Interpolation et Régression Linéaires

by Loïc Quertenmont, PhD

LINFO1113 - 2019-2020

Programme

Cours	1	Librairies mathématiques, représentation des nombres en Python et erreurs liées
Cours	2,3	Résolution des systèmes linéaires
Cours	**4,5**	**Interpolation et Régression Linéaires**
Cours	6,7	Zéro d'équation
Cours	8,9	Différentiation numérique
Cours	10	Intégration numérique
Cours	11,12	Introduction à l'optimisation
Cours	13	Rappel / Répétition

Outline

- **Introduction**
- **Polynomial interpolation**
  - **Lagrange**
  - **Newton**
  - **Neville**
  - **Radial Function Interpolation**
- **Cubic Spline**
- **Least-Square Fit**
- **B-Spline**

Cubic Spline Interpolation

Limit of the interpolation

- We have seen how to interpolate a function $g(x)$ by a polynomial $f^{(n)}(x) = \sum_{i=0}^{n} a_i x^i$
- The interpolation error

$$e^{(n)}(x) = g(x) - f^{(n)}(x)$$

depends on the number n of data points used for the interpolation.

  - We saw that the error could actually increases with the number of points → Runge Phenomena

- In practice, we don't want to work with polynomials with a high level degree

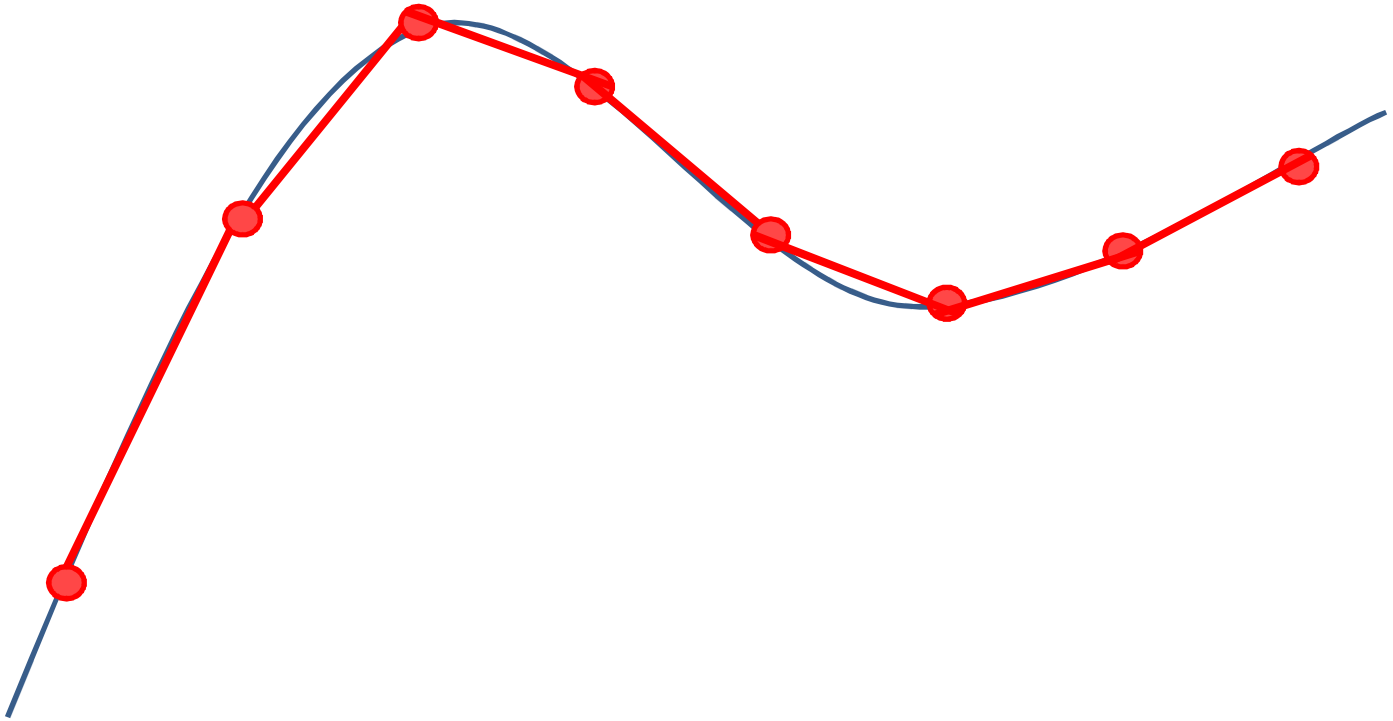
Piecewise Interpolation

- Given n data points

- **Global Interpolation** *of* $g(x)$ = Approximate $g(x)$ by a polynomial $f(x)$ that passes through all n points

- **Piecewise Interpolation** = Approximate $g(x)$ *by a function* $f(x)$ *that consists of* $n-1$ *polynomials of identical degree, each one interpolating a small piece of* $g(x)$

Piecewise Interpolation: Linear



- Quite easy to implement
- Result is not smooth (not differentiable in the data points)

Cubic spline

- A **cubic spline** uses cubic polynomials to interpolate the intervals $[x_i, x_{i+1}]$ between two neighbor data points (x_i, y_i) and (x_{i+1}, y_{i+1})

- But: There is an infinite number of cubic polynomials through two points!

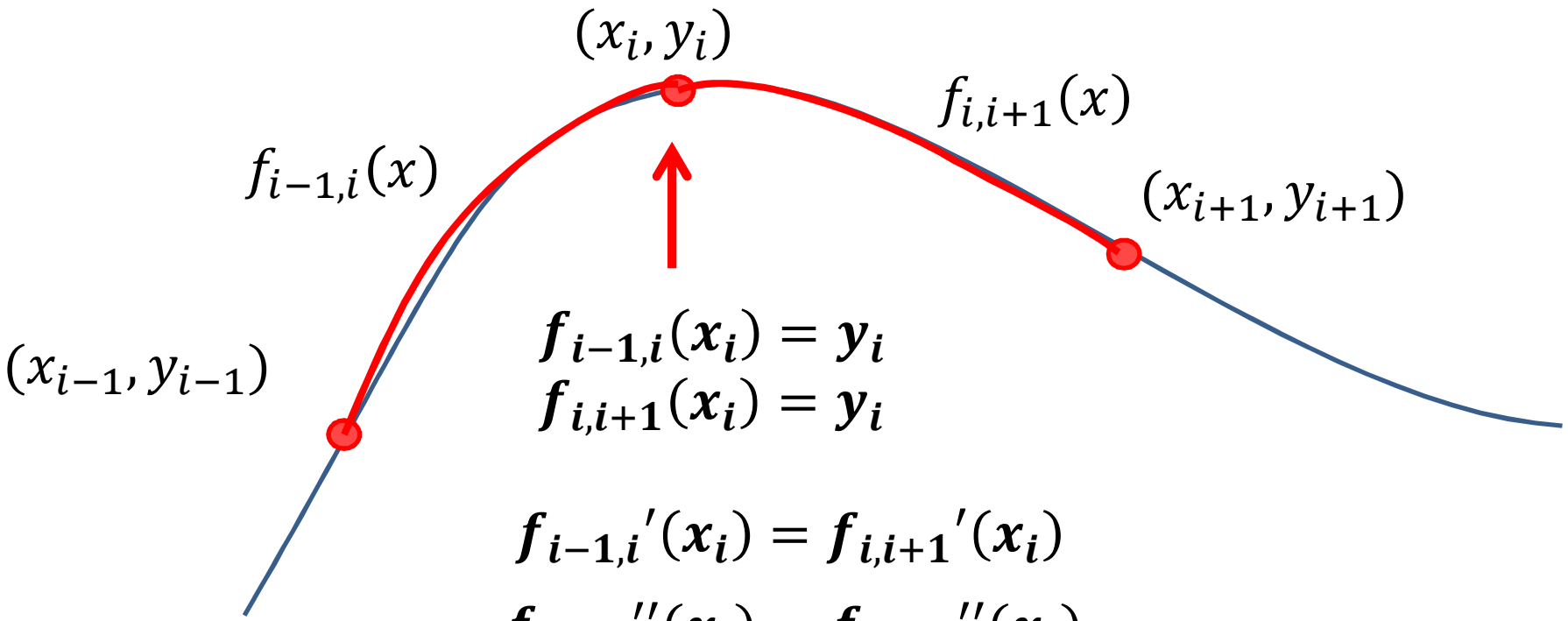
- We will choose the cubic polynomials such that the final result spline $f(x)$ is twice differentiable
  - Twice differentiable is "smooth enough" for many real world applications
  - Imagine a robot following a path: no abrupt turns or speed changes

Polynomials for Cubic Splines

- For two points (x_i, y_i) and (x_{i+1}, y_{i+1}) we want to find the cubic polynomial $f_{i,i+1}(x)$ such that
  - $f_{i,i+1}(x_i) = y_i$
  - $f_{i,i+1}(x_{i+1}) = y_{i+1}$

  and the smoothness requirements such that the resulting spline is twice differentiable in x_i:
  - ${f_{i,i+1}}'(x_i) = {f_{i-1,i}}'(x_i)$
  - ${f_{i,i+1}}''(x_i) = {f_{i-1,i}}''(x_i)$

Polynomials for Cubic Splines (2)



- **The behavior is very similar to a beam string being fixed at each knot i by a pin**
- **There is no bending moment at the two end pins**
  - **Second derivative is null at the end points**

Cubic Splines Development (1)

- What are the coefficients of each piece-wise polynomials ?

- Let's call $\mathbf{k_i}$ the second derivative at knot I
  - $\boldsymbol{k_i} \coloneqq \boldsymbol{f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i)}$

- Second derivative at end points is null
  - $\boldsymbol{k_0 = k_n = 0}$

- Second derivative between two points is linear
  - Using Lagrange two point interpolation, we have
  - $\boldsymbol{f''_{i,i+1}(x) = k_i l_i(x) + k_{i+1} l_{i+1}(x)}$
  - with $l_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}}$ and $l_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i}$

Cubic Splines Development (2)

- If we expand, we have

$$\boldsymbol{f''_{i,i+1}(x) = \frac{k_i(x-x_{i+1})-k_{i+1}(x-x_i)}{x_i-x_{i+1}}}$$

- Integrating twice, we have

$$f_{i,i+1}(x) = \frac{k_i(x-x_{i+1})^3-k_{i+1}(x-x_i)^3}{6(x_i-x_{i+1})} + Cx + D$$

Where $Cx + D$ are integration constant

- This can be rewritten as ($\boldsymbol{C = A - B}$ and $\boldsymbol{D = -Ax_{i+1} + Bx_i}$)

$$f_{i,i+1}(x) = \frac{k_i(x-x_{i+1})^3-k_{i+1}(x-x_i)^3}{6(x_i-x_{i+1})} + A(x - x_{i+1}) + B(x - x_i)$$

Cubic Splines Development (3)

- $f_{i,i+1}(x) = \frac{k_i(x-x_{i+1})^3 - k_{i+1}(x-x_i)^3}{6(x_i-x_{i+1})} + \boldsymbol{A}(x - x_{i+1}) + \boldsymbol{B}(x - x_i)$

- Imposing $\boldsymbol{f_{i,i+1}(x_i) = y_i}$
  - $f_{i,i+1}(x_i) = \frac{k_i(x_i-x_{i+1})^3}{6(x_i-x_{i+1})} + A(x_i - x_{i+1}) = y_i$

$$\rightarrow \boldsymbol{A = \frac{y_i}{(x_i-x_{i+1})} - \frac{k_i}{6}(x_i - x_{i+1})}$$

- Imposing $\boldsymbol{f_{i,i+1}(x_{i+1}) = y_{i+1}}$
  - $f_{i,i+1}(x_{i+1}) = \frac{-k_{i+1}(x_{i+1}-x_i)^3}{6(x_i-x_{i+1})} + B(x_{i+1} - x_i) = y_{i+1}$

$$\rightarrow \boldsymbol{B = \frac{y_{i+1}}{(x_i-x_{i+1})} - \frac{k_{i+1}}{6}(x_i - x_{i+1})}$$

- Replacing A and B by their values
  - $$f_{i,i+1}(x) = \frac{k_i}{6}\left[\frac{(x-x_{i+1})^3}{x_i-x_{i+1}} - (x - x_{i+1})(x_i - x_{i+1})\right] - \frac{k_{i+1}}{6}\left[\frac{(x-x_i)^3}{x_i-x_{i+1}} - (x - x_i)(x_i - x_{i+1})\right] + \frac{y_i(x-x_{i+1}) - y_{i+1}(x-x_i)}{x_i-x_{i+1}}$$

Cubic Splines Development (4)

- **We can find the solution for k_i by imposing $f'_{i-1,i}(x_i) = f'_{i,i+1}(x_i)$**
  - **After a lot of algebra, we get:**

$$k_{i-1}(x_{i-1} - x_i) + 2k_i(x_{i-1} - x_{i+1}) + k_{i+1}(x_i - x_{i+1})$$
$$= 6\left(\frac{y_{i-1}-y_i}{x_{i-1}-x_i} - \frac{y_i-y_{i+1}}{x_i-x_{i+1}}\right) \quad \text{for } i = 1, 2, \ldots, n-1$$

- **This is a system of linear equation in k_i which has the form of a tri-diagonal matrix.**
  - **This is something we can solve easily with the LUdecomp3 algorithm we have seen.**

- **In the particular case, where the data are evenly spaced**

$$x_{i-1} - x_i = x_i - x_{i+1} = -h$$

- **The equations simplifies to**

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2}(y_{i-1} - 2y_i + y_{i+1}) \quad \text{for } i = 1, 2, \ldots, n-1$$

