Interpolation et Régression Linéaires

by Loïc Quertenmont, PhD

LINFO1113 - 2019-2020

Programme

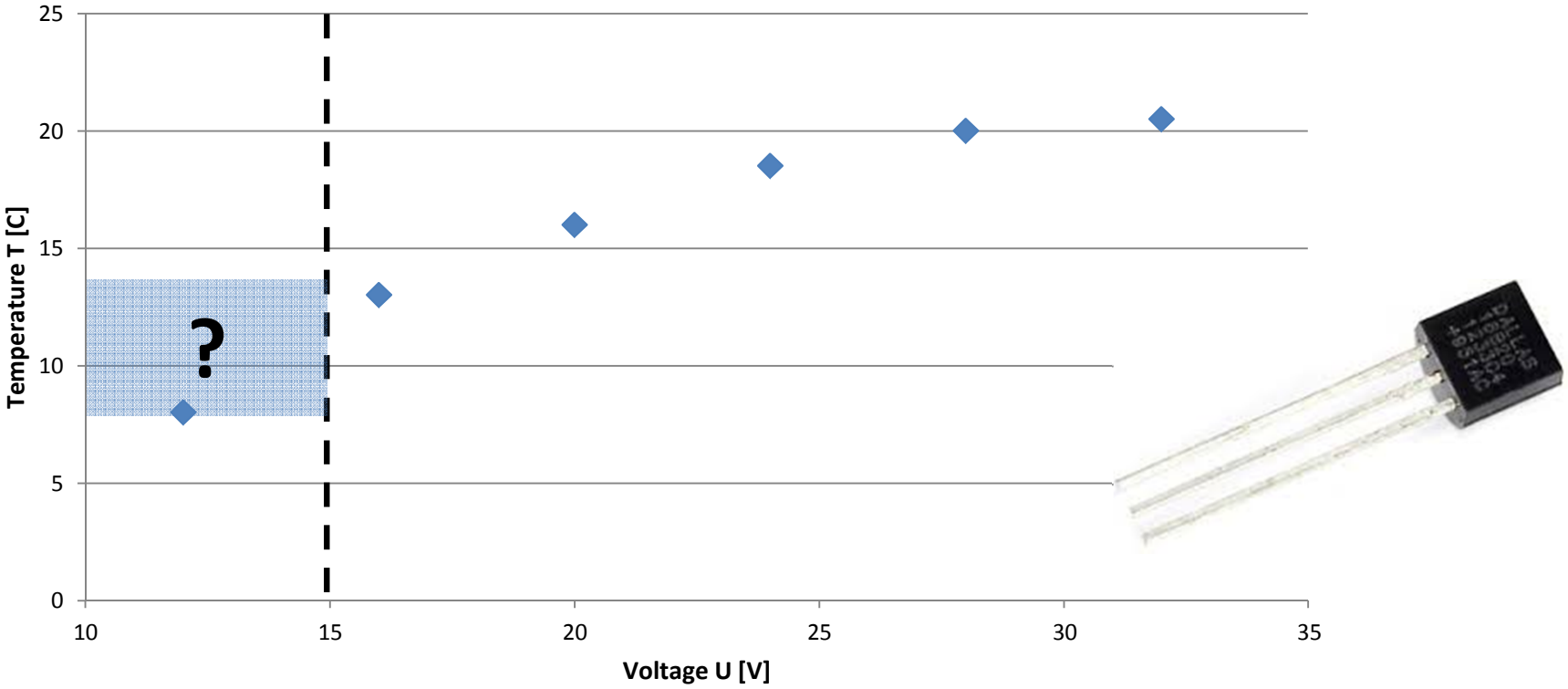
Cours	1	Librairies mathématiques, représentation des nombres en Python et erreurs liées
Cours	2,3	Résolution des systèmes linéaires
Cours	**4,5**	**Interpolation et Régression Linéaires**
Cours	6,7	Zéro d'équation
Cours	8,9	Différentiation numérique
Cours	10	Intégration numérique
Cours	11,12	Introduction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

Outline

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

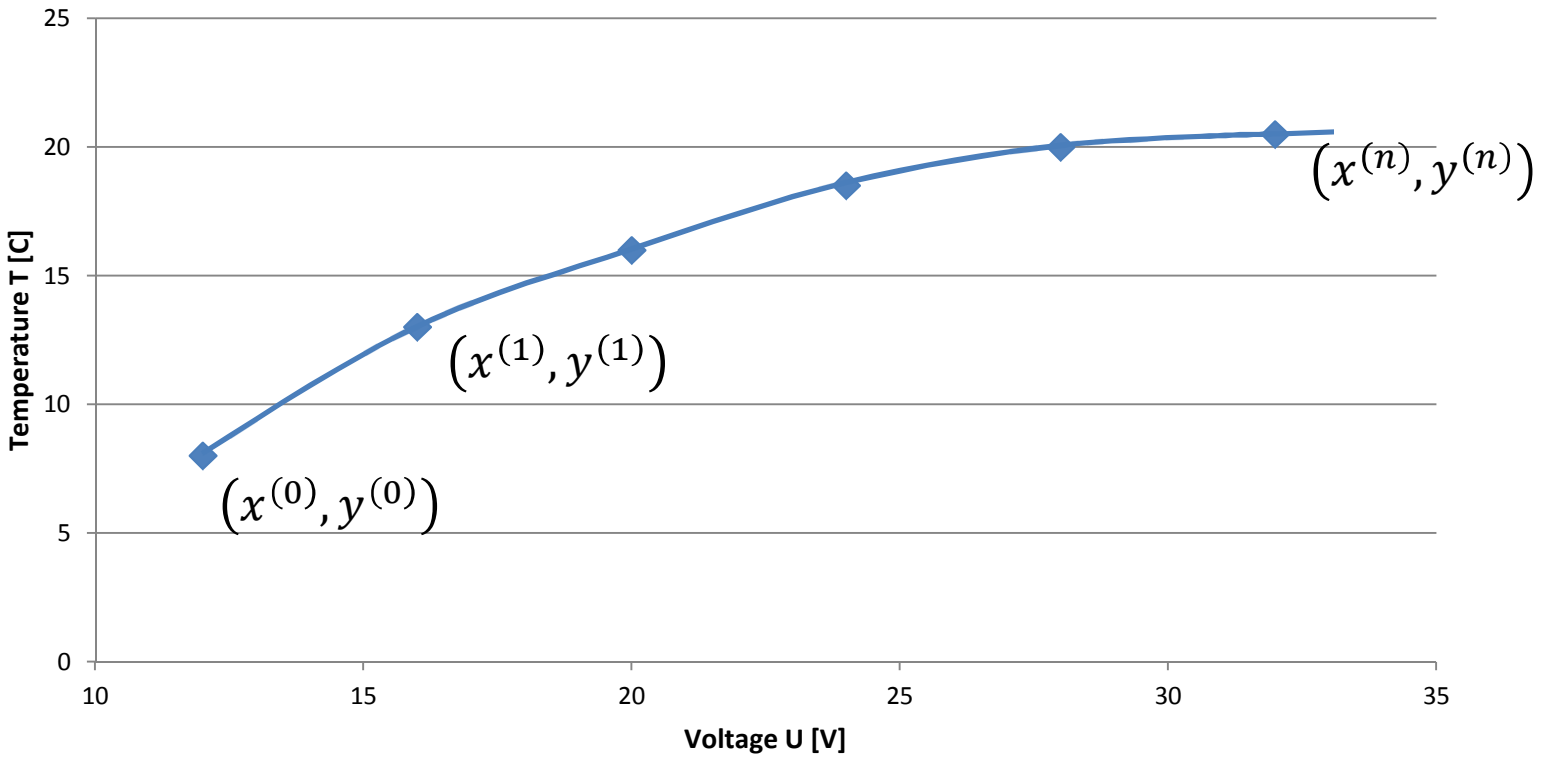
Example

- 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

- Can we find a function $T = f(U)$ that helps us to calculate T for all U?
- Calibration:
  - measuring f(U) for a set of values
- Interpolation



