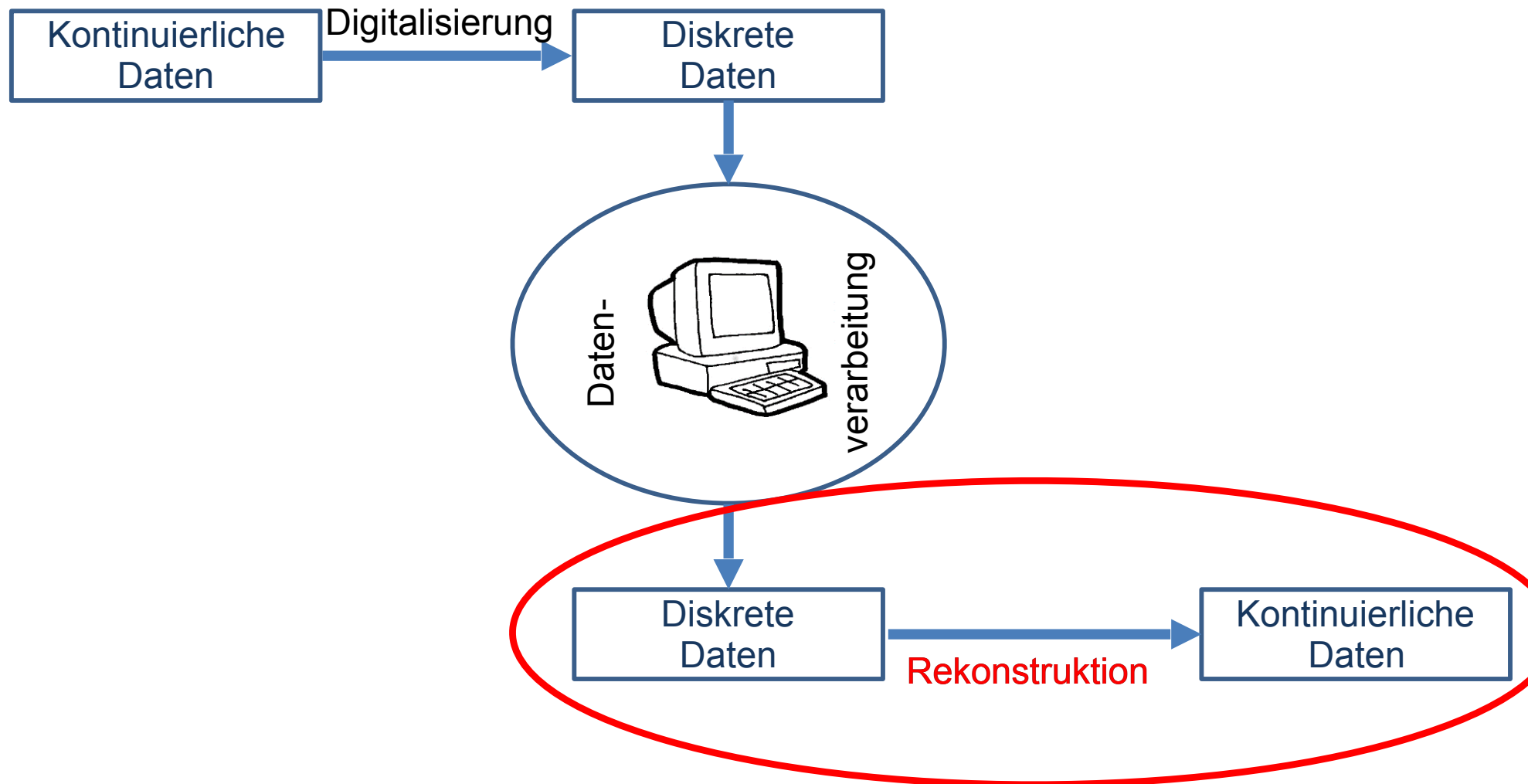


Algorithmik kontinuierlicher Systeme

Rekonstruktion kontinuierlicher Daten – Interpolation 1D



- Rekonstruktion kontinuierlicher Daten aus diskreten Daten



- Lokale Verfahren vs. globale Verfahren
- Lokale Verfahren
 - *nearest neighbor* : unstetig, $O(h)$
 - linear: stetig, $O(h^2)$
 - Catmull-Rom: glatt (diff.-bar, C^1), $O(h^3)$
 - kubische Splines: glatt (diff.-, bar, C^2), $O(h^4)$
- Schätzen von Ableitungen : vorwärts, rückwärts, zentral
- Rechenaufwand:
 - Aufwand für Suche ...
 - der Rest ist $O(1)$
- Globale Verfahren
 - Polynominterpolation
 - B-Spline-Interpolation

- Polynome vom Grad $n-1$ bilden einen n -dimensionalen Vektorraum.

$$\mathbb{P}_n = \text{span} \{ 1, x, x^2, \dots, x^{n-1} \} = \left\{ \sum_{i=0}^{n-1} c_i x^i \right\}$$

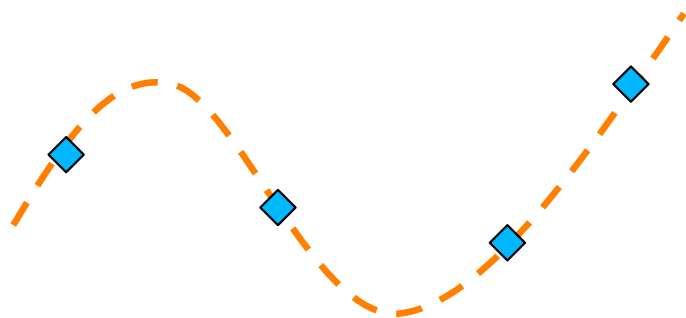
Taylorbasis
Monombasis

- Polynome werden in der Informatik sehr gerne verwendet, weil deren Auswertung für Computer einfach ist (→ Hornerschema s.u.)
 - im Gegensatz zu anderen mathematischen Funktionen.
- Polynome werden deshalb häufig zur Approximation anderer (komplizierterer) Funktionen eingesetzt (→ Taylorentwicklung)
- Die Taylorbasis (oder Monombasis) oben ist nur eine Möglichkeit den Polynomraum darzustellen - Varianten später.

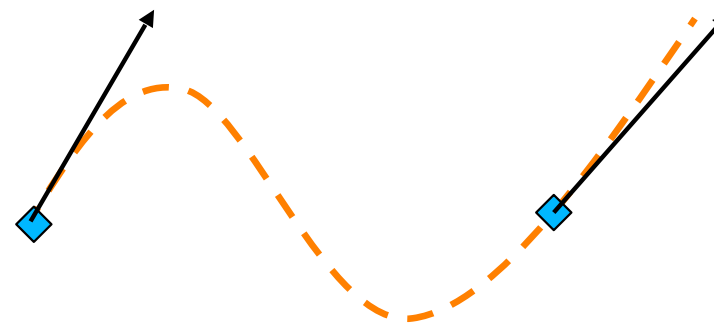
Interpolation: von der zu approximierenden Funktion werden nur Werte an (wenigen) Stützstellen verwendet, die Werte dazwischen werden durch ein „Polynominterpolation“ berechnet.

Werden nur die Punkte selbst verwendet, spricht man von „Lagrange-Interpolation“.

Werden zusätzliche auch Informationen über die Richtungen (Ableitung der Funktion) verwendet, dann spricht man von „Hermite-Interpolation“ bzw. von Interpolation mit doppelten Stützstellen.



Lagrange Interpolation



Hermite Interpolation

- Für die (Lagrange-)Polynominterpolanten sind gegeben:
 - **Stützstellen** $\{x_1, x_2, \dots, x_n\}$
 - **Stützwerte** $\{y_1, y_2, \dots, y_n\}$

- Ein Polynom $p(x) = \sum_{i=0}^{n-1} c_i x^i$ interpoliert diese Werte

sofern $p(x_1) = y_1, \dots, p(x_n) = y_n$

das heißt:

$$p(x_1) = \sum_{i=0}^{n-1} c_i x_1^i = y_1$$

$$p(x_2) = \sum_{i=0}^{n-1} c_i x_2^i = y_2$$

\vdots

$$p(x_n) = \sum_{i=0}^{n-1} c_i x_n^i = y_n$$

- $p(x) = c_0 + c_1 x + \dots + c_{n-1} x^{n-1}$ eingesetzt in die n Interpolationsbedingungen führt auf ein lineares Gleichungssystem mit n Gleichungen für die n unbekannten Koeffizienten c_i .

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}}_A \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix}}_c = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_y$$

- A heißt Vandermonde-Matrix,
- Vandermonde-Determinante $\det(A) = \prod_{1 \leq j < k \leq n} (x_j - x_k) \neq 0$

- Die Matrix A ist die sogenannte Vandermonde-Matrix.
 - ▶ Sie ist quadratisch, voll besetzt, nicht singulär (sofern $x_i \neq x_j$)
 - aber
 - ▶ schon bei moderatem n (und äquidistanten Stützstellen) sehr schlecht konditioniert

n	$\kappa(A_{\text{Vandermonde}})$		
3	13.91		
4	154.46		
5	2592.89		
6	57688.70		
7	1597316.16		
8	52937735		
9	2043725290		
10	9.00778E10		
11	4.46282E12		
12	2.45502E14		
13	1.48460E16		
14	9.79006E17		
15	7.02701E19		

- Das System $A c = y$ ist eindeutig lösbar, da $\det(A) \neq 0$:
- Folgerung:
Zu n Stützstellen gibt es ein eindeutig bestimmtes Interpolationspolynom vom Grad $n-1$
- Man löse das Gleichungssystem $A c = y$ mit der Vandermondematrix A z.B. mit LR-Zerlegung (oder Gauss-Elimination)
 - Aufwand $O(n^3)$
 - Numerisch sehr problematisch (da schlecht konditioniert)
 - *Gibt's, denn da nichts besseres?*

Divide-et-Impera Ansatz

Rekursion

Wie könnte es gehen?

