Computational practicum: Q2 – Lecture 6 Interpolation and regression

Zoë J.G. Gromotka, Ferdinand Grozema, Artur M. Schweidtmann, Tanuj Karia

With support from Sophia Rupprecht and Qinghe Gao

Computational Practicum
Dept. Chemical Engineering
Delft University of Technology

TU Delft

Process Intelligence RESEARCH

Delft Institute of Applied Mathematics

Learning objectives

After successfully completing this lecture, you are able to…

- explain interpolation and regression.
- discuss the advantages and limitations of different interpolation and regression methods.
- use Python libraries’ built-in functions for constructing Lagrange polynomials and Splines.
- derive the nominal equation for linear regression problems.
- implement the three ways to solve linear (or polynomial) regression problems from scratch.
- apply Python libraries’ built-in functions to nonlinear regression problems.

Agenda

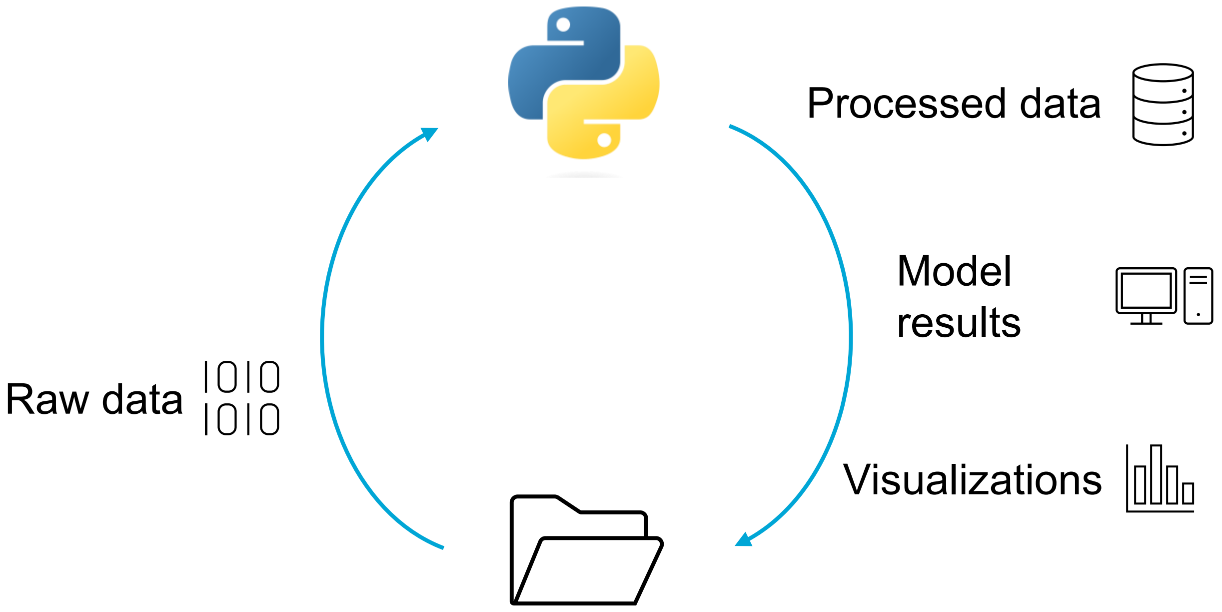
- Data loading and saving in Python
- Introduction
- Interpolation
- Regression

Agenda

- **Data loading and saving in Python**
- Introduction
- Interpolation
- Regression

Loading and saving data

- In scientific computing, data is often loaded and saved.
  - Loading raw data, e.g., from experimental setups
  - Saving processed data, model results, visualizations
- In Python, many types of data can be loaded and saved easily.
  - Text
  - Tabular data
  - ...



Loading raw text data

- Read raw text data with the `open` function.
- `with` … `as`: Context manager. Ensures that the file is closed properly if an error occurs.
- The `"r"` flag means the file is opened in read mode.
- Use `read` to get the entire content as string.
- Alternatively, iterate over the lines with a for-loop.

```python
# Open a file and get the content as string
with open("file.txt", "r") as file:
    data = file.read()
print(data)

# Or, iteratively get the lines of the file as
# string
with open("file.txt", "r") as file:
    for line in file:
        print(line)
```

Saving raw text data

- Save raw text data similarly with the `open` function.
- Use the `"w"` flag to open in write mode.
- Use `write` to write the string data in the file.
- This will overwrite any existing content of output.txt, not add a new line.
- If output.txt does not exist yet, a new file output.txt is created.

```python
# Open a file and get the content as string
with open("output.txt", "w") as file:
    file.write(data)
```

Saving and loading numpy arrays

- Numpy offers functions to save and load arrays quickly.
- For this, numpy has its own binary file format .npy.
- However, this is not an open file format and can only be used by numpy.

```python
import numpy as np

data = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

# Save the array to an .npy file
np.save("array.npy", data)

# Load the array back from the .npy file
loaded_data = np.load("array.npy")
```

Loading tabular data

- To load tabular data from csv, use the Pandas library to load it to numpy.
- The Pandas library is specialized for handling tabular data. We recommend taking a closer look: https://pandas.pydata.org/docs/getting_started/intro_tutorials/

```python
import pandas as pd

# Load the CSV into a Pandas DataFrame
df = pd.read_csv("data.csv")

# Convert the DataFrame to a NumPy array
data = df.to_numpy()

print(data)
```

Saving matplotlib visualizations

- Matplotlib is frequently used to plot results.
- Matplotlib figures can be saved with the `savefig` function.

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)
fig, ax = plt.subplots()

ax.plot(x, y, label="Sine Wave")
ax.set_title("Sine Wave Plot")
ax.set_xlabel("X-axis")
ax.set_ylabel("Y-axis")
ax.legend()