Introduction

- We start from a set of measure (x_i, y_i)

x_0	x_1	x_2	x_3	…	…	x_{n-1}	x_n
y_0	y_1	y_2	y_3	…	…	y_{n-1}	y_n

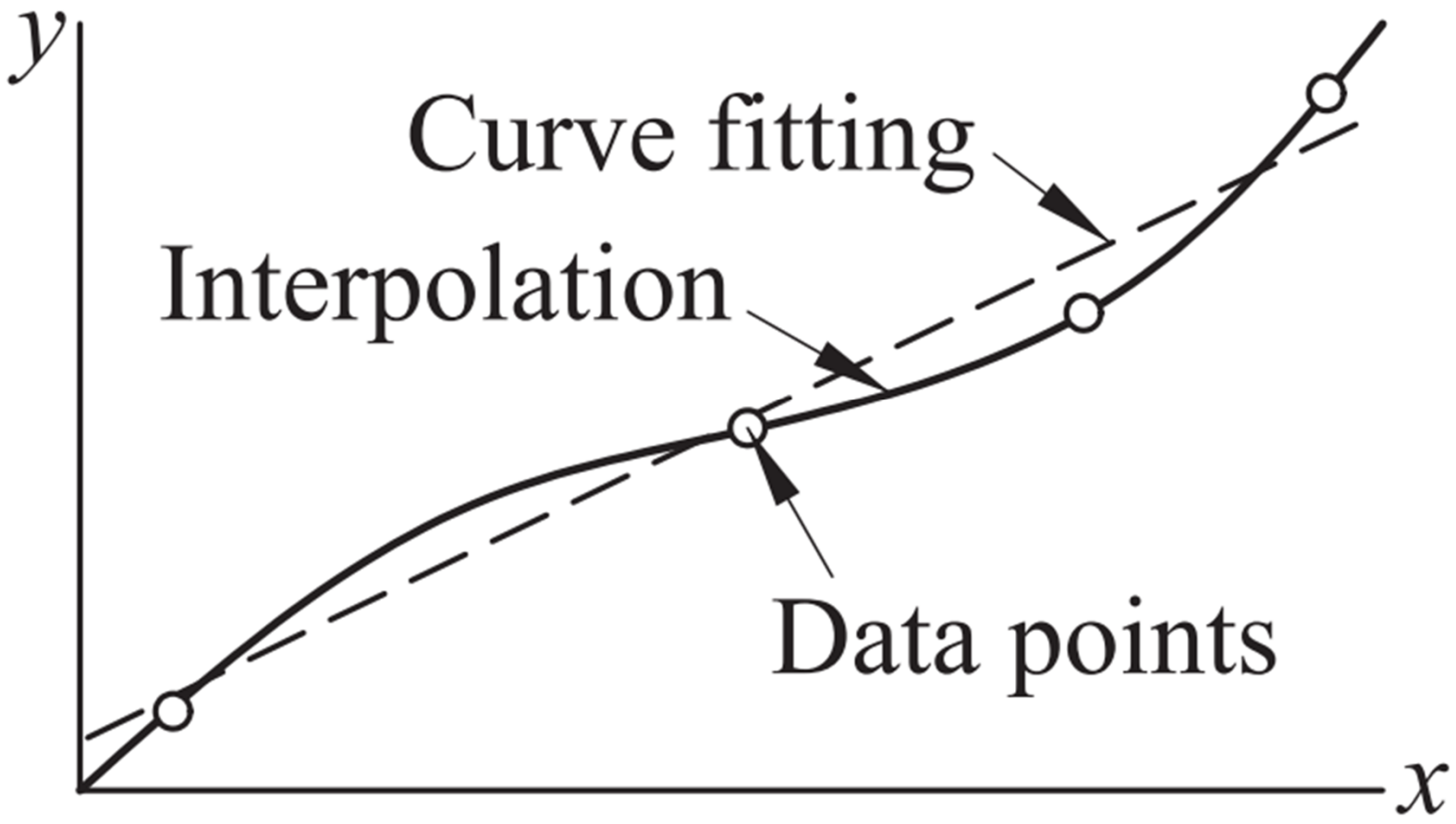
  - The source of the data can be
    - Measurements (from instruments or experiments)
    - Numerical computations
- We want to get a find a curve y(x) from this set of data, that is define on the entire data domain

Introduction

- **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
  - $y_i = f(x_i) + \varepsilon_i$

Introduction



Polynomial Interpolation

Vandermonde (naïve) Solution (1)

- We will use this polynomial for interpolation:

$$f(x) = a_1 + a_2 x + a_3 x^2 \ldots + a_n x^{n-1}$$

- We know that $f(x_i) = y_i$ for all i

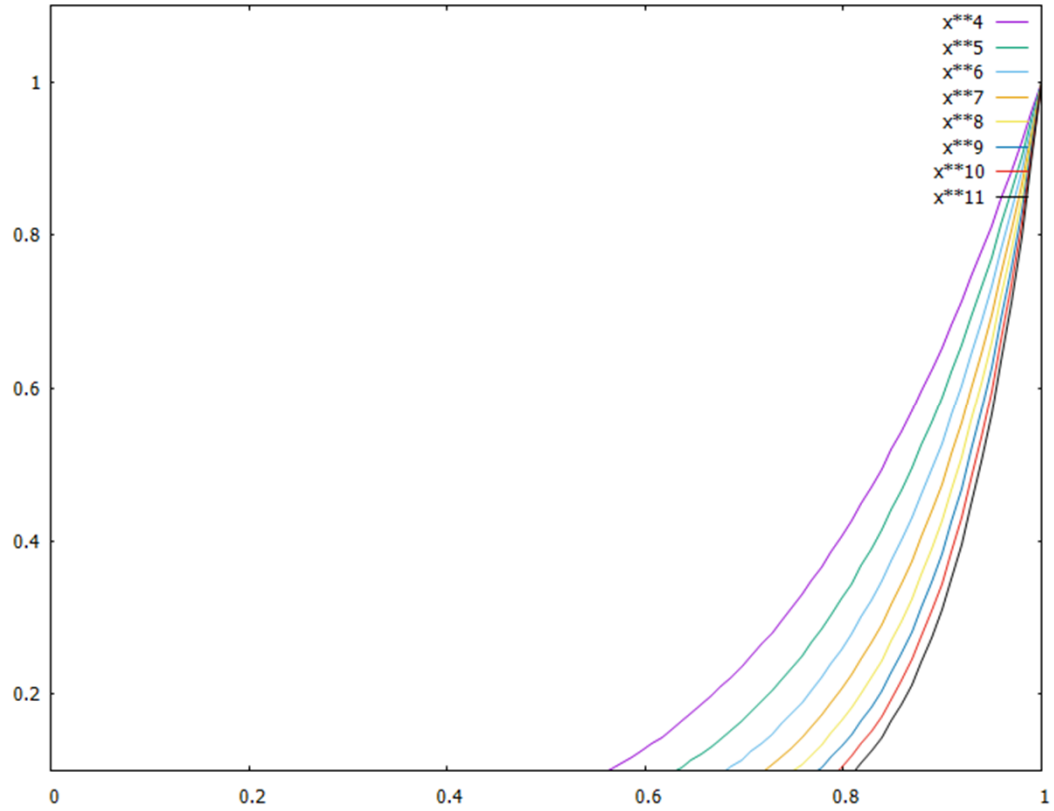
$$\begin{pmatrix} 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & (x_m)^2 & \cdots & (x_m)^{n-1} \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y \\ \vdots \\ y_m \end{pmatrix}$$

- We can find a_i by solving the Vandermonde system

- For $n = m$, the system has a unique solution if all rows are linearly independent (= the matrix has rank n)
- There exists only one polynomial of degree $n - 1$ with $f(x_i) = y_i$ for n points!