~~$p(x)$ interpoliert x_1, x_2, \dots, x_k~~

~~$q(x)$ interpoliert $x_{k+1}, x_{k+2}, \dots, x_{2k}$~~

~~$F(p, q)$ interpoliert $x_1, x_2, \dots, x_k, x_{k+1}, x_{k+2}, \dots, x_{2k}$~~

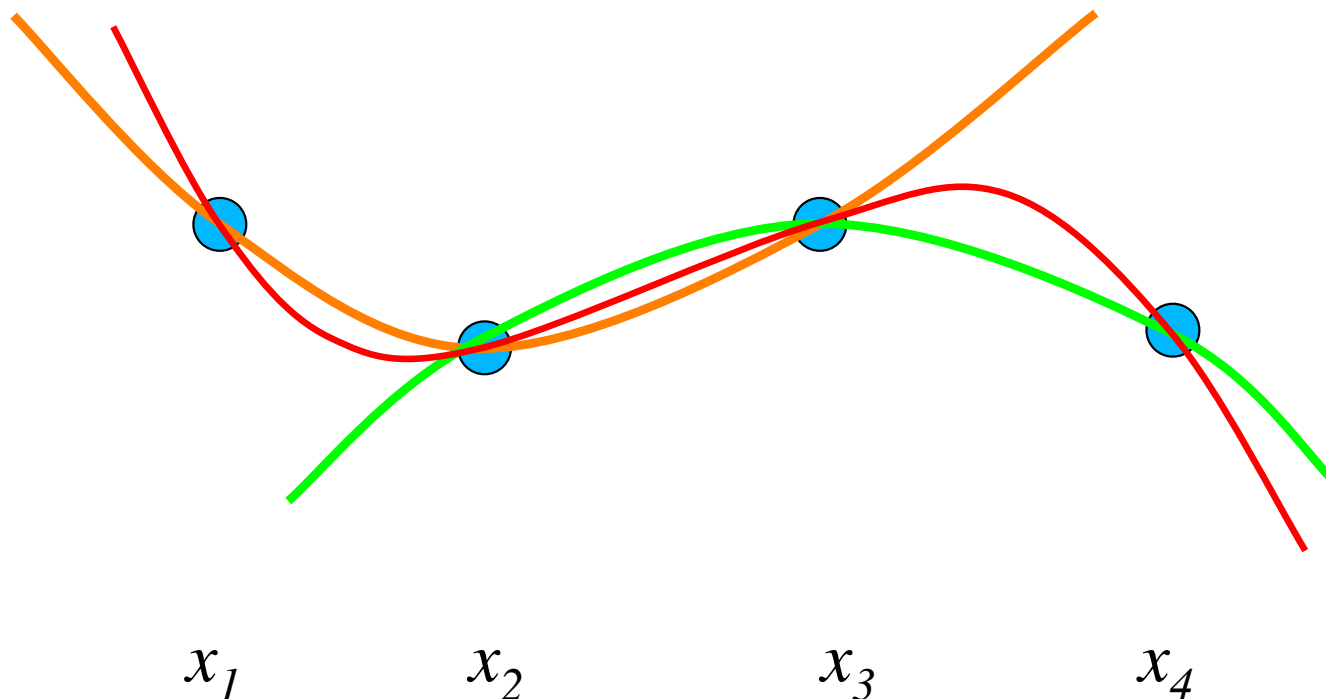
So geht es leider nicht!
(Es wäre zu schön gewesen)

Ansatz

$p_{1,k-1}(x)$ interpoliert an x_1, x_2, \dots, x_k (Polynomgrad $k-1$)

$p_{2,k-1}(x)$ interpoliert an x_2, x_3, \dots, x_{k+1} (Polynomgrad $k-1$)

Brechne daraus $p_{1,k}$ das an x_1, x_2, \dots, x_{k+1} interpoliert (Polynomgrad k)



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

- Warum funktioniert das:
 - Ist $x=x_i$, dann „übernimmt“ das linke Polynom.
 - Ist $x=x_{i+k}$, dann „übernimmt“ das rechte Polynom.
 - Bei den Stützstellen dazwischen „kooperieren“ beide Polynome.
 - Die Sortierung der Stützstellen ist nicht relevant.
- Die Formel kann man als Rekursion für (reelle/komplexe) Zahlen, also die Werte der Polynome an der Stelle x lesen
- Es ist aber genau so gut eine Formel für Funktionen:
 - Eine Funktion, die aus zwei Funktionen eine dritte berechnet.
- Terminierung der Rekursion:
 - Die Interpolation einer einzigen Stützstelle (x_i, y_i) liefern trivialerweise die konstanten Polynome $p_{i,0}(x) = y_i$

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

Wenn man einen Datentyp „Polynom“ (`class Polynom` in Python/C++) hat, kann man diese Formel zur Konstruktion des interpolierenden Polynoms verwenden.

Besonders attraktiv wäre dies natürlich in funktionalen Programmierprachen (wie Lisp, Maple, auch Python,...).

Am einfachsten und effizientesten geht es, wenn man eine geeignete Datenstruktur (mathematisch gesehen: Polynombasis) verwendet, nämlich die Polynomdarstellung nach Newton. Dann kommt man auf die Newtonsche Interpolationsformel (siehe heute später in der Vorlesung).

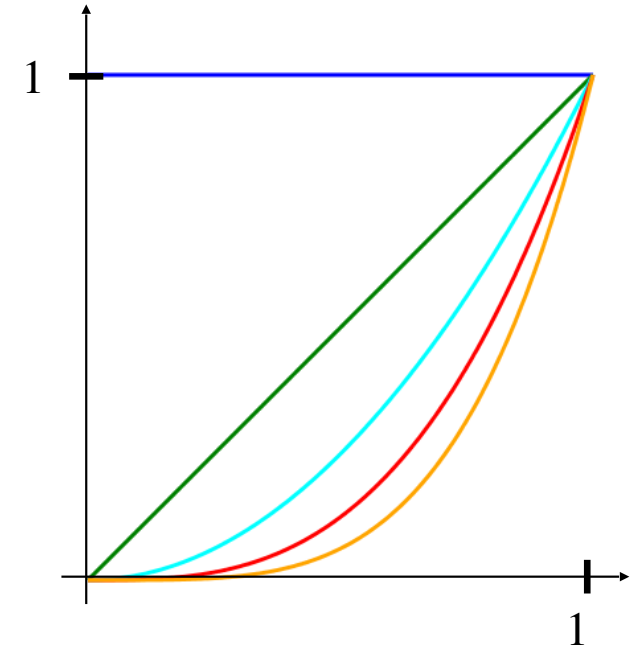
Aber zunächst ist die Rekursionsformel noch unbrauchbar:

Der Aufwand explodiert, wenn man die Rekursionsformel naiv verwendet: Bei Polynomgrad n sind $O(2^n)$ Aufrufe erforderlich! Das ist viel zu teuer.

Nur wenn man nutzt, dass die Rekursion immer wieder die gleichen Polynome berechnet, kann man ein effizienten Algorithmus erhalten.

Das führt auf das Aitken-Neville-Dreiecksschema und die Newtoninterpolation: siehe unten.

- Bisher : Darstellung in der
 - Taylor-Basis bzw. Monom-Basis $\{1, x, x^2, x^3, \dots, x^{n-1}\}$
 - zu speichern: Die n Koeffizienten
 - Effiziente Auswertung mit dem Horner-Schema (s.u.)



- **Alternativen:**
 1. Bernstein-Polynome
 2. Lagrange-Polynome
 3. Newton-Polynome

- bilden jeweils eine Basis von \mathbb{P}_n

- Alternative 1: Basis aus **Bernstein**-Polynomen

$$B_i^{n-1}(x) = \binom{n-1}{i} (1-x)^{n-1-i} x^i \quad \mathbb{P}_{n-1} = \text{span} \{ B_0^{n-1}(x), B_1^{n-1}(x), \dots, B_{n-1}^{n-1}(x) \}$$

- Basis für Polynome vom Grad $n-1$
- Sehr beliebt in Graphik beim geometrischen Modellieren
→ Gegenstand der aktuellen Übungsaufgabe
- Für Polynominterpolation eher **nicht** geeignet

- ▶ Alternative 2: Basis aus **Lagrange**-Polynomen
zu gegebenen Stützstellen $\{x_1, x_2, \dots, x_n\}$

$$L_i(x) = \frac{(x - x_1) \cdot \dots \cdot (x - x_{i-1}) \cdot (x - x_{i+1}) \cdot \dots \cdot (x - x_n)}{(x_i - x_1) \cdot \dots \cdot (x_i - x_{i-1}) \cdot (x_i - x_{i+1}) \cdot \dots \cdot (x_i - x_n)} \quad (i = 1..n)$$

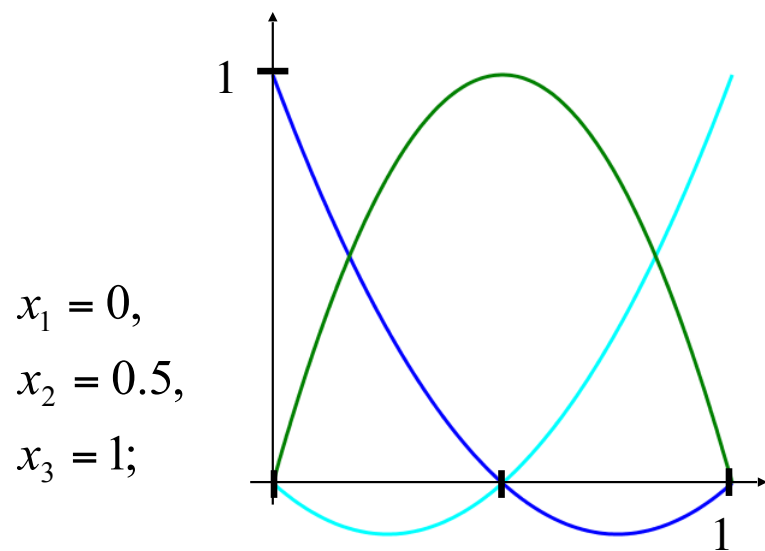
- Eigenschaften der Lagrange-Polynome:
 - Die L_i sind linear unabhängig und bilden eine Basis von IP_{n-1}
 - $L_i(x_j) = \delta_{ik}$ (= Kroneckersymbol)
 - \rightarrow das (eindeutige) Interpolationspolynom ist gegeben durch

$$p(x) = \sum_{i=1}^n y_i L_i(x)$$

- Hauptanwendung der Lagrange-Polynome sind **theoretische Überlegungen** (z.B. zur Entwicklung von speziellen Algorithmen). Für die algorithmische **Berechnung** des Interpolationspolynoms selbst sind sie eher schlecht geeignet!