# Save the figure to a png file
fig.savefig("sine_wave_plot.png",
            format="png",
            dpi=300)
```

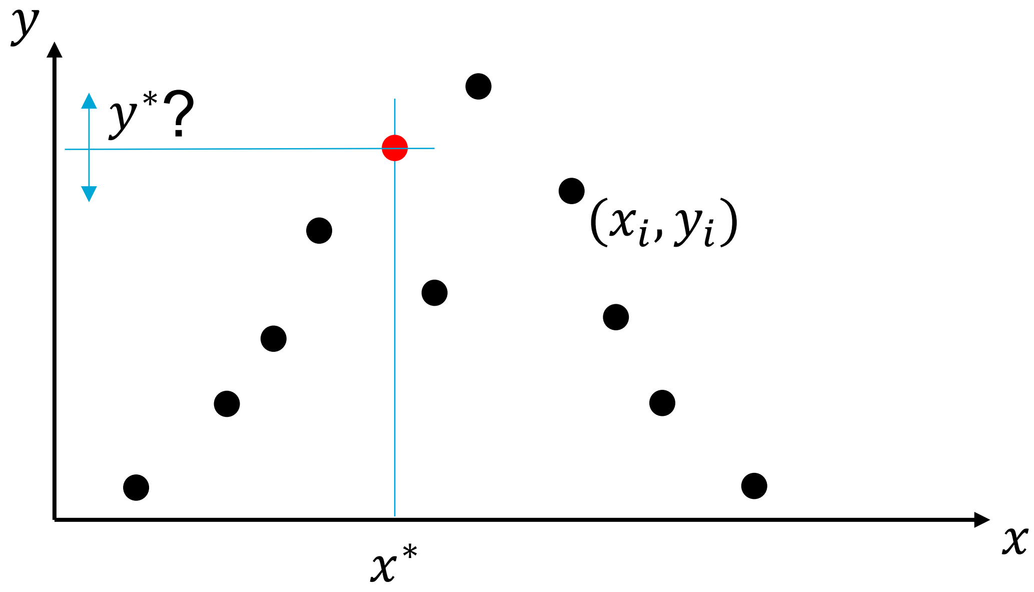
Agenda

- Data loading and saving in Python
- **Introduction**
- Interpolation
- Regression

Today's challenge

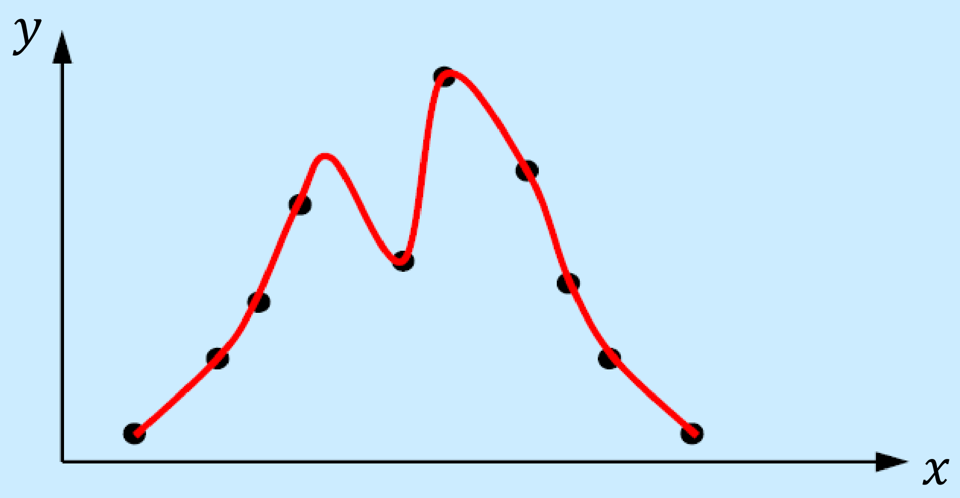
- We are given a set of N datapoints $\mathcal{D} = \{(x_i, y_i) | 1 \leq i \leq N\}$

- For x^* not in $\mathcal{D}$, how can we best find a <u>suitable</u> corresponding y^*?



Interpolation vs. regression

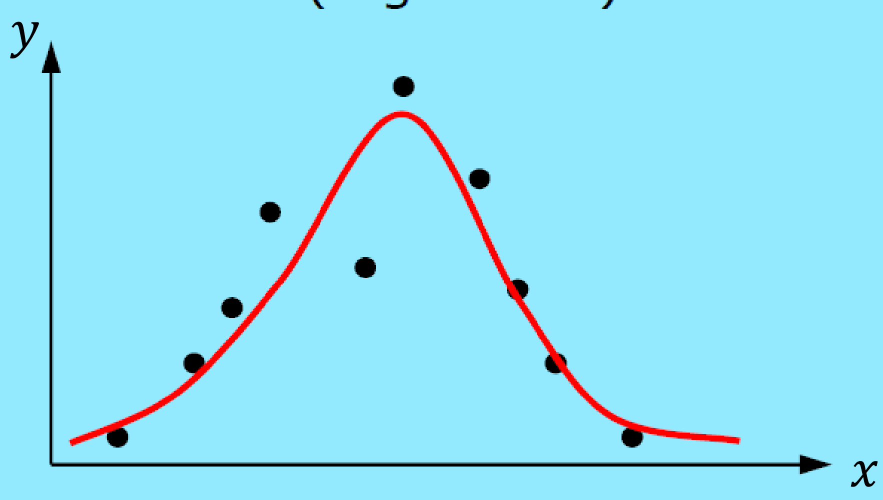
Interpolation



The curve passes through all the data points.

Estimating values between known data points

Approximation
(regression)



The curve does not have to pass through all the data points (but can pass through).

Fitting a model to data to describe general trends and make predictions

Examples of regression and interpolation (1/2)

- Thermodynamic Property Estimation:
  - Use interpolation (e.g., cubic spline) to estimate properties like enthalpy, specific heat, and entropy from tabulated data.
  - Regression to model temperature and pressure dependence of thermodynamic properties.
- Reaction Kinetics:
  - Regression to fit experimental data to rate laws (e.g., Arrhenius equation).
  - Nonlinear regression for complex reaction mechanisms.
- Process Optimization:
  - Regression used to build surrogate models for optimizing reactor and process parameters.
  - Polynomial regression to model relationships between operating conditions and product yield or purity.

Examples of Regression and Interpolation (2/2)

- Heat and Mass Transfer Correlations:
  - Regression for developing empirical correlations from experimental data (e.g., heat transfer coefficients, diffusivities).
  - Nonlinear regression for fitting complex transport models.
- Process Control and Monitoring:
  - Interpolation in real-time data for process variable estimation between measured points.
  - Regression to develop predictive models for fault detection or process optimization.
- Fluid Dynamics:
  - Use interpolation to estimate pressure and velocity profiles in fluid flow systems.
  - Regression for modeling pressure drop across packed beds or fluidized reactors.
- Chemical Reactor Design:
  - Fitting experimental conversion vs. time data using regression for reactor sizing and design. Predictive modeling of product distributions in multi-step reactions.

Agenda

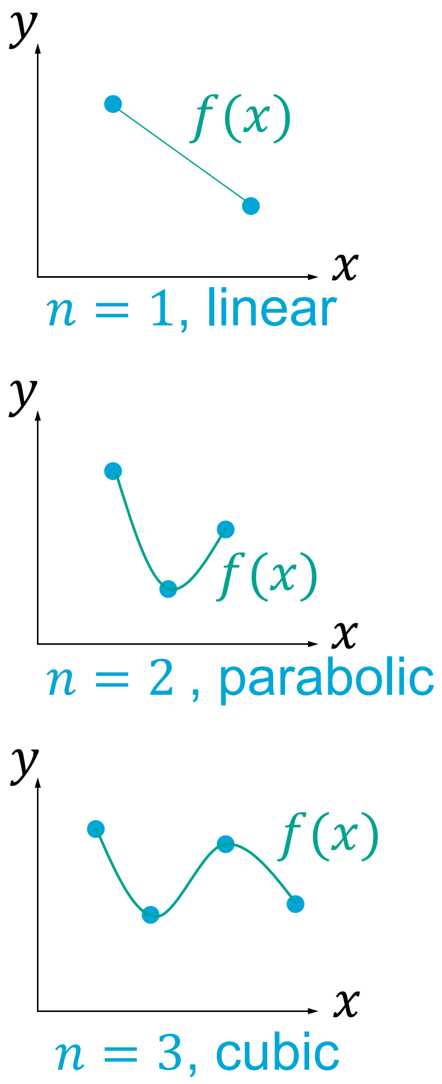
- Data loading and saving in Python
- Introduction
- **Interpolation**
  - **Polynomial interpolation**
  - Piecewise interpolation
    - Linear
    - Quadratic
    - Cubic
- Regression

Polynomial Interpolation

- Challenge: $n + 1$ discrete, precise data points $(\hat{x}_i, \hat{y}_i)$
- Task: Find a polynomial $f(x)$ of degree n

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

that *passes through all points*.



Excursus: Why not conventional polynomials?

Consider using conventional polynomials of the form

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

for interpolation. To determine the coefficients $a_0, a_1, \ldots, a_n$, the system of equations

$$f(\hat{x}_0) = a_0 + a_1 \hat{x}_0 + a_2 \hat{x}_0^2 + \cdots + a_n \hat{x}_0^n$$
$$f(\hat{x}_1) = a_0 + a_1 \hat{x}_1 + a_2 \hat{x}_1^2 + \cdots + a_n \hat{x}_1^n$$
$$\vdots$$
$$f(\hat{x}_n) = a_0 + a_1 \hat{x}_n + a_2 \hat{x}_n^2 + \cdots + a_n \hat{x}_n^n$$

must be solved.

See Q1 Lecture 3

These systems of equations are often **ill-conditioned**[1,2] and thus the computed coefficients $a_0, a_1, \ldots, a_n$ are highly inaccurate, especially for large n.

[1] Numerical Recipes: The Art of Scientific Computing, Third Edition, by W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Version 3.04 (2011), Chapter 3.5 Coefficients of the Interpolating Polynomial
[2] Victor Y. Pan. How bad are vandermonde matrices? April 2015. doi: 10.48550/ARXIV.1504.02118.

Excursus: What is an ill-conditioned system?

THIS SLIDE IS NOT RELEVANT FOR EXAM

Consider $\boldsymbol{Ax} = \boldsymbol{b}$. *How large is the effect of relative errors in the input $\boldsymbol{\delta_b}$ on the error of the solution $\boldsymbol{\delta_x}$?*

→ The condition of a system can be seen as a <u>measure for system sensitivity to perturbations $\widetilde{\boldsymbol{b}}$ in the input</u>. To quantify the condition of a problem described by a nonsingular matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, the <u>relative condition number</u> is computed given a matrix norm $\|\cdot\|$ according to

$$\kappa_{\|\cdot\|} = \frac{\boldsymbol{\delta_x}}{\boldsymbol{\delta_b}} = \frac{\|\boldsymbol{A}^{-1}\boldsymbol{b} - \boldsymbol{A}^{-1}\widetilde{\boldsymbol{b}}\|}{\|\boldsymbol{x}\|} \cdot \frac{\|\boldsymbol{b}\|}{\|\boldsymbol{b} - \widetilde{\boldsymbol{b}}\|} = \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|.$$

The condition number is a property of the system $Ax = b$, not the algorithm used for solving it!

A problem is considered ill-conditioned if $\kappa_{\|\cdot\|}$ is large indicating a large amplification of errors in the input on errors in the solution.

Lagrange interpolating polynomials

- We do not want to solve/approximate the solution of a linear system of equations to compute the coefficients of our interpolating polynomial!
- Alternative: Lagrange interpolating polynomial $f_n(x)$:

$$f_n(x) = \sum_{i=0}^{n} L_i(x)\hat{y}_i \quad \text{with} \quad L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - \hat{x}_j}{\hat{x}_i - \hat{x}_j}$$

- The Lagrange interpolating polynomial $f_n(x)$ passes through all given data points by creating fractions $L_i(x)$, which cancel out to 1 at the given datapoints $(\hat{x}_i, \hat{y}_i)$ (i.e., at the knots).

Lagrange interpolating polynomials: Example 1/2

$$f_n(x) = \sum_{i=0}^{n} L_i(x)\hat{y}_i \quad \text{with} \quad L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-\hat{x}_j}{\hat{x}_i-\hat{x}_j}$$

- Examples:
  - Two datapoints $(\hat{x}_0, \hat{y}_0), (\hat{x}_1, \hat{y}_1) \rightarrow n = 1$

$$f_1(x) = \frac{x-\hat{x}_1}{\hat{x}_0-\hat{x}_1}\hat{y}_0 + \frac{x-\hat{x}_0}{\hat{x}_1-\hat{x}_0}\hat{y}_1$$

$$f_1(x=\hat{x}_0) = \underbrace{\cancel{\frac{\hat{x}_0-\hat{x}_1}{\hat{x}_0-\hat{x}_1}}}_{=1}\hat{y}_0 + \underbrace{\cancel{\frac{\hat{x}_0-\hat{x}_0}{\hat{x}_1-\hat{x}_0}}}_{=0}\hat{y}_1 = \hat{y}_0$$

Lagrange interpolating polynomials: Example 2/2

$$f_n(x) = \sum_{i=0}^{n} L_i(x)\hat{y}_i \quad \text{with} \quad L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-\hat{x}_j}{\hat{x}_i-\hat{x}_j}$$

- Examples:
  - Three datapoints $(\hat{x}_0, \hat{y}_0), (\hat{x}_1, \hat{y}_1), (\hat{x}_2, \hat{y}_2) \rightarrow n = 2$

$$f_2(x) = \frac{x-\hat{x}_1}{\hat{x}_0-\hat{x}_1}\frac{x-\hat{x}_2}{\hat{x}_0-\hat{x}_2}\hat{y}_0 + \frac{x-\hat{x}_0}{\hat{x}_1-\hat{x}_0}\frac{x-\hat{x}_2}{\hat{x}_1-\hat{x}_2}\hat{y}_1 + \frac{x-\hat{x}_0}{\hat{x}_2-\hat{x}_0}\frac{x-\hat{x}_1}{\hat{x}_2-\hat{x}_1}\hat{y}_2$$

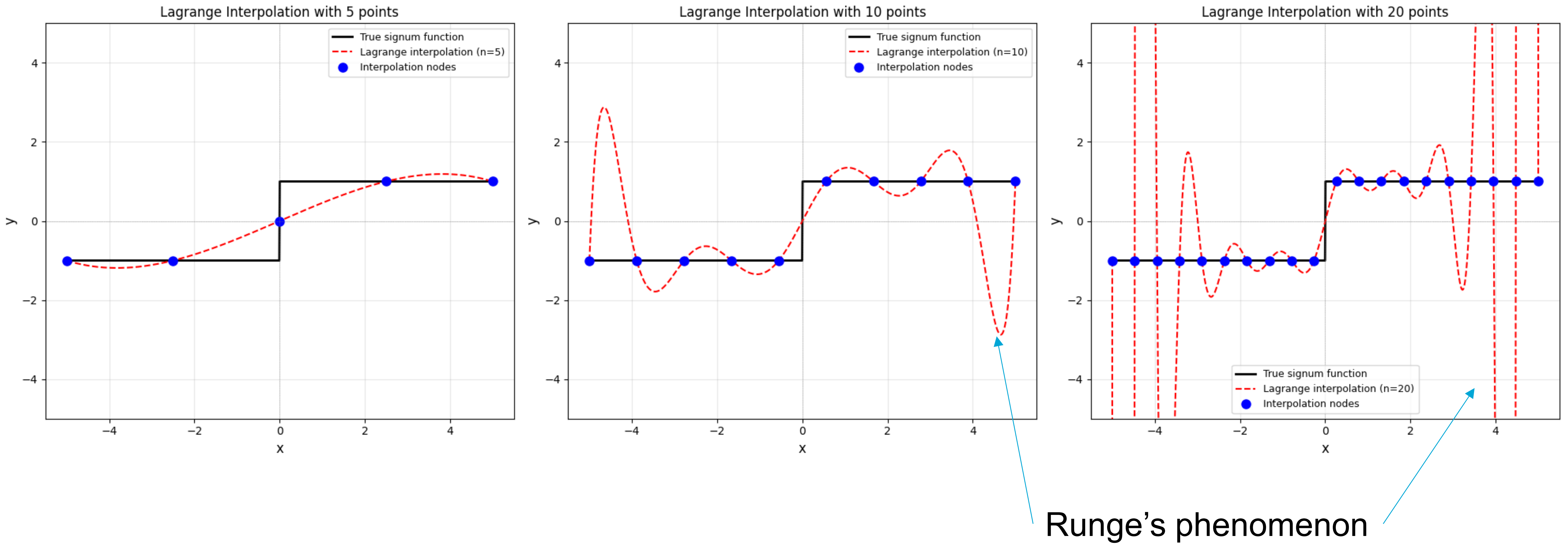
$$f_2(x=\hat{x}_0) = \underbrace{\cancel{\frac{\hat{x}_0-\hat{x}_1}{\hat{x}_0-\hat{x}_1}}}_{=1}\underbrace{\cancel{\frac{\hat{x}_0-\hat{x}_2}{\hat{x}_0-\hat{x}_2}}}_{=1}\hat{y}_0 + \underbrace{\cancel{\frac{\hat{x}_0-\hat{x}_0}{\hat{x}_1-\hat{x}_0}}}_{=0}\frac{\hat{x}_0-\hat{x}_2}{\hat{x}_1-\hat{x}_2}\hat{y}_1 + \underbrace{\cancel{\frac{\hat{x}_0-\hat{x}_0}{\hat{x}_2-\hat{x}_0}}}_{=0}\frac{\hat{x}_0-\hat{x}_1}{\hat{x}_2-\hat{x}_1}\hat{y}_2 = \hat{y}_0$$

Lagrange interpolation with SciPy

- Main function: scipy.interpolate.lagrange()
- Purpose: Creates a polynomial that passes through a given set of points.
- Key Features: Simple to use for small data sets.
- **Returns** the interpolation polynomial as a **callable object**.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import lagrange

# Sample data points
x_points = np.array([0, 1, 2, 3])
y_points = np.array([1, 2, 0, 2])

# Compute the Lagrange polynomial
poly = lagrange(x_points, y_points)

# Generate a smooth curve for visualization
x_fine = np.linspace(min(x_points), max(x_points), 500)
y_fine = poly(x_fine)


# Plot the results
plt.plot(x_points, y_points, 'o', label="Data Points")
plt.plot(x_fine, y_fine, '-', label="Lagrange Polynomial")
```

