polynomial interpolation one might probably get into troubles. Likewise, it makes no real sense to build and use Newton-Cotes quadrature rules with too big n. Instead, one resorts to applying the quadrature to N equally sized subintervals $I_j = [x_{j-1}, x_j]$ (for $j = 1, ..., N$), where $x_j = a+hj$ (for $j = 0, ..., N$) and $h = (b-a)/N$. This gives rise to the so-called *composite Newton-Cotes quadrature rules*. This is illustrated in figure 1.13.

[14] Hence, odd number of nodes.

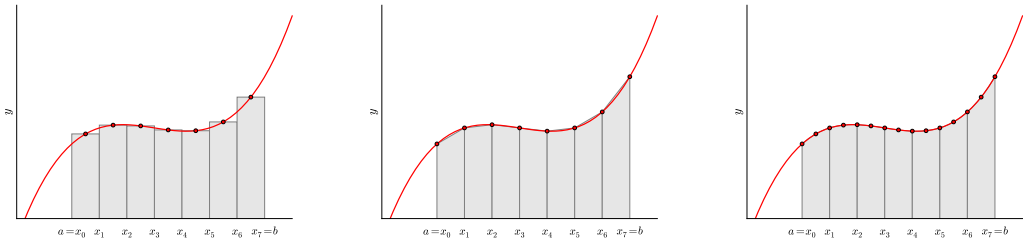

Figure 1.13: Composite midpoint (left), trapezoidal (middle) and Simpson (right) quadrature rule.

The popular composite midpoint, trapezoidal and Simpson rule are given by

$$Q_0^N[f] = h\sum_{k=1}^{N} f\left(\frac{x_{k-1}+x_k}{2}\right) \tag{1.50}$$

$$Q_1^N[f] = h\left(\frac{1}{2}f(a) + \sum_{k=1}^{N-1} f(x_k) + \frac{1}{2}f(b)\right) \tag{1.51}$$

$$Q_2^N[f] = \frac{h}{6}\left(f(a) + 2\sum_{k=1}^{N-1} f(x_k) + 4\sum_{k=1}^{N} f\left(\frac{x_{k-1}+x_k}{2}\right) + f(b)\right), \tag{1.52}$$

where

$$x_k = a + hk\ ,\ k = 0, 1, ..., N \text{ and } h = \frac{b-a}{N}. \tag{1.53}$$

One can show, that these composite quadrature rules are of the following orders of accuracy

$$\left|Q_0^N[f] - I[f]\right| = O(h^2) \tag{1.54}$$

$$\left|Q_1^N[f] - I[f]\right| = O(h^2) \tag{1.55}$$

$$\left|Q_2^N[f] - I[f]\right| = O(h^4). \tag{1.56}$$

So far, we have constructed quadrature rules by finding an interpolation polynomial through $n+1$ equidistantly spaced nodes and then computed the quadrature weights by exactly integrating this polynomial. This gives obviously a degree of exactness of at least $q = n$. However, one can show that $n+1$ quadrature nodes and weights can be chosen in such a manner that the degree of exactness is maximized to $q = 2n+1$. This leads to so-called *Gaussian quadrature*:

$$G_n[f] = \sum_{j=1}^{n} w_j f(x_j) \approx \int_{-1}^{1} f(x)\mathrm{d}x. \tag{1.57}$$

The values for the nodes and weights can be found in tables. Here we list just the first few

n	Nodes x_j	Weights w_j	$q = 2n-1$
1	$x_1 = 0$	$w_1 = 2$	1
2	$x_1 = -\sqrt{1/3}$	$w_1 = 1$	3
	$x_2 = +\sqrt{1/3}$	$w_2 = 1$	
3	$x_1 = -\sqrt{3/5}$	$w_1 = \frac{5}{9}$	5
	$x_2 = 0$	$w_2 = \frac{8}{9}$	
	$x_3 = +\sqrt{3/5}$	$w_3 = \frac{5}{9}$	

The first difference to the quadrature formulas we have seen so far is that the Gaussian quadrature nodes and weights are usually given for a specific integration interval, namely $I = [-1, 1]$. A second minor difference: the sum starts at $j = 1$ and not at 0.

In order to use Gauss quadratures on general intervals $I = [a, b]$, one has to make a small variable substitution

$$t = \frac{b-a}{2}x + \frac{a+b}{2} \longrightarrow \mathrm{d}t = \frac{b-a}{2}\mathrm{d}x \tag{1.58}$$

giving

$$I[f] = \int_a^b f(t)\mathrm{d}t = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right)\mathrm{d}x. \tag{1.59}$$

Then we can apply Gauss quadrature as

$$G_n[f] = \frac{b-a}{2}\sum_{j=1}^{n} w_j f\left(\frac{b-a}{2}x_j + \frac{a+b}{2}\right). \tag{1.60}$$

Again, while nobody will ask you to know by heart the nodes and weights of Gaussian quadrature, a popular exam question is to apply it to a general interval $I = [a, b]$.

Gauss quadrature rules are derived through so-called orthogonal polynomials. Orthogonality for polynoms is defined like for usual "vectors" with the innver product (a.k.a. scalar product) replaced accordingly

$$\langle f, g\rangle = \int_a^b f(x)g(x)\mathrm{d}x,$$

where f and g are two real functions (e.g. polynomials). Anyway, we don't need to know that for the exam...

Let us make an example.