► Alternative 2: Basis aus **Lagrange**-Polynomen zu gegebenen Stützstellen $\{x_1, x_2, \dots, x_n\}$

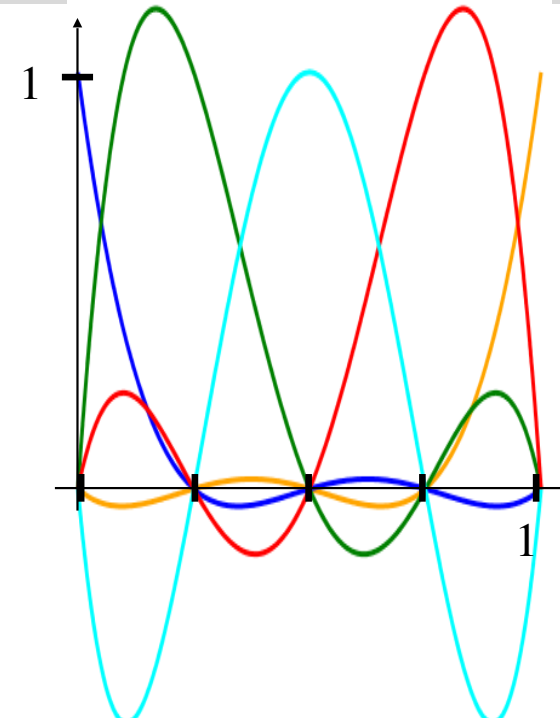
$$L_i(x) = \frac{(x - x_1) \cdot \dots \cdot (x - x_{i-1}) \cdot (x - x_{i+1}) \cdot \dots \cdot (x - x_n)}{(x_i - x_1) \cdot \dots \cdot (x_i - x_{i-1}) \cdot (x_i - x_{i+1}) \cdot \dots \cdot (x_i - x_n)} \quad (i = 1..n)$$



$$L_1(x) = \frac{(x - 0.5)(x - 1)}{(0 - 0.5)(0 - 1)} = 2x^2 - 3x + 1$$

$$L_2(x) = \frac{(x - 0)(x - 1)}{(0.5 - 0)(0.5 - 1)} = -4x^2 + 4x$$

$$L_3(x) = \frac{(x - 0)(x - 0.5)}{(1 - 0)(1 - 0.5)} = 2x^2 - x$$

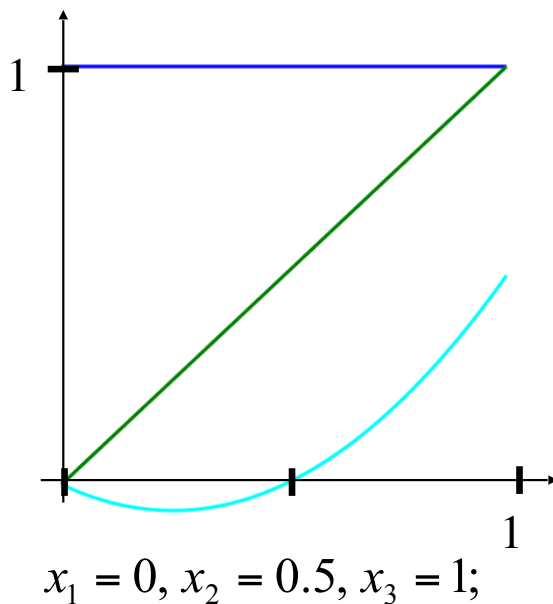


$x_1 = 0,$
 $x_2 = 0.25,$
 $x_3 = 0.5,$
 $x_4 = 0.75,$
 $x_5 = 1;$

$$L_i(x_j) = \begin{cases} 1 & \text{falls } j = i \\ 0 & \text{sonst} \end{cases}$$

► Alternative 3: Basis aus **Newton**-Polynomen zu gegebenen Stützstellen $\{x_1, x_2, \dots, x_n\}$

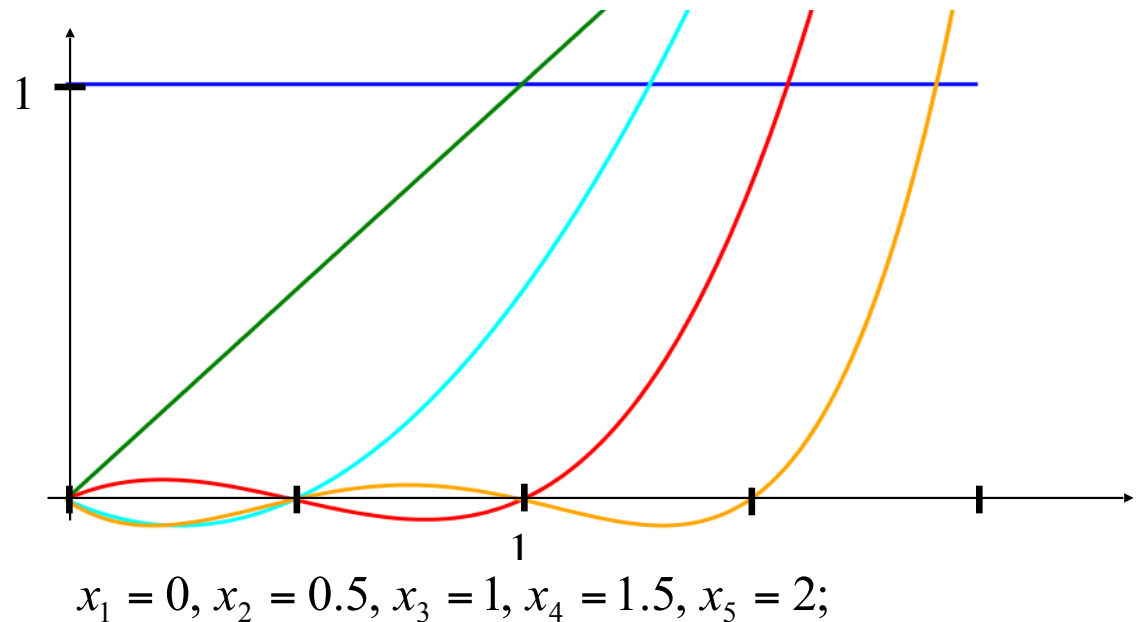
$$N_i(x) = (x - x_1) \cdot (x - x_2) \cdot \dots \cdot (x - x_{i-1}) \quad (i = 1..n)$$



$$N_1(x) = 1,$$

$$N_2(x) = (x - 0) = x,$$

$$N_3(x) = (x - 0)(x - 0.5) = x^2 - 0.5x;$$



...

$$N_4(x) = (x - 0)(x - 0.5)(x - 1),$$

$$N_5(x) = (x - 0)(x - 0.5)(x - 1)(x - 1.5);$$

- Alternative 3: Basis aus **Newton**-Polynomen
zu gegebenen Stützstellen $\{x_1, x_2, \dots, x_n\}$

$$N_i(x) = (x - x_1) \cdot (x - x_2) \cdot \dots \cdot (x - x_{i-1}) \quad (i = 1..n)$$

Verwende die Basis $\{N_1(x), N_2(x), \dots, N_n(x)\}$ zur Lösung der Interpolationsaufgabe, Ansatz:

$$p(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)(x - x_2) + \dots + a_{n-1}(x - x_1)(x - x_2) \dots (x - x_{n-1})$$

- Mit dieser Basis läßt sich das Interpolationsproblem effizient und stabil lösen
→ Algorithmus von **Aitken-Neville**
- Weiterer Vorteil:
In dieser Basis ist das Interpolationspolynom effizient für alle x
(mit einem Horner-artigen Schema) auswertbar.

