2. Interpolation

Closing the Gaps of Discretization . . .

2.1. Preliminary Remark

The Approximation Problem

- **Problem**: Approximate a (fully or partially unknown or just complicated) function $f(x)$ with a function $p(x)$ that is simple to construct and to deal with (evaluate, differentiate, integrate). $p(x)$ is called **approximant**.
- The difference $f - p$ should not exceed a certain tolerance either pointwise or in an averaging norm (e.g. least squares sum or Euclidean norm) or it should fulfill certain design criteria.
- **Examples**:
  – In computers, functions such as the exponential function or the trigonometric functions are approximated up to computing accuracy by extremely fast converging series consisting of suitable polynomials and are computed this way. The familiar Taylor series are generally not suited for this kind of approximation, because they don't converge fast enough, i.e. too many terms are needed to reach the required accuracy. The construction of more appropriate series (or rather families of functions, often orthogonal polynomials) is a problem of **approximation theory**.

Chebyshev polynomials:

$$T_0(x) = 1$$
$$T_1(x) = x$$
$$T_{k+1}(x) = 2x \cdot T_k(x) - T_{k-1}(x)$$

$$a_0 = \frac{1}{\pi}\int_{-1}^{1}\frac{f(x)}{\sqrt{1-x^2}}\mathrm{d}x, \qquad a_k = \frac{2}{\pi}\int_{-1}^{1}\frac{f(x)T_k(x)}{\sqrt{1-x^2}}\mathrm{d}x, \qquad k = 1, 2, \ldots$$

Chebyshev approximation:

$$f(x) := \sin(\pi x) \approx \sum_{k=0}^{n} a_k \cdot T_k(x)$$

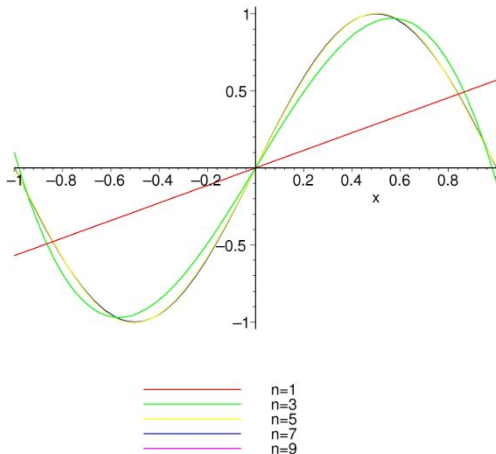
Table:

$n = 0:$	0
$n = 1, 2:$	$0.592306858x$
$n = 3, 4:$	$2.569980700x - 2.667666686x^3$
$n = 5, 6:$	$3.091392515x - 4.753313948x^3 + 1.668517810x^5$
$n = 7, 8:$	$3.139276852x - 5.136388641x^3 + 2.434667195x^5 - 0.4377996486x^7$
$n = 9:$	$3.141527662x - 5.166399408x^3 + 2.5427059x^5 - 0.5818513224x^7 + 0.6402296612x^9$

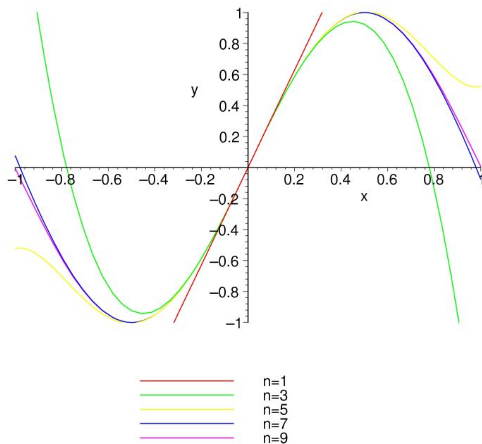
Taylor expansion:

$$f(x) \approx \pi x - \frac{\pi^3 x^3}{6} + \frac{\pi^5 x^5}{120} - \frac{\pi^7 x^7}{5040} + \frac{\pi^9 x^9}{362880} - \ldots$$

Chebyshev Polynomials – Convergence



Taylor Polynomials – Convergence

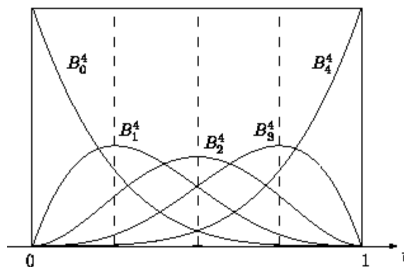


– In CAGD and in computer graphics, one is interested in easily and intuitively controllable representations of curves and curved surfaces, whose shape is roughly defined by so called control points. A famous class of those **freeform curves** or **freeform surfaces** are the **Bézier curves** or **Bézier surfaces**, respectively. The former are defined by

$$X(t) := \sum_{i=0}^{n} b_i \cdot B_i^n(t)\,, \qquad b_i \in \mathbb{R}^d\,, \qquad B_i^n(t) := \binom{n}{i}(1-t)^{n-i}t^i\,,$$

$$i = 0, ..., n\,,$$

with d-dimensional control points b_i and **Bernstein polynomials** $B_i^n(t)$.