Vandermonde (naïve) Solution (2)

- Complexity: $O(n^3)$
- Bad: Will not work well for large n
  - Why?
  - For large k the terms x^k and x^{k+1} become very similar:



Vandermonde (naïve) Solution (3)

- That means the right columns of

$$\begin{pmatrix} 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & (x_m)^2 & \cdots & (x_m)^{n-1} \end{pmatrix}$$

  become very similar for large n

- The system $XA = Y$ becomes *ill-conditioned*: We have nearly dependent columns

- This makes the solution numerically unstable.

Lagrange's Method

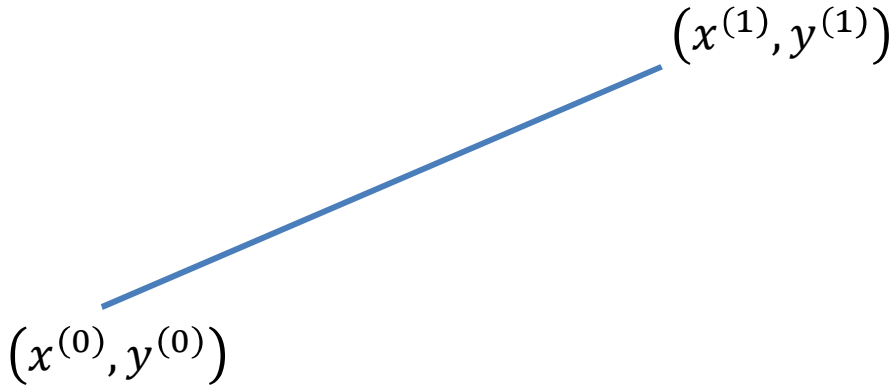
- **Let's find another method**
- Simplest interpolant is a polynomial

- It is always possible to construct a **unique** polynomial of degree **n** that passes through **n+1** distinct points

  – 1 point → pol. degree 0 (horizontal line)
  – 2 points → pol. degree 1 (line)
  – 3 points → pol. degree 2

Lagrange's Method (1)

- Let's build a generic polynomial that passes through two points (n=1)



- $P_1(x) = a\,x + b$
- $P_1(x_0) = y_0$
- $P_1(x_1) = y_1$

$$a = \frac{y_0 - y_1}{x_0 - x_1}$$

$$b = \frac{x_0 y_1 - x_1 y_0}{x_0 - x_1}$$

$$P_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0}$$

Lagrange's Method (2)

$$P_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0}$$

- We can verify that

  $P_1(x_0) = y_0$ and $P_1(x_1) = y_1$

- Because:

$$l_0(x) := \frac{x - x_1}{x_0 - x_1} = 1 \text{ for } x = x_0 \text{ and } = 0 \text{ for } x = x_1$$

- Similarly

$$l_1(x) := \frac{x - x_0}{x_1 - x_0} = 0 \text{ for } x = x_0 \text{ and } = 1 \text{ for } x = x_1$$

Lagrange's Method (3)

$$P_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0}$$

- **Can we generalize to 3 points (n=2) ?**
- $P_2(x) = y_0\, l_0(x) + y_1 l_1(x) + y_2 l_2(x)$
  - With $l_i(x_j)$ with the form a $x^2 + bx + c$
  - With $l_i(x_j) = 0$ for $i \neq j$
  - With $l_i(x_j) = 1\ for\ i = j$

$$P_2(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Lagrange's Method (3)

$$P_2(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

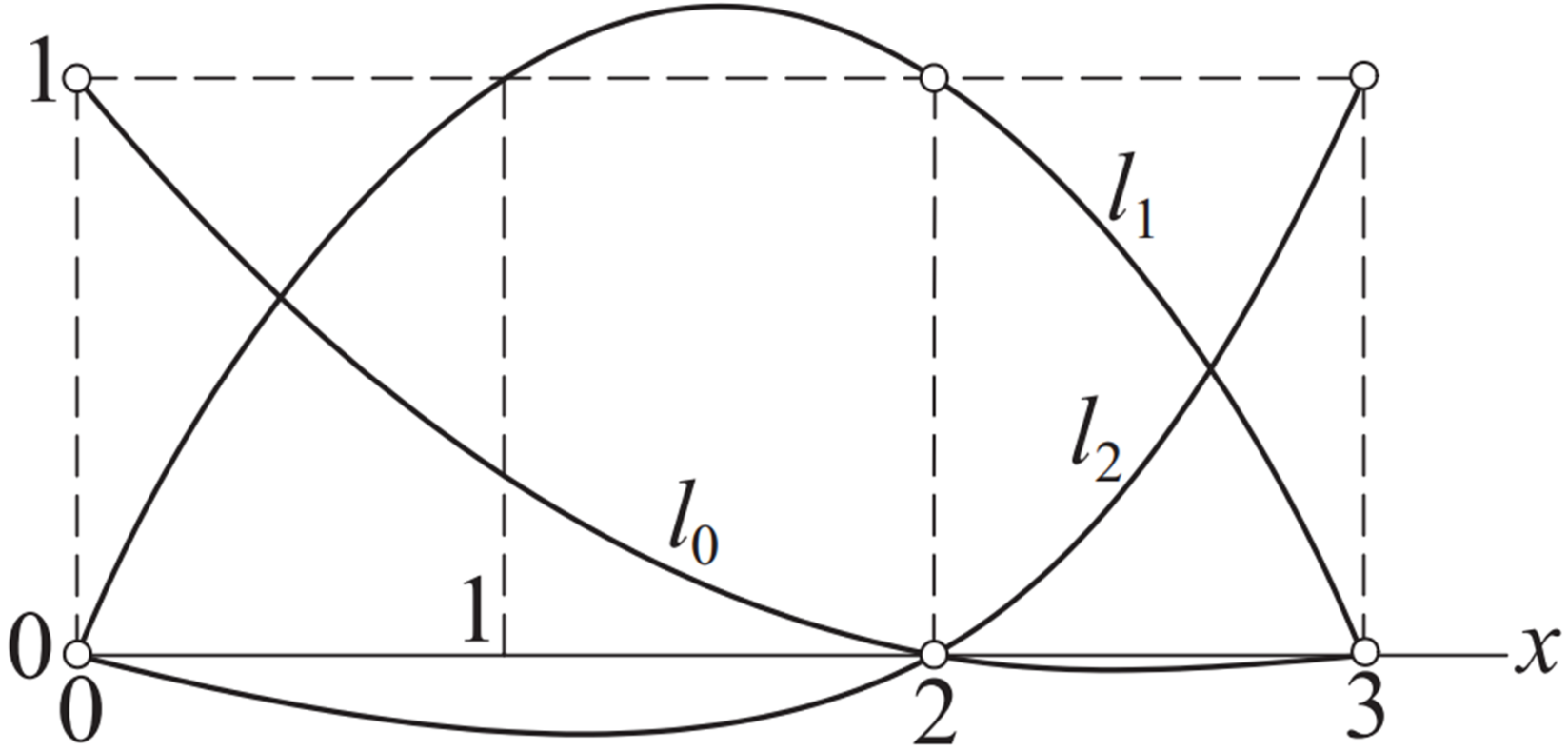
- **Can we generalize to n+1 points (degree n) ?**
- $P_n(x) = \sum_i y_i \, l_i(x)$
  - With $l_i(x_j)$ with the form $a x^2 + bx + c$
  - With $l_i(x_j) = \delta_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases}$

$$l_i(x) = \frac{(x-x_0)}{(x_i-x_0)} \frac{(x-x_1)}{(x_i-x_1)} \cdots \frac{(x-x_{i-1})}{(x_i-x_{i-1})} \frac{(x-x_{i+1})}{(x_i-x_{i+1})} \cdots \frac{(x-x_n)}{(x_i-x_n)} = \prod_{j \neq i} \frac{(x-x_j)}{(x_i-x_j)}$$

$$P_n(x) = \sum_i \left(y_i \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)} \right)$$