- Alternative 3: Basis aus **Newton**-Polynomen zu gegebenen Stützstellen $\{x_1, x_2, \dots, x_n\}$

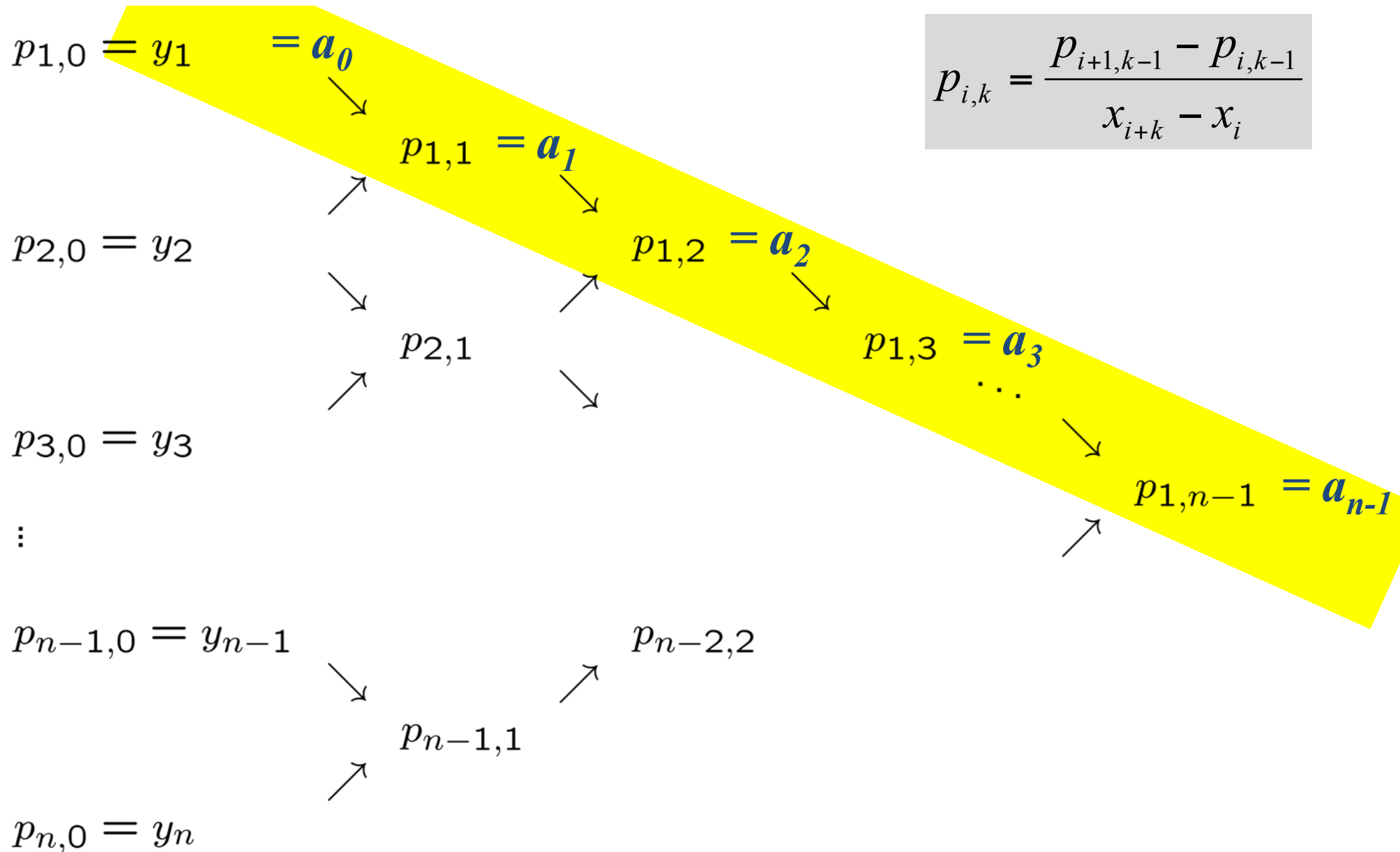
$$p(x) = a_0 + a_1(x-x_1) + a_2(x-x_1)(x-x_2) + \dots + a_{n-1}(x-x_1)(x-x_2)\dots(x-x_{n-1})$$

- Die Interpolationsbedingungen $p(x_j) = y_j$ führen auf ein Gleichungssystem mit unterer Dreiecksmatrix:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & x_2 - x_1 & 0 & & 0 \\ 1 & x_3 - x_1 & (x_3 - x_1)(x_3 - x_2) & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 1 & x_n - x_1 & (x_n - x_1)(x_n - x_2) & & N_n(x_n) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}$$

- Der Algorithmus von **Aitken-Neville**

$$p_{i,k} = \frac{p_{i+1,k-1} - p_{i,k-1}}{x_{i+k} - x_i}$$



- Der Algorithmus von Aitken-Neville

$$p(x) = a_0 + a_1(x-x_1) + a_2(x-x_1)(x-x_2) + \dots + a_{n-1}(x-x_1)(x-x_2)\dots(x-x_{n-1})$$

- Pseudo-Code zur Berechnung der Koeffizienten a_i :

```
# Initialisierung:
for i=1..n
    pi,0 = Yi
# Spaltenweises Vorgehen:
for k=1..n-1
    # Durchlaufen der Spalte:
    for i=1..n-k-1
        pi,k = (pi+1,k-1 - pi,k-1) / (xi+k - xi)
# Rückgabe der Ergebnisse:

ai = p1,i ; (i = 0..n-1)
```

- Aufwand: $O(n^2)$

- Effiziente Auswertung des Newton-Polynoms mit Horner-artigem Schema:
- Klassisches Hornerschema für

$$p(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} = (\dots(a_{n-1}x + a_{n-2})x + \dots + a_1)x + a_0$$

► PseudoCode :

```
sum = a[n-1]
for i=n-2..0
    sum = sum*x+a[i]
```

- Modifiziertes Hornerschema

$$\begin{aligned} p(x) &= a_0 + a_1(x - x_1) + \dots + a_{n-1}(x - x_1) \cdot \dots \cdot (x - x_{n-1}) = \\ &= (\dots(a_{n-1}(x - x_{n-1}) + a_{n-2})(x - x_{n-2}) + \dots + a_1)(x - x_1) + a_0 \end{aligned}$$

► PseudoCode

```
sum = a[n-1]
for i=n-2..0
    sum = sum*(x-x[i+1])+a[i]
```

- Die Bernstein-Polynome vom Grad n

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i \quad (i = 0, 1, 2, \dots, n)$$

bilden ein Basis der Polynome (vom Grad n)

- Die Binomial-Koeffizienten $\binom{n}{i} = \frac{n!}{i! \cdot (n-i)!}$

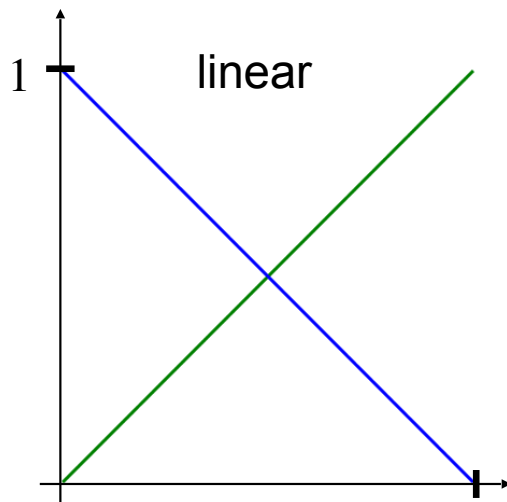
- Rekursive Definition
(Pascalsches Dreieck) $\binom{n+1}{i} = \binom{n}{i} + \binom{n}{i-1}$

- Binomische Formel $(a+b)^n = \sum_{i=0}^n \binom{n}{i} \cdot a^{n-i} b^i$

$$\begin{array}{ccccccc}
 & & & & 1 & & & & \\
 & & & & & 1 & & 1 & \\
 & & & & & & 1 & 2 & 1 \\
 & & & & & & & 1 & 3 & 3 & 1 \\
 & & & & & & & & 1 & 4 & 6 & 4 & 1
 \end{array}$$

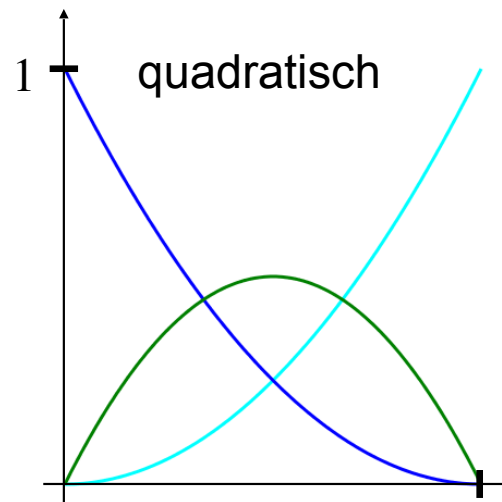
- Alternative 1: Basis aus **Bernstein-Polynomen**

$$B_i^{n-1}(x) = \binom{n-1}{i} (1-x)^{n-1-i} x^i \quad \mathbb{P}_{n-1} = \text{span} \{ B_0^{n-1}(x), B_1^{n-1}(x), \dots, B_{n-1}^{n-1}(x) \}$$