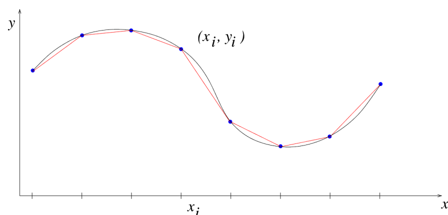


The Interpolation Problem

- **Interpolation** or **intermediate value computation** is a central problem of numerics.
- **Problem**: A special case of approximation: The values of f and p have to be equal at certain points. The approximant then becomes an **interpolant**.
- Often, instead of an explicit f, only discrete points (x_i, y_i) are given, where $p(x_i) = y_i$ has to be fulfilled. The values y_i can also be vectors – in this case points in space are interpolated.
- **Examples**:
  - Given measure or control points are to be connected with a non-linear curve (in 2D) or surface (in 3D), for example when designing a car body.
  - **Linear interpolation** is often used intuitively – whether justified or not: A program with input data of size n needs two minutes, it needs three minutes with input data of size $2n$. How long do we have to wait for the result for input data of size $1.5n$?

Notation for the Problem of Interpolation

- For reasons of simplicity, the following only deals with dimension 1. Everything introduced can be generalized for higher dimensions.
- The abscissa $x_i, i = 0, 1, ..., n$ which are to “fasten” the interpolant (i.e. the interpolant is defined by the values given) are called **nodes** or **support abscissas**.
- The distances $h_i := x_{i+1} - x_i$ between two nodes are called **mesh widths**. If $h_i = h$ is constant, the nodes are called **equidistant**.
- The given values y_i are called **support ordinates**. They can either be given explicitly or as function values $y_i = f(x_i)$ of a real-valued function f.



- The pairs (x_i, y_i) are called **support points**.
- Because of their simple structure, **polynomials** are particularly popular as interpolants:
  - $\mathbb{P}_n$ refers to the vector space of all polynomials with real coefficients of degree less or equal to n in a variable x.
  - It holds that $\dim(\mathbb{P}_n) = n + 1$. An example for a basis is provided by the **monomials** x^i, $i = 0, ..., n$:

$$p(x) := \sum_{i=0}^{n} a_i \cdot x^i.$$

  - As it is generally known, with the differential operator D^k for the k-th derivative with respect to x it holds:

$$D^{n+1}p = 0 \qquad \forall p \in \mathbb{P}_n .$$

- However, the polynomial interpolation is far from being the only option:
  - It is possible to glue together piecewise polynomials in order to get polynomial splines, which have several essential advantages (see section 2.3).
  - It is also possible to interpolate with rational functions (especially favored in CAGD), with trigonometric functions (see section 2.4), or with exponential functions.

Variations of the Interpolation Problem

- There are different ways to be faced with the problem of interpolation in concrete applications. The two most common amongst them are the following:
- **simple nodes**:
  - This is the only case examined in this chapter.
  - For each of the pairwise different x_i, a value y_i is prescribed.
  - This problem of interpolation is called **Lagrange interpolation**
  - Example: Determine the quadratic polynomial p with $p(-1) = p(1) = 1$ and $p(0) = 0$; solution: $p(x) = x^2$.
- **multiple nodes**:
  - For each node x_i, function value y_i and derivative y_i' are specified.
  - This interpolation problem is called **Hermite interpolation**.
  - Example: Determine the quadratic polynomial q with $q(0) = q'(0) = 0$ and $q''(0) = 2$; solution: $q(x) = x^2$.
  - Specifying values of derivatives can be used to smoothly glue together the polynomial pieces (i.e. without sharp bends etc.).

2.2. Interpolation with Polynomials

The Polynomial Interpolant and its Error

- The interpolation problem in case of polynomials:
  - $p \in \mathbb{P}_n$ is called **polynomial interpolant** for f for the nodes $a = x_0 < x_1 < x_2 < ... < x_n = b$, if

$$p(x_i) \;=\; f(x_i) \;=:\; y_i \qquad \forall i \in \{0, 1, ..., n\}\,.$$

  - The definition is done analogously if, instead of a function to be interpolated, only a discrete set of data $y_0, \ldots, y_n$ is given.
  - Therefore, the number of nodes determines the degree of the interpolation polynomial: p has degree n, if the support points (x_i, y_i) do not belong to a polynomial of lower degree.
  - Thus, a large number of nodes usually results in polynomials of high degrees, which – as we will see soon – are a source of numerical problems.
  - The existence and uniqueness of the solution of this interpolation problem is known from calculus. There is always exactly one polynomial $p(x)$ of minimal degree that interpolates the $n+1$ given points.

- When using the interpolant p between the support points instead of the function f, **interpolation errors** occur:
  - The difference $f(x) - p(x)$ is called **error term** or **remainder**. Later in this section, we will show

$$f(x) - p(x) \;=\; \frac{D^{n+1} f(\xi)}{(n+1)!} \cdot \prod_{i=0}^{n} (x - x_i)$$

    for an intermediate point ξ,

$$\xi \;\in\; [\min(x_0, \ldots, x_n, x), \max(x_0, \ldots, x_n, x)] \,.$$

    In case of sufficiently smooth functions f (i.e. functions that can be differentiated a sufficient number of times), this relation allows the estimate of the interpolation error.
- But first, we will deal with the question how to construct the polynomial interpolant.