Interpolating polynomials for the Signum function



How can we prevent such oscillations?

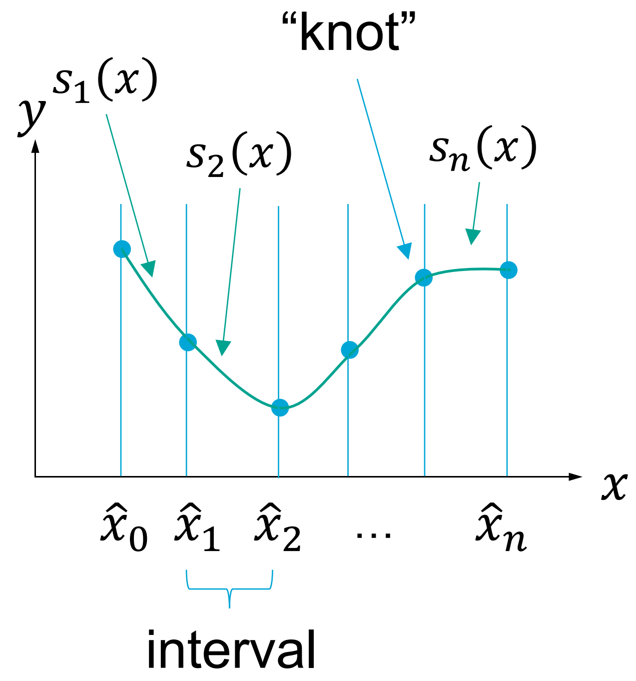
https://en.wikipedia.org/wiki/Runge%27s_phenomenon

Agenda

- Data loading and saving in Python
- Introduction
- **Interpolation**
  - Polynomial interpolation
  - **Piecewise interpolation**
    - Linear
    - Quadratic
    - Cubic
- Regression

Piecewise (or spline) interpolation

- Break the data into smaller intervals and fit lower-order polynomials $s_i(x)$ within each interval
- **Splines** are piecewise polynomials going (1) through all given data points and (2) satisfying certain continuity conditions [1]
- Ensures better control over oscillations compared to single high-degree polynomials.
- Types of piecewise interpolation:
  - Linear
  - Quadratic
  - Cubic



[1] J. Trygg, J. Gabrielsson, and T. Lundstedt. Background Estimation, Denoising, and Preprocessing, pages 1–8. Elsevier,2009. ISBN 9780444527011. doi: 10.1016/b978-044452701-1.00097-1.

Agenda

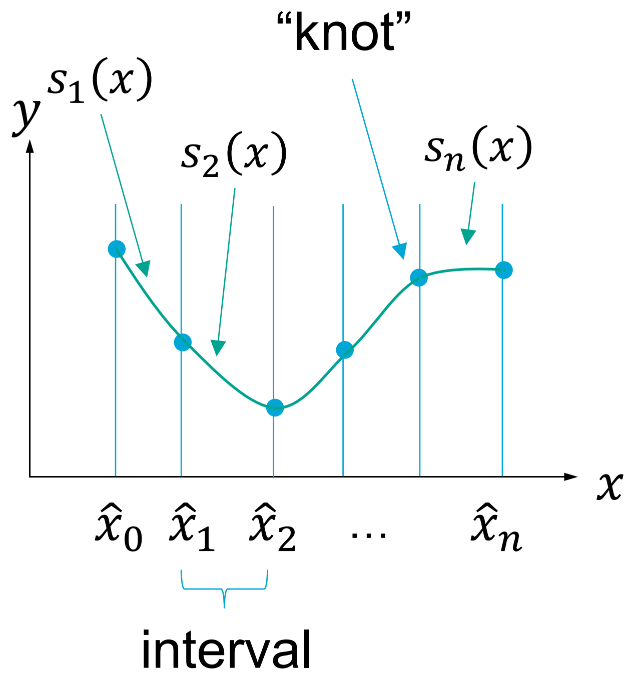
- Data loading and saving in Python
- Introduction
- **Interpolation**
  - Polynomial interpolation
  - **Piecewise interpolation**
    - **Linear**
    - Quadratic
    - Cubic
- Regression

Piecewise interpolation: linear spline

Generic linear function: $\boxed{s_i(x) = a_i x + b_i.}$

This can be explicitly calculated given the two neighbouring data points

$$s_i(x) = \frac{f(\hat{x}_i) - f(\hat{x}_{i-1})}{\hat{x}_i - \hat{x}_{i-1}}(x - \hat{x}_{i-1}) + f(\hat{x}_{i-1}).$$



Piecewise linear interpolation with SciPy

- `linear_interpolator = interp1d(x_points, y_points, kind='linear')`
- Interpolation method: kind='linear' (default).
- Returns a callable function for evaluating intermediate values.

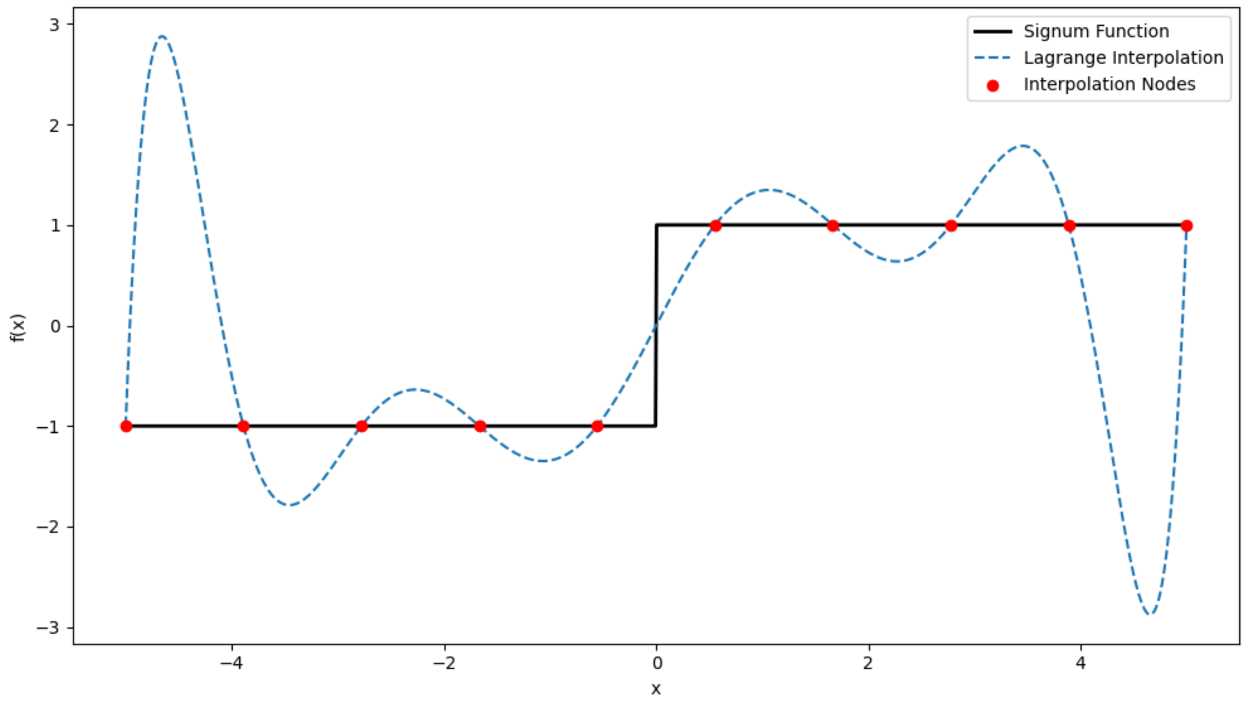
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d