$$B_0^1(x) = 1 - x$$

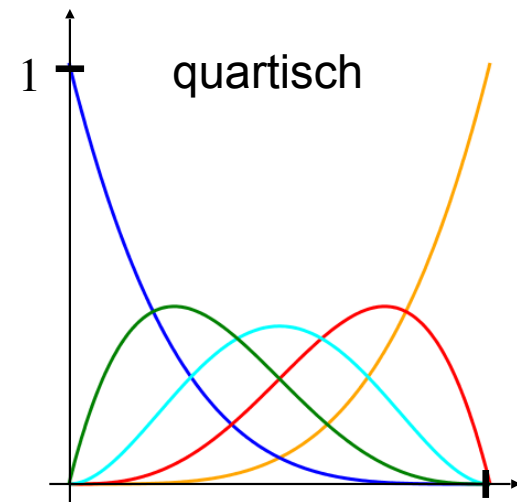
$$B_1^1(x) = x$$



$$B_0^2(x) = (1-x)^2$$

$$B_1^2(x) = 2(1-x)x$$

$$B_2^2(x) = x^2$$



$$B_0^4(x) = (1-x)^4$$

$$B_1^4(x) = 4(1-x)^3x$$

$$B_2^4(x) = 6(1-x)^2x^2$$

$$B_3^4(x) = 4(1-x)x^3$$

$$B_4^4(x) = x^4$$

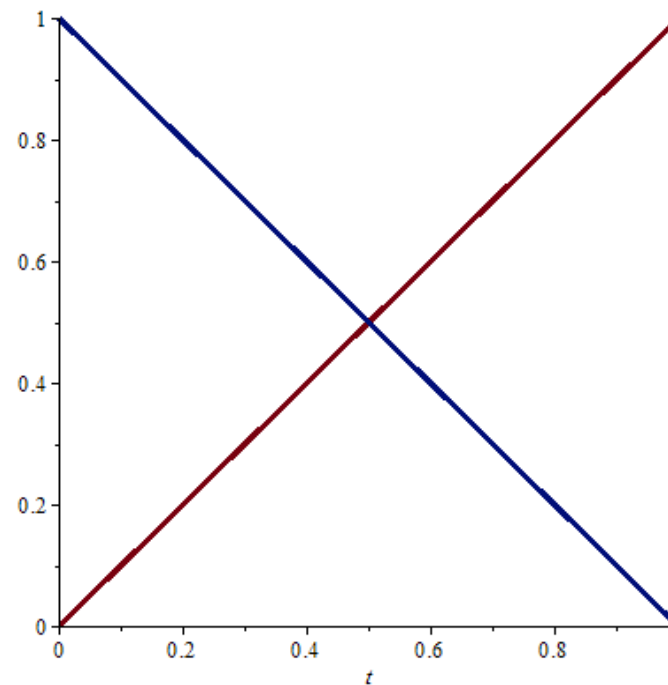
► **Eigenschaften:**

★ $B_0^n(x) \geq 0$ falls $0 < x < 1$

★ $\sum_{i=0}^n B_i^n(x) = 1$

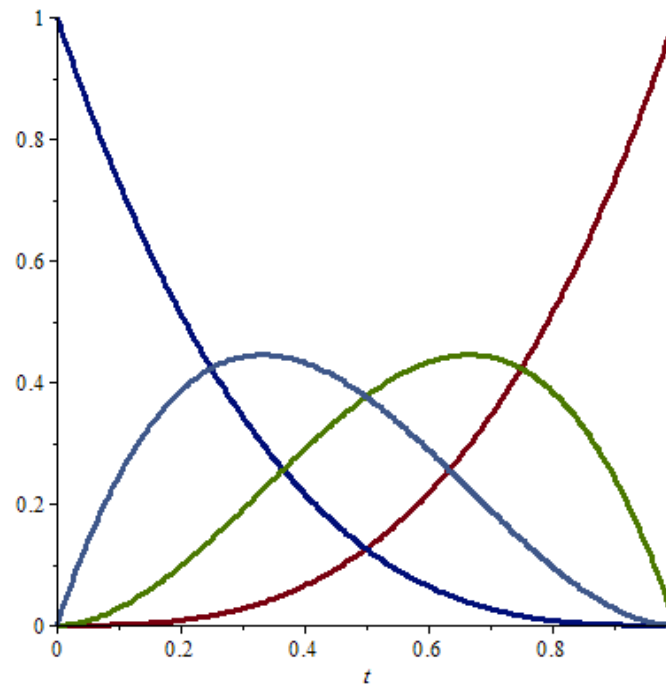
- Bernstein-Polynome vom Grad $n=1$

$$B_0^1(t) = 1 - t, \quad B_1^1(t) = t;$$



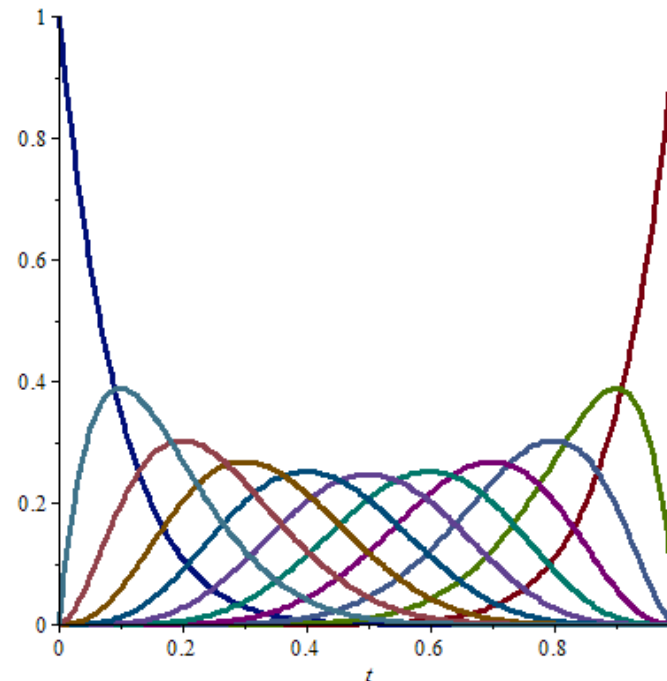
- Bernstein-Polynome vom Grad $n=3$
(kubische Bernstein-Polynome)

$$B_0^3(t) = (1-t)^3, B_1^3(t) = 3(1-t)^2 t, B_2^3(t) = 3(1-t)t^2, B_3^3(t) = t^3;$$



- Bernstein-Polynome vom Grad $n=10$

$$B_0^{10}(t) = (1-t)^{10}, B_1^{10}(t) = 10(1-t)^9 t, \dots, B_{10}^{10}(t) = t^{10};$$



$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i \quad (i = 0, 1, 2, \dots, n)$$

- Es gilt

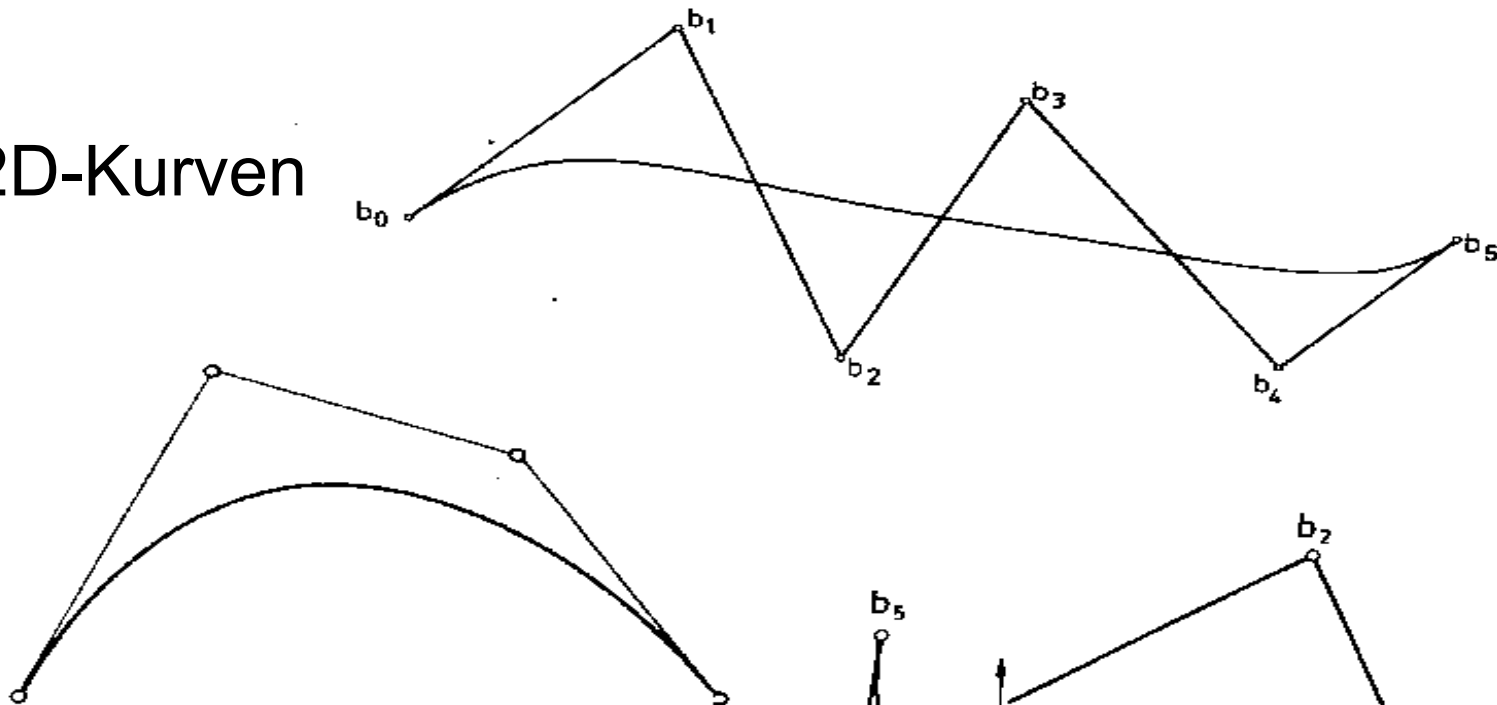
$$0 \leq B_i^n(t) \leq 1 \quad \text{für } t \in [0, 1]$$