Representation 1: Lagrange Polynomials

- The simplest approach for construction is the familiar **point** or **incidence sampling**:
  - Put up $p(x)$ with general coefficients:

$$p(x) := \sum_{i=0}^{n} a_i \cdot x^i .$$

  - Insert for every data point (x_i, y_i) the respective value pair and solve the resulting system of $n+1$ equations for the $n+1$ unknowns $a_0, ..., a_n$.
- An alternative approach uses **Lagrange polynomials** $L_k(x)$ of degree n,

$$L_k(x) := \prod_{i:i\neq k} \frac{x - x_i}{x_k - x_i} ,$$

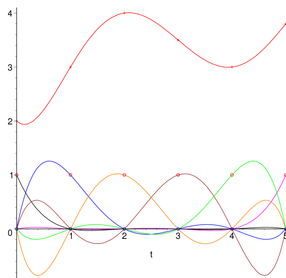
to determine the interpolant:

$$p(x) := \sum_{k=0}^{n} y_k \cdot L_k(x) .$$

- Properties of the Lagrange polynomials:
  - L_k disappears at all nodes except x_k:

$$L_k(x_i) = \delta_{ik} = \begin{cases} 1 & \text{for } i = k \\ 0 & \text{otherwise.} \end{cases}$$

  - The L_k are linearly independent and form a basis of $\mathbb{P}_n$.
  - The polynomial $p(x)$ defined above is indeed the sought interpolant.
- Note that the formula above does not only deliver $p(x)$ for a fixed x but also a compact representation of the polynomial.



Representation 2: The Scheme of Aitken and Neville

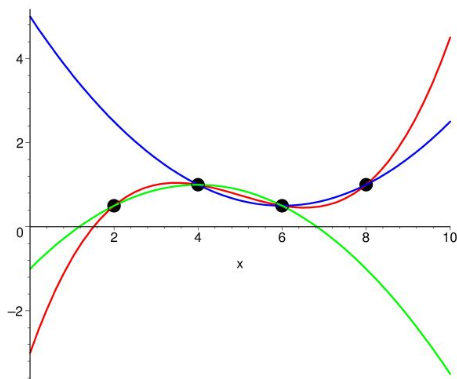
- Instead of an explicit or closed representation of the polynomial $p(x)$, the evaluation of $p(x)$ at one or more intermediate points is often all one is interested in. Interestingly, such a problem does not require an explicit formulation of the interpolating polynomial p beforehand. An elegant *recursive* possibility for evaluation is given by the **scheme of Aitken and Neville**:
  - Define auxiliary polynomials $p_{i,k}$ of degree k which interpolate the $k+1$ support points (x_l, y_l) for $l = i, ..., i+k$, $(p_{i,0} = y_i \ \ \forall_i = 0, \ldots, n)$.
  - The $p_{i,k}(x)$ follow the recursion formula

$$p_{i,k}(x) \ = \ \frac{x_{i+k} - x}{x_{i+k} - x_i} \cdot p_{i,k-1}(x) \ + \ \frac{x - x_i}{x_{i+k} - x_i} \cdot p_{i+1,k-1}(x) \, .$$

    It is easy to verify the formula using the properties of interpolation and the uniqueness of the interpolation problem.

- This scheme can be used for different purposes:
  - For any concrete value x it will deliver the polynomial value $p(x)$ as $p_{0,n}(x)$.
  - Maple, for example, can treat x as a variable, such that we get the interpolating polynomial p as $p_{0,n}$ for the result explicitly, i.e. in closed representation.

- Due to the uniqueness of the interpolation problem, the p found here is of course identical to the above sum of Lagrange polynomials – only the way how to compute p is different.



The Scheme of Aitken and Neville (2)

- We give the Aitken-Neville algorithm in pseudocode:

```
for i=0 to n do
    p[i,0] := y[i]
od;
for k=1 to n do
    for i=0 to n-k do
        p[i,k] := (x[i+k]-x)/(x[i+k]-x[i])*p[i,k-1] +
                  (x-x[i])/((x[i+k]-x[i])*p[i+1,k-1];
    od;
od;
```

- It basically depends on the concrete problem whether the specification of a closed representation for p pays off:
  - If polynomial values are only to be specified for one or a few nodes, the direct computation of the values $p(x)$ is the better choice.
  - If many evaluations are needed, the explicit computation of the polynomial (i.e. of all its coefficients) can pay off.

Representation 3: Newton's Interpolation Formula

- Another possibility of representing the polynomial interpolant requires the so called **divided differences**:
  - The highest coefficient of the polynomial $p_{i,k}$ introduced before is denoted by

    $$[x_i, \ldots, x_{i+k}]f$$

    and is called **divided difference of f of order k for $x_i, \ldots, x_{i+k}$**, i.e.

    $$p_{i,k}(x) - [x_i, \ldots, x_{i+k}]f \cdot x^k \in \mathbb{P}_{k-1} .$$

  - For the divided differences, a recursion formula can be deduced from the scheme of Aitken and Neville as well:

    $$[x_i, \ldots, x_{i+k}]f = \frac{[x_{i+1}, \ldots, x_{i+k}]f - [x_i, \ldots, x_{i+k-1}]f}{x_{i+k} - x_i} .$$

  - This recursion formula also leads to a Neville-like triangular tableau.
  - It holds that

    $$[x_i]f = f(x_i) = y_i$$

    as well as

    $$[x_i, x_{i+1}]f = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} .$$

  - Neither for the scheme of Aitken and Neville nor for the divided differences the order of nodes is relevant.