Cubic Splines Algorithm

- Making an algorithm
  - They are two steps:
    1. **Finding the curvature coefficients (k_i)**
    2. Evaluating the function $f_{i,i+1}(x)$ at x

```
import numpy as np
from LUdecomp3 import *

def curvatures(xData,yData):
    n = len(xData) - 1
    c = np.zeros(n)
    d = np.ones(n+1)
    e = np.zeros(n)
    k = np.zeros(n+1)
    c[0:n-1] = xData[0:n-1] - xData[1:n]
    d[1:n] = 2.0*(xData[0:n-1] - xData[2:n+1])
    e[1:n] = xData[1:n] - xData[2:n+1]
    k[1:n] =6.0*(yData[0:n-1] - yData[1:n]) \
                /(xData[0:n-1] - xData[1:n]) \
            -6.0*(yData[1:n] - yData[2:n+1])   \
                /(xData[1:n] - xData[2:n+1])
    LUdecomp3(c,d,e)
    LUsolve3(c,d,e,k)
    return k
```

- We search x such that Ax = b, where
  A is tri-diagonal
  b is defined by data
- The algorithm is uses the LUdecomp3 function that we developed to solve tri-diagonal linear systems
- The rest of the code is just shaping the data into the three diagonal vectors c,d and e AND the constant vector k

Cubic Splines Algorithm

- Making an algorithm
  - They are two steps:
    1. Finding the curvature coefficients (k_i)
    2. **Evaluating the function $f_{i,i+1}(x)$ at x**

```
def evalSpline(xData,yData,k,x):

   def findSegment(xData,x):
       iLeft = 0
       iRight = len(xData)- 1
       while 1:
           if (iRight-iLeft) <= 1: return iLeft
           i =(iLeft + iRight)/2
           if x < xData[i]: iRight = i
           else: iLeft = i

   i = findSegment(xData,x)
   h = xData[i] - xData[i+1]
   y = ((x - xData[i+1])**3/h - (x - xData[i+1])*h)*k[i]/6.0 \
     - ((x - xData[i])**3/h - (x - xData[i])*h)*k[i+1]/6.0   \
     + (yData[i]*(x - xData[i+1])                            \
      - yData[i+1]*(x - xData[i]))/h
   return y
```

- We identify on which segment x lies This is done via the bisection method *The method will be discussed later*
- Then we simply evaluate the function $f_{i,i+1}(x)$ with the $x_i, y_i, x_{i+1}, y_{i+1}$ corresponding to this segment and the specific value of x

Cubic Splines Example (1)

- **Data:**
- **What is the value of f at x=1.5**

x	1	2	3	4	5
y	0	1	0	1	0

- **Evenly spaced, with h = 1 →**

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2}(y_{i-1} - 2y_i + y_{i+1})$$

- $\boldsymbol{k_0 = 0}$
- $\boldsymbol{k_0 + 4\,k_1 + k_2 = 6(0 - 2 + 0) = -12}$
- $\boldsymbol{k_1 + 4\,k_2 + k_3 = 6(1 - 0 + 1) = 12}$
- $\boldsymbol{k_2 + 4\,k_3 + k_4 = 6(0 - 2 + 0) = -12}$
- $\boldsymbol{k_4 = 0}$

$$\boldsymbol{k_1 = k_3 = -30/}$$
$$\boldsymbol{k_2 = 36/7}$$

Cubic Splines Example (2)

- **X = 1.5 lies on the segment between knots 0 and 1**

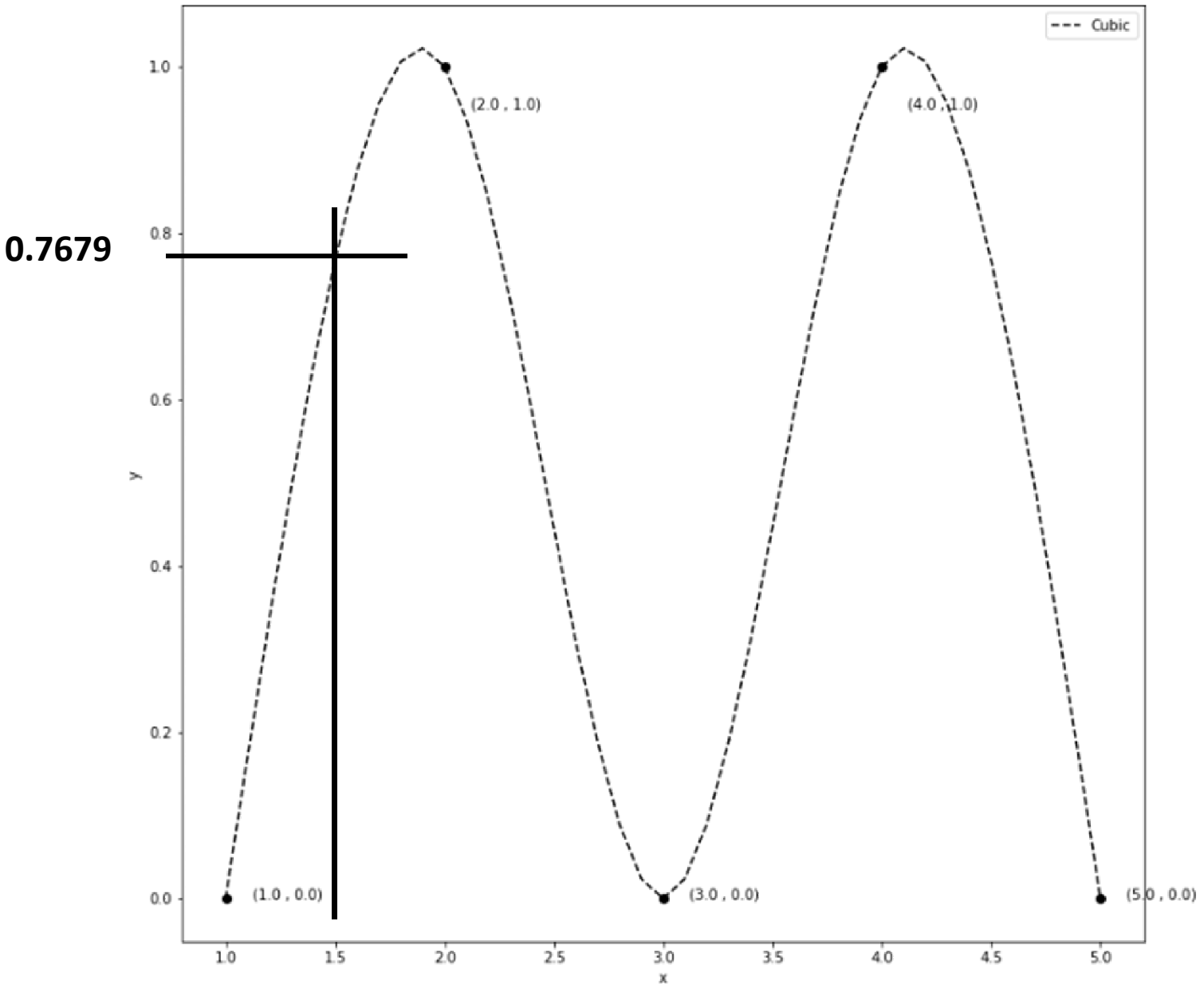
$$f_{i,i+1}(x) = \frac{k_i}{6}\left[\frac{(x-x_{i+1})^3}{x_i - x_{i+1}} - (x-x_{i+1})(x_i - x_{i+1})\right] - \frac{k_{i+1}}{6}\left[\frac{(x-x_i)^3}{x_i - x_{i+1}} - (x-x_i)(x_i - x_{i++1})\right] + \frac{y_i(x-x_{i+1}) - y_{i+1}(x-x_i)}{x_i - x_{i+1}}$$

$$f_{0,1}(x) = \frac{k_1}{6}[(x-x_0)^3 - (x-x_0)] - y_0(x-x_1) + y_1(x-x_0)$$

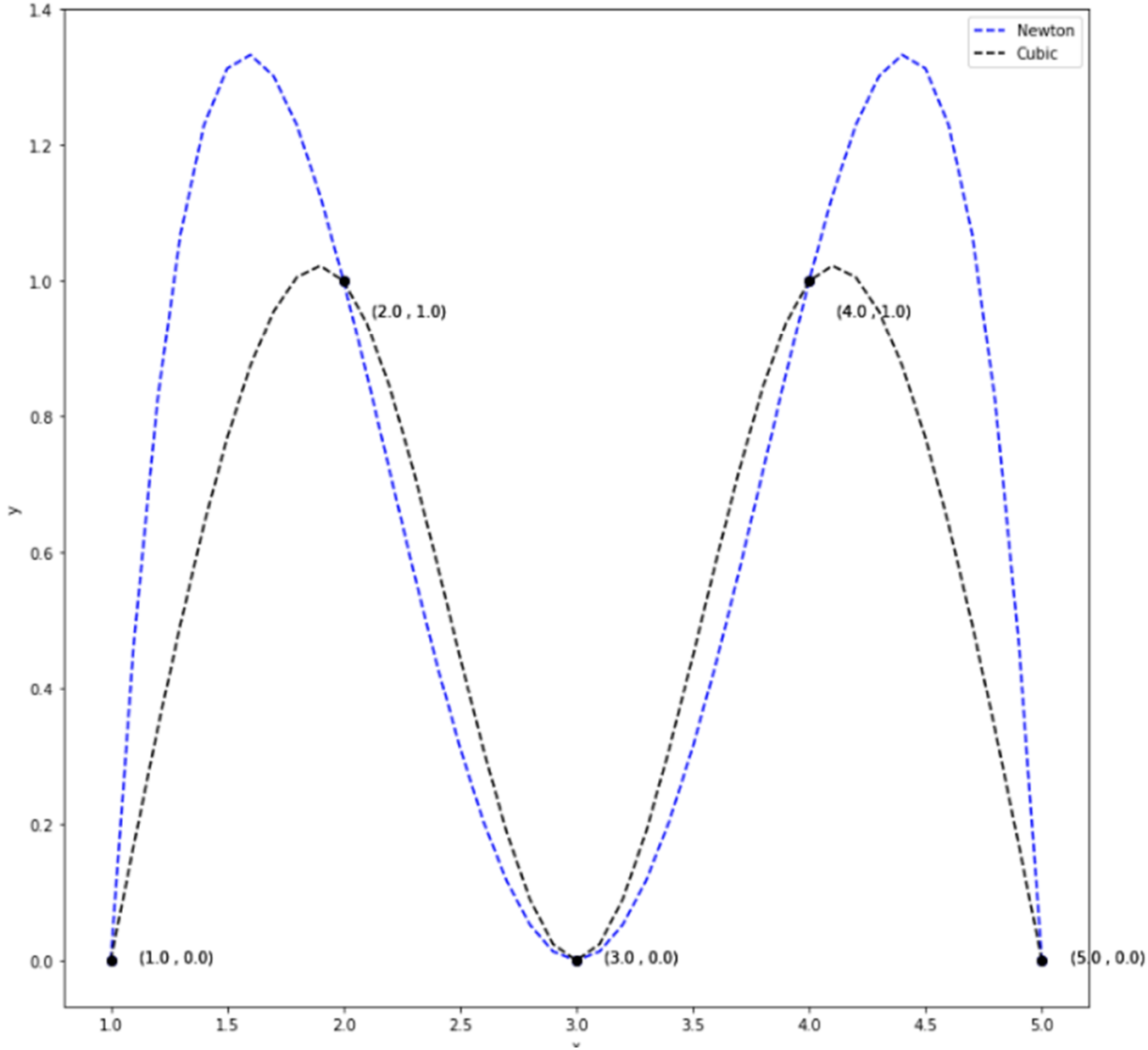
$$f_{0,1}(1.5) = \frac{1}{6}\frac{-30}{7}[(1.5-1)^3 - (1.5-1)] - 0 + 1(1.5-1)$$

$$f_{0,1}(1.5) = 0.7679$$

Cubic Splines Example (3)



Cubic Splines Example (3)



Border conditions

- Sometimes it is preferable to replace one or both of the end condition ($k_0 = k_n = 0$)

- We can for instance impose on the starting point that
  - $\boldsymbol{f'_{0,1}(x) = 0}$ (zero slope) instead of
  - $\boldsymbol{f''_{0,1}(x) = 0}$ (zero curvature)

- $\boldsymbol{f'_{0,1}(x)} = \frac{k_0}{6}\left[\frac{3(x-x_1)^2}{x_0-x_1} - (x_0 - x_1)\right] - \frac{k_1}{6}\left[\frac{3(x-x_0)^2}{x_0-x_1} - (x_0 - x_1)\right] + \frac{y_0-y_1}{x_0-x_1}$

- $\boldsymbol{f'_{0,1}(x_0)} = 0 \rightarrow \frac{k_0}{3}(x_0 - x_1) + \frac{k_1}{6}(x_0 - x_1) + \frac{y_0-y_1}{x_0-x_1} = 0$

- $\boldsymbol{2k_0 + k_1 = \frac{y_0-y_1}{(x_0-x_1)^2}}$

Border conditions

- **If we solve for specific data at x=2.6:**
- $2k_0 + k_1 = \frac{y_0 - y_1}{(x_0 - x_1)^2} = 0$

x	0	1	2	3
y	1	1	0.5	0

- The other curvature equations are unchanged (evenly spaced):
  - $k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2}(y_{i-1} - 2y_i + y_{i+1})$
  - $k_3 = 0$ (zero curvature at the end)