$$B_i^n(t) \quad \text{hat eine } i\text{-fache Nullstelle in } t = 0$$

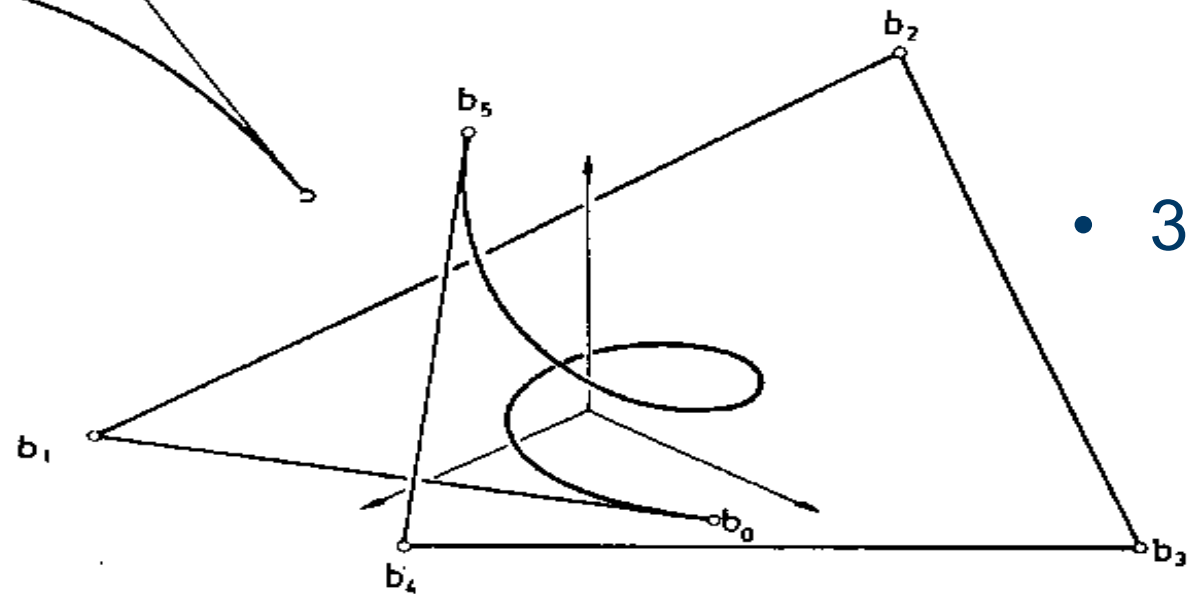
$$B_i^n(t) \quad \text{hat eine } (n-i)\text{-fache Nullstelle in } t = 1$$

$$\sum_{i=0}^n B_i^n(t) = 1 \quad \text{für alle } t$$

- 2D-Kurven



- 3D-Kurve



- Geometrie des Kontrollpolygons spiegelt (grob) die Geometrie der Kurve wider
- **Formeigenschaften**
 1. Interpolation der Endpunkte
 2. In den Endpunkten tangential an das Kontrollpolygon
 3. Bézier-Kurve liegt in der konvexen Hülle der Kontrollpunkte
 4. affine Invarianz
 5. Variationsreduzierend
- Konsequenz: Modifikation der Kontrollpunkte führt zu vorhersehbarer (intuitiver) Änderung der Bézier-Kurve.

- **Endpunkt-Interpolation**

$$C(0) = \sum_{i=0}^n b_i \cdot B_i(0) = b_0$$

$$C(1) = \sum_{i=0}^n b_i \cdot B_i(1) = b_n$$

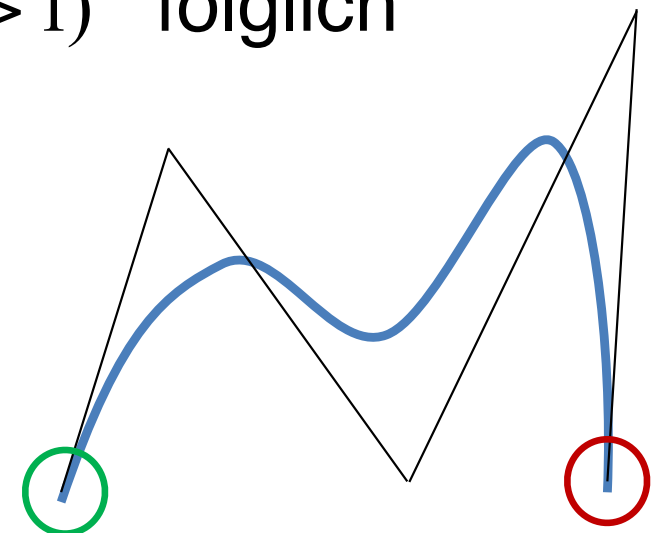
- **Tangentenbedingung**

$$C'(t) = \sum_{i=0}^n b_i \cdot B_i'(t), \text{ dabei}$$

$$B_0'(0) = -n, \quad B_1'(0) = n, \quad B_i'(0) = 0 \quad (i > 1) \quad \text{folglich}$$

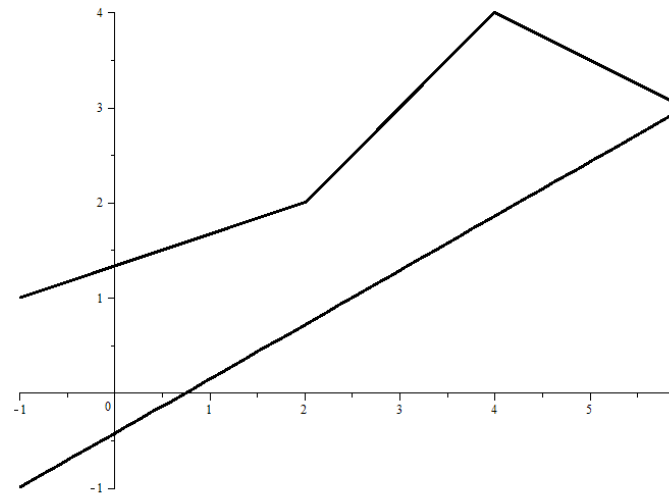
$$C'(0) = \sum_{i=0}^n b_i \cdot B_i'(0) = n(b_1 - b_0)$$

$$C'(1) = \sum_{i=0}^n b_i \cdot B_i'(1) = n(b_n - b_{n-1})$$

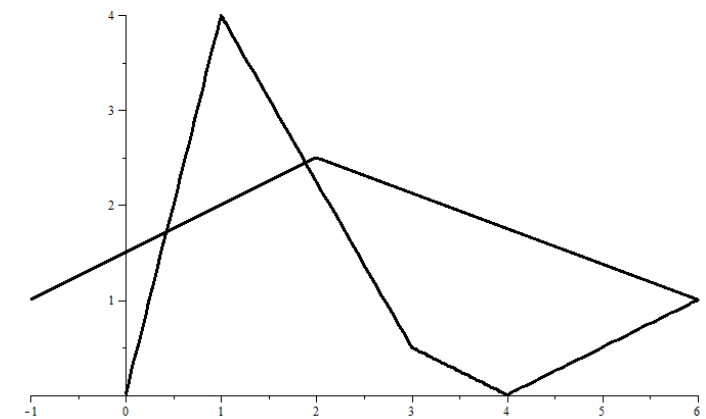


- Konvexe Hülle:
Bézier-Kurve liegt in **konvexer Hülle** ihrer Kontrollpunkte

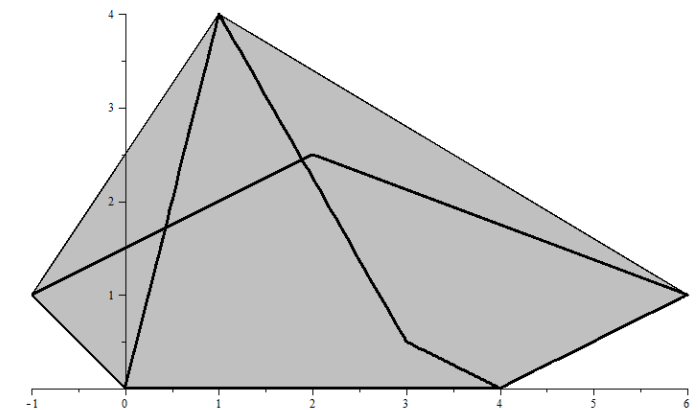
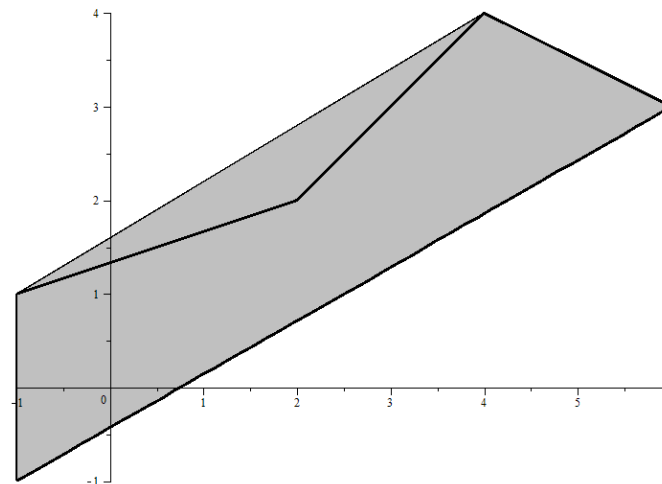
- Beispiele



convex hull property



convex hull property



- (Rekursive) Auswertung der Bernstein Polynome: teuer

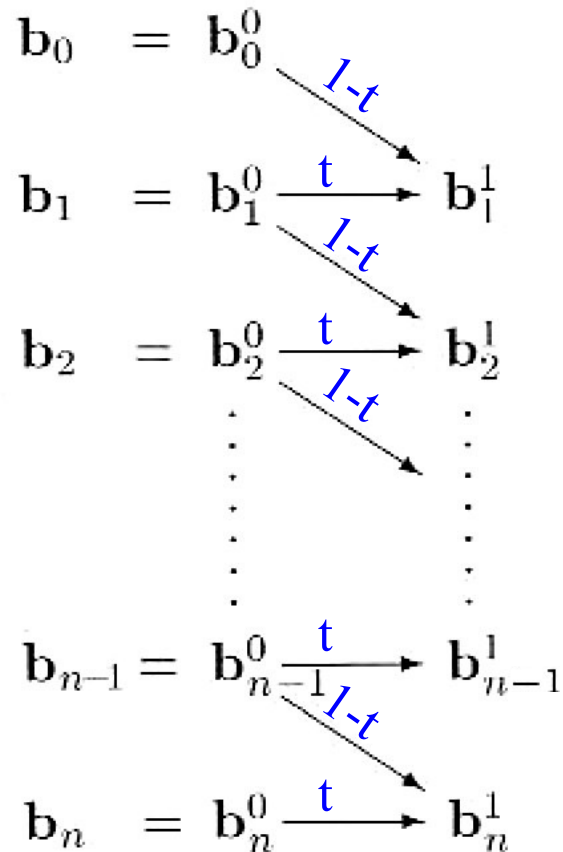
$$B_i^{r+1}(t) = (1-t)B_i^r(t) + tB_{i-1}^r(t)$$

$$B_i^0(t) = \begin{cases} 1 & \text{falls } i = 0 \\ 0 & \text{sonst} \end{cases}$$