Newton's Interpolation Formula (2)

- Using divided differences, we get a second closed representation for $p(x)$, **Newton's interpolation formula**:

$$
\begin{aligned}
p(x) \quad := \quad & [x_0]f \; + \\
& [x_0, x_1]f \cdot (x - x_0) \; + \\
& [x_0, x_1, x_2]f \cdot (x - x_0) \cdot (x - x_1) \; + \\
& [x_0, \ldots, x_3]f \cdot (x - x_0) \cdot (x - x_1) \cdot (x - x_2) \; + \\
& \ldots \\
& [x_0, \ldots, x_n]f \cdot \prod_{i=0}^{n-1} (x - x_i)
\end{aligned}
$$

- The appeal of this representation is its **incremental** character:
  - When adding another node x_{n+1}, only another summand

    $$[x_0, \ldots, x_{n+1}]f \cdot \prod_{i=0}^{n} (x - x_i)$$

    has to be added to the actual interpolant for $x_0, \ldots, x_n$ in its Newton form.
  - The computations so far do not get lost when subsequently increasing the degree of the polynomial.
  - It can be shown easily that neither the Lagrange nor the Aitken-Neville representation possesses this useful attribute.

The Estimation of the Interpolation Error

- Now, we get to the initially mentioned interpolation error $f(x) - p(x)$ at an arbitrary intermediate point x. For its estimate, the divided differences will help:
  - If f is n times continuously differentiable in $[x_0, x_n]$, then there is an intermediate value ξ in this interval with

    $$[x_0, ..., x_n]f \; = \; \frac{D^n f(\xi)}{n!} \,.$$

    The proof is analogous to the mean value theorem known from calculus.
  - Now, examine an $\bar{x} \in [x_0, x_n]$ which is not a node. Further, let $P(x)$ denote the polynomial of degree $n+1$ that, in addition to the interpolation properties of $p(x)$, also interpolates our function f (now also assumed to be $n+1$ times continuously differentiable) in $\bar{x}$. Then it holds:

$$\begin{aligned} f(\bar{x}) - p(\bar{x}) \; = \; P(\bar{x}) - p(\bar{x}) \quad &= \quad [x_0, \ldots, x_n, \bar{x}]f \cdot \prod_{i=0}^{n} (\bar{x} - x_i) \\ &= \quad \frac{D^{n+1} f(\xi)}{(n+1)!} \cdot \prod_{i=0}^{n} (\bar{x} - x_i)\,; \end{aligned}$$

- For equidistant nodes with mesh width $h := x_{i+1} - x_i$, for example, the error can be estimated by

$$|f(\bar{x}) - p(\bar{x})| \;\leq\; \frac{\max_{[a,b]} |D^{n+1} f(x)|}{n+1} \cdot h^{n+1} \;=\; O(h^{n+1})\,.$$

- However, the error estimation also shows that the equidistant choice of nodes is not optimal. Rather, an $(n+1)$-th node $\bar{x}$ should be chosen to minimize the maximum value of the product on the right side.

The Condition of Polynomial Interpolation

- How well-conditioned or ill-conditioned is the problem of interpolation?
  - Input data: the nodes x_i, the sampling values y_i, as well as the intermediate point x at which the function value is to be reconstructed by interpolation.
  - Solution: the value $y := p(x)$
- Therefore, there are different 'condition numbers' – depending on the input data with respect to which the sensitivity is to be examined.
- It is easiest to describe the sensitivity of $p(x)$ regarding variations in the point of evaluation x – it is just described by $p'(x)$ and though it is bounded by $[a, b]$ in the case of polynomials, but it can become very big, particularly with high degrees of $p(x)$.
- In practice, the sensitivity regarding variations in the nodes and especially variations of the supporting values is more important. From

$$y = p(x) = \sum_{k=0}^{n} y_k \cdot L_k(x)$$

  it immediately follows that

$$\frac{\partial y}{\partial y_k} = L_k(x) .$$

  Thus, the size of the Lagrange polynomials sets this condition.
- To get a feeling for the order of magnitude the values of the Lagrange polynomials can reach, we will examine an example.

An Example to the Conditioning of Polynomial Interpolation

- Let the following situation be given:

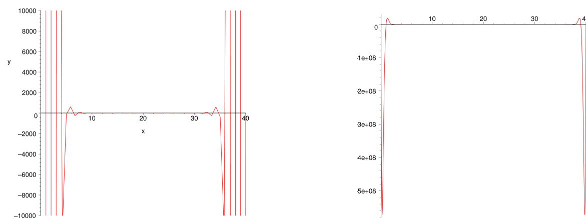
$$x_i \;:=\; i,\; i = 0, 1, ..., 40 \;=:\; n; \qquad k \;:=\; 20; \qquad x \;\in\,]x_0, x_1[\, = \,]0, 1[$$

- This results in the following estimation for the value of L_{20} in x:

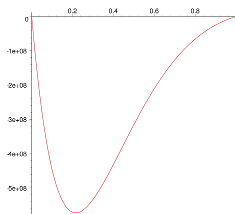
$$\begin{aligned}
|L_{20}(x)| &= \left| \prod_{i \neq 20} \frac{x - x_i}{x_{20} - x_i} \right| = \prod_{i \neq 20} \frac{|x - x_i|}{|20 - x_i|} \\
&= \frac{x}{20} \cdot \frac{1 - x}{19} \cdot \prod_{i=2}^{19} \frac{i - x}{20 - i} \cdot \prod_{i=21}^{40} \frac{i - x}{i - 20} \\
&\geq \frac{x - x^2}{380} \cdot \prod_{i=2}^{19} \frac{i - 1}{20 - i} \cdot \prod_{i=21}^{40} \frac{i - 1}{i - 20} \\
&= \frac{x - x^2}{380} \cdot \frac{18!}{18!} \cdot \frac{39!}{19! \cdot 20!} \\
&\geq 1.8 \cdot 10^8 \cdot (x - x^2)
\end{aligned}$$

- Particularly, it holds $|\; L_{20}(0.5) \;| \geq \; 4.5 \cdot 10^7$.