# Sample data points
x_points = np.array([0, 1, 2, 3])
y_points = np.array([1, 2, 0, 2])

# Create a piecewise linear interpolator
linear_interpolator = interp1d(x_points, y_points,
kind='linear')

# Generate points for plotting the interpolated
function
x_fine = np.linspace(min(x_points), max(x_points),
500)
y_fine = linear_interpolator(x_fine)


# Plot the results
plt.plot(x_points, y_points, 'o', label="Data
Points")
plt.plot(x_fine, y_fine, '-', label="Lagrange
Polynomial")
```

Scipy.interpolate.interp1d: legacy

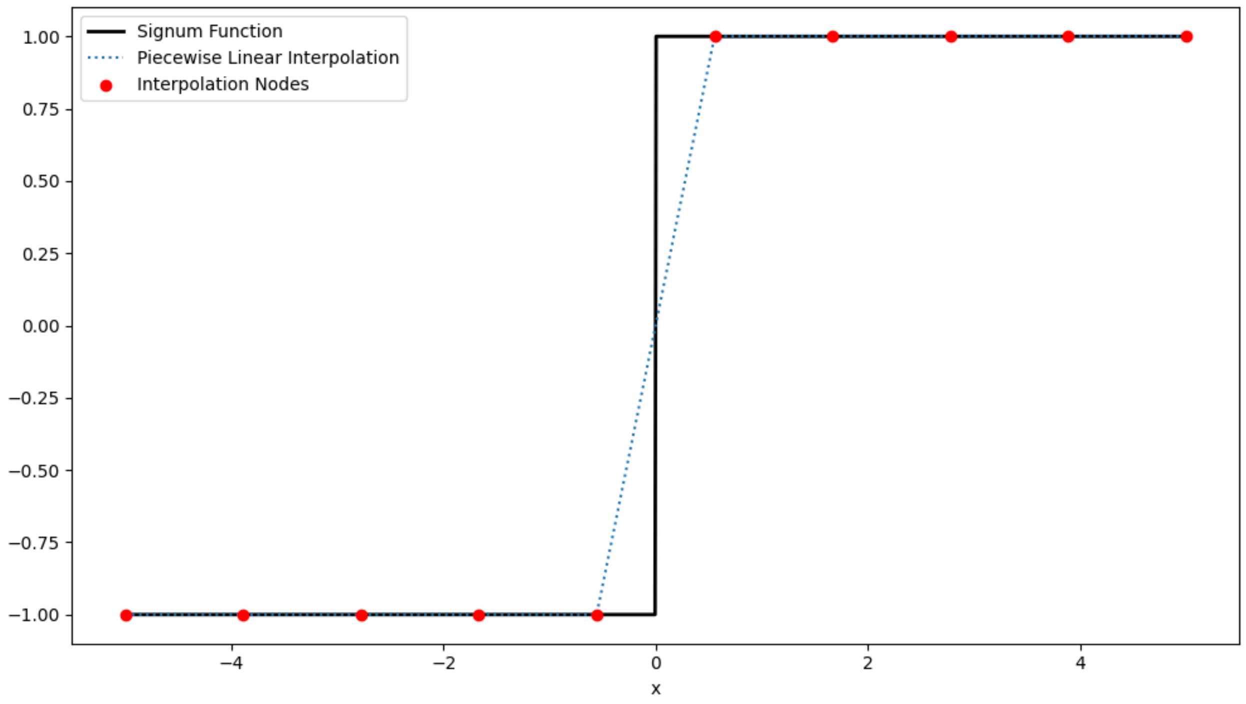
- Class scipy.interpolate.interp1d is considered legacy and will not receive updates
- Alternatives for 1D interpolation using scipy.interpolate:
  - Piecewise linear interpolation using numpy.interp
  - Cubic splines using the CubicSpline class of scipy.interpolate
  - scipy.interpolate.make_interp_spline using Bsplines

Piecewise interpolation avoid Runge's phenomenon

Lagrange interpolating polynomial
with 10 points



Piecewise interpolation
(1^{st} degree polynomial)



Excursus: Continuity

Let $f: I \to \mathbb{R}$, I an interval, and $x_0 \in I$.

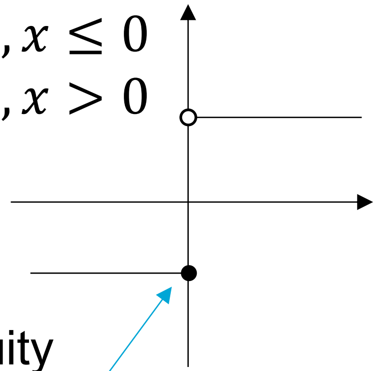
- f is continuous at x_0 if

$$\lim_{x \to x_0} f(x) = f(x_0)$$

- i.e. for each $\epsilon > 0$ there exists a $\delta > 0$ with $|f(x) - f(x_0)| < \epsilon$, if $|x - x_0| < \delta$.
- f is continuous on I if f is continuous at every point in I.

Examples:

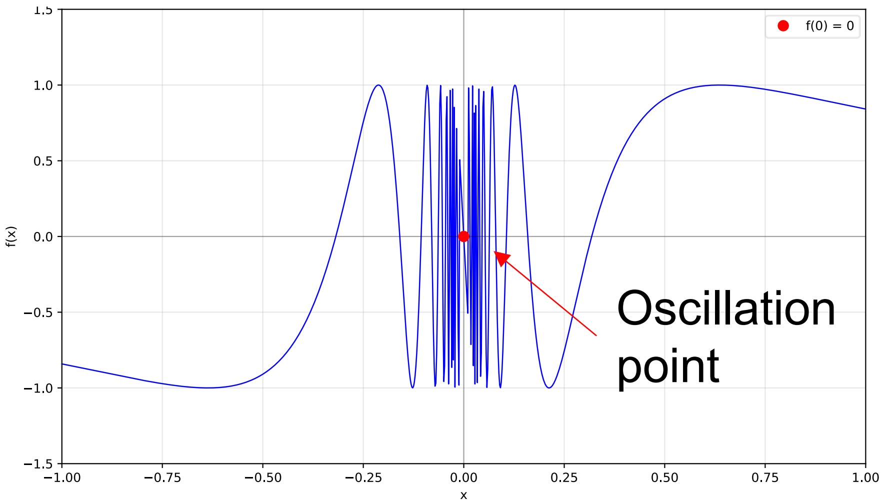
$$f(x) = \begin{cases} 0, x \leq 0 \\ 1, x > 0 \end{cases}$$



Discontinuity at $x_0 = 0$

$$f(x) = \begin{cases} \sin\left(\frac{1}{x}\right), x \neq 0 \\ 0, x = 0 \end{cases}$$

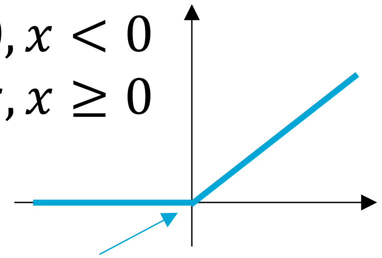
$\lim_{x \to 0} \sin\left(\frac{1}{x}\right)$ does not exist → $f(x)$ is not continuous at 0.



Excursus: Smoothness

- "A function of **class C_k** is a function of smoothness at least k; that is, a function of class C_k is a function that has a **kth derivative that is continuous** in its domain." [1]
- Examples:
  - C_0 Continuous but not differentiable
  - C_1 Continuous and differentiable but non-continuous second derivative
  - C_2 Continuous, differentiable, and continuous second derivative

$$f(x) = \begin{cases} 0, x < 0 \\ x, x \geq 0 \end{cases}$$



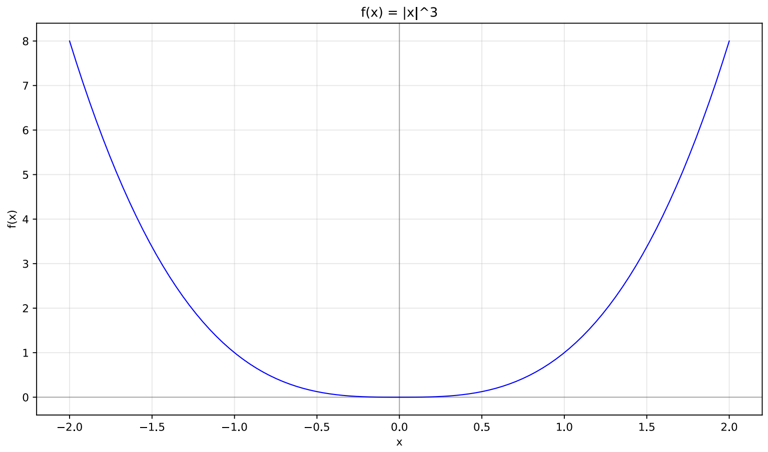
Non-differentiable at 0, but continuous→ C_0 and not C_1

$$f(x) = |x|^3$$

- Second derivative exists and is continuous
- But the second derivative is non-differentiable at 0

→ Non-continuous third derivative

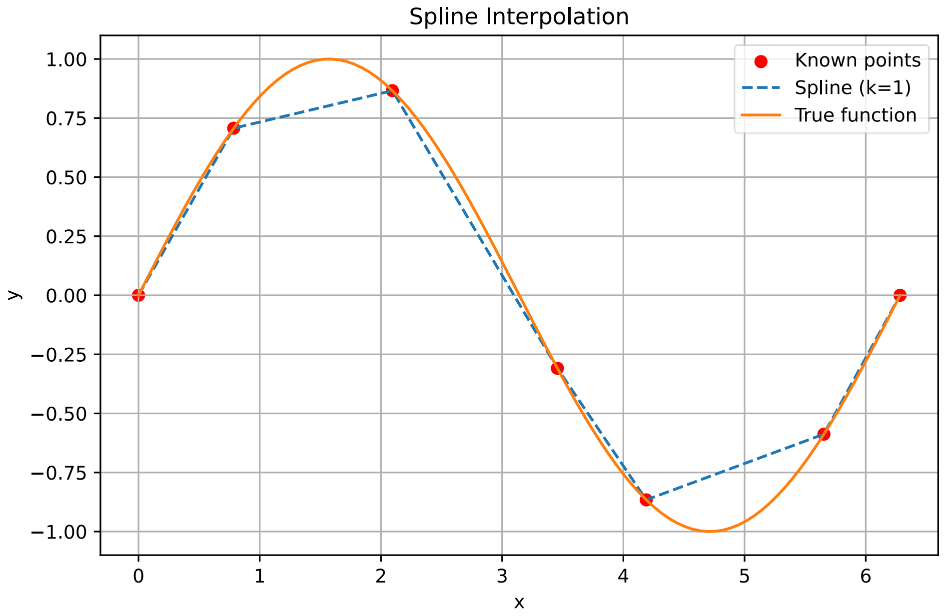
→ C_2 but not C_3



[1] https://en.wikipedia.org/wiki/Smoothness

Advantages and disadvantages of linear splines

- **Advantages**:
  - Simple and fast: Easy to compute and implement, especially for real-time application
  - Works well for datasets with low curvature.
  - Avoid Runge's phenomenon
- **Disadvantages**:
  - Accuracy limitations: Poor fit for highly curved data.
  - Continuous but not differentiable function (C_0) [1]
  - No physical model equations (→ difficult to interpret)



[1] https://en.wikipedia.org/wiki/Smoothness

Agenda

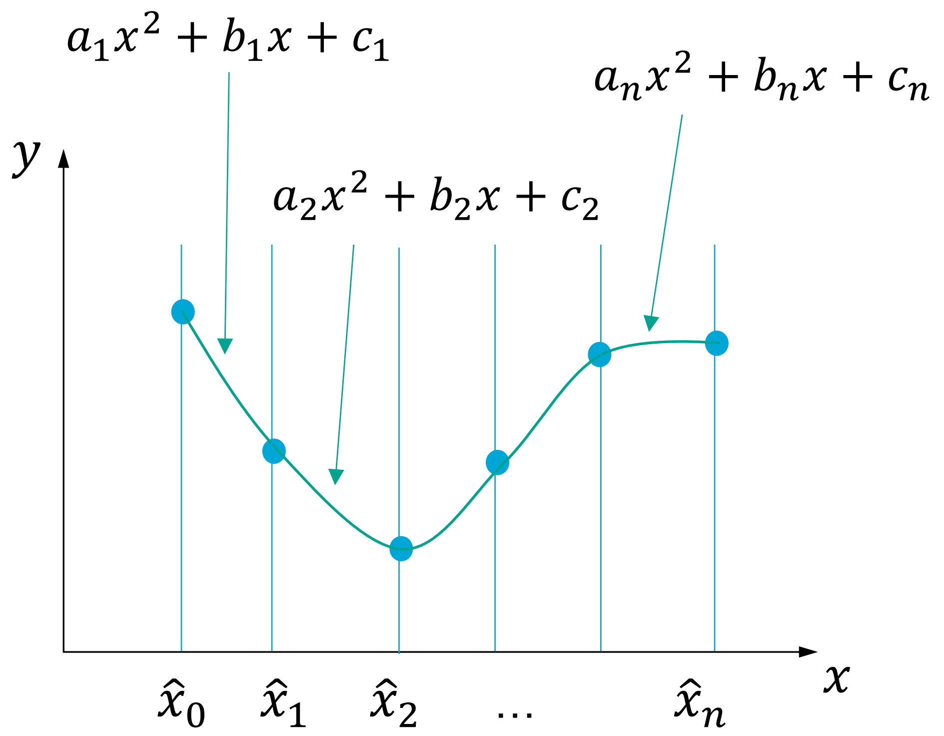
- Data loading and saving in Python
- Introduction
- **Interpolation**
  - Polynomial interpolation
  - **Piecewise interpolation**
    - Linear
    - **Quadratic**
    - Cubic
- Regression

Piecewise interpolation: Quadratic spline

- Generic quadratic function:

$$s_i(x) = a_i x^2 + b_i x + c_i$$

- To construct n linear splines for $n + 1$ datapoints, we have **$3n$ unknowns** $(a, b, c$ for each spline).
- → We need to specify $3n$ conditions.



Quadratic spline: derivation (1/2)

1. *Function values of adjacent polynomials must be equal at interior points and equal to known values.*

$$\left.\begin{aligned} a_{i-1}\hat{x}_{i-1}^2 + b_{i-1}\hat{x}_{i-1} + c_{i-1} &= f(\hat{x}_{i-1}) \\ a_i\hat{x}_{i-1}^2 + b_i\hat{x}_{i-1} + c_i &= f(\hat{x}_{i-1}) \end{aligned}\right\}$$

for $i = 2, \ldots, n$
→ $2(n-1)$ conditions

2. *Function values must be equal to known values at endpoints.*

$$\left.\begin{aligned} a_0\hat{x}_0^2 + b_0\hat{x}_0 + c_0 &= f(\hat{x}_0) \\ a_n\hat{x}_n^2 + b_n\hat{x}_n + c_n &= f(\hat{x}_n) \end{aligned}\right\}$$

for $i = 0$ and $i = n$
→ 2 conditions

3. *First order derivatives of adjacent polynomials must be equal at interior points*, i.e., $f'(x) = 2ax + b$.

$$2a_{i-1}\hat{x}_{i-1} + b_{i-1} = 2a_i\hat{x}_{i-1} + b_i \quad \}$$

for $i = 2, \ldots, n$
→ $n - 1$ conditions

Quadratic spline: derivation (2/2)

How many conditions have we specified now?

$$2n - 2 + 2 + n - 1 = 3n - 1$$

→ We need to specify one more condition!

4. *Second order derivative is zero at first point.*

$$a_1 = 0$$

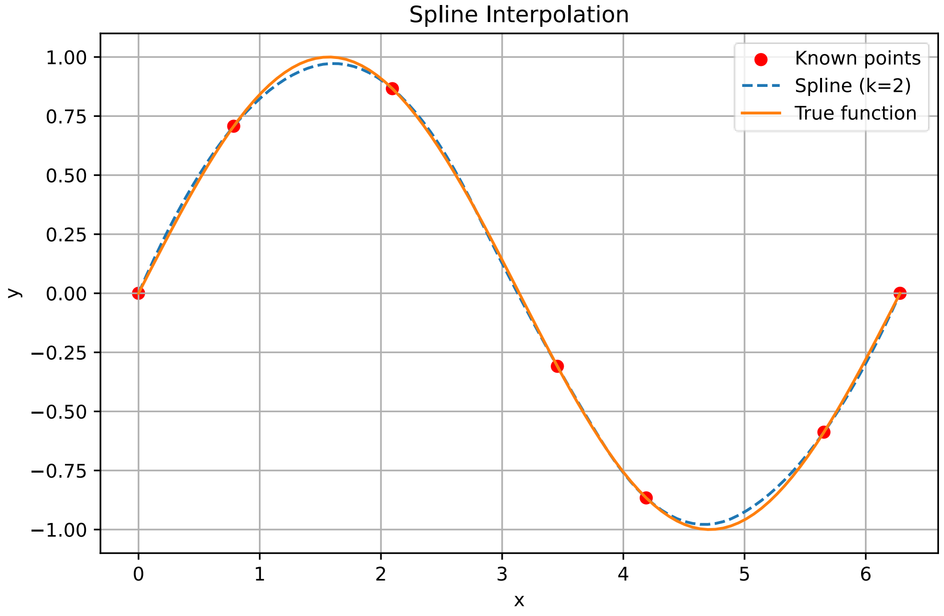
Piecewise quadratic interpolation with SciPy

- `linear_interpolator = interp1d(x_points, y_points, kind='quadratic')`
- Interpolation method: kind=quadratic'.
- Returns a callable function for evaluating intermediate values.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d


# Sample data points
x_points = np.array([0, 1, 2, 3])
y_points = np.array([1, 2, 0, 2])

# Create a piecewise linear interpolator
linear_interpolator = interp1d(x_points, y_points,
kind='quadratic')

# Generate points for plotting the interpolated
function
x_fine = np.linspace(min(x_points), max(x_points),
500)
y_fine = linear_interpolator(x_fine)

# Plot the results
plt.plot(x_points, y_points, 'o', label="Data
Points")
plt.plot(x_fine, y_fine, '-', label="Lagrange
Polynomial")
```