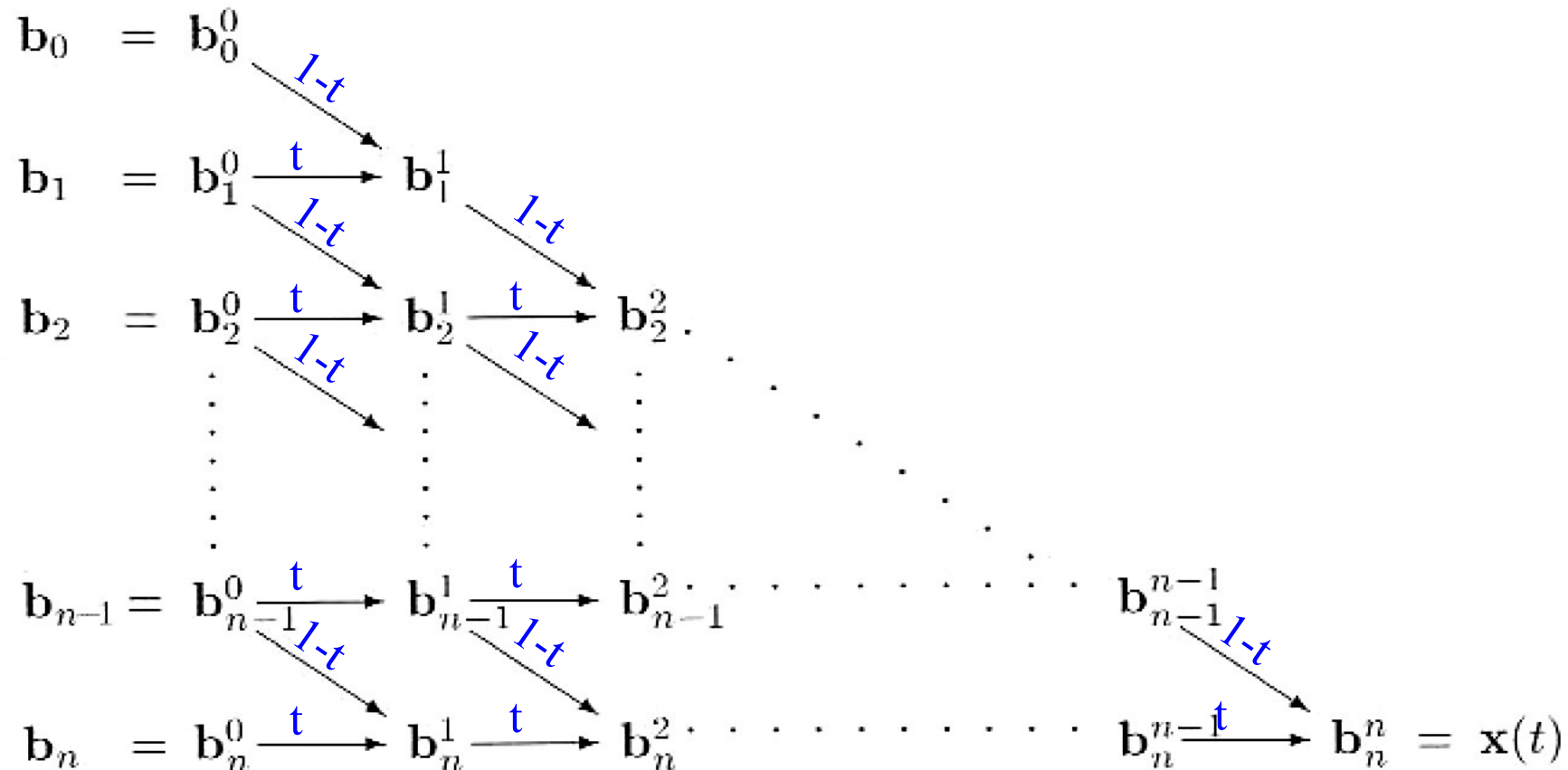
- oder durch **Horner-Bézier**: effizient

$$\begin{aligned} C(t) &= \sum_{i=0}^n \mathbf{b}_i B_i^n(t) = \sum_{i=0}^n \mathbf{b}_i \binom{n}{i} (1-t)^{n-i} t^i \\ &= (1-t)^n \left(\tilde{\mathbf{b}}_0 + \tilde{\mathbf{b}}_1 \left(\frac{t}{1-t} \right)^1 + \dots + \tilde{\mathbf{b}}_{n-1} \left(\frac{t}{1-t} \right)^{n-1} + \tilde{\mathbf{b}}_n \left(\frac{t}{1-t} \right)^n \right) \\ &= t^n \left(\tilde{\mathbf{b}}_0 \left(\frac{1-t}{t} \right)^n + \tilde{\mathbf{b}}_1 \left(\frac{1-t}{t} \right)^{n-1} + \dots + \tilde{\mathbf{b}}_{n-1} \left(\frac{1-t}{t} \right)^1 + \tilde{\mathbf{b}}_n \right) \end{aligned}$$

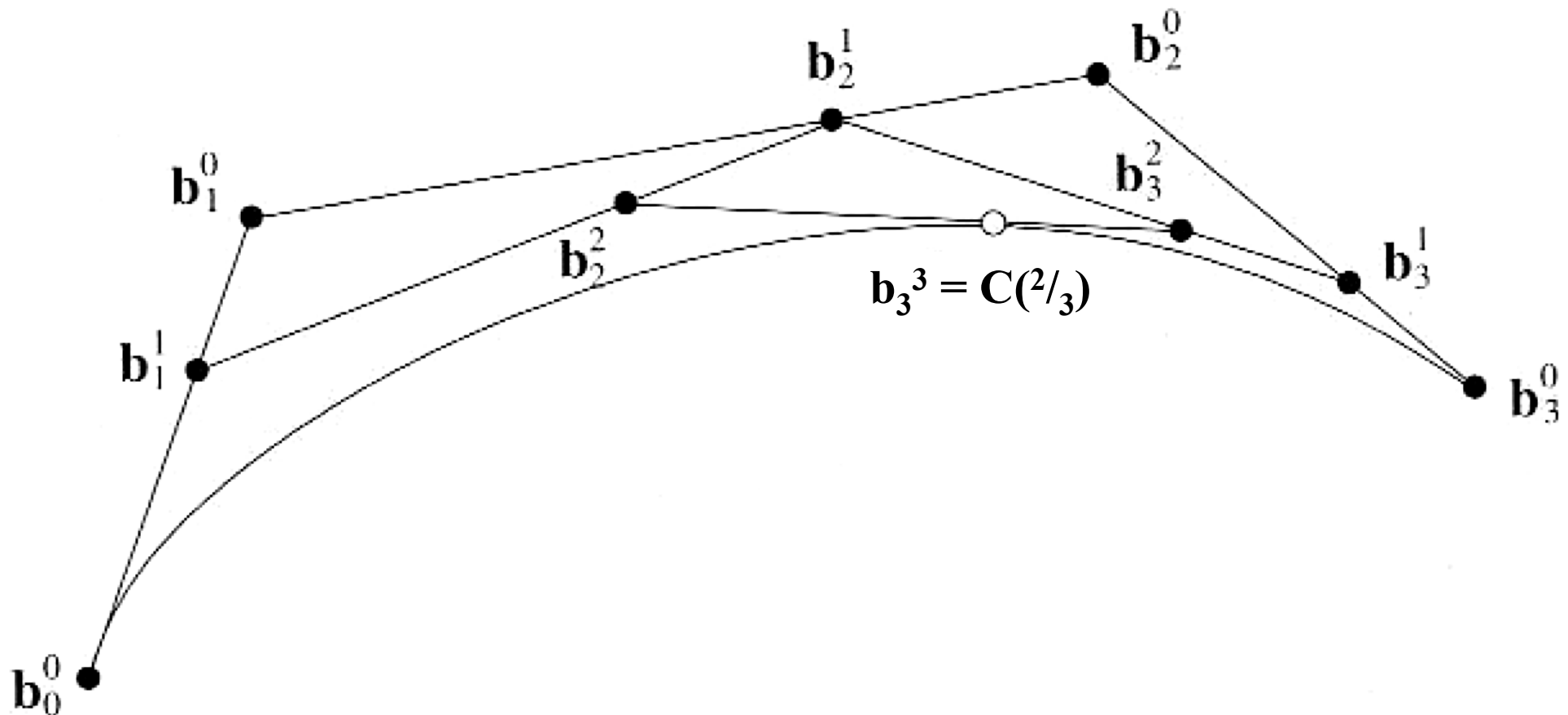
- Oder durch fortgesetzte lineare Interpolation:
der **Algorithmus von de Casteljau** (stabil)



- Oder durch fortgesetzte lineare Interpolation:
der **Algorithmus von de Casteljau** (stabil)



- der **Algorithmus von de Casteljau** (geometrisch)