- This is an elementary result: Small errors in central supporting values are dramatically increased at the borders of the examined interval by polynomial interpolation.
- For large n (7 or 8 and up), polynomial interpolation is extremely ill-conditioned and therefore factually useless.
- For this reason, better methods must be found in this regard.



Lagrange polynomial $L_{20}(x)$ in $[0, 40]$



Lagrange polynomial $L_{20}(x)$ in $[0, 1]$

2.3. Polynomial Splines

Definition of Polynomial Splines

- Remember the main disadvantages of polynomial interpolation:
  - Number of data points and polynomial degree are inflexibly chained together.
  - For bigger n (and therefore for a bigger number of nodes, which frequently occurs in practice), polynomial interpolation is useless just because of matters of condition.

- Idea for remedy:
  - Glue together pieces of polynomials of lower degree to form a global interpolant for a big number of nodes as well.
  - This is exactly what **polynomial splines** or, in short, **splines** do, which we will introduce in a general way, without specific references to interpolation problems.
  - Let again be $a = x_0 < x_1 < ... < x_n = b$ and $m \in \mathbb{N}$. The x_i are called **knots**. We will only examine the special case of **simple** knots i.e. $x_i \neq x_j$ for $i \neq j$.

- A function $s: [a,b] \to \mathbb{R}$ is called **spline of order m** or **of degree $m-1$**, if the following holds:
  - $s(x) = p_i(x)$ on $[x_i, x_{i+1}]$ with $p_i \in \mathbb{P}_{m-1}$, $i = 0, 1, ..., n-1$,
  - $s \in \mathcal{C}^{m-2}([a,b])$.
- Between two neighboring knots s is a polynomial of degree $m-1$ and globally (particularly in the knots themselves) s is $m-2$ times continuously differentiable.
- s is also called **piecewise** polynomial.

Examples, Characteristics, and Applications

- What does a polynomial spline look like? For this, we examine the simplest cases:
  - $m = 1$: s is a step function (piecewise continuous, continuity is not even given in knots)
  - $m = 2$: s is a frequency polygon (piecewise linear, continuous in knots)
  - $m = 3$: quadratic spline (piecewise quadratic, globally one time continuously differentiable)
  - $m = 4$: cubic spline (piecewise cubic, globally twice continuously differentiable)
- About the name: The English word *spline* describes an elastic wooden slat used in shipbuilding.
- Evidently, with fixed knots and fixed degree, the set of all splines forms a vector space. By the way, the dimension of this space is $n + m - 1$ (a global $p_0 \in \mathbb{P}_{m-1}$ defines m degrees of freedom; for every interior knot $x_1, ..., x_{n-1}$ another degree of freedom is added for a possible jump in the $(m-1)$-th derivative – as all lower derivatives are continuous).

- Main fields of application for splines:
  - **Interpolation**: A spline is to be constructed in a way such that it fulfills $n + m - 1$ interpolation constraints – according to the number of its degrees of freedom. The nodes of the interpolation can but do not have to be knots.
  - **CAGD**: Here the spline is to be constructed in a way that it takes the shape of the curve defined by control points (important in context with **free form curves and surfaces**).

Interpolation with Cubic Splines

- Now we will actually use splines for interpolation. **Cubic splines**, the case of $m = 4$, are widespread for interpolation purposes. Moreover, they are easy to construct and to analyze. Examine the following special case:
  - simpe knots: $a = x_0 < x_1 < ... < x_n = b$
  - locally: spline $s \in \mathbb{P}_3$ on every subinterval $[x_i, x_{i+1}]$, $i = 0, 1, ..., n-1$
  - globally: spline $s \in \mathcal{C}^2([a, b])$
  - Let the nodes be exactly the knots, then the interpolation constraints are

$$s(x_i) \;=\; y_i\,, \qquad i = 0, 1, ..., n\,.$$

- With this, we get:
  - We have $n + m - 1 = n + 3$ degrees of freedom for the determination of the concrete spline (dimension of the underlying vector space, see above)
  - Of those, $n + 1$ are needed for the above interpolation constraints.
  - With this, 2 degrees of freedom remain, which we can use for further conditions to s. More on this later.
  - Firstly, we will set about 'building' our cubic spline.

The Local Polynomial Segments

- For every subinterval $[x_i, x_{i+1}]$ define the local cubic polynomial as a function of the four parameters y_i and y_{i+1} (function values in both nodes involved) as well as y_i' and y_{i+1}' (first derivative in those points):

$$\begin{aligned} s(x) &= p_i\left(\frac{x - x_i}{x_{i+1} - x_i}\right) =: p_i(t) \\ &=: y_i \cdot \alpha_1(t) + y_{i+1} \cdot \alpha_2(t) + (x_{i+1} - x_i) \cdot \left(y_i' \cdot \alpha_3(t) + y_{i+1}' \cdot \alpha_4(t)\right) \end{aligned}$$

with $\alpha_1, \alpha_2, \alpha_3, \alpha_4 \in \mathbb{P}_3$, $t \in [0,1]$ and

$$\begin{aligned} \alpha_1(0) &= 1\,, & \alpha_1(1) &= \alpha_1'(0) = \alpha_1'(1) = 0 & \Rightarrow \quad \alpha_1(t) &:= 1 - 3t^2 + 2t^3 \\ \alpha_2(1) &= 1\,, & \alpha_2(0) &= \alpha_2'(0) = \alpha_2'(1) = 0 & \Rightarrow \quad \alpha_2(t) &:= 3t^2 - 2t^3 \\ \alpha_3'(0) &= 1\,, & \alpha_3(0) &= \alpha_3(1) = \alpha_3'(1) = 0 & \Rightarrow \quad \alpha_3(t) &:= t - 2t^2 + t^3 \\ \alpha_4'(1) &= 1\,, & \alpha_4(0) &= \alpha_4(1) = \alpha_4'(0) = 0 & \Rightarrow \quad \alpha_4(t) &:= -t^2 + t^3 \end{aligned}$$