Lagrange's Method (4)

Example for P_2



Lagrange's Method (5)

- Error of the interpolation

$$f(x) - P_n(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_n)}{(n+1)!} f^{(n+1)}(\xi)$$

- ξ is somewhere between (x_0, x_n)

- It is interesting to note that:
  - The error is 0 if $x=x_i$ (by construction)
  - The error grows the farther we are from x_i

Lagrange: Exercise

Use Lagrange to compute y at x=1 ?

Data =

x	0	2	3
y	7	11	28

$$P_n(x) = \sum_i y_i \, l_i(x) \qquad l_i(x) = \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)}$$

$$\ell_0 = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{(1 - 2)(1 - 3)}{(0 - 2)(0 - 3)} = \frac{1}{3}$$

$$\ell_1 = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = \frac{(1 - 0)(1 - 3)}{(2 - 0)(2 - 3)} = 1$$

$$\ell_2 = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{(1 - 0)(1 - 2)}{(3 - 0)(3 - 2)} = -\frac{1}{3}$$

$$y = y_0\ell_0 + y_1\ell_1 + y_2\ell_2 = \frac{7}{3} + 11 - \frac{28}{3} = 4$$

Newton's Method (1)

- Lagrange Method is conceptually simple
- But the algorithm is not very efficient

- A more efficient method is Newton's method
  Where the polynomial have the form:

$$
\begin{aligned}
P_n(x) = a_0 \\
&+ (x - x_0)a_1 \\
&+ (x - x_0)(x - x_1)a_2 \\
&+ \cdots \\
&+ (x - x_0)(x - x_1)\cdots(x - x_{n-1})a_n
\end{aligned}
$$

Newton's Method (2)

- Example with 4points (n=3)

$$P_3(x) = a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2 + (x - x_0)(x - x_1)(x - x_2)a_3$$
$$= a_0 + (x - x_0)\{a_1 + (x - x_1)[a_2 + (x - x_2)a_3]\}$$

- Can be computed easily by recurrence relations:

$$P_0(x) = a_3$$

$$P_1(x) = a_2 + (x - x_2)P_0(x)$$

$$P_2(x) = a_1 + (x - x_1)P_1(x)$$

$$P_3(x) = a_0 + (x - x_0)P_2(x)$$

Newton's Method (3)

- Generalizing to arbitrary n

$$P_0(x) = a_n$$

$$P_k(x) = a_{n-k} + (x - x_{n-k})\,P_{k-1}(x), \quad k = 1, 2, \ldots, n$$

- The algorithm implementation is then just:

```
p = a[n]
for k in range(1,n+1):
    p = a[n-k] + (x - xData[n-k])*p
```

– with xData the x-coordinate of the n data points

Newton's Method (4)

- OK, But…
  - **what are the coefficients a_i ?**
- We force the polynomials to pass through the data points (x_i, y_i)

$$y_i = P_n(x_i)$$

- This lead to the system of equations:

$$y_0 = a_0$$

$$y_1 = a_0 + (x_1 - x_0)a_1$$

$$y_2 = a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2$$

$$\vdots$$

$$y_n = a_0 + (x_n - x_0)a_1 + \cdots + (x_n - x_0)(x_n - x_1)\cdots(x_n - x_{n-1})a_n$$

Newton's Method (5)

$$y_0 = a_0$$

$$y_1 = a_0 + (x_1 - x_0)a_1$$

$$y_2 = a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2$$

$$\vdots$$

$$y_n = a_0 + (x_n - x_0)a_1 + \cdots + (x_n - x_0)(x_n - x_1)\cdots(x_n - x_{n-1})a_n$$

$$a_0 = y_0$$

$$a_1 = \nabla y_1$$

$$a_2 = \nabla^2 y_2$$

$$a_n = \nabla^n y_n$$

- Divided difference

$$\nabla y_i = \frac{y_i - y_0}{x_i - x_0}, \quad i = 1, 2, \ldots, n$$

$$\nabla^2 y_i = \frac{\nabla y_i - \nabla y_1}{x_i - x_1}, \quad i = 2, 3, \ldots, n$$

$$\nabla^3 y_i = \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2}, \quad i = 3, 4, \ldots n$$

$$\vdots$$

$$\nabla^n y_n = \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}}$$

Newton's Method (6)

- Calculating by hand
  - It is easier if we store the values in the "Coefficients Table"

x_0	$y_0 = a_0$				
x_1	y_1	$\nabla y_1 = a_1$			
x_2	y_2	∇y_2	$\nabla^2 y_2 = a_2$		
x_3	y_3	∇y_3	$\nabla^2 y_3$	$\nabla^3 y_3 = a_3$	
x_4	y_4	∇y_4	$\nabla^2 y_4$	$\nabla^3 y_4$	$\nabla^4 y_4 = a_4$

  - **All the coefficients of the polynomial are on the diagonal of the table**

Newton's Method (7)

- Example

x	-2	1	4	-1	3	-4
y	-1	2	59	4	24	-53

i	x_i	y_i	∇y_i
0	-2	-1	
1	1	2	?
2	4	59	?
3	-1	4	?
4	3	24	?
5	-4	-53	?

$$\nabla y_i = \frac{y_i - y_0}{x_i - x_0}$$

(2 - (-1)) / (1 – (-2)) = 1

(59 - (-1)) / (4 – (-2)) = 10

Newton's Method (7)

- Example

x	-2	1	4	-1	3	-4
y	-1	2	59	4	24	-53

We see that all these data lies on a cubic polynomial (degree 3)

i	x_i	y_i	∇y_i	$\nabla^2 y_i$	$\nabla^3 y_i$	$\nabla^4 y_i$	$\nabla^5 y_i$
0	-2	-1					
1	1	2	1				
2	4	59	10	?			
3	-1	4	5	?	?		
4	3	24	5	?	?	?	
5	-4	-53	26	?	?	?	?