Algorithmus von *de Casteljau* für $t = 2/3$ und $n = 3$

- Algorithmus von *de Casteljau*: PseudoCode

```
# Initialisierung
for i = 0..n
     $b_i^0 = b_i;$ 

# deCasteljau Pyramide/Dreieck
for k = 1..n
    for i = k..n
         $b_i^k = (1-t) * b_{i-1}^{k-1} + t * b_i^{k-1}$ 

return  $b_n^n$  # Kurvenpunkt C(t)
```

Gegeben n Stützstellen $\{x_1, x_2, \dots, x_n\}$, die Werte sind Samples einer $f : y_i = f(x_i)$. Falls f n -mal differenzierbar ist, gilt für das Interpolationspolynom $p(x)$:

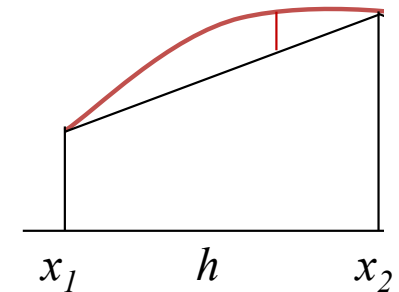
$$f(x) - p(x) = (x - x_1) \cdot (x - x_2) \cdot \dots \cdot (x - x_n) \cdot \frac{f^{(n)}(\xi)}{n!}$$

dabei ist ξ ein Wert zwischen $\min\{x, x_1, x_2, \dots, x_n\}$ und $\max\{x, x_1, x_2, \dots, x_n\}$

Speziell $n=2$: lineare Interpolation

$$f(x) - p(x) = (x - x_1)(x - x_2) \frac{f^{(2)}(\xi)}{2} \quad \text{folglich}$$

$$|f(x) - p(x)| \leq \frac{h^2}{8} \max\{|f^{(2)}(\xi)|\}$$



- Verlauf der Funktion $|w(x)|$
- $w(x)$ oszilliert und wächst zu den Rändern hin (und über die Ränder hinaus, wenn man „Extrapolation“ machen möchte) stark an
- Dort muss mit sehr viel größeren Rekonstruktionsfehlern gerechnet werden!

$$w(x) = (x - x_1) \cdot (x - x_2) \cdot \dots \cdot (x - x_n)$$

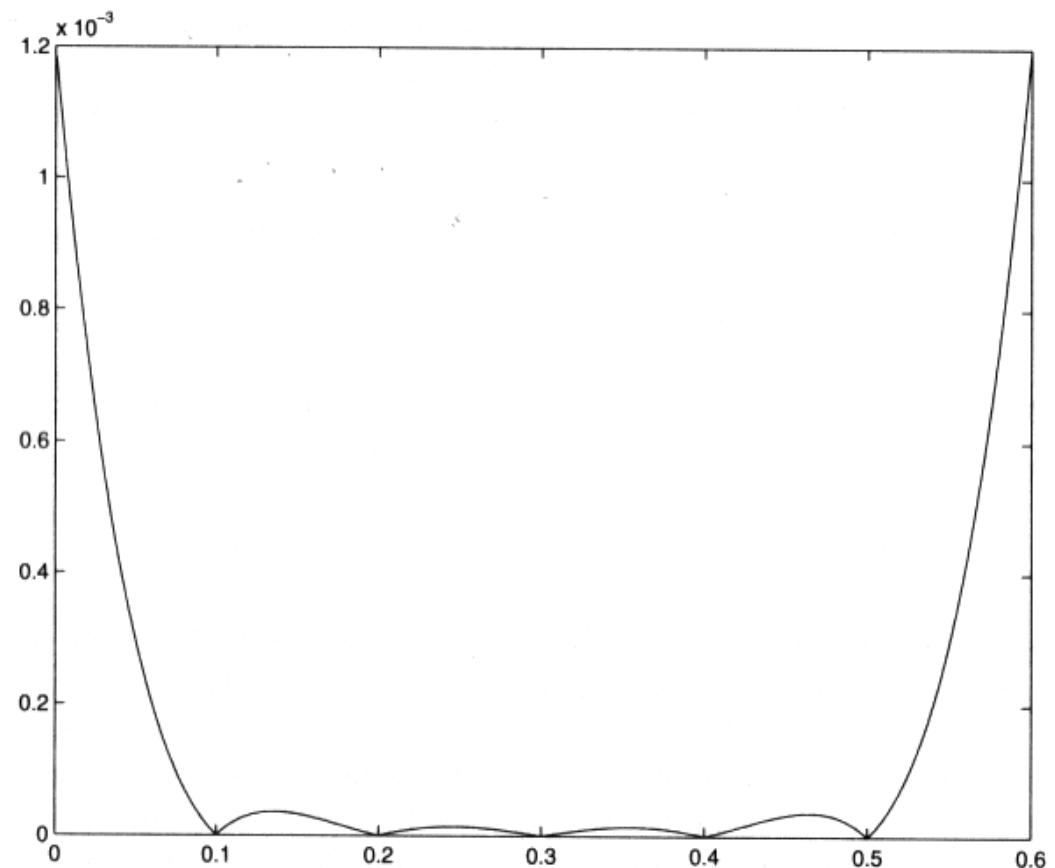
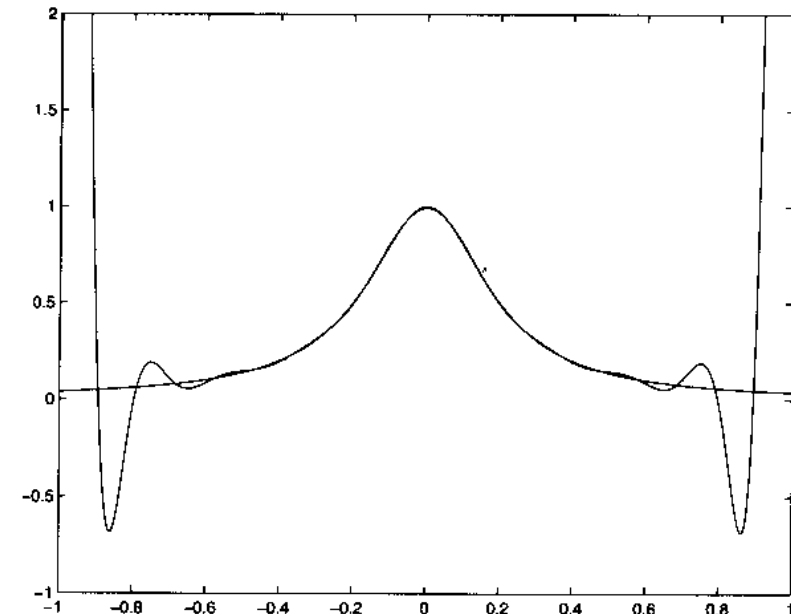


Abb. 14.5. Verlauf der Funktion $w(x)$ zu den Stützstellen 0.1, 0.2, 0.3, 0.4 und 0.5.

- Ein Beispiel : Runge Funktion

$$f(x) = \frac{1}{1 + 25x^2}, \quad (x \in [-1, 1])$$

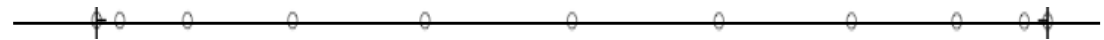
- ▶ 21 äquidistante Stützstellen im Intervall $[-1, 1]$
- ▶ Polynominterpolant vom Grad 20



- Selbst bei Verwendung von mehr äquidistanten Stützstellen wird das Ergebnis nicht besser (**keine Konvergenz!**)

- Ein Beispiel : Runge Funktion $f(x) = \frac{1}{1 + 25x^2}$, $(x \in [-1, 1])$
- ★ Ausweg in diesem Fall: Verwendung nicht äquidistanter Stützstellen sog. Tschebycheff-Stützstellen

$$x_i = -\cos\left(\frac{i-1}{n-1} \cdot \pi\right), \quad i = 1, 2, \dots, n$$



- In der Praxis oft die Lösung:
Vermeide Polynominterpolation für größeres n
- Verwende alternative Verfahren:
 - Catmull-Rom
 - B-Spline Interpolation

- Polynome sind zur Interpolation/Rekonstruktion geeignet, aber nur wenn der Polynomgrad nicht zu hoch ist.
- Andernfalls
 - Wird das Gleichungssystem für die Polynomkoeffizienten sehr schlecht konditioniert
 - Der Fehler oszilliert und wird an den Rändern des Interpolationsintervalls sehr groß
 - Man kann diese Probleme durch geeignete andere Wahl der Stützstellen (teilweise) kompensieren.
- Lagrange-Polynome für theoretische Untersuchungen hilfreich aber ungeeignet zur numerischen Behandlung
- Andere Polynom-Basen (Bernstein, Lagrange, Newton)
- Praktische Berechnung: Algorithmus von Aitken-Neville
 - numerisch stabil und Aufwand $O(n^2)$
 - Effiziente Auswertung mittels Hornerschema

Konditionszahl
(bzgl. Euklidischer
Norm) für
**Vandermonde-
Matrix** und
**„Newton“-
Matrix** für n
äquidistante
Stützstellen:
 $\{0,1,2,\dots,n-1\}$

n	$\kappa(M_{\text{Vandermonde}})$	$\kappa(M_{\text{Newton}})$	$\kappa(M_N) / \kappa(M_V)$
3	13.91	6.18	0.444
4	154.46	18.69	0.121
5	2592.89	74.17	0.029
6	57688.70	368.83	0.639E-2
7	1597316.16	2206.21	0.138E-2
8	52937735	15415.01	0.291E-3
9	2043725290	123173.74	0.603E-4
10	0.900778E11	1107665.81	0.123E-4
11	0.446282E13	11070260.73	0.123E-4
12	0.245502E15	121720944.9	0.496E-6
13	0.148460E17	1460178399	0.984E-7
14	0.979006E18	0.189775E11	0.194E-7
15	0.702701E20	0.265633E12	0.378E-8