- It is easy to see that:

$$s(x_i) = p_i(0) = y_i\,, \qquad s(x_{i+1}) = p_i(1) = y_{i+1}\,,$$

$$\frac{ds}{dx}(x_i) = \frac{dp_i}{dt}(0) \cdot \frac{dt}{dx}(x_i) = y_i'\,, \qquad \frac{ds}{dx}(x_{i+1}) = \frac{dp_i}{dt}(1) \cdot \frac{dt}{dx}(x_{i+1}) = y_{i+1}'\,;$$

- Note that this ansatz already guarantees global continuity and continuous differentiability!

The Global Gluing Together

- In the representation using local polynomials we have $2n+2$ degrees of freedom ($n+1$ values of knots and $n+1$ derivatives of knots), of which we already fixed $n+1$ by the $n+1$ constraints of interpolation.
- Therefore, we have $n+1$ degrees of freedom left to ensure continuity of the second derivative at the $n-1$ interior knots. From this point of view, 2 free parameters are left as well.
- We differentiate all local polynomials twice and estimate continuity at the $n-1$ 'glued' points by equating the second derivative of p_{i-1} in $t=1$ and the second derivative of p_i in $t=0$:

$$y'_{i-1}\frac{1}{h_{i-1}} + y'_i\left(\frac{2}{h_{i-1}} + \frac{2}{h_i}\right) + y'_{i+1}\frac{1}{h_i} \;=\; 3\left(\frac{y_i - y_{i-1}}{h_{i-1}^2} + \frac{y_{i+1} - y_i}{h_i^2}\right),$$

$$h_i := x_{i+1} - x_i, i = 1, \ldots, n-1\,.$$

- Obviously, this is a tridiagonal system of linear equations of dimension $n-1$ – a first motivation to intensely occupy ourselves with the numerical solution of systems of linear equations in the chapters 4 and 6.

- How should the conditions for the two spare degrees of freedom be defined? Usually, **boundary conditions** are being set. In this case, three methods are common:
  - providing the slope at the boundaries, i.e. of y'_0 and y'_n
  - providing the second derivatives at the boundaries (**natural boundary conditions**)
  - **periodic boundary conditions**: The first and second derivatives at both boundary points respectively have to be equal, thus $y'_0 = y'_n$ and $s''(x_0) = s''(x_n)$.

Costs and Performance

- How much does cubic spline interpolation cost?
  - A (depending on the choice of boundary conditions possibly slightly perturbated) tridiagonal system of equations has to be solved.
  - It's easy to realize that this can be done with $O(n)$ arithmetic operations (e.g. by Gaussian elimination known from linear algebra).
  - This is to be compared with the $O(n^3)$ arithmetic operations (reminiscence to linear algebra or see chapter 4) used for solving a full system of equations at the incidence sample or rather with the quadratic computational effort of the scheme of Aitken and Neville.
- Which accuracy does the spline operation provide?
  - As we are using local polynomials of third degree and the nodes have distance $h := x_{i+1} - x_i$, the error estimation delivers (analogously to section 2.2)

    $$| \, f(x) - s(x) \, | = \; O(h^4) \, .$$

  - This is comparable to the order of error $O(h^{n+1})$ of polynomial interpolation. However, there, an extremely high differentiability is assumed and, for bigger n, the problems of condition detected earlier occur.

2.4. Trigonometric Interpolation

The Principle of Trigonometric Interpolation

- In **trigonometric interpolation**, complex functions are considered which are defined on the unit circle of the complex plane – this is also called **representation in frequency space**.
- Let n nodes be given on the unit circle of the complex plane,

$$z_j := e^{\frac{2\pi i}{n} j}, \qquad j = 0, 1, ..., n-1,$$

  as well as n nodal values v_j. To be found is the interpolant

$$p(z), \quad z = e^{2\pi i t}, t \in [0,1],$$

  with

$$p(z_j) = v_j, \quad j = 0, 1, ..., n-1, \qquad p(z) = \sum_{k=0}^{n-1} c_k z^k = \sum_{k=0}^{n-1} c_k e^{2\pi i k t}.$$

- The ansatz for $p(z)$ is made as a linear combination of exponential functions or – after separating the real part from the imaginary part – with sine and cosine functions.
- To find this p means de facto to calculate the coefficients c_k, those being nothing else but the coefficients of the (discrete) Fourier transform.
- For this problem, we will discuss an almost legendary algorithm, the **fast Fourier transform (FFT)**, in the following.