- $2k_0 + k_1 = 0$
- $k_0 + 4k_1 + k_2 = -3$
- $k_1 + 4k_2 + k_3 = 0$
- $k_3 = 0$

$$k_0 = 0.461^5$$
$$k_1 = -092^3$$
$$k_2 = 0.230^8$$
$$k_3 = 0$$

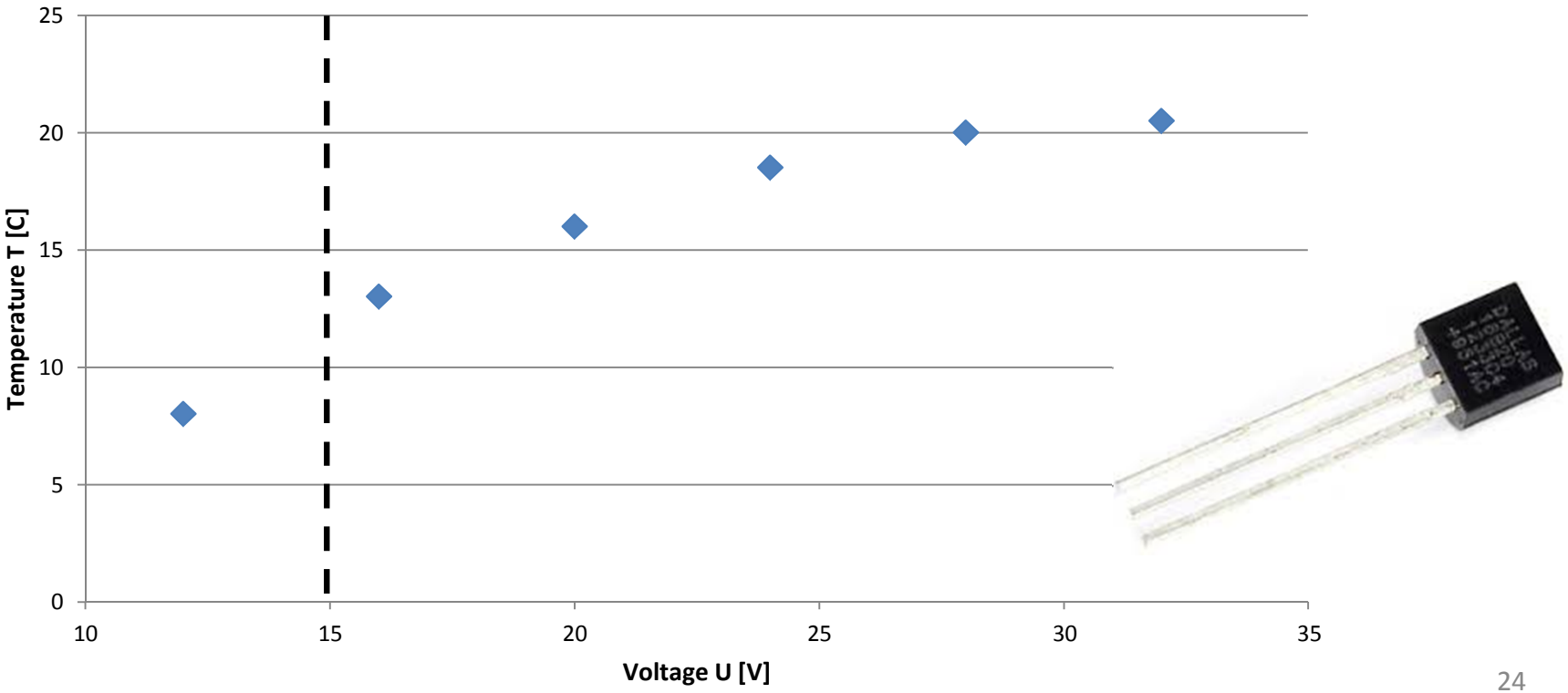
$$f_{2,3}(x) = \frac{k_2}{6}[-(x - x_3)^3 + (x - x_3)] - \frac{k_3}{6}[-(x - x_2)^3 + (x - x_2)] - y_2(x - x_3) + y_3(x - x_2)$$

$$f_{2,3}(x) = \frac{0.230^8}{6}[-(-0.4)^3 + (-0.4)] - 0 - 0.5(-0.4) + 0 = \mathbf{0.187^1}$$

Least-Squares Fit

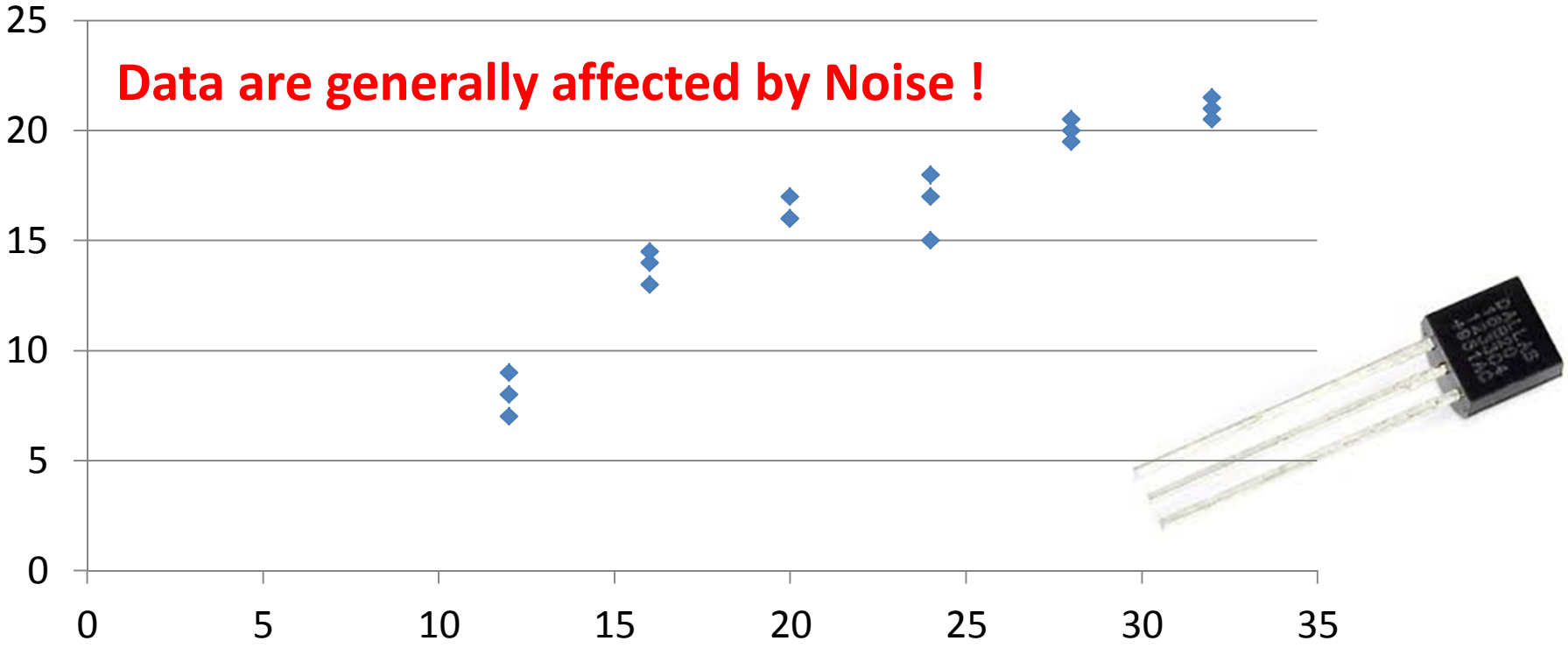
Recall the Example

- An electronic temperature sensor outputs a voltage U that depends on the temperature T
  - The response T = f(U) is generally **non linear**
  - **What is the temperature for U=15V ?**



Example

- Often, when we perform the calibration, we get slightly different response curves.
  - Bellow, we performed the calibration 3 times.
  - **What value should we use for the interpolation ?**



Reminder

- **Interpolation** (and Extrapolation)
  - The curve pass through the points (x_i, y_i)
  - $y_i = f(x_i)$

- **Curve fitting**
  - The curve is **smooth** but does not necessary go through the data points
  - The curve “approximate” the data
  - $y_i = f(x_i) + \varepsilon_i$

Curve fitting (1)

- If the data are obtained from experiments, they typically contain a significant amount of random noise caused by measurement errors.

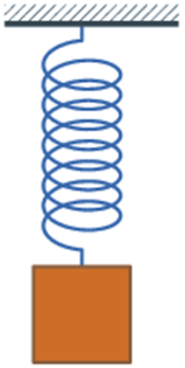
- We want to find a **smooth curve** that fits the data points **“on average”**.

- This curve should have a simple form (i.e. low-order polynomial), so as not to reproduce the noise in the data

Curve fitting (2)

- Let's define the function to be fitted through the n+1 data points (x_i, y_i)
- $\boldsymbol{f(x) = f(x; a_0, a_1, \dots a_m)}$
  - $\boldsymbol{f}$ depends on m+1 variables ($\boldsymbol{a_i}$)
  - $m < n$
  - The form of $\boldsymbol{f}$ could be anything, and is generally defined by the theory behind the data
    - I.e, for a mass-spring system

$$f(t) = a_0 t e^{-a_1 t}.$$



  - The only question we are trying to solve is… what are the parameters $\boldsymbol{a_i}$ that fits best the data

Least-Square (1)

- What is meant by the “best” fit?

- If the noise is confined to the y-coordinate:

$$x_i = \tilde{x}_i$$
$$y_i = \tilde{y}_i + \varepsilon_i$$

- The most commonly used measure is the least-squares fit, Where we minimize

$$S(a_0, a_1, \ldots, a_m) = \sum_{i=0}^{n} [y_i - f(x_i)]^2 \qquad \approx \sum_i {r_i}^2$$

r_i is called the residuals.
It’s the discrepancy between the data points and the fitted function

Least-Square (2)

- Optimal values for the parameters $\boldsymbol{a_k}$ is given by

$$\frac{\partial S}{\partial a_k} = 0, \quad k = 0, 1, \ldots, m.$$

- The function is often non-linear in $\boldsymbol{a_k}$ and may be difficult to solve (in general)

- Often, $\boldsymbol{f}$ is chosen to be a linear combination of $\boldsymbol{f_j(x)}$

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + \cdots + a_m f_m(x)$$

- If we take, $f_0(x) = 1, f_1(x) = x, f_2(x) = x^2$, and so on.
  We have a polynomial fit

Least-Square (3)

- The spread of the data about the fitting curve is quantified by the **standard deviation**, defined as

$$\sigma = \sqrt{\frac{S}{n-m}}$$

- **In the case, $n = m$**
  - We have an **interpolation**
  - Both the numerator and the denominator of σ are zero
    - σ is indeterminate

Linear Regression

- Fitting a straight line to the data
  - This is known as a “linear regression”
  - $f(x) = a + bx$
- Least Square:

$$S(a, b) = \sum_{i=0}^{n} [y_i - f(x_i)]^2 = \sum_{i=0}^{n} (y_i - a - bx_i)^2$$

- Minimizing

$$\frac{\partial S}{\partial a} = \sum_{i=0}^{n} -2(y_i - a - bx_i) = 2\left[a\,(n+1) + b\sum_{i=0}^{n} x_i - \sum_{i=0}^{n} y_i\right] = 0$$

$$\frac{\partial S}{\partial b} = \sum_{i=0}^{n} -2(y_i - a - bx_i)x_i = 2\left(a\sum_{i=0}^{n} x_i + b\sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i\right) = 0$$

Linear Regression

- Dividing both equations by 2 (n + 1) and rearranging terms, we get

$$a + \bar{x}b = \bar{y} \qquad \bar{x}a + \left(\frac{1}{n+1}\sum_{i=0}^{n} x_i^2\right) b = \frac{1}{n+1}\sum_{i=0}^{n} x_i y_i$$

- Where we define, the mean of the x_i and y_i

$$\bar{x} = \frac{1}{n+1}\sum_{i=0}^{n} x_i \qquad \bar{y} = \frac{1}{n+1}\sum_{i=0}^{n} y_i$$

- Solving, we get: $a = \dfrac{\bar{y}\sum x_i^2 - \bar{x}\sum x_i y_i}{\sum x_i^2 - n\bar{x}^2}$ $b = \dfrac{\sum x_i y_i - \bar{x}\sum y_i}{\sum x_i^2 - n\bar{x}^2}$
  - This form is susceptible to roundoff errors (the two terms in each numerator as well as in each denominator can be roughly equal)
  - The following form is equivalent, but less affected by roundoff

$$b = \frac{\sum y_i(x_i - \bar{x})}{\sum x_i(x_i - \bar{x})} \qquad a = \bar{y} - \bar{x}b$$

Linear Form (including polynomials)

- General form for $\boldsymbol{f}$, where f_i is a basis function

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + \ldots + a_m f_m(x) = \sum_{j=0}^{m} a_j f_j(x)$$

- Least Square:

$$S = \sum_{i=0}^{n} \left[y_i - \sum_{j=0}^{m} a_j f_j(x_i) \right]^2$$

- Minimization:

$$\frac{\partial S}{\partial a_k} = -2 \left\{ \sum_{i=0}^{n} \left[y_i - \sum_{j=0}^{m} a_j f_j(x_i) \right] f_k(x_i) \right\} = 0, \quad k = 0, 1, \ldots, m$$

- Can be re-arranged (by swapping the sums), as:

$$\sum_{j=0}^{m} \left[\sum_{i=0}^{n} f_j(x_i) f_k(x_i) \right] a_j = \sum_{i=0}^{n} f_k(x_i) y_i, \quad k = 0, 1, \ldots, m$$

Linear Form (including polynomials)

$$\sum_{j=0}^{m}\left[\sum_{i=0}^{n} f_j(x_i) f_k(x_i)\right] a_j = \sum_{i=0}^{n} f_k(x_i) y_i, \quad k = 0, 1, \ldots, m$$

- In matrix notation, we have: **Aa = b** with:

$$A_{kj} = \sum_{i=0}^{n} f_j(x_i) f_k(x_i) \qquad b_k = \sum_{i=0}^{n} f_k(x_i) y_i$$

- This is known as the ***normal equations*** of the least square fit

- It can be solved like a system of linear equation
- Since $A_{kj} = A_{jk}$, Choleski decomposition algorithm can be used