Advantages and disadvantages of quadratic splines

- **Advantages**:
  - Smoothness: Once continuously differentiable.
  - Simplicity: Simpler to construct compared to single higher-order polynomials
- **Disadvantages**:
  - Continuous and differentiable but non-continuous second derivative (C_1) [1] *[Note that this may depend on the implementation, some may be only C_0]*
- No physical model equations (→ difficult to interpret)



[1] https://en.wikipedia.org/wiki/Smoothness

Agenda

- Data loading and saving in Python
- Introduction
- **Interpolation**
  - Polynomial interpolation
  - **Piecewise interpolation**
    - Linear
    - Quadratic
    - **Cubic**
- Regression

Cubic spline: derivation (1/2)

Generic cubic spline function: $f_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$

→ For $n+1$ datapoints, i.e., n intervals, we need to specify $4n$ conditions

1. *Function values must be equal at interior points.*

$$a_{i-1}\hat{x}_{i-1}^3 + b_{i-1}\hat{x}_{i-1}^2 + c_{i-1}\hat{x}_{i-1} + d_{i-1} = f(\hat{x}_{i-1})$$
$$a_i\hat{x}_{i-1}^3 + b_i\hat{x}_{i-1}^2 + c_i\hat{x}_{i-1} + d_i = f(\hat{x}_{i-1})$$

for $i = 2, \dots, n$
→ $2(n-1)$ conditions

2. *Function values must be equal to known values at endpoints.*

$$a_0\hat{x}_0^3 + b_0\hat{x}_0^2 + c_0\hat{x}_0 + d_0 = f(\hat{x}_0)$$
$$a_n\hat{x}_n^3 + b_n\hat{x}_n^2 + c_n\hat{x}_n + d_n = f(\hat{x}_n)$$

for $i = 0$ and $i = n$
→ 2 conditions

3. *First order derivatives of adjacent polynomials must be equal at interior points.*

$$3a_{i-1}\hat{x}_{i-1}^2 + 2b_{i-1}\hat{x}_{i-1} + c_{i-1} = 3a_i\hat{x}_{i-1}^2 + 2b_i\hat{x}_{i-1} + c_i$$

for $i = 2, \dots, n$
→ $n-1$ conditions

Cubic spline: derivation (2/2)

4. *Second order derivatives of adjacent polynomials must be equal at interior points.*

$$6a_{i-1}\hat{x}_{i-1} + 2b_{i-1} = 6a_i\hat{x}_{i-1} + 2b_i$$

for $i = 2, \ldots, n$
→ $n - 1$ conditions

5. *Second order derivatives at endpoints are zero.*

$$6a_n\hat{x}_n + 2b_n = 0$$

$$6a_0\hat{x}_0 + 2b_0 = 0$$

for $i = 0$ and $i = n$
→ 2 conditions

→ Ensures <u>smoothness</u> by matching the first and second derivatives at the interval boundaries: avoid oscillations.