The Discrete Fourier Transform

- With the p introduced earlier and the abbreviation

$$\omega := e^{\frac{2\pi i}{n}}$$

the following problem has to be solved: Find n complex numbers $c_0, ..., c_{n-1}$, which fulfill

$$v_j = p(\omega^j) = \sum_{k=0}^{n-1} c_k \omega^{jk} \qquad \text{for} \qquad j = 0, 1, ..., n-1\,.$$

- With a bit of calculus ($\bar{\omega}$ complex conjugate to ω), it can be shown that

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} v_j \bar{\omega}^{jk} \qquad \text{for} \qquad k = 0, 1, ..., n-1\,.$$

- We denote by c and v the n-dimensional vectors of the discrete Fourier coefficients or of the DFT input, respectively. Furthermore, let the matrix M be given as $M = (\omega^{jk})_{0 \leq j,k \leq n-1}$. Then, in matrix vector notation, we have

$$v = M \cdot c, \; c = \frac{1}{n} \cdot \overline{M} \cdot v\,.$$

- The formula for calculating the coefficients is given by the **discrete Fourier transform (DFT)** of the input data v_k.
- The formula for calculating the values v_j out of the Fourier coefficients c_k is called **inverse discrete Fourier transform (IDFT)**.
- Obviously, the number of arithmetic operations needed for DFT and IDFT is of order $O(n^2)$. In contrast, the FFT algorithm does with $O(n \log n)$ operations for suitable n.

The FFT Algorithm – Basic Principle (1)

- **Basic idea**: Break down sums of length n into two partial sums of half length and continue recursively until reaching the trivial sum of length 1.
- For this purpose, we confine ourselves to the case of $n = 2^p$, i.e. to powers of 2, from the outset.
- In the following, we will examine the inverse discrete Fourier transform (IDFT). The method for the actual DFT follows from this clearly through transitioning to the complex conjugate and dividing by n:

$$v = IDFT(c),\ c = DFT(v) \ = \ \frac{1}{n} \cdot \overline{IDFT(\bar{v})}\,.$$

- For $v = IDFT(c)$, we get with $n = 2m$ for $j = 0, 1, ..., m-1$

$$\begin{aligned} v_j \ = \ \sum_{k=0}^{n-1} c_k \cdot e^{\frac{2\pi ijk}{n}} \quad &= \quad \sum_{k=0}^{n/2-1} c_{2k} \cdot e^{\frac{2\pi ij2k}{n}} + \sum_{k=0}^{n/2-1} c_{2k+1} \cdot e^{\frac{2\pi ij(2k+1)}{n}} \\ &= \quad \sum_{k=0}^{m-1} c_{2k} \cdot e^{\frac{2\pi ijk}{m}} + \omega^j \cdot \sum_{k=0}^{m-1} c_{2k+1} \cdot e^{\frac{2\pi ijk}{m}}\,. \end{aligned}$$

- The entries still missing $v_m, ..., v_{n-1}$ (the upper half of the indices) emerge, following the same principle, according to

$$\begin{aligned} v_{m+j} &= \sum_{k=0}^{m-1} c_{2k} \cdot e^{\frac{2\pi i(j+m)2k}{n}} + \sum_{k=0}^{m-1} c_{2k+1} \cdot e^{\frac{2\pi i(j+m)(2k+1)}{n}} \\ &= \sum_{k=0}^{m-1} c_{2k} \cdot e^{\frac{2\pi ijk}{m}} - \omega^j \cdot \sum_{k=0}^{m-1} c_{2k+1} \cdot e^{\frac{2\pi ijk}{m}} \end{aligned}$$

for $j = 0, 1, ..., m-1$, as

$$e^{\frac{2\pi i(j+m)}{n}} = e^{\pi i} \cdot e^{\frac{\pi ij}{m}} = -e^{\frac{\pi ij}{m}} = -\omega^j \,.$$

- This corresponds to a structure

$$v_j = A + \omega^j \cdot B, \; v_{m+j} = A - \omega^j \cdot B \,,$$

where the components A and B themselves are Fourier sums of half length.

The FFT Algorithm – Basic Principle (2)

- We can reduce the computation of $v = IDFT(c)$ to

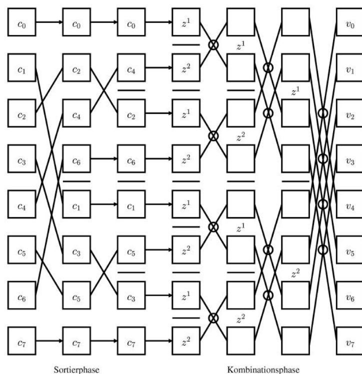
  $$IDFT(c_0, c_2, ..., c_{n-2}) \qquad \text{and} \qquad IDFT(c_1, c_3, ..., c_{n-1}),$$

  which then have to be linearly combined appropriately to get the first and second half of v.
- "Combining appropriately" is to say combining with the aid of the so called **Butterfly scheme**.
- This procedure will now be repeated recursively. With this, we can give a first recursive program in pseudocode:

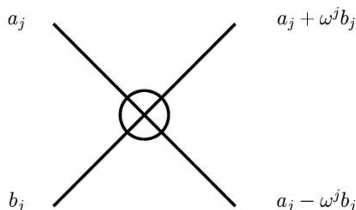
```
funct IDFT(c[0],...,c[n-1],n);
if (n=1) then do v[0]:=c[0] od;
else do
  m:=n/2;
  z1:=IDFT(c[0],c[2],...,c[n-2],m);
  z2:=IDFT(c[1],c[3],...,c[n-1],m);
  omega:=exp(2*pi*i/n);
  for j:=0 to m-1 do
    v[j]:=z1[j]+omega^j*z2[j];
    v[m+j]:=z1[j]-omega^j*z2[j];
  od;
  return(v[0],...,v[n-1]);
od;
```

Analysis of the FFT Algorithm

- The basic algorithm above is recursive; for better runtime and memory efficiency, iterative variations were developed.
- The FFT algorithm naturally breaks down into a **sorting phase** (rearranging of the `c[k]`) and a **combining phase** (butterfly operation). This can be demonstrated for the example of $n = 8$:



- Here, the butterfly operation stands for



- From this picture, we intuitively learn two things:
  - The underlying principles of the FFT algorithm can be found again in completely different places such as dynamic networks such as **Shuffle-Exchange**.
  - The computing complexity of the FFT algorithm is of order $O(n \log(n))$.