Particular Case: Polynomial Fit

- Fit with a **polynomial of degree m**
- The fitting function has the form: $f(x) = \sum_{j=0}^{m} a_j x^j.$
- The *normal equation* of the least square fit

$$A_{kj} = \sum_{i=0}^{n} f_j(x_i) f_k(x_i) \qquad b_k = \sum_{i=0}^{n} f_k(x_i) y_i$$

- Become

$$A_{kj} = \sum_{i=0}^{n} x_i^{j+k} \qquad b_k = \sum_{i=0}^{n} x_i^k y_i$$

- Or:

$$\mathbf{A} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 & \dots & \sum x_i^m \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \dots & \sum x_i^{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_i^{m-1} & \sum x_i^m & \sum x_i^{m+1} & \dots & \sum x_i^{2m} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^m y_i \end{bmatrix}$$

Polynomial Fit Implementation

```
import numpy as np
import math
from gaussPivot import *

def polyFit(xData,yData,m):
    a = np.zeros((m+1,m+1))
    b = np.zeros(m+1)
    s = np.zeros(2*m+1)
    for i in range(len(xData)):
        temp = yData[i]
        for j in range(m+1):
            b[j] = b[j] + temp
            temp = temp*xData[i]
        temp = 1.0
        for j in range(2*m+1):
            s[j] = s[j] + temp
            temp = temp*xData[i]
    for i in range(m+1):
        for j in range(m+1):
            a[i,j] = s[i+j]
    return gaussPivot(a,b)
```

1. We compute all the term appearing in the vector ‘b’

2. We compute all the UNIQUE terms appearing in the matrix and save them in the ‘s’ vector: $n, \sum x_i, \sum x_i^2, \ldots, \sum x_i^{2m}$

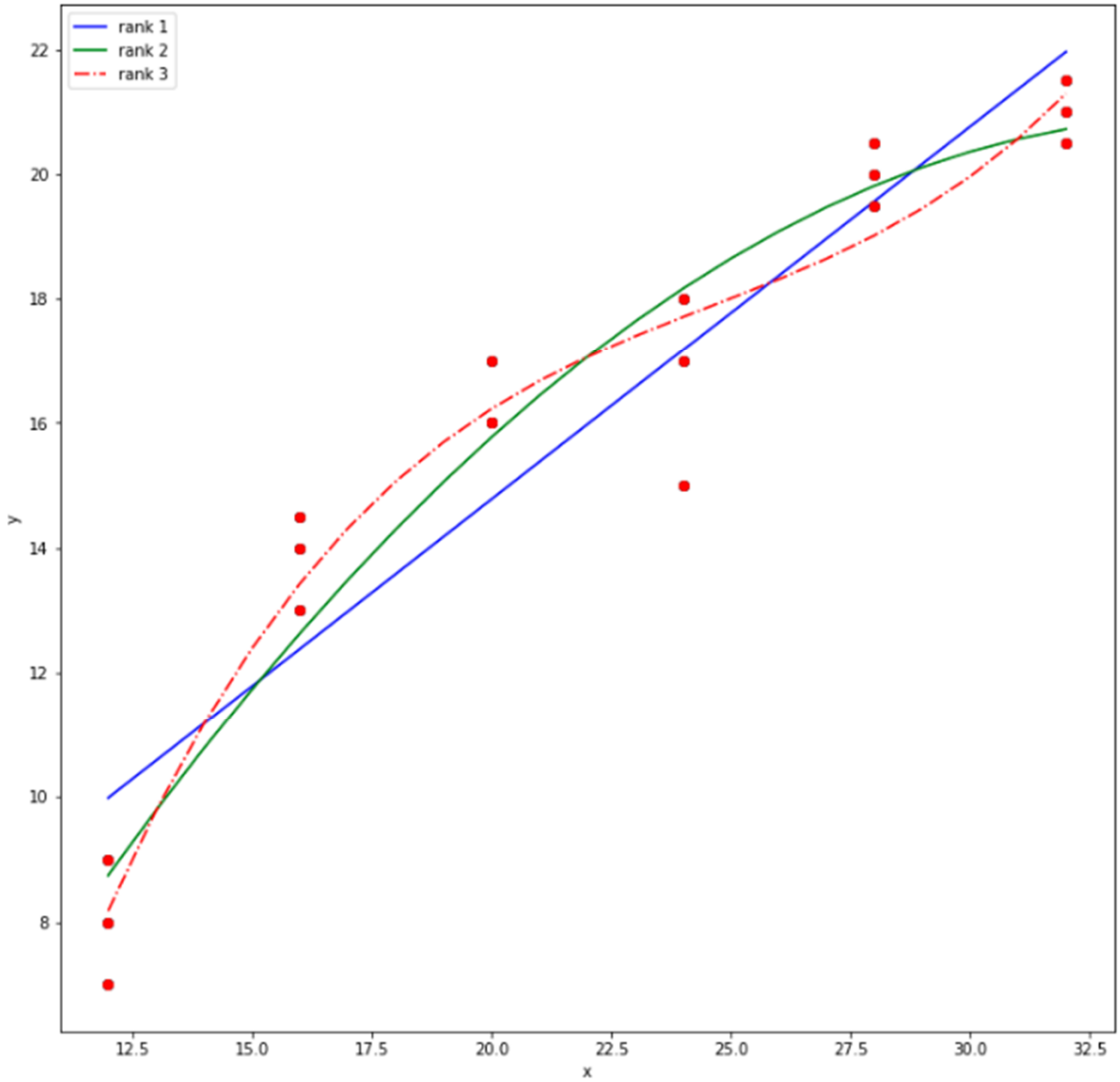
3. We fill the matrix A from the content of the ‘s’ vector

Polynomial Fit Implementation

- Plotting the results
- Once we have the coefficients of the polynomial, evaluating it is simple

```
def plotPoly(xData,yData,coeff,xlab='x',ylab='y'):
    m = len(coeff)
    x1 = min(xData)
    x2 = max(xData)
    dx = (x2 - x1)/20.0
    x = np.arange(x1,x2 + dx/10.0,dx)
    y = np.zeros((len(x)))*1.0
    for i in range(m):
        y = y + coeff[i]*x**i
    plt.plot(xData,yData,'o',x,y,'-')
    plt.xlabel(xlab); plt.ylabel(ylab)
    plt.grid (True)
    plt.show()
```

Fits on the temperature example



- **Pol Coefficients:**
- **Rank 1**
  - **2.798**
  - **0.598**
- **Rank 2**
  - **-7.370**
  - **1.621**
  - **-0.023**
- **Rank3**
  - **-33.67**
  - **5.688**
  - **-0.218**
  - **0.0029**

Data Weighting (1)

- In many cases, the accuracy / uncertainty on the data varies from point to point
  - For instance, if the data are measured with different instruments (analog thermometer vs digital thermometer)

- In that case, we may want to assign a confidence factor (or weight) on each data point: W_i
- In that case, we minimize the sum of the squares of the weighted residuals $r_i = W_i\,[y_i - f(x_i)]$,

$$S(a_0, a_1, \ldots, a_m) = \sum_{i=0}^{n} W_i^2\,[y_i - f(x_i)]^2$$

- **We will force the function f to be closer to data points with a smaller error (higher weight)**

Data Weighting (2)

- In the case of straight line fit:

$$f(x) = a + bx$$

- Least Square:

$$S(a,b) = \sum_{i=0}^{n} W_i^2 (y_i - a - bx_i)^2$$

- Minimization:

$$\frac{\partial S}{\partial a} = -2\sum_{i=0}^{n} W_i^2 (y_i - a - bx_i) = 0 \rightarrow a\sum_{i=0}^{n} W_i^2 + b\sum_{i=0}^{n} W_i^2 x_i = \sum_{i=0}^{n} W_i^2 y_i$$

$$\frac{\partial S}{\partial b} = -2\sum_{i=0}^{n} W_i^2 (y_i - a - bx_i) x_i = 0 \rightarrow a\sum_{i=0}^{n} W_i^2 x_i + b\sum_{i=0}^{n} W_i^2 x_i^2 = \sum_{i=0}^{n} W_i^2 x_i y_i$$

Data Weighting (3)

- If we define the weighted average:

$$\hat{x} = \frac{\sum W_i^2 x_i}{\sum W_i^2} \qquad \hat{y} = \frac{\sum W_i^2 y_i}{\sum W_i^2}$$

- We obtain

$$b = \frac{\sum W_i^2 y_i (x_i - \hat{x})}{\sum W_i^2 x_i (x_i - \hat{x})} \qquad a = \hat{y} - b\hat{x}.$$

- Note that this is very similar to what we obtained for unweighted case:

$$b = \frac{\sum y_i (x_i - \bar{x})}{\sum x_i (x_i - \bar{x})} \qquad a = \bar{y} - \bar{x}b$$

Multi-Variate Linear Regression (1)

- General form for $\boldsymbol{f}$, where f_i is a basis function

$$\mathbf{Y} = f(\mathbf{x}) = \sum_{i=0}^{m} a_m x_m = \mathbf{XA}$$

  - Where $\mathbf{X} \in \mathbb{R}^{m \times n}$, $\mathbf{Y} \in \mathbb{R}^{m}$ and $\mathbf{A} \in \mathbb{R}^{n}$
  - **Where $x_0 = 1$ (to get the constant term a_0)**

- Least Square:

$$\begin{aligned} S &= \|\mathbf{Y} - \mathbf{XA}\|_2^2 \\ &= (\mathbf{Y} - \mathbf{XA})^{\mathrm{T}}(\mathbf{Y} - \mathbf{XA}) \\ &= \left(\mathbf{Y^T} - \mathbf{A^T X^T}\right)(\mathbf{Y} - \mathbf{XA}) \\ &= \mathbf{Y^T Y} - \mathbf{Y^T XA} - \mathbf{A^T X^T Y} + \mathbf{A^T X^T XA} \end{aligned}$$

All those terms are scalars (1x1 matrices), thus:

$$S = \mathbf{Y^T Y} + \mathbf{A^T X^T XA} - \mathbf{2A^T X^T Y}$$

- Minimization:

$$\frac{\partial(\mathbf{Y^T Y} + \mathbf{A^T X^T XA} - \mathbf{2A^T X^T Y})}{\partial \mathbf{A}} = 0$$

Multi-Variate Linear Regression (2)

- Minimization:

$$\frac{\partial(\mathbf{Y^T Y} + \mathbf{A^T X^T X A} - \mathbf{2 A^T X^T Y})}{\partial \mathbf{A}} = 0$$

- Which means:

$$\frac{\partial(\mathbf{Y^T Y} + \mathbf{A^T X^T X A} - \mathbf{2 A^T X^T Y})}{\partial a_i} = 0 \quad \text{for i} = 1, \ldots, \text{n}$$

- which gives (after a lot of calculations):

$$\mathbf{2 X^T X A} - \mathbf{2 X^T Y} = \mathbf{0}$$

- Or: Solve A with cholesky

$$\mathbf{X^T X A} = \mathbf{X^T Y} \quad \leftrightarrow \quad \mathbf{A} = (\mathbf{X^T X})^{-1} \mathbf{X^T Y}$$

- These are again the **normal equation** of the least square fit

Multi-Variate Linear Regression (3)

Remarks on Multi-Variate Linear Regression:

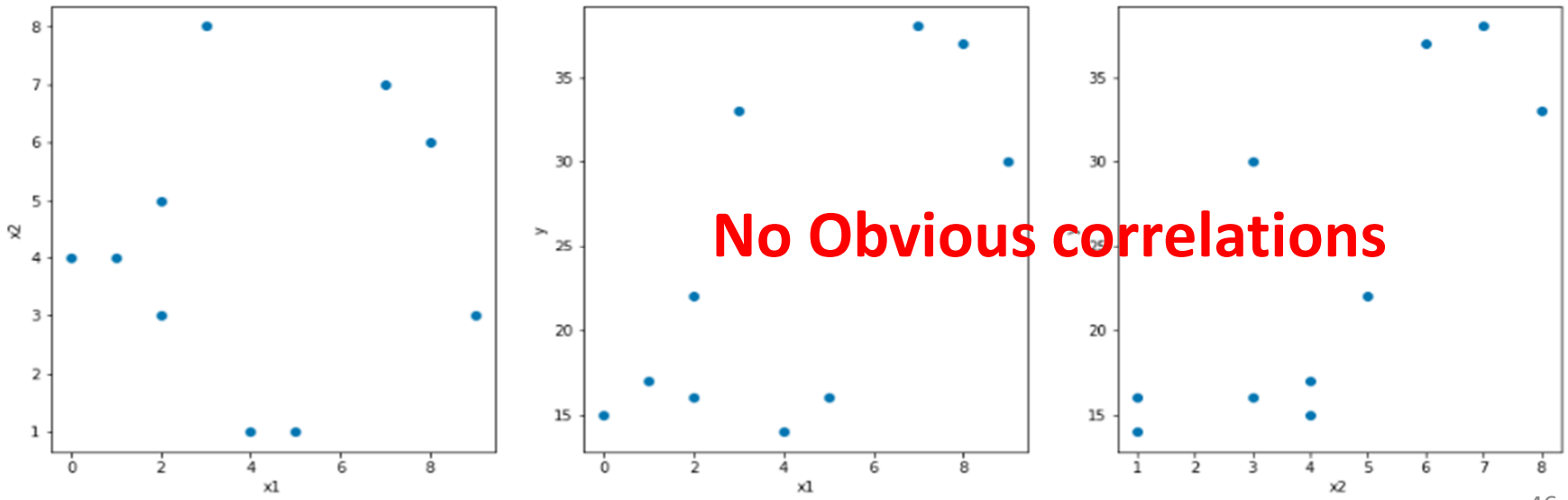
- **Basics of machine-learning** (aka “artificial intelligence”)
- Simple and yet very efficient for understanding and solving real-life problems
- One of the most used tools in data science, economics, …