Newton's Method (8)

- Making an algorithm
  - Notation: m = n+1

```
a = yData.copy()
for k in range(1,m):
    for i in range(k,m):
        a[i] = (a[i] - a[k-1])/(xData[i] - xData[k-1])
```

  - At start, 'a' contains the y-coordinates of the data
    - It's the y column of the coefficient table
  - Each pass through the loop, generate the next column of the coefficient table
    - It overwrite the component in 'a' from indices 'k'
  - At the end, 'a' contains the diagonal of the table
    - The polynomial coefficients

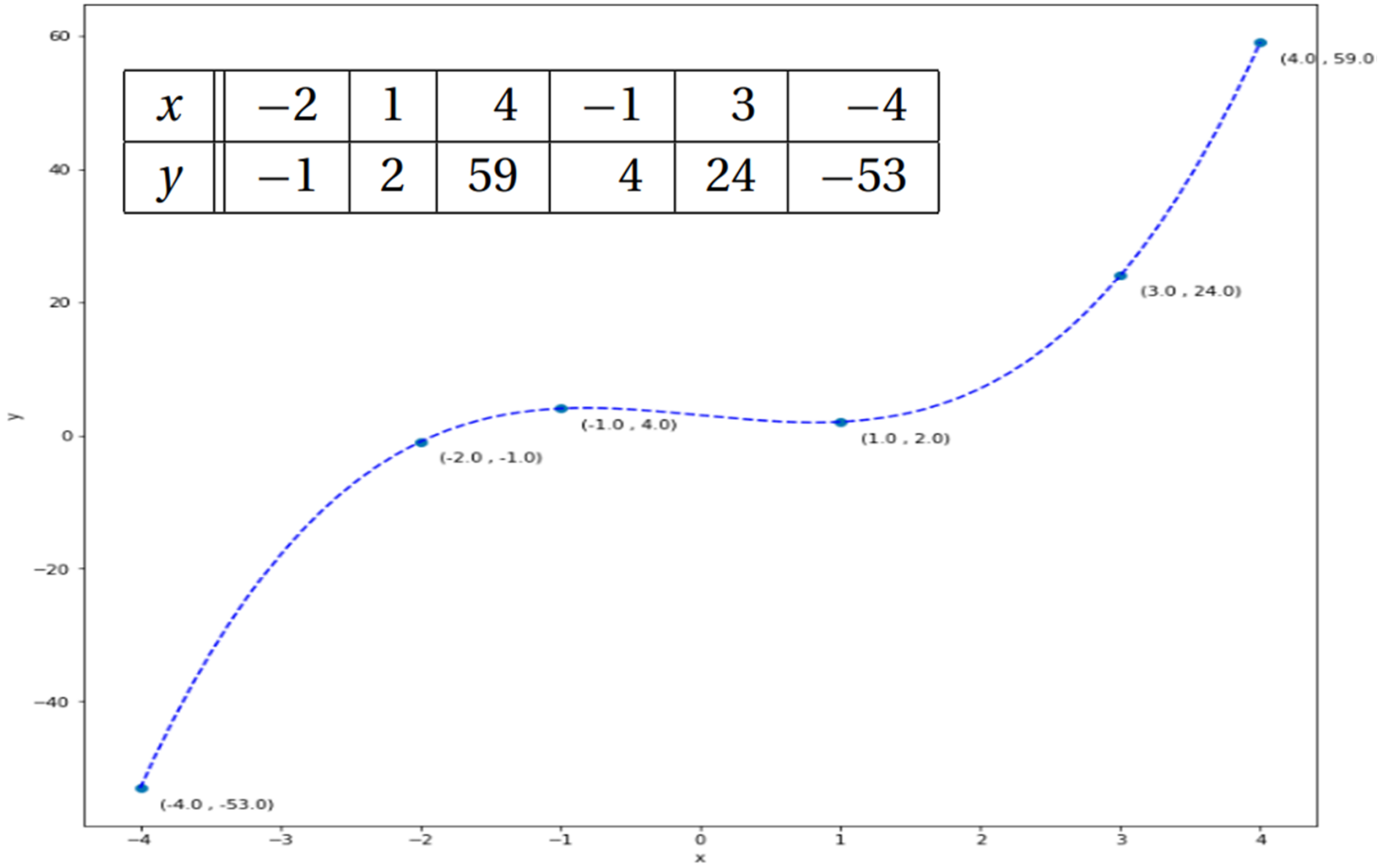
Newton's Implementation

```
def coeffts(xData,yData):
    m = len(xData)  # Number of data points
    a = yData.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xData[k:m] - xData[k-1])
    return a
```

```
def evalPoly(a,xData,x):
    n = len(xData) - 1  # Degree of polynomial
    p = a[n]
    for k in range(1,n+1):
        p = a[n-k] + (x -xData[n-k])*p
    return p
```

Example

x	-2	1	4	-1	3	-4
y	-1	2	59	4	24	-53



Neville's Method (1)

- Newton's requires two steps of computations
  - Computation of the coefficients
  - Evaluation of the polynomials

- It's OK if we need to interpolate the data at many different values of x.

- But if we need to interpolate for only one value, Neville's algorithm is more efficient

  - It involves only one step

Neville's Method (2)

- **Let's define**, $P_k[x_i, x_{i+1}, \ldots, x_{i+k}]$
- The polynomial of degree k that passes through the k+1 data points $(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{i+k}, y_{i+k})$

- **For a single data point, we have:**

$$P_0[x_i] = y_i$$

- **For two data points:**

$$P_1[x_i, x_{i+1}] = \frac{(x - x_{i+1})\,P_0[x_i] + (x_i - x)\,P_0[x_{i+1}]}{x_i - x_{i+1}}$$

  – We can verify that:

$$P_1[x_i, x_{i+1}] = y_i \text{ when } x = x_i$$

$$P_1[x_i, x_{i+1}] = y_{i+1} \text{ when } x = x_{i+1}$$

Neville's Method (3)

- **For three data points:**

$$P_2[x_i, x_{i+1}, x_{i+2}] = \frac{(x - x_{i+2})\,P_1[x_i, x_{i+1}] + (x_i - x)\,P_1[x_{i+1}, x_{i+2}]}{x_i - x_{i+2}}$$

- We can verify that:
  - **For $x = x_i$** $\quad P_2[x_i, x_{i+1}, x_{i+2}] = P_1[x_i, x_{i+1}] = y_i$
  - **For $x = x_{i+2}$** $\quad P_2[x_i, x_{i+1}, x_{i+2}] = P_2[x_{i+1}, x_{i+2}] = y_{i+2}$
  - **For $x = x_{i+1}$** $P_2[x_i, x_{i+1}, x_{i+2}] = \frac{(x_{i+1} - x_{i+2})y_{i+1} + (x_i - x_{i+1})y_{i+1}}{x_i - x_{i+2}} = y_{i+1}$
    - Because for $x = x_{i+1}$ we have $P_1[x_i, x_{i+1}] = P_1[x_{i+1}, x_{i+2}] = y_{i+1}$

Neville's Method (4)

- **We can deduce a recursive formulate from the pattern**

$$P_k[x_i, x_{i+1}, \ldots, x_{i+k}]$$
$$= \frac{(x - x_{i+k})\, P_{k-1}[x_i, x_{i+1}, \ldots, x_{i+k-1}] + (x_i - x)\, P_{k-1}[x_{i+1}, x_{i+2}, \ldots, x_{i+k}]}{x_i - x_{i+k}}$$

- **Tableau for Neville's Method**

	$k = 0$	$k = 1$	$k = 2$	$k = 3$
x_0	$P_0[x_0] = y_0$	$P_1[x_0, x_1]$	$P_2[x_0, x_1, x_2]$	$P_3[x_0, x_1, x_2, x_3]$
x_1	$P_0[x_1] = y_1$	$P_1[x_1, x_2]$	$P_2[x_1, x_2, x_3]$	
x_2	$P_0[x_2] = y_2$	$P_1[x_2, x_3]$		
x_3	$P_0[x_3] = y_3$			

Neville's Method (5)