An Application: Compression of Picture Data

- Let a gray scale picture be considered as a matrix of gray scale values. Every matrix entry $g_{i,j}$ then corresponds to the gray scale value of the pixel in row i and column j of the picture.
- To stay independent of the number of the gray scales available, we allow $g_{i,j} \in [0, 1]$, where zero is equal to the color black.
- The **JPEG method** uses the special frequency distribution of "natural" images to store them at a high compression rate. Losses occurring due to compression are accepted – the compression is **lossy**. The compression is done in the following way:
  - Color images are transformed into the YUV color model (Y stands for intensity, U and V identify hue and saturation).
  - The picture is divided into blocks of 8×8 pixels. In each of those blocks, a **fast cosine transform FCT** is done which is explained on the next slide.
  - The frequency values are **quantized** – only a certain number of discrete color values is accepted, intermediate values have to be rounded accordingly. This step is mainly responsible for the occurring loss of quality.
  - Finally, the quantized data get compressed appropriately.

The Cosine Transform

- Image data is neither periodic nor does it have complex parts. Therefore, the 'conventional' Fourier transform is useless here.
- The JPEG method uses the following variation, the so called **Cosine transform**:
  - transformation:

$$F_k := \sum_{j=0}^{N-1} f_j \cos\left(\frac{\pi k (j+\frac{1}{2})}{N}\right) \qquad k = 0, 1, ..., N-1$$

  - inverse transformation:

$$f_j := \frac{2}{N} \sum_{k=0}^{N-1}{}' F_k \cos\left(\frac{\pi k (j+\frac{1}{2})}{N}\right) := \frac{2}{N}\left(\frac{F_0}{2} + \sum_{k=1}^{N-1} F_k \cos\left(\frac{\pi k (j+\frac{1}{2})}{N}\right)\right)$$

$$j = 0, ..., N-1$$

- Instead of the complex exponential terms as basic components, here, only real cosine terms appear.

- For 2 D – the relevant case for image data processing – the following representation results:

$$F_{k_1,k_2} := \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} f_{j_1,j_2} \cos\left(\frac{\pi k_1 (j_1 + \frac{1}{2})}{N_1}\right) \cos\left(\frac{\pi k_2 (j_2 + \frac{1}{2})}{N_2}\right)$$

$$f_{j_1,j_2} := \frac{4}{N_1 N_2} \sum_{k_1=0}^{N_1-1}{}' \sum_{k_2=0}^{N_2-1}{}' F_{k_1,k_2} \cos\left(\frac{\pi k_1 (j_1 + \frac{1}{2})}{N_1}\right) \cos\left(\frac{\pi k_2 (j_2 + \frac{1}{2})}{N_2}\right)$$

Reduction to FFT:

- We want to use our FFT knowledge in the search for a faster algorithm for solving the discrete cosine transform.
- For this, we will reduce the 2D cosine transform above to the familiar FFT in three steps:
  - The 2D cosine transform can be realized by composing 1D cosine transforms.
  - By mirroring the values of F_k or rather f_j the cosine transform can be transferred into a modified Fourier transform of double length. With

$$\tilde{f}_j := \begin{cases} f_j & j = 0, ..., N-1, \\ f_{2N-j-1} & j = N, ..., 2N-1 \end{cases} \quad \text{and}$$

$$\tilde{F}_k := \begin{cases} 2F_k & k = 0, ..., N-1, \\ 0 & k = N, \\ -2F_{2N-k} & k = N+1, ..., 2N-1 \end{cases}$$

    we get

$$\tilde{f}_j = \frac{1}{2N} \sum_{k=0}^{2N-1} \tilde{F}_k e^{\left(\frac{-2\pi i (j+\frac{1}{2})k}{2N}\right)}, \qquad \text{and} \qquad \tilde{F}_k = \sum_{j=0}^{2N-1} \tilde{f}_j e^{\left(\frac{2\pi i (j+\frac{1}{2})k}{2N}\right)}.$$

  - By scaling $\tilde{F}_k$ and $\tilde{f}_j$ adequately, the introduced standard representation of the (I)DFT results.

2.5. Examples for Applications of Interpolation

Interpolation in Computer Science

- **Tables** – formerly indispensable for functions such as the logarithm or trigonometric functions, today they are actually only used for distribution functions such as the normal distribution – print function values at discrete nodes. If intermediate values are wanted, interpolation is necessary.
- In every **function plot** discrete points have to be interpolated.
- Arbitrary nonlinear curves and surfaces – so called **freeform curves** or **freeform surfaces** – play a big role in geometric modeling and computer graphics.
- For the creation of **computer animations**, e.g. of cartoon sequences, the explicit creation of every single frame usually is too costly computationally for real-time applications – after all, about 25 images per second are needed for a visually appealing (i.e. especially jerk free) animation. For this reason, often only certain keyframes are computed and certain interim frames are interpolated, which is also called **inbetweening** in this context.
- When robots are moving (e.g. playing soccer), discrete target points are computed – the way from the momentary position to the next target is then determined by interpolation.

- For **picture reconstruction**, the original has to be reconstructed as exactly as possible out of the compressed or faulty image data – another case for interpolation.
- Eventually, the problem of interpolation occurs at every **D/A conversion** of a signal.