Example (1)

- Find the 2d-plane define in the 3d space (x1,x2, y) that fit the following data:

X1	1	3	7	9	2	8	5	4	2	0
X2	4	8	7	3	5	6	1	1	3	4
Y	17	33	38	30	22	37	16	14	16	15

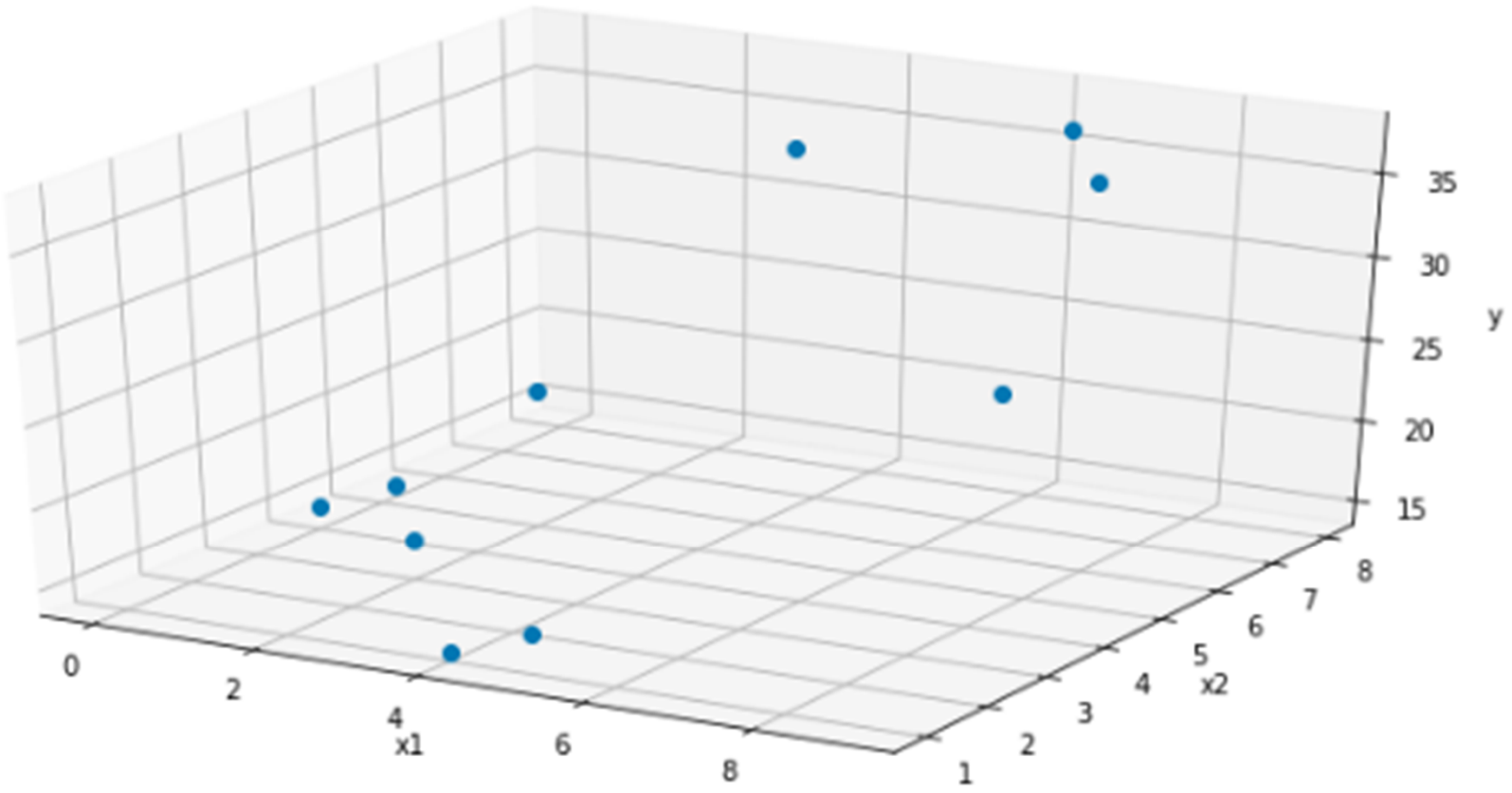
- See how the data looks like



No Obvious correlations

Example (2)

- See in 3D : y = f(x1, x2)



A trend is more visible…, large x1 or x2 have large y

Example (3)

we need to solve the OLS normal equation

$$\mathbf{X}^{\mathbf{T}}\mathbf{X}\mathbf{A} = \mathbf{X}^{\mathbf{T}}\mathbf{Y}$$

We are looking for A and we know X and Y

1) we know x1,x2,y

```
]: x1 = x1[:,np.newaxis] #column vector
   x2 = x2[:,np.newaxis] #column vector
   y  = y [:,np.newaxis] #column vector

   print("x1, shape=", x1.shape, ":")
   print(x1)
   print("")
   print("x2, shape=", x2.shape, ":")
   print(x2)
   print("")
   print("y, shape=", y.shape, ":")
   print(y)
```

executed in 10ms, finished 10:12:43 2019-10-17

```
x1, shape= (10, 1) :
[[1.]
 [3.]
 [7.]
 [9.]
 [2.]
 [8.]
 [5.]
 [4.]
 [2.]
 [0.]]
```

```
x2, shape= (10, 1) :
[[4.]
 [8.]
 [7.]
 [3.]
 [5.]
 [6.]
 [1.]
 [1.]
 [3.]
 [4.]]
```

```
y, shape= (10, 1) :
[[17.]
 [33.]
 [38.]
 [30.]
 [22.]
 [37.]
 [16.]
 [14.]
 [16.]
 [15.]]
```

Example (4)

- We should build the full matrix X which concatenate all the variables we want to for the linear regression
  - a constant term (always=1)
  - The variable x1
  - The variable x2

2) we build x = [1, x1, x2]

```
: X = np.hstack([np.ones(shape=x1.shape), x1, x2])
print("X, shape=", X.shape, ":")
print(X)
```

executed in 13ms, finished 10:12:43 2019-10-17

```
X, shape= (10, 3) :
[[1. 1. 4.]
 [1. 3. 8.]
 [1. 7. 7.]
 [1. 9. 3.]
 [1. 2. 5.]
 [1. 8. 6.]
 [1. 5. 1.]
 [1. 4. 1.]
 [1. 2. 3.]
 [1. 0. 4.]]
```

Example (5)

- We compute the two parts that we need to solve the OLS general equation: $\mathbf{X^T X A = X^T Y}$
  - $\mathbf{X^T X}$
  - $\mathbf{X^T Y}$

3) we compute $X^T X$ and $X^T Y$

```
: XTX = np.matmul( np.transpose(X), X)
  print("XTX, shape=", XTX.shape, ":")
  print(XTX)
  print("")
  XTY = np.matmul( np.transpose(X), y)
  print("XTY, shape=", XTY.shape, ":")
  print(XTY)
```

- **Question: What are the dimensions of $\mathbf{X^T X}$ and $\mathbf{X^T Y}$?**

```
XTX, shape= (3, 3) :
[[ 10.  41.  42.]
 [ 41. 253. 177.]
 [ 42. 177. 226.]]

XTY, shape= (3, 1) :
[[ 238.]
 [1160.]
 [1158.]]
```

Example (6)

- $\mathbf{X^T X}$ is always symmetric and positively defined,
  so se can use cholesky to find A by solving the system $\mathbf{X^T X A = X^T Y}$

```
L = choleski(XTX.copy())
A = choleskiSol(L, XTY.copy())

print("A, shape=", A.shape, ":")
print(A)
```

executed in 39ms, finished 10:12:43 2019-10-17

- **Question: What is the dimension of A ?**

```
A, shape= (3, 1) :
[[3.]
 [2.]
 [3.]]
```

Example (7)

- We can compute the Y approx. from x1,x2 and A
  - $\mathbf{Y} = \mathbf{XA}$

```
Y_OLS = np.matmul(X,A)
print("Y_OLS, shape=", Y_OLS.shape, ":")
print(Y_OLS)
print("")
print("comparing with Y:")
print(np.hstack([Y_OLS,y,Y_OLS-y]))
```

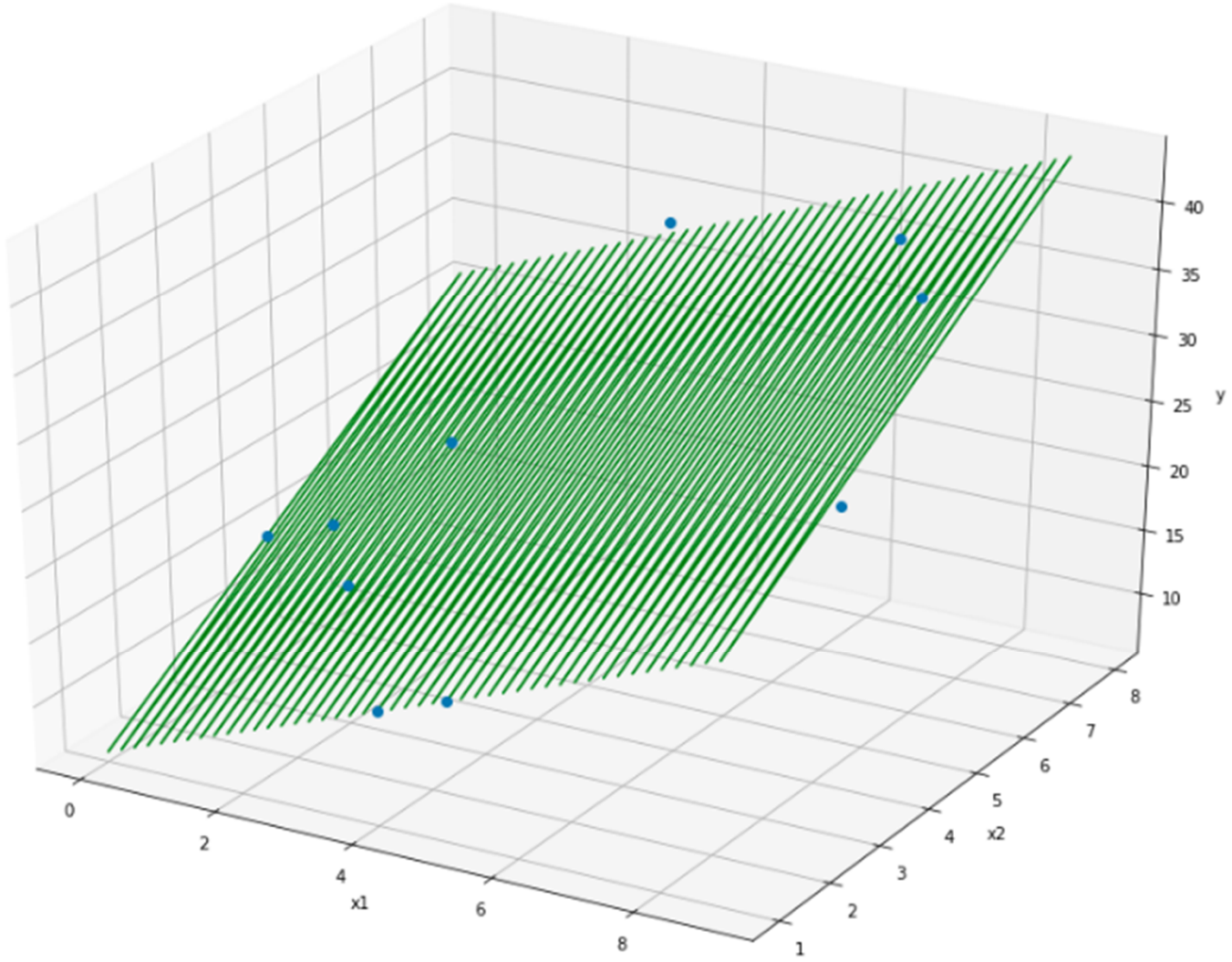
executed in 9ms, finished 10:12:44 2019-10-17

- **Question: What is the dimension of Y_OLS ?**
  - **(10,1)**
- **Compare Y_OLS with Y (real data)**

Y_OLS	Y	residuals
1.70000000e+01	1.70000000e+01	-7.10542736e-15
3.30000000e+01	3.30000000e+01	7.10542736e-15
3.80000000e+01	3.80000000e+01	7.10542736e-15
3.00000000e+01	3.00000000e+01	-7.10542736e-15
2.20000000e+01	2.20000000e+01	-3.55271368e-15
3.70000000e+01	3.70000000e+01	0.00000000e+00
1.60000000e+01	1.60000000e+01	-1.42108547e-14
1.40000000e+01	1.40000000e+01	-1.42108547e-14
1.60000000e+01	1.60000000e+01	-1.06581410e-14
1.50000000e+01	1.50000000e+01	-8.88178420e-15

Example (8)