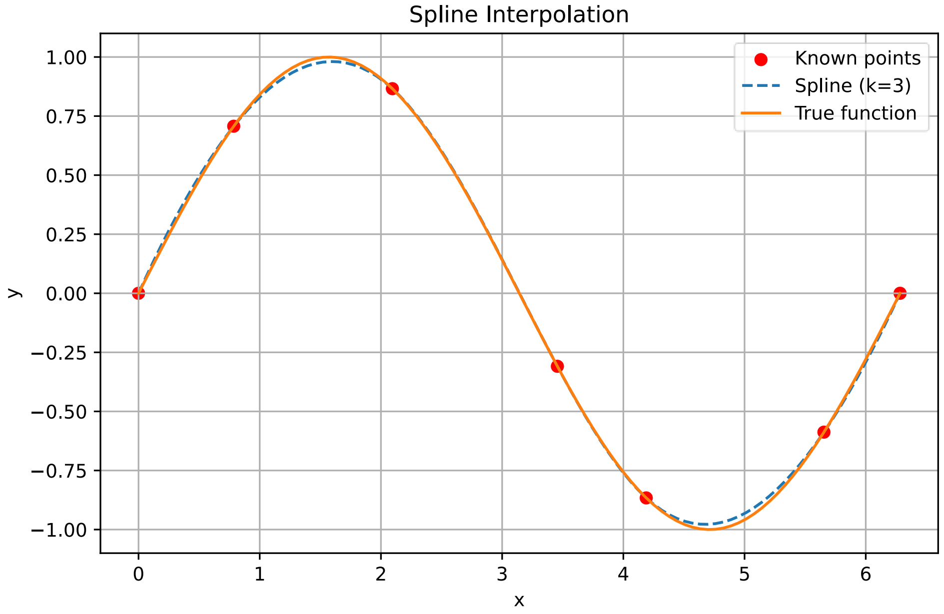
Piecewise **cubic** interpolation with SciPy

- Interpolation method: kind=cubic'.
- Returns a callable function for evaluating intermediate values.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d


# Sample data points
x_points = np.array([0, 1, 2, 3])
y_points = np.array([1, 2, 0, 2])

# Create a piecewise linear interpolator
linear_interpolator = interp1d(x_points, y_points,
kind='cubic')

# Generate points for plotting the interpolated
function
x_fine = np.linspace(min(x_points), max(x_points),
500)
y_fine = linear_interpolator(x_fine)