- Example, Evaluate the function at x=0
  Given the data:

x	-2	1	4	-1
y	-1	2	59	4

$$P_1[x_i, x_{i+1}] = \frac{(x - x_{i+1})\,P_0[x_i] + (x_i - x)\,P_0[x_{i+1}]}{x_i - x_{i+1}}$$

$$P_2[x_i, x_{i+1}, x_{i+2}] = \frac{(x - x_{i+2})\,P_1[x_i, x_{i+1}] + (x_i - x)\,P_1[x_{i+1}, x_{i+2}]}{x_i - x_{i+2}}$$

x_i	$y_i = P_0[\,]$	$P_1[\,,\,]$	$P_2[\,,\,,\,]$	$P_3[\,,\,,\,,\,]$
-2	-1	?	?	?
1	2	?	?	
4	59	?		
-1	4			

Neville's Method (6)

- Making an algorithm to compute the table
  - Notation: m = n+1

```
y = yData.copy()
for k in range (1,m):
    for i in range(m-k):
        y[i] = ((x - xData[i+k])*y[i] + (xData[i] - x)*y[i+1])/ \
               (xData[i]-xData[i+k])
```

  - At start, 'y' contains the y-coordinates of the data
    - It's the y column of the Neville's table
  - Each pass through the outer loop, generate the next column of the Neville's table
    - It overwrite the component in 'y' from indices 'k'
  - At the end, 'y' contains the diagonal **(evaluated at x)**
    - The final value of the interpolant evaluated at x is y[0]

Remarks on polynomial interpolation (1)

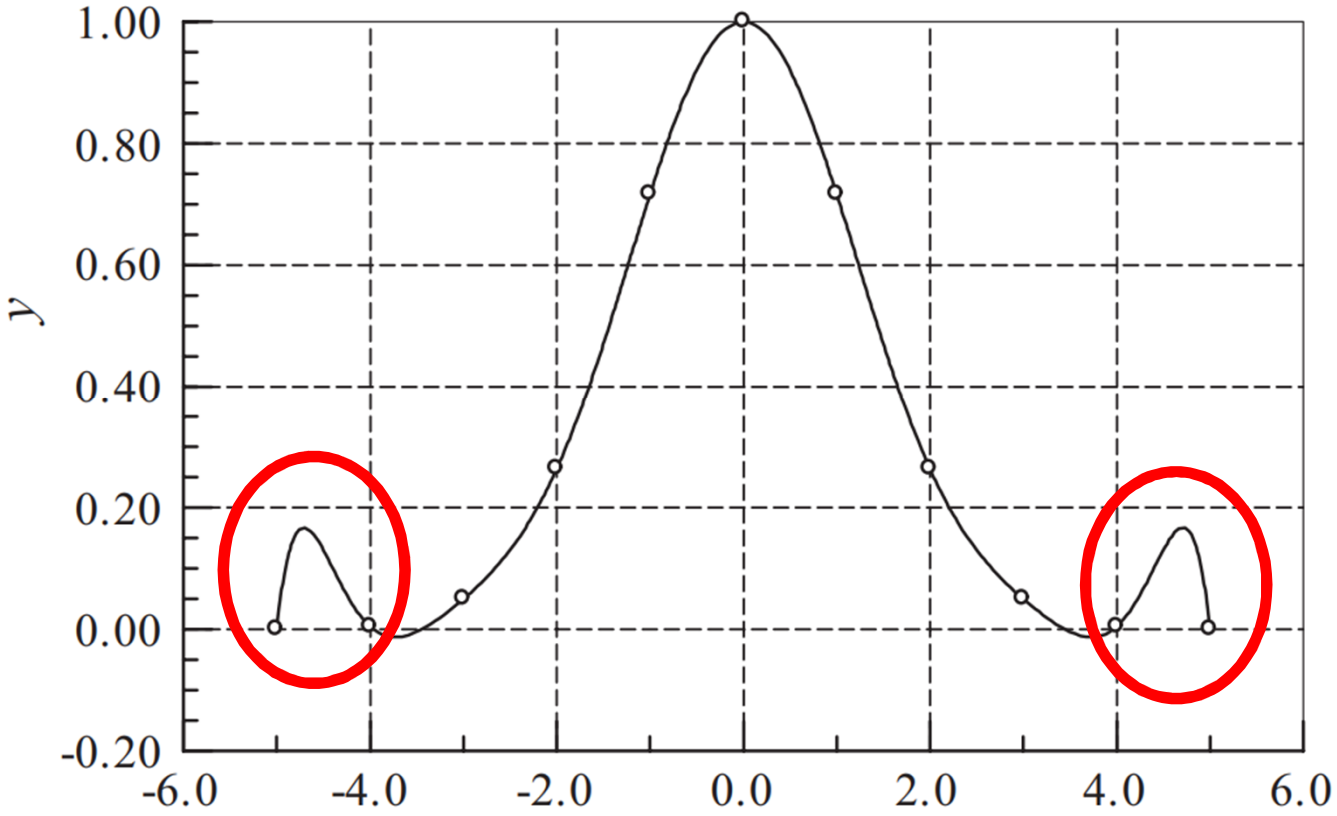
- **How the error of the polynomial approximation evolves with n (the degree of the polynomial) ?**
  - Does it increase or decrease ?
- We could believe that the error→0 when n→∞
  - Lagrange thought that too
  - But this is incorrect, in fact:
    error→∞ when n→∞
  - Known as the Runge Phenomena

- **Polynomial interpolation should be carried out with the fewest feasible number of points**

Remarks on polynomial interpolation (2)

Runge Phenomena

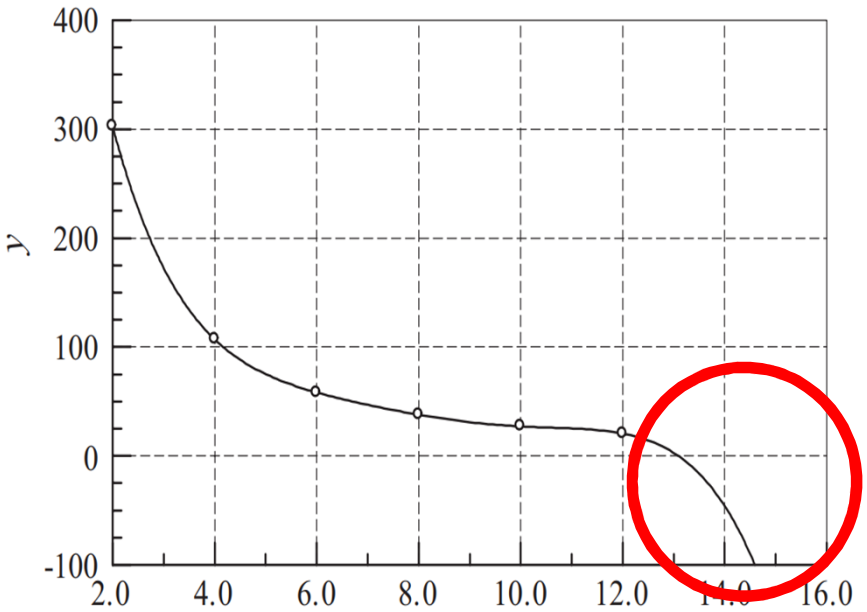
Do you see the problem ?