- Drawing the plane defined by **A**
- **Initial data points are also shown**



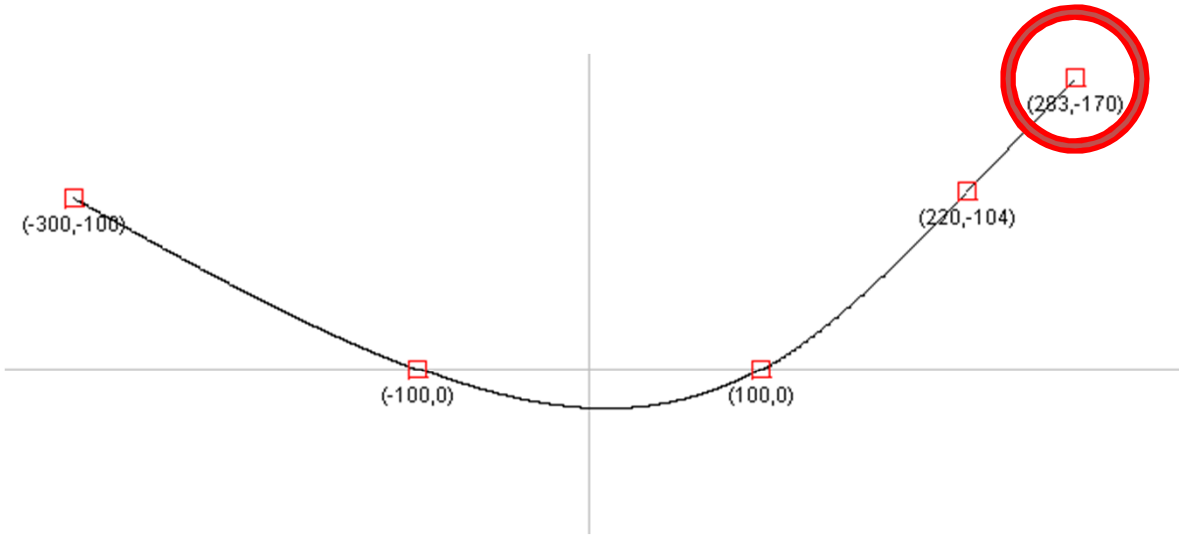
Introduction to B-Splines (Bonus)

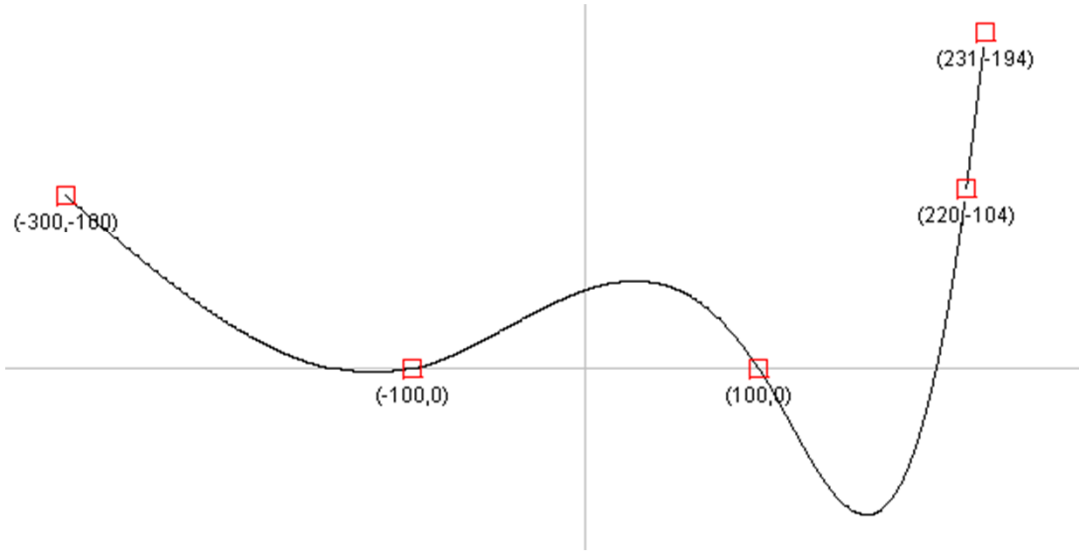
We come back to the interpolation

We are not in regression anymore!

Drawbacks of interpolation approaches (1)

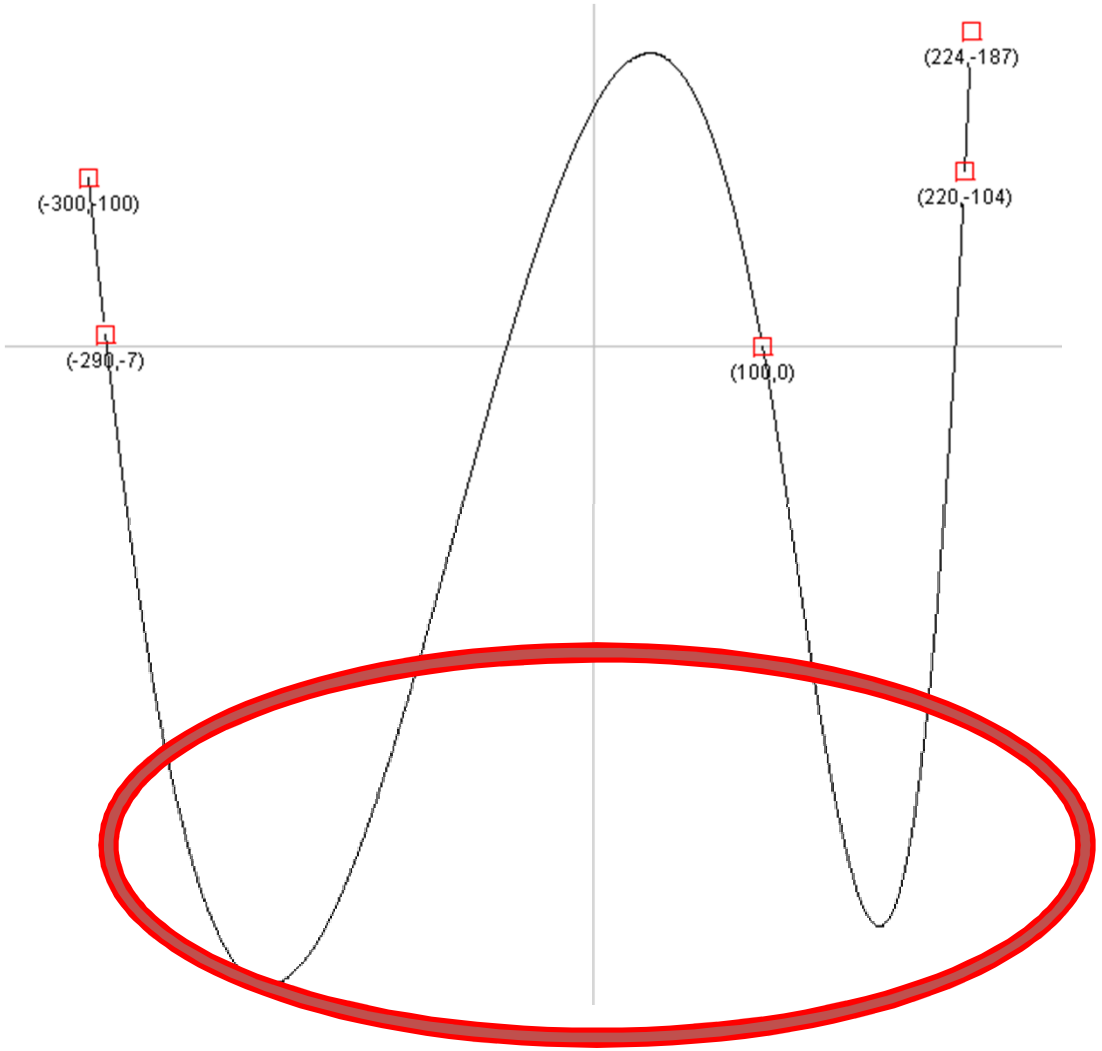
- Changing one point changes the entire curve (even with piece-wise cubic splines!)





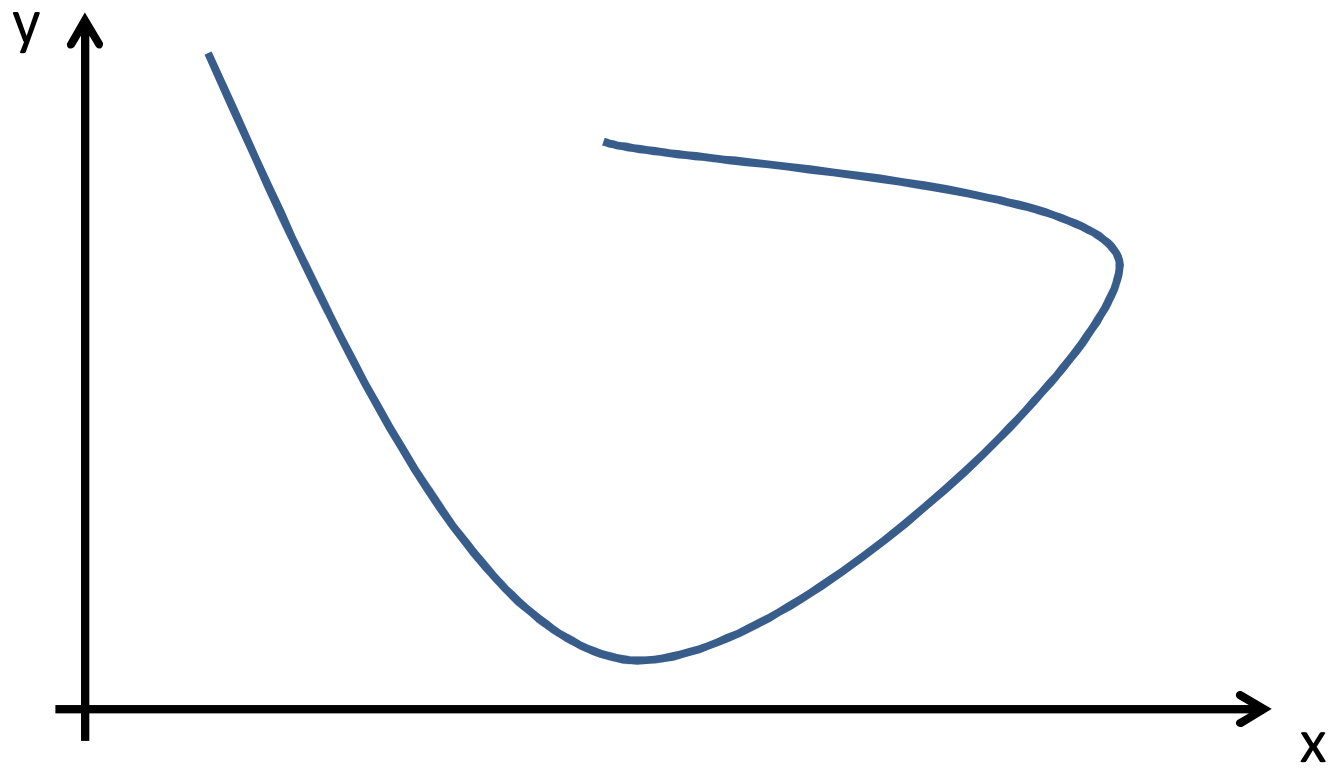
Drawbacks of interpolation approaches (2)

- The curve can “run away” from the data points



Drawbacks of interpolation approaches (3)

- The curve has always the form $y = f(x)$
- This curve is not possible because it cannot be described by a function of x:



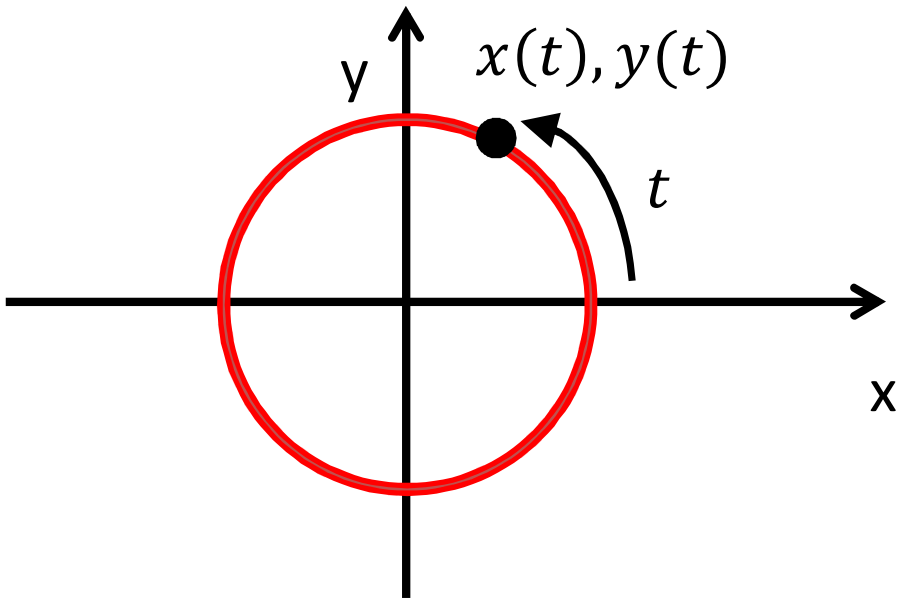
B-Splines

- The shown three drawbacks make the methods that we have seen so far (global/piece-wise interpolation) hard to use for design, computer graphics, games, etc.

- We will now see *B-splines curves*
  1. Points only have *local* influence on the curve
  2. The curve cannot “run away” from the points
  3. B-spline curves are parametric curves of the form $x(t), y(t)$. Not functions of the form $y = f(x)$.

What is a Parametric Curve?