# Plot the results
plt.plot(x_points, y_points, 'o', label="Data
Points")
plt.plot(x_fine, y_fine, '-', label="Lagrange
Polynomial")
```

Advantages and disadvantages of **cubic** splines

- **Advantages**:
  - Smooth transitions: Provides a continuous curve with smooth derivatives at interval boundaries (C_2) [1]
  - Accurate: Better fit for data with curvature.
  - Reduces oscillation: Handles larger datasets without the issues seen in higher-degree polynomial interpolation.
- **Disadvantages**:
  - Requires solving a system of equations to compute coefficients.
  - No physical model equations (→ difficult to interpret)



[1] https://en.wikipedia.org/wiki/Smoothness

Live coding: Linear interpolation in a steam table

- Open Colab: [Interpolation]

	t (°C)								
P (bar)	0	20	50	100	150	200	250	300	350
1	−0.085 2	0.206 7	0.462 3						
5	−0.083 8	0.207 2	0.462 2	0.753 9	1.024				
10	−0.082 0	0.207 9	0.462 0	0.753 0	1.022				
50	−0.067 8	0.213 3	0.460 5	0.745 5	1.007	1.347	1.936		
100	−0.049 9	0.220 1	0.458 9	0.736 6	0.990 2	1.312	1.848	3.189	
150	−0.032 0	0.227 2	0.457 4	0.728 1	0.974 0	1.281	1.772	2.883	
200	−0.014 2	0.234 3	0.456 2	0.720 0	0.958 7	1.251	1.704	2.648	6.923
250	+0.003 3	0.241 6	0.455 1	0.712 2	0.944 2	1.224	1.643	2.460	5.162
300	0.020 5	0.248 9	0.454 2	0.704 7	0.930 3	1.198	1.589	2.306	4.276
350	0.037 3	0.256 2	0.453 4	0.697 5	0.917 2	1.175	1.539	2.176	3.718
400	0.053 5	0.263 6	0.452 8	0.690 7	0.904 6	1.152	1.494	2.065	3.324
450	0.069 0	0.270 9	0.452 3	0.684 1	0.892 6	1.131	1.453	1.968	3.027
500	0.083 6	0.278 2	0.452 0	0.677 7	0.881 1	1.111	1.415	1.884	2.791
600	0.110 0	0.292 6	0.451 7	0.665 7	0.859 6	1.075	1.348	1.742	2.439
700	0.131 7	0.306 5	0.451 8	0.654 5	0.839 7	1.042	1.290	1.626	2.186
800	0.147 5	0.319 6	0.452 3	0.644 1	0.821 3	1.012	1.238	1.530	1.994
900	0.156 3	0.331 7	0.453 0	0.634 3	0.804 2	0.984 4	1.193	1.448	1.843
1000	0.157 6	0.342 6	0.454 0	0.625 2	0.788 2	0.959 4	1.152	1.377	1.720

$^{a}\beta$ in 10^{-3}/K.

Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Wrapping up spline interpolation (1/2)

- **Advantages**:
  - Reduces Oscillations: Less prone to the oscillatory behavior seen in high-degree polynomial interpolation.
  - Flexibility: Adapts somehow better to datasets with unevenly spaced data points (compared to simple interpolation).
  - Maintains smooth transitions between intervals.
- **Disadvantages**:
  - May require some computational effort (particularly for cubic splines).
  - Cannot reliably predict outside the dataset's bounds; results are undefined or nonsensical.
  - Becomes computationally expensive and less stable as the dataset grows
  - Typically focuses on one independent variable and doesn't handle multivariate relationships well.
  - No physical model equations (→ difficult to interpret)

Wrapping up spline interpolation (2/2)

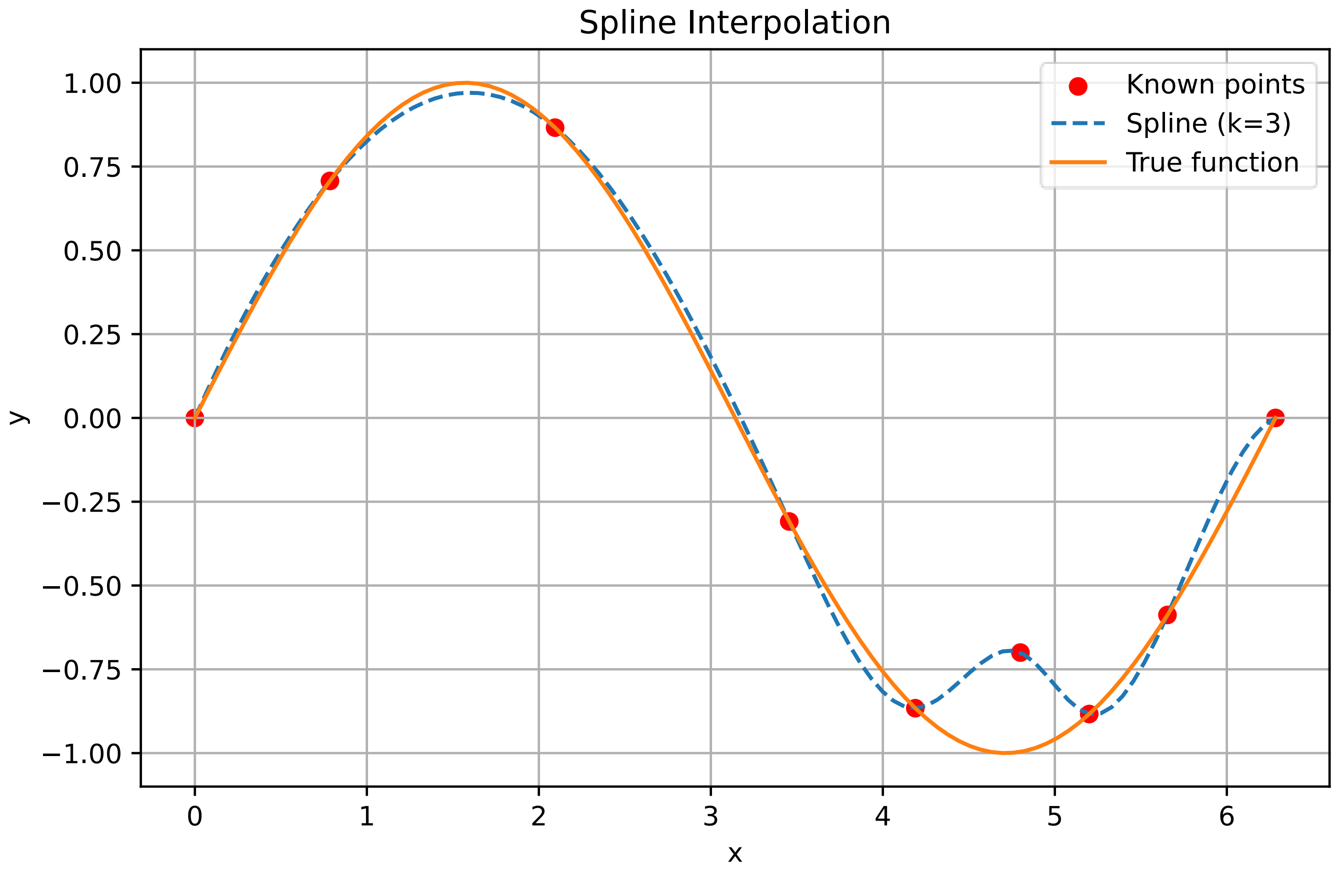
- **Applications**:
  - Frequently used in data tables for thermodynamic properties.
  - Used in fluid dynamics for smooth curve generation.

Agenda

- Data loading and saving in Python
- Introduction
- Interpolation
- **Regression**
  - Linear regression
  - Polynomial regression
  - Nonlinear regression

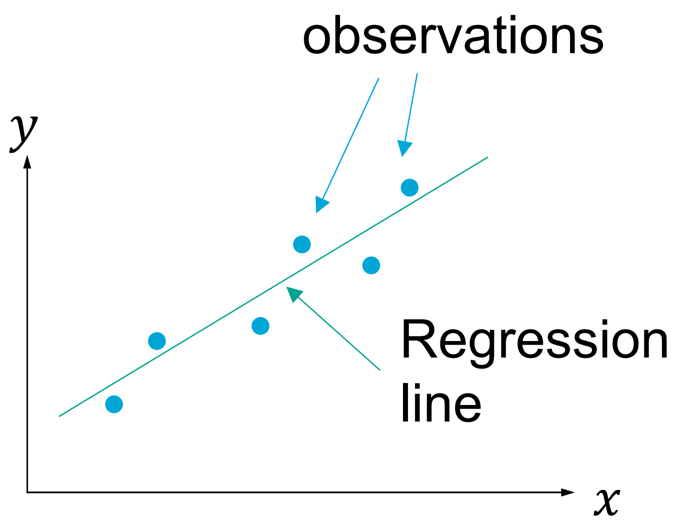
Introduction to Regression

- Guess what: Data has noise.



Introduction to Regression

- Regression: Find a model that best fits the data <u>without the constraint of passing though all datapoints necessarily</u>
- Accounts for data variability and noise by finding a "best-fit" curve
- Interpretation: Can use physical model equations that are easier to interpret
- Types of Regression:
  - Linear regression: Fit straight line
  - Polynomial regression: Fit higher degree polynomials
  - Nonlinear regression: Fitting parameters of general nonlinear models (or functions)



Agenda

- Data loading and saving in Python
- Introduction
- Interpolation
- **Regression**
  - **Linear regression**
  - Polynomial regression
  - Nonlinear regression

Linear regression: general idea

We consider a set of n paired observations $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
We want to find a straight line, i.e., a linear function

$$\hat{y} = w_0 + w_1 x$$

that best represents the observed data.

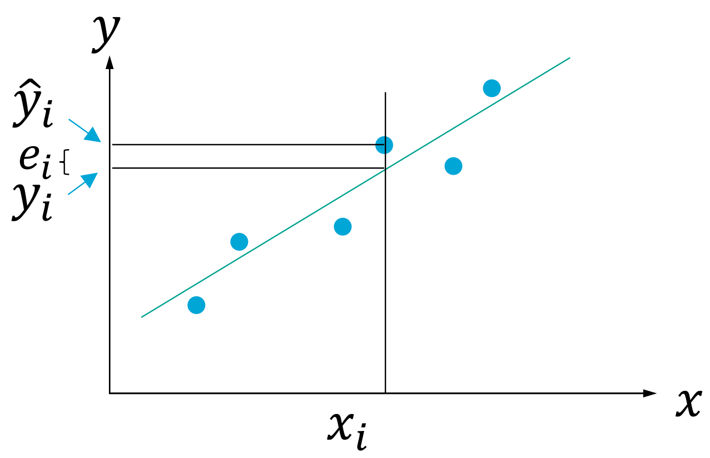
The **residual** at a point i is defined as $e_i = y_i - \hat{y}(x_i)$.

How do we define the best fit?

One option: **Residual Sum of Squares (RSS)**:

$$RSS = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - w_0 - w_1 x_i)^2$$

→ Our task: Find (w_0, w_1) such that RSS is minimal.



One dimensional input → multidimensional input

- In the following, we no longer consider scalar inputs x_i, but multidimensional inputs $\boldsymbol{X}_i \in \mathbb{R}^D$ such that $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \dots, x_{iD})$ and $x_{ij} \in \mathbb{R}$. For each input $\boldsymbol{X}_i \in \mathbb{R}^D$, we observe a scalar output $y_i \in \mathbb{R}$ such that $\boldsymbol{y} \in \mathbb{R}^N$.
- For N observations, we handle data pairs $(\boldsymbol{X}, \boldsymbol{y})$ with $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{y} \in \mathbb{R}^N$. The data pairs are each denoted by $(\boldsymbol{X}_0, y_0)$, $(\boldsymbol{X}_1, y_1)$, …, $(\boldsymbol{X}_N, y_N)$.
- We refer to the individual $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \dots, x_{iD})$ as input features or feature vectors. An input $\boldsymbol{X}$ $\in \mathbb{R}^{N \times D}$ thus consists of N stacked feature vectors.

Linear regression: Theory

Given a vector of inputs $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \cdots, x_{ij}, \cdots, x_{iD})$ associated with a scalar output $y_i \in \mathbb{R}$, we aim to approximate the output $\hat{y}_i$ via the linear model $f(\boldsymbol{X}_i, \boldsymbol{w})$ such that

$$\hat{y}_i = f(\boldsymbol{X}_i, \boldsymbol{w}) = w_0 + \sum_{j=1}^{D} x_{ij} w_j$$

where

- $\hat{y}_i$ is the prediction of the model
- $\boldsymbol{w}^T = (w_0, w_1, w_2, \dots, w_D)$, $\boldsymbol{w} \in \mathbb{R}^{D+1}$ are model parameters (or coefficients) for $j = 0, \dots, D$
- w_0 is the intercept (also known as bias)

Reformulate to matrix/vector form

Given a vector of inputs $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \cdots, x_{ij}, \cdots, x_{iD})$ associated with a scalar output $y_i \in \mathbb{R}$, we aim to approximate the output $\hat{y}_i$ via the linear model $f(\boldsymbol{X}_i, \boldsymbol{w})$ such that

$$\hat{y}_i = f(\boldsymbol{X}_i, \boldsymbol{w}) = w_0 + \sum_{j=1}^{D} x_{ij} w_j$$