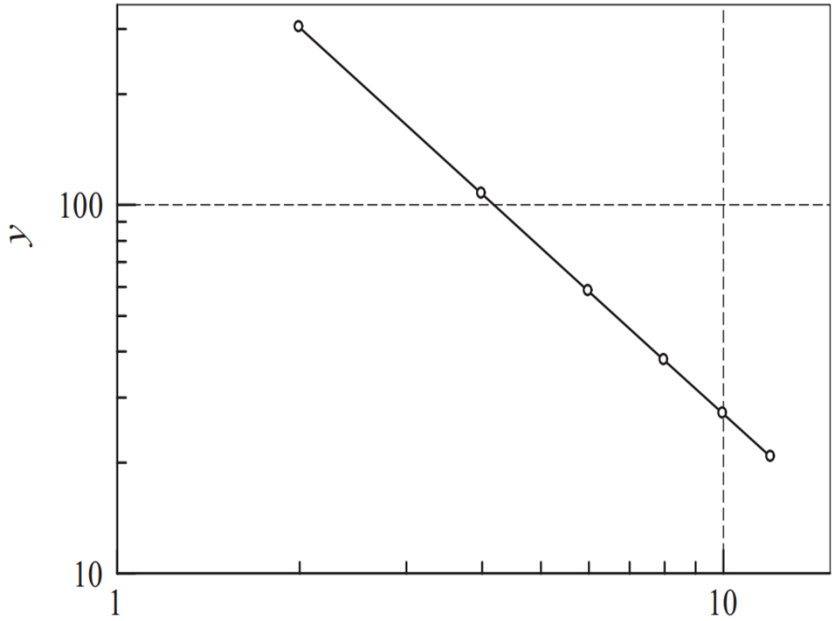


Polynomials with a large degree (here, n=10)
have a tendency to oscillate

Remarks on polynomial extrapolation

- **Polynomial extrapolation → interpolate outside the range**
- **Extrapolation is dangerous**
- **If there is no other options:**
  - **Visually inspect that the extrapolation make sense**
  - **Use a low order polynomial (with the nearest points)**
  - **Work with a log(x) vs log(y)**





Inverse Interpolation

- **Interpolation can be use to estimate the inverse of a function**
  - $y = f(x)$
  - $x = f^{-1}(y)$
- **We can exchange x and y in Neville's algorithm, to get an approximation of $f^{-1}(y)$**
- **Example: Find the root of f given the data:**
  - **Root the f → x such that $f(x) = 0$ → root = $f^{-1}(0)$**

x	4.0	3.9	3.8	3.7
y	−0.06604	−0.02724	0.01282	0.05383

i	y_i	$P_0[\,] = x_i$	$P_1[\,,\,]$	$P_2[\,,\,,\,]$	$P_3[\,,\,,\,,\,]$
0	−0.06604	4.0	?	?	?
1	−0.02724	3.9	?	?	
2	0.01282	3.8	?		
3	0.05383	3.7			

Inverse Interpolation

i	y_i	$P_0[\,] = x_i$	$P_1[\,,\,]$	$P_2[\,,\,,\,]$	$P_3[\,,\,,\,,\,]$
0	−0.06604	4.0	3.8298	3.8316	3.8317
1	−0.02724	3.9	3.8320	3.8318	
2	0.01282	3.8	3.8313		
3	0.05383	3.7			

- **Root estimation based on cubic interpolation over 4 points**
  - **3.8317**

- **But actually, all numbers in the table are estimates of the root based on**
  - **Different points**
  - **Different degree of the polynomials**

Radial Function Interpolation (1)

- Some data are better interpolated by **rational functions**
  - **Rational functions is the quotient of two polynomials**

$$R(x) = \frac{P_m(x)}{Q_n(x)} = \frac{a_1 x^m + a_2 x^{m-1} + \cdots + a_m x + a_{m+1}}{b_1 x^n + b_2 x^{n-1} + \cdots + b_n x + b_{n+1}}$$

- As R(x) is a ratio, it can be scaled such that one of the coeficient is unity
  - Generally, $b_{n+1} = 1$
  - m+n+1 undetermined coefficients → m+n+1 data points

- **Diagonal rational function**
  - Degree of the numerator (m) equals degree of the denominator (n)
    if m+n is even → **m = n**
  - Degree of the numerator (m) equals degree of the denominator minus 1 (n-1)
    if m+n is odd → **m = n-1**

  - Interpollation can be carried out with a Neville-type of algorithm

Radial Function Interpolation (2)

- Recursive formula

$$R[x_i, x_{i+1}, \ldots, x_{i+k}] = R[x_{i+1}, x_{i+2}, \ldots, x_{i+k}] + \frac{R[x_{i+1}, x_{i+2}, \ldots, x_{i+k}] - R[x_i, x_{i+1}, \ldots, x_{i+k-1}]}{S}$$

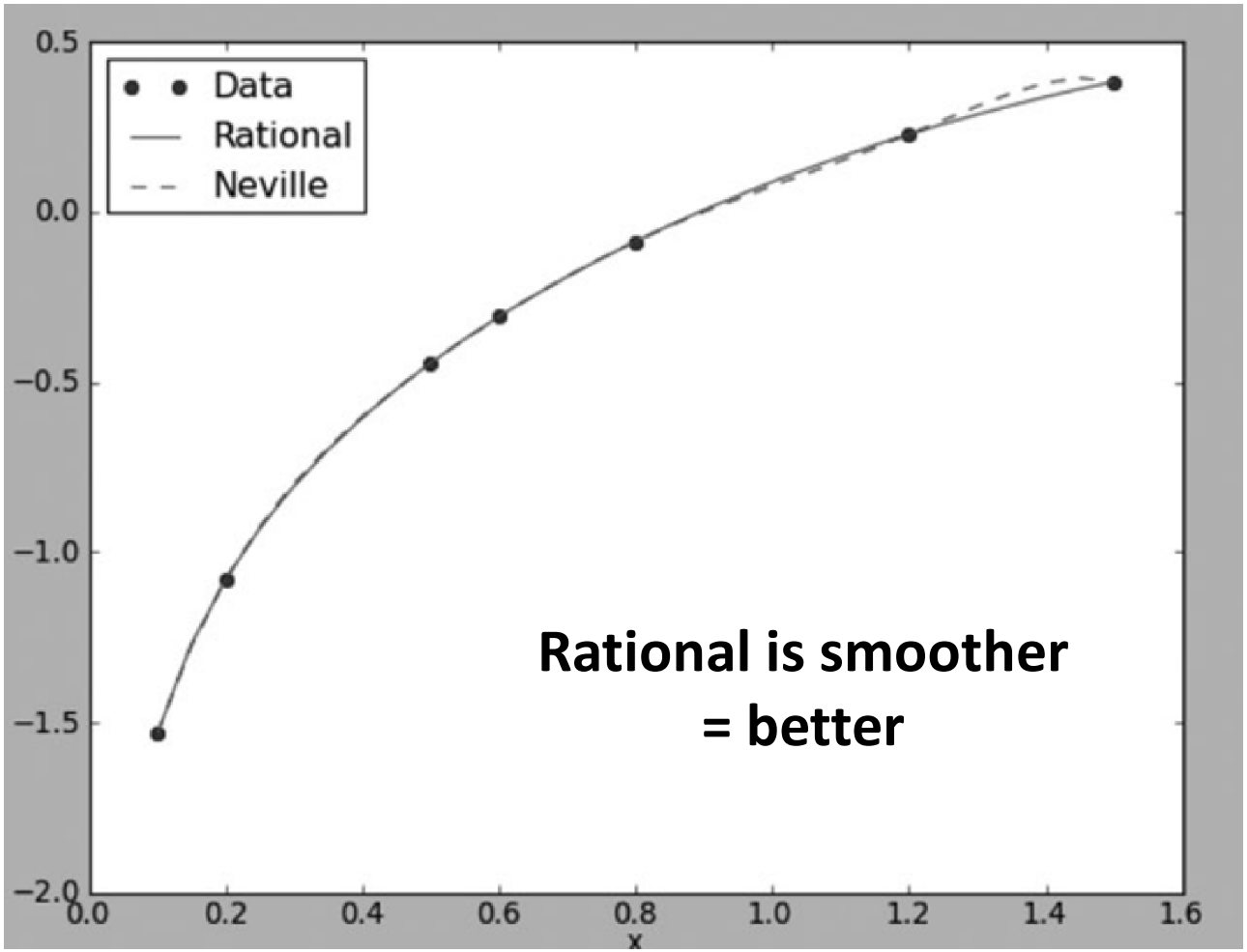
where

$$S = \frac{x - x_i}{x - x_{i+k}} \left(1 - \frac{R[x_{i+1}, x_{i+2}, \ldots, x_{i+k}] - R[x_i, x_{i+1}, \ldots, x_{i+k-1}]}{R[x_{i+1}, x_{i+2}, \ldots, x_{i+k}] - R[x_{i+1}, x_{i+2}, \ldots, x_{i+k-1}]} \right) - 1$$