- (Two-dimensional) Parametric curves are described by *parametric equations* with a parameter t
- Example: Circle with radius r
  - $x(t) = r \cdot \cos(t)$
  - $y(t) = r \cdot \sin(t)$ with $t \in [0,2\pi]$



B-Spline basis functions

- A ***B-spline basis function***

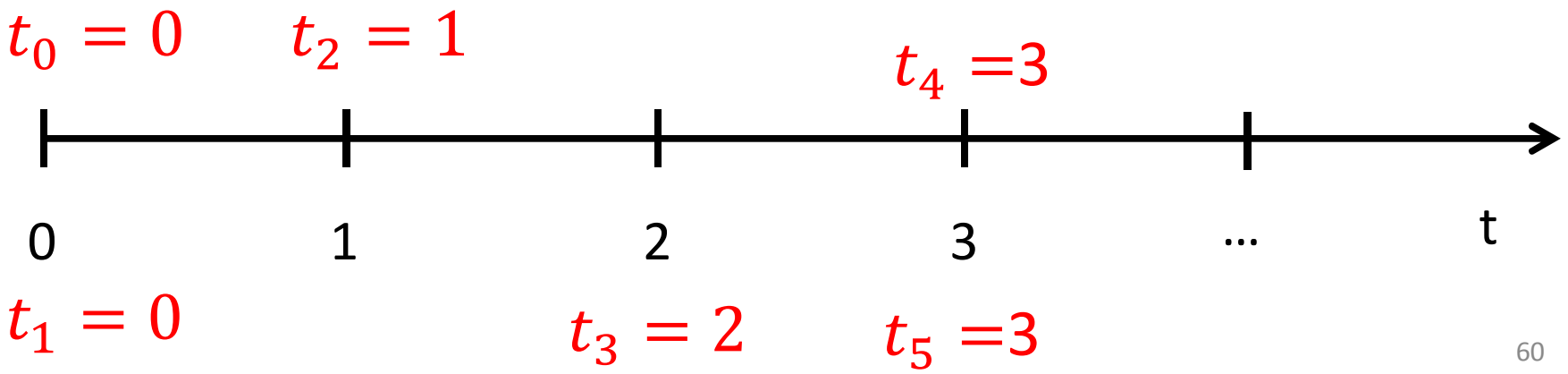
$$B_i^m(t): \mathbb{R} \to [0,1]$$

  is defined using a list of *nodes*

$$T = (t_0, t_1, \dots) \text{ with } t_0 \leq t_1 \leq t_2 \leq \dots$$

- For *uniform* B-splines, we require $t_i \in \mathbb{N}$

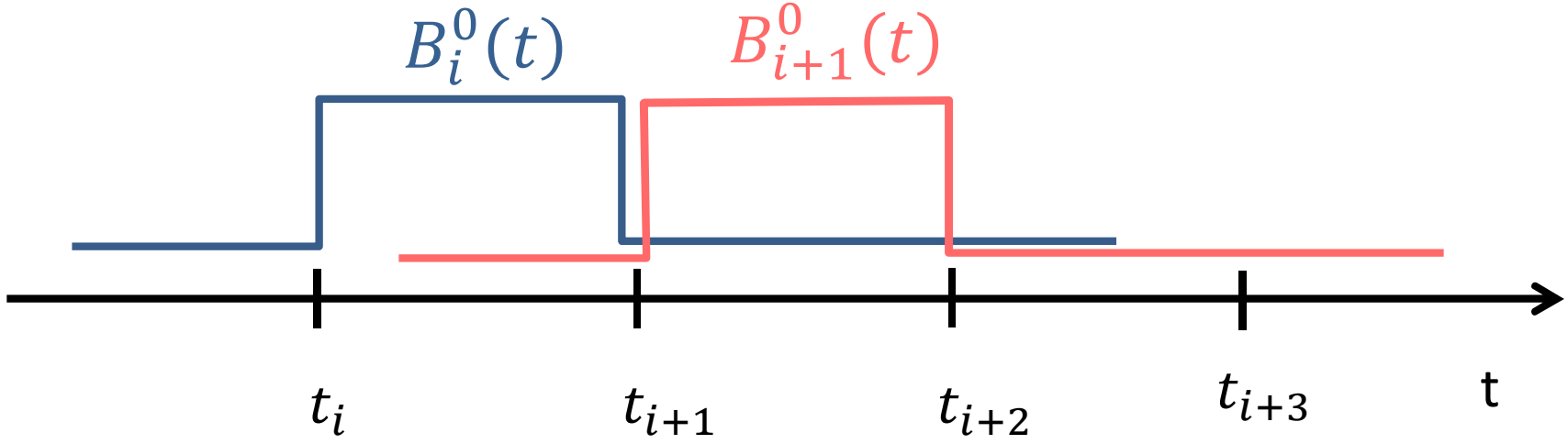
- Example:



Basis function for $m = 0$

- The basis function $B_i^0(t)$ is defined as

$$B_i^0(t) = \begin{cases} 1, & t_i \le t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$



Basis functions for $m > 0$

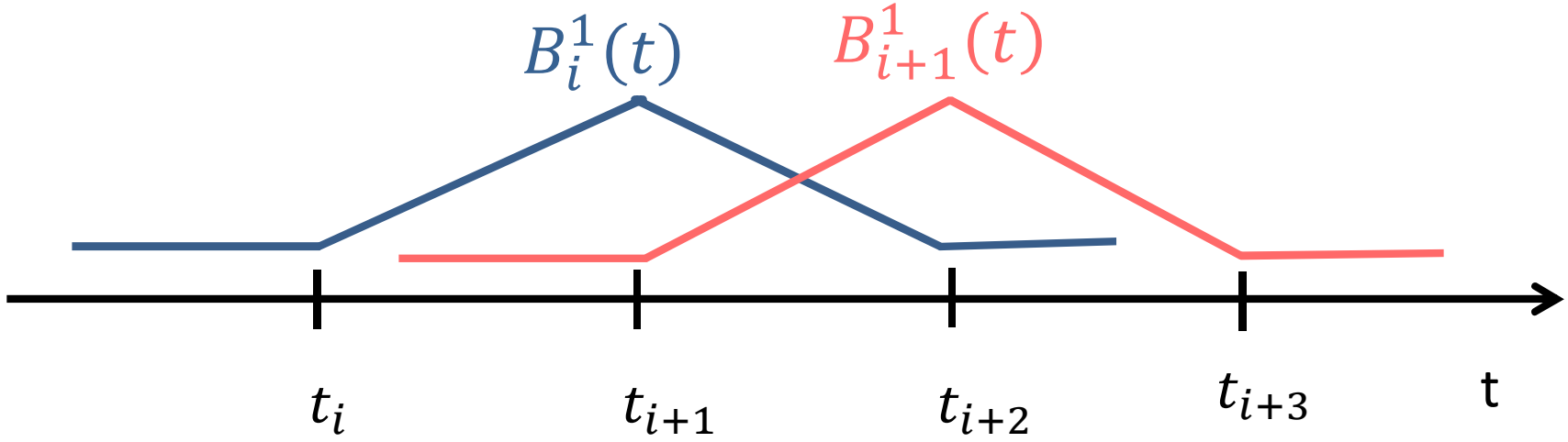
- For $m > 0$, the Basis functions are defined as

$$B_i^m(t) = \frac{t - t_i}{t_{i+m} - t_i} B_i^{m-1}(t) + \frac{t_{i+m+1} - t}{t_{i+m+1} - t_{i+1}} B_{i+1}^{m-1}(t)$$

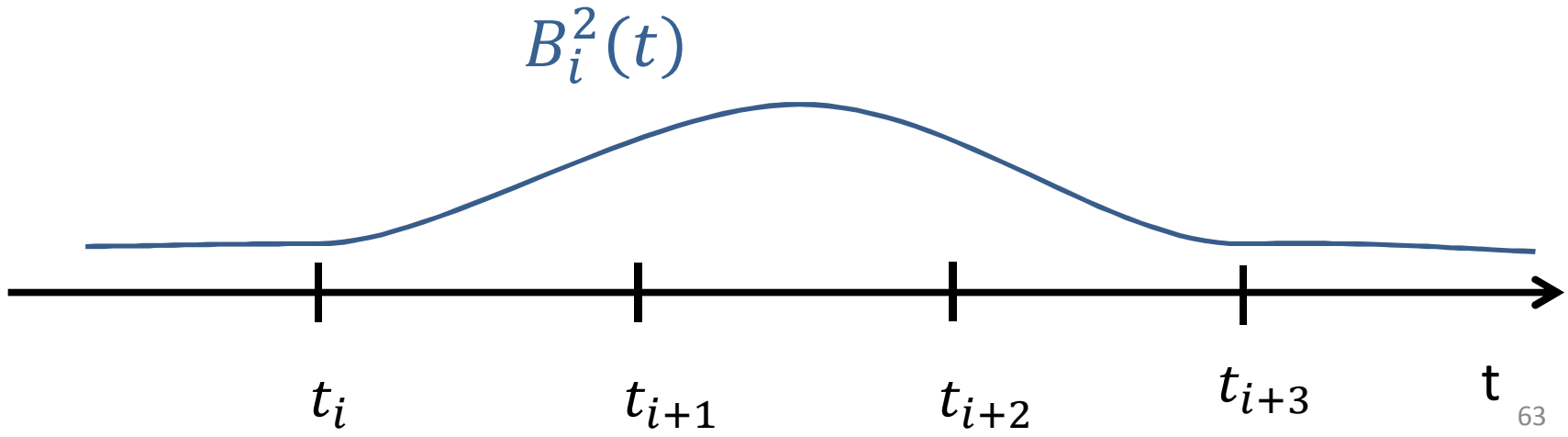
- Note for implementation:
  - If $t_{i+m} - t_i = 0$ then ignore the term $\frac{t-t_i}{t_{i+m}-t_i} B_i^{m-1}(t)$
  - If $t_{i+m+1} - t_{i+1} = 0$ then ignore $\frac{t_{i+m+1}-t}{t_{i+m+1}-t_{i+1}} B_{i+1}^{m-1}(t)$

Basis functions for $m = 1$ and $m = 2$

- Example for $m = 1$ (linear)



- Example for $m = 2$ (quadratic)



Properties of the Basis functions

$B_i^m(t)$ has the following properties:

- $B_i^m(t)$ is composed of piece-wise polynomials of degree m
- $B_i^m(t) \in [0,1]$
- $B_i^m(t) = 0$ for t outside $[t_i, t_{i+m+1}]$
- For $t \in [t_m, t_n]$ it holds

$$\sum_{i=0}^{n} B_i^m(t) = 1$$

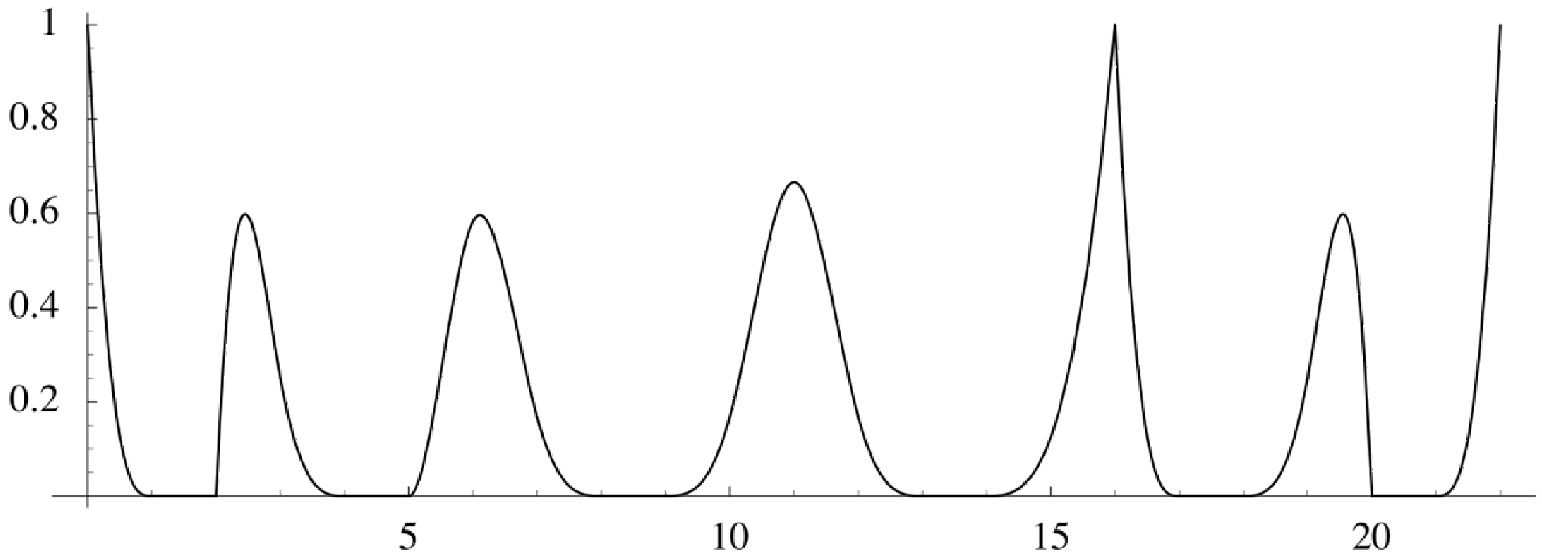
Differentiability of $B_i^m(t)$

- $B_i^m(t)$ is $m-1$ times differentiable on $[t_i, t_{i+m+1}[$ if $t_0 < t_1 < t_2 <$ …, i.e. if all nodes are different

- <u>However:</u>
  - If we choose k identical nodes $t_i = t_{i+1} = \cdots = t_{i+k-1}$, $B_i^m(t)$ will only be $m-k$ times differentiable

Example

Source: University Oslo



- From left to right:
  - $B_0^3(t)$ for nodes $(t_0, t_1, t_2, t_3, t_4) = (0,0,0,0,1)$
  - $B_2^3(t)$ for (2,2,2,3,4)
  - $B_5^3(t)$ for (5,5,6,7,8)
  - $B_9^3(t)$ for (9,10,11,12,13)
  - $B_{14}^3(t)$ for (14,16,16,16,17)
  - $B_{18}^3(t)$ for (18,19,20,20,20)
  - $B_{21}^3(t)$ for (21,22,22,22,22)

Why is $B_i^m(t)$ called a basis function?