Example 1.7. We compute

$$\int_0^1 \frac{1}{1+x}\mathrm{d}x = \log(2)$$

with the composite trapezoidal Q_1^N, Simpson Q_2^N and $n = 3$ Gauss quadrature G_3^N. In figure 1.14 is shown the error as a function of the number of subintervals N. We see that the composite trapezoidal and Simpson rule have the expected orders of accuracy. Moreover, we see that the Gaussian quadrature G_3^N has order of accuracy 6. Also note that the error does not go below $\sim 10^{-16}$, which is again a consequence of te finite precision of computers. The figure was produced with the MATLAB© scripts `func.m`, `trapez.m`, `simpson.m`, `gauss3.m` and `error.m`. ▲

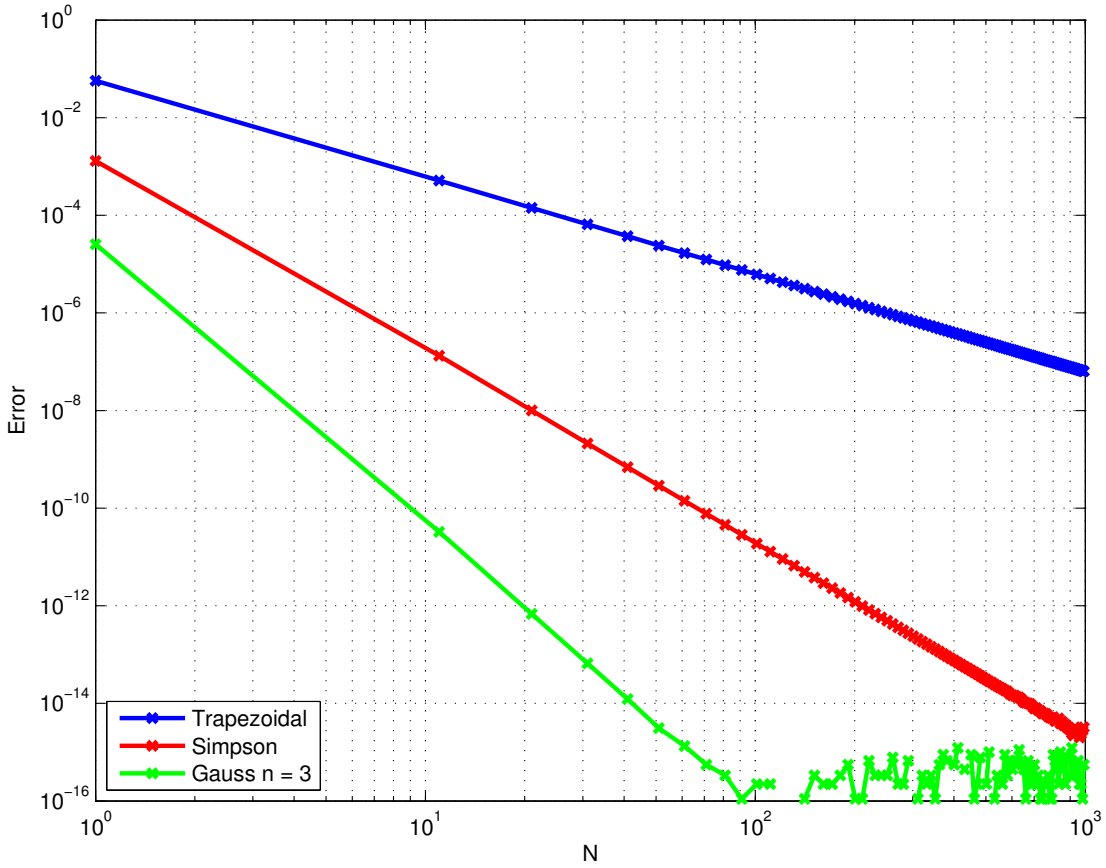

Figure 1.14: Error for composite trapezoidal, Simpson and Gauss $n = 3$.

1.4 Review questions

Here a few review questions[15] for the present chapter:

(a) What's interpolation good for?

(b) Why do the interpolation nodes have to be different?

(c) What's the difference between the monomial and Lagrange basis for polynomial interpolation?

(d) Does an interpolating polynomial depend on the choice of the basis?

(e) On what does the error depend when approximating a function by polynomial interpolation?

(f) What is meant by a "(sufficiently) smooth function"?

(g) What is piecewise polynomial interpolation?

[15] FAQs at exams...

(h) What shortcomings of polynomial interpolation are improved upon by piecewise polynomial interpolation?

(i) Define the big-O notation.

(j) Define order of accuracy.

(k) What is the difference between manual or symbolic differentiation and numerical differentiation?

(l) Why is numerical differentiation useful?

(m) What is a finite difference approximation?

(n) How are formulas for numerical differentiation derived?

(o) How to determine the order of accuracy of a finite difference formula?

(p) What problems can appear with numerical differentiation? (Remember that computers have a finite precision...)

(q) What is a quadrature rule?

(r) How did we derive quadrature rules?

(s) What is the degree of exactness of a quadrature rule?

(t) What is the order of accuracy of a quadrature rule?

1.5 Bibliography

[1] Uri M. Ascher and Chen Greif. *A First Course in Numerical Methods*. Society for Industrial & Applied Mathematics (SIAM), jun 2011. doi: 10.1137/9780898719987. URL http://dx.doi.org/10.1137/9780898719987.

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in FORTRAN. The art of scientific computing*. 1992.

[3] Hans Rudolf Schwarz and Norbert Köckler. Numerische mathematik. 2011. doi: 10.1007/978-3-8348-8166-3. URL http://dx.doi.org/10.1007/978-3-8348-8166-3.