	$k=-1$	$k=0$	$k=1$	$k=2$	$k=3$
x_1	0	$R[x_1] = y_1$	$R[x_1, x_2]$	$R[x_1, x_2, x_3]$	$R[x_1, x_2, x_3, x_4]$
x_2	0	$R[x_2] = y_2$	$R[x_2, x_3]$	$R[x_2, x_3, x_4]$	
x_3	0	$R[x_3] = y_3$	$R[x_3, x_4]$		
x_4	0	$R[x_4] = y_4$			

Radial Function Interpolation (3)

x	0.1	0.2	0.5	0.6	0.8	1.2	1.5
y	−1.5342	−1.0811	−0.4445	−0.3085	−0.0868	0.2281	0.3824



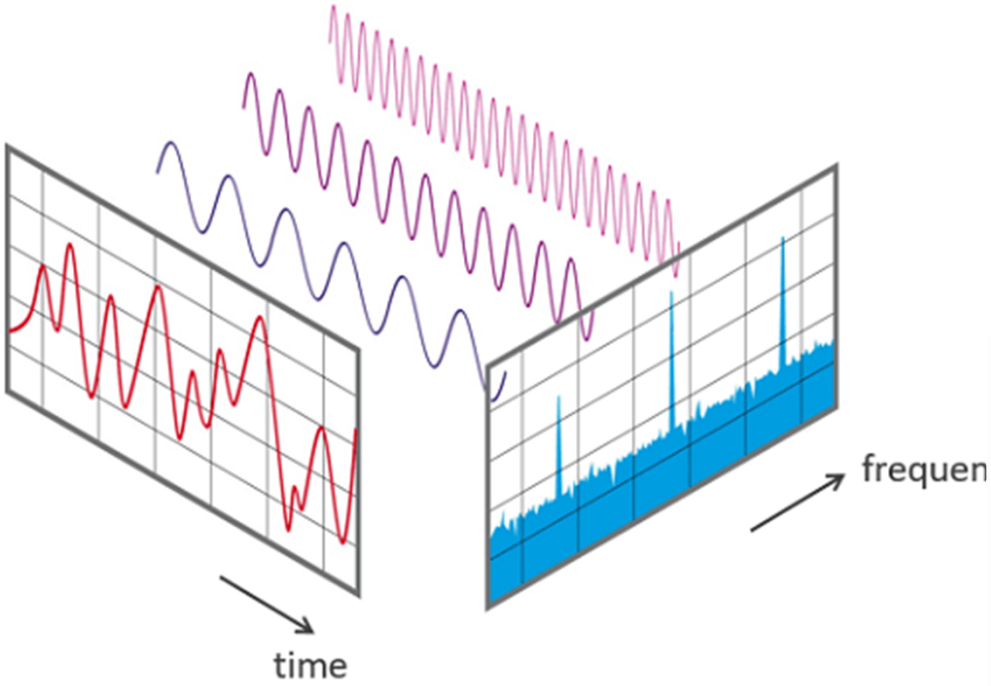
Trigonometric Interpolation

- There exists many other forms of interpolations

- For instance, if $\boldsymbol{f}$ is a periodic function,
  We will interpolate it with a basis of periodical functions
- $y_i = \phi(x_i)$
- $\phi(x) = a_0 + \sum_{k=1}^{K}(a_k \cos(kx) + b_k \sin(kx))$
- $\phi(x) = \sum_{k=-K}^{K} c_k e^{ikx}$ (with Euler notation)

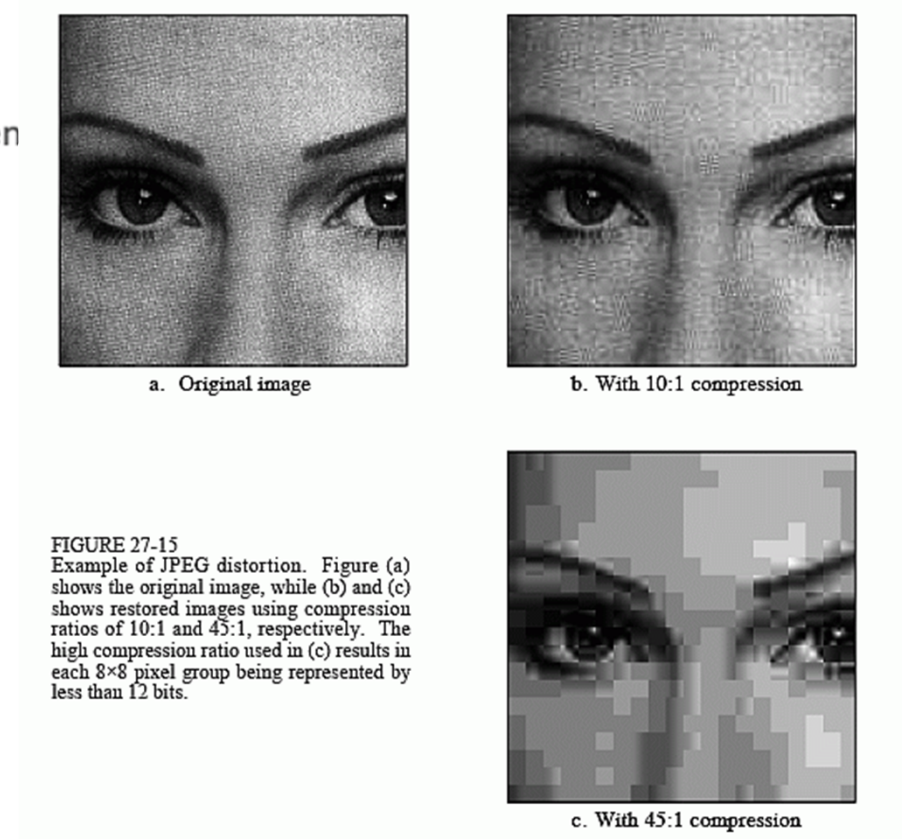
- 2K+1 unknowns → Unique solution if we have 2K+1 points
- In the special case where the x_i are evenly spaced,
  - This problem is the **Discrete Fourier Transform**
  - Efficient algorithm implementation**: Fast Fourier Transform (FFT)**

Application of FFT



- **Signal Decomposition**
- **Signal Filtering**

- **Compression**
  - **JPEG**
  - **MP3**


a. Original image

b. With 10:1 compression

c. With 45:1 compression

FIGURE 27-15
Example of JPEG distortion. Figure (a) shows the original image, while (b) and (c) shows restored images using compression ratios of 10:1 and 45:1, respectively. The high compression ratio used in (c) results in each 8×8 pixel group being represented by less than 12 bits.

Source:https://www.dspguide.com/ch27/6.htm