- It can be shown that for the linear combination

$$s(t) = \sum_{i=0}^{n} c_i B_i^m(t)$$

  - the coefficients $c_i \in \mathbb{R}$ are unique,
  - i.e., there is no other $c_0', \dots, c_n'$ that gives the same $s(t)$.
- In fact, ***any* spline curve of degree $\boldsymbol{m}$** can be expressed by a unique linear combination of $B_i^m(t)$

- **Differentiability**: As for the basis functions, the differentiability of $s(t)$ depends on the choice of the nodes: **If you repeat a node $\boldsymbol{k}$ times**, $s(t)$ will only be $\boldsymbol{m - k}$ **times differentiable**

B-Spline Curves

- A B-spline curve of degree m is a linear combination of the basis functions:

  - $x(t) = \sum_{i=0}^{n-1} x^{(i)} \, B_i^m(t)$
  - $y(t) = \sum_{i=0}^{n-1} y^{(i)} \, B_i^m(t) \quad \text{for } 0 \leq t \leq n - m$

  with n control points $\left(x^{(i)}, y^{(i)}\right), 0 \leq i < n$

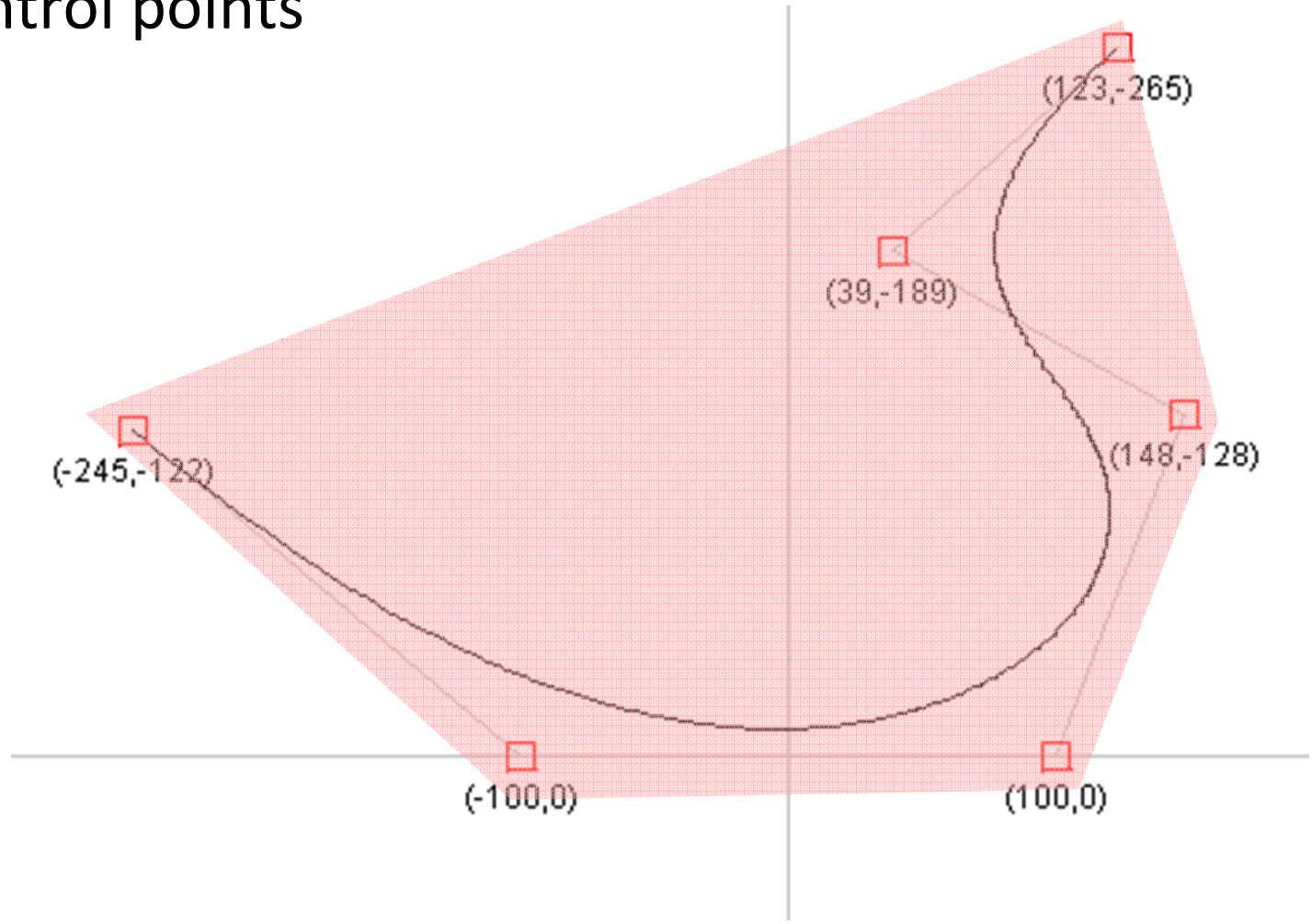
Properties of the B-spline curve

- Consequence of $B_i^m(t) = 0$ for t outside $[t_i, t_{i+m+1}]$:
  - a control point $(x^{(i)}, y^{(i)})$
    only has influence on the curve for $t \in [t_i, t_{i+m+1}[$

- Consequence of $\sum_{i=0}^{n} B_i^m(t) = 1$:
  - all points $x(t), y(t)$ of the curve are inside the *convex hull* of the control points

Convex hull (Enveloppe convexe)

- A B-spline always stays inside the *convex hull* of the control points



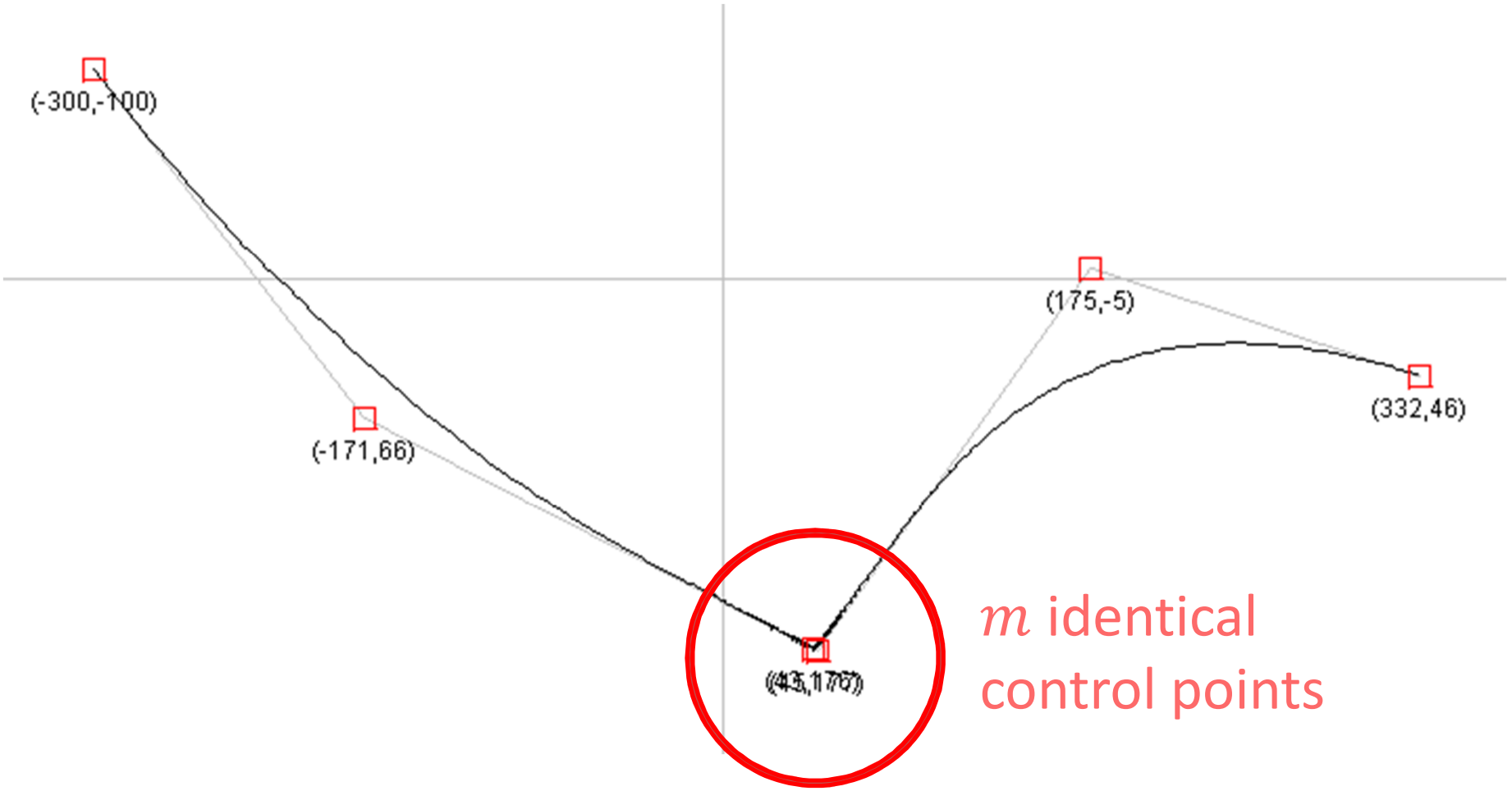
Differentiability

- Again, the differentiability of the curve depends on the choice of the nodes:
- If you repeat a node k times, the curve will only be $m - k$ times differentiable

- A similar effect happens if you choose identical control points

Impact of repeating a control point

- B-spline with $\boldsymbol{m = 3}$ and three identical control points



Node placement

- For the exercises, we set m identical nodes at the beginning and m identical nodes at the end
- This has the nice effect, that the curve starts and ends exactly in the first and last control point

$$t_i = \begin{cases} 0, & for\ i \leq m \\ n - m, & for\ i \geq n \\ i - m, & otherwise \end{cases}$$

Conclusion on B-Splines

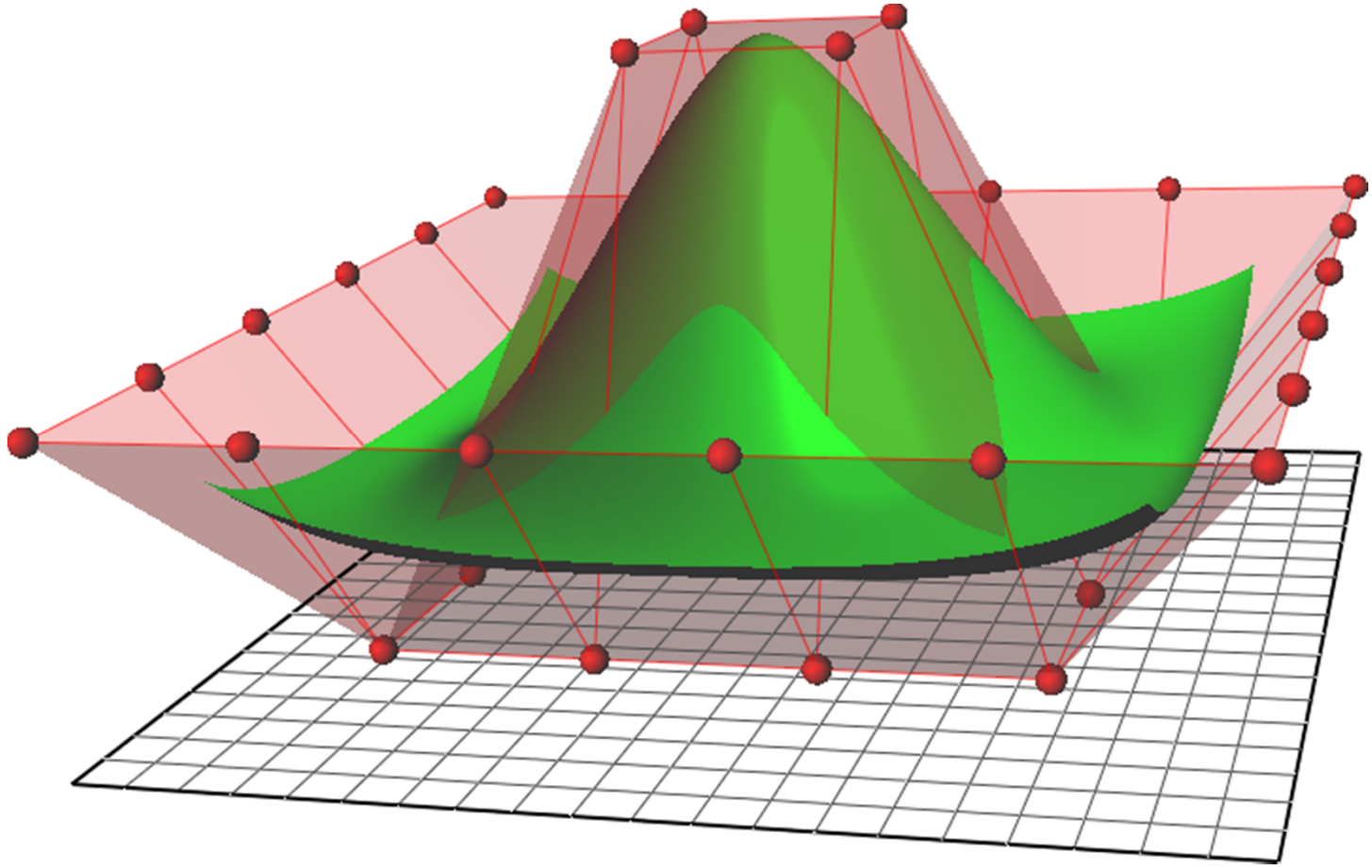
- B-spline curves are not interpolations: the curve does not go through the control points

- But they have nice properties:
  - Curve stays inside the convex hull
  - Control points only have *local* influence on the shape of the curve
  - Easy to construct basis functions for higher degrees
  - By repeating nodes or control points, you can make curves with “sharp bends”

NURBS

- B-splines have some limitations on the form of the curve
  - For example, you cannot make a perfect circle
- Non-Uniform Rational B-Splines (NURBS) are a generalization of B-Splines
  - More flexible, circles possible
- NURBS are also used in 3D graphics
  - Two curve parameters s and t

NURBS for surfaces



https://de.wikipedia.org/wiki/Non-Uniform_Rational_B-Spline#/media/File:NURBS_surface.png