- We add a constant value $\widetilde{\boldsymbol{X}}_i = (1, x_{i1}, x_{i2}, \cdots, x_{ij}, \cdots, x_{iD})$ such that

$$\hat{y}_i = f\left(\widetilde{\boldsymbol{X}}_i, \boldsymbol{w}\right) = \sum_{j=0}^{D} x_{ij} w_j$$

- Now, we can write the linear model in matrix form:

$$\hat{\boldsymbol{y}} = f\left(\widetilde{\boldsymbol{X}}, \boldsymbol{w}\right) = \widetilde{\boldsymbol{X}}\boldsymbol{w}$$

- In the following, we write $\widetilde{\boldsymbol{X}} \coloneqq \boldsymbol{X}$ for simplicity.

Minimizing the RSS to find w

- We aim to find the parameters $\hat{\boldsymbol{w}}$ that <u>minimize</u> the residual sum of squares (RSS) :

$$\min_{\boldsymbol{w}} RSS(\boldsymbol{w})$$

$$\text{with} \quad RSS(\boldsymbol{w}) = \sum_{n=1}^{N} (y_i - \hat{y}_i)^2 = \sum_{n=1}^{N} \left(y_i - \sum_{j=0}^{D} x_{ij} w_j \right)^2$$

$$\textbf{Matrix form}: \ RSS(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})$$

- Optimality conditions
  - Necessary condition for optimality: The RSS’s derivative is zero at the stationary point.
  - $RSS(\cdot)$ is a quadratic function w.r.t. the parameters $\boldsymbol{w}$. → RSS is <u>convex</u>.
  - The stationary point is a global minimum.

See Q2
Lecture 4

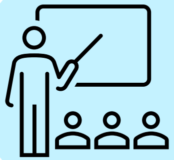
Minimizing the residual sum-of-squares (1/2)

$$RSS(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{Xw})^T(\boldsymbol{y} - \boldsymbol{Xw}) = \boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{Xw} - (\boldsymbol{Xw})^T\boldsymbol{y} + (\boldsymbol{Xw})^T\boldsymbol{Xw}$$
$$= \boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{Xw} - \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{Xw}$$

Differentiating $RSS(\boldsymbol{w})$ w.r.t. $\boldsymbol{w}$ and setting the resulting equation equal to zero, we can derive the **Normal Equation** (see full derivation [1], there are different ways to arrive the same answer):

$$\nabla_{\boldsymbol{w}} RSS(\boldsymbol{w}) = \nabla_{\boldsymbol{w}}\left(\boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{Xw} - \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{Xw}\right)$$
$$\boldsymbol{0} = \boldsymbol{0} - \boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{2w}^T\boldsymbol{wX}$$
$$\boldsymbol{0} = -\boldsymbol{2X}^T\boldsymbol{y} + \boldsymbol{2X}^T\boldsymbol{Xw}$$
$$\left(\boldsymbol{X}^T\boldsymbol{X}\right)\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y}$$
$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

- The model parameters $\boldsymbol{w}$ can be directly calculated unless $\boldsymbol{X}^T\boldsymbol{X}$ is singular (or non-invertible)
- When is $\boldsymbol{X}^T\boldsymbol{X}$ singular?
  - If the determinant is zero ($\boldsymbol{det}\left(\boldsymbol{X}^T\boldsymbol{X}\right) = \boldsymbol{0}$), it implies singularity.
  - Often because there is linear dependence among the features in $\boldsymbol{X}$


See Q1 Lecture 3

[1] https://link.springer.com/referenceworkentry/10.1007/978-0-387-32833-1_286#:~:text=Normal%20equations%20are%20equations%20obtained,of%20a%20multiple%20linear%20regression

Minimizing the residual sum-of-squares (2/2)

- **Option A**: Solve $\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$ analytically (Direct method)
  - Inversion of $\boldsymbol{X}^T\boldsymbol{X}$ necessary, e.g., using numpy.linalg.inv($\boldsymbol{X}^T\boldsymbol{X}$), or solve linear system of linear equations numpy.linalg.solve($\boldsymbol{X}^T\boldsymbol{X}$, $\boldsymbol{X}^T\boldsymbol{y}$)

See Q1 Lecture 3

- **Option B**: Iterative approximation of the solution of the linear system of equations (Indirect method) using, e.g., Jacobi method or Gauss Seidel

$$\underbrace{(\boldsymbol{X}^T\boldsymbol{X})}_{\boldsymbol{A}}\underbrace{\boldsymbol{w}}_{\boldsymbol{x}} = \underbrace{\boldsymbol{X}^T\boldsymbol{y}}_{\boldsymbol{b}}.$$

See Q1 Lecture 3

- **Option C**: Solve an optimization problem with the objective function $L(\boldsymbol{w})$ using e.g., scipy.optimize.minimize(objective, x0=0)

$$L(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})$$

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = -2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})$$

See Q2 Lecture 4

Live coding: Linear regression

- Open Colab: Linear regression
- using option A

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Agenda

- Data loading and saving in Python
- Introduction
- Interpolation
- **Regression**
    - Linear regression
    - **Polynomial regression**
    - Nonlinear regression

Polynomial regression: Theory

- Polynomial regression: Transformation of input variables (features) such that we use a polynomial to fit our data. We only consider one-dimensional inputs now, i.e., $\boldsymbol{X} \in \mathbb{R}^N$.

$$\hat{y}_i = w_0 + w_1 x_i + w_2 x_i^2 + \cdots + w_M x_i^M = \sum_{j=0}^{M} w_j x_i^j \, .$$

  - M is the order of the polynomial.
  - x_i^j denotes x_i raised to the power of j.
  - The polynomial coefficients w_0, ..., w_M are represented by the parameter vector $\boldsymbol{w} \in \mathbb{R}^{M+1}$.
- Basic idea: precompute nonlinear "features" and then solve a linear regression problem
- The model is <u>nonlinear w.r.t the inputs $\boldsymbol{X}$</u> but still <u>linear w.r.t the parameters $\boldsymbol{w}$</u>!

Example: Creating polynomial features

- Suppose we have an input $\boldsymbol{X} = (5,3,4)$, and we want to create polynomial features up to 2nd degree:

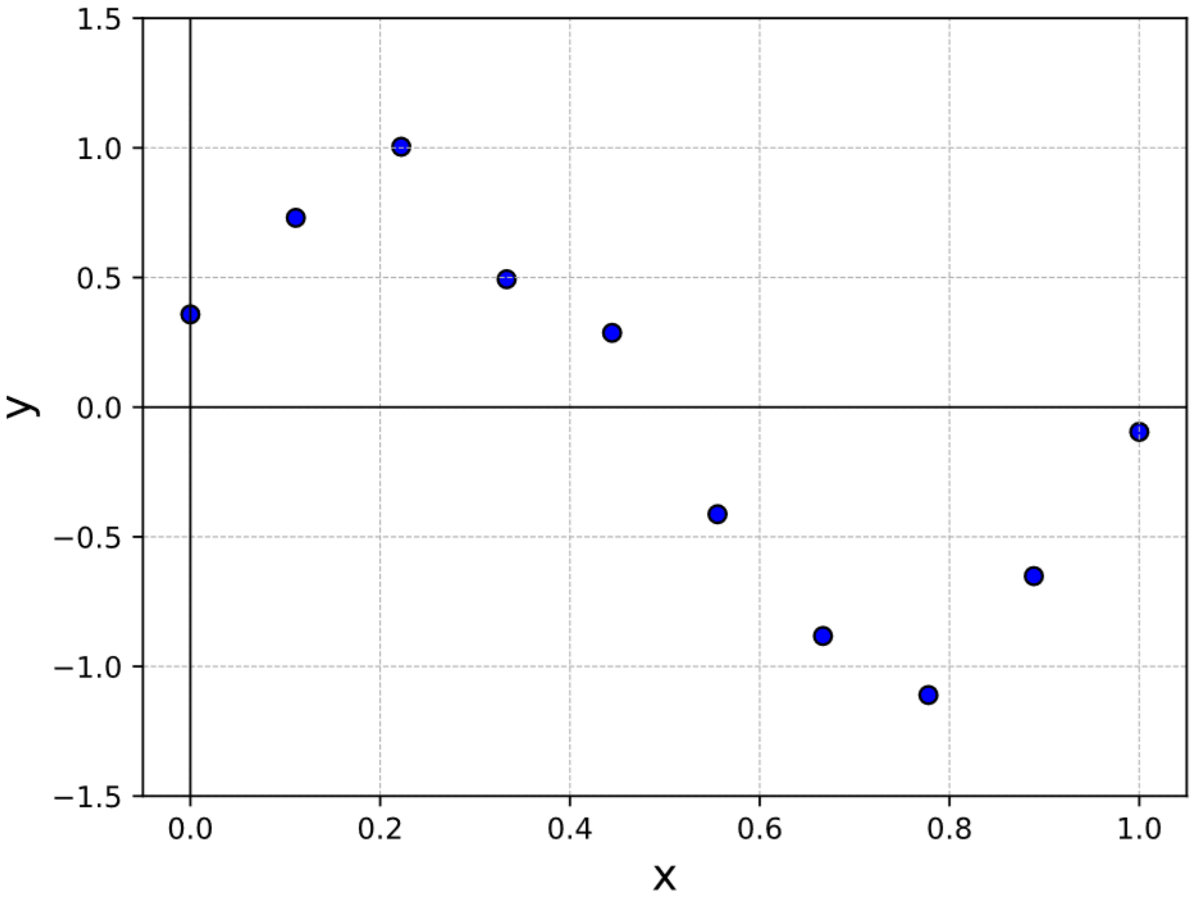
$$\hat{y}(\boldsymbol{X}, \boldsymbol{w}) = w_0 + w_1 x + w_2 x^2$$

- To calculate polynomial features of degree 2, you would transform each x value into x and x^2 and the polynomial features are:

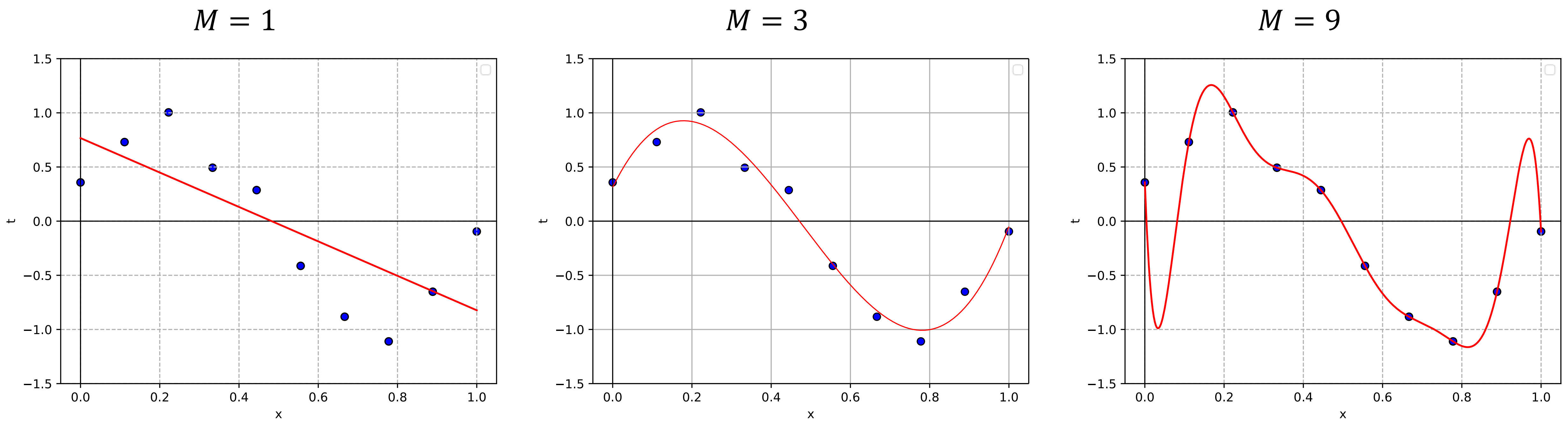
$$\hat{y}(\boldsymbol{X}, \boldsymbol{w}) = \begin{bmatrix} 1 & 5 & 5^2 \\ 1 & 3 & 3^2 \\ 1 & 4 & 4^2 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 25 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

- The first column of ones is for the intercept (degree 0), the second column is the original feature x (degree 1), and the third column is x^2 (degree 2).

What is the “correct” polynomial degree?



How to select the order of polynomial curve fitting?



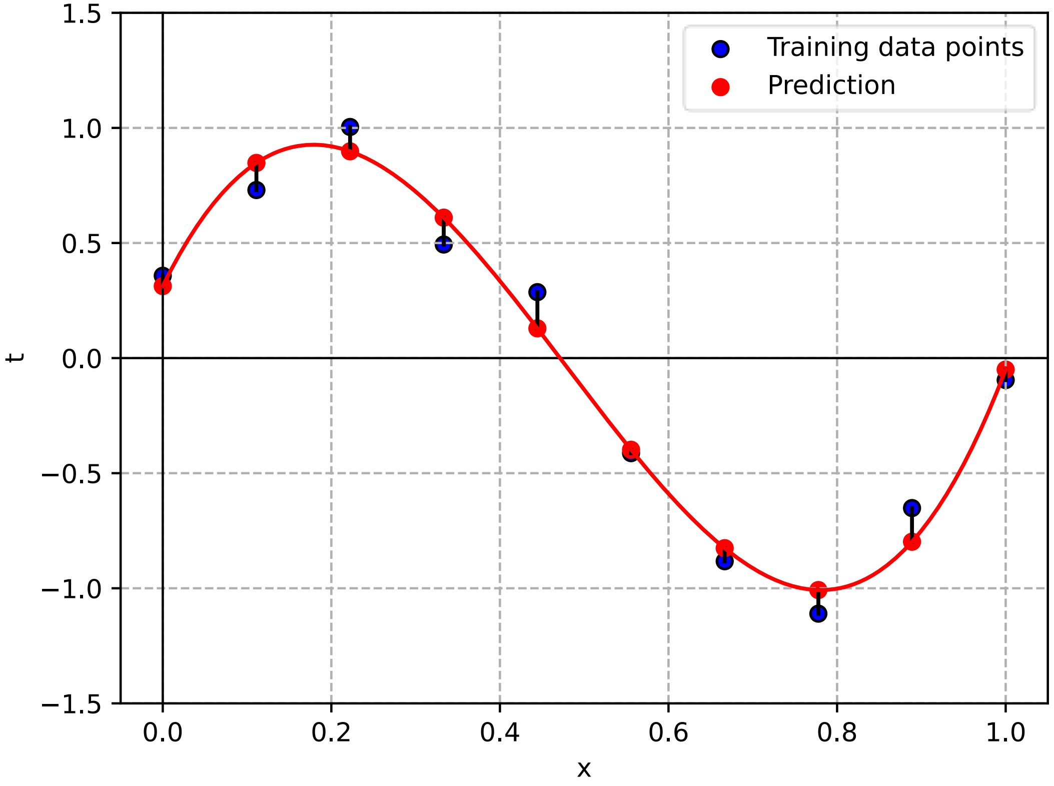
Polynomial curve fitting: Evaluation

- The performance is measured by mean square error with true values t_n and predictions $\hat{y}(x_n, \boldsymbol{w})$:

$$E(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}\{\hat{y}(x_n, \boldsymbol{w}) - t_n\}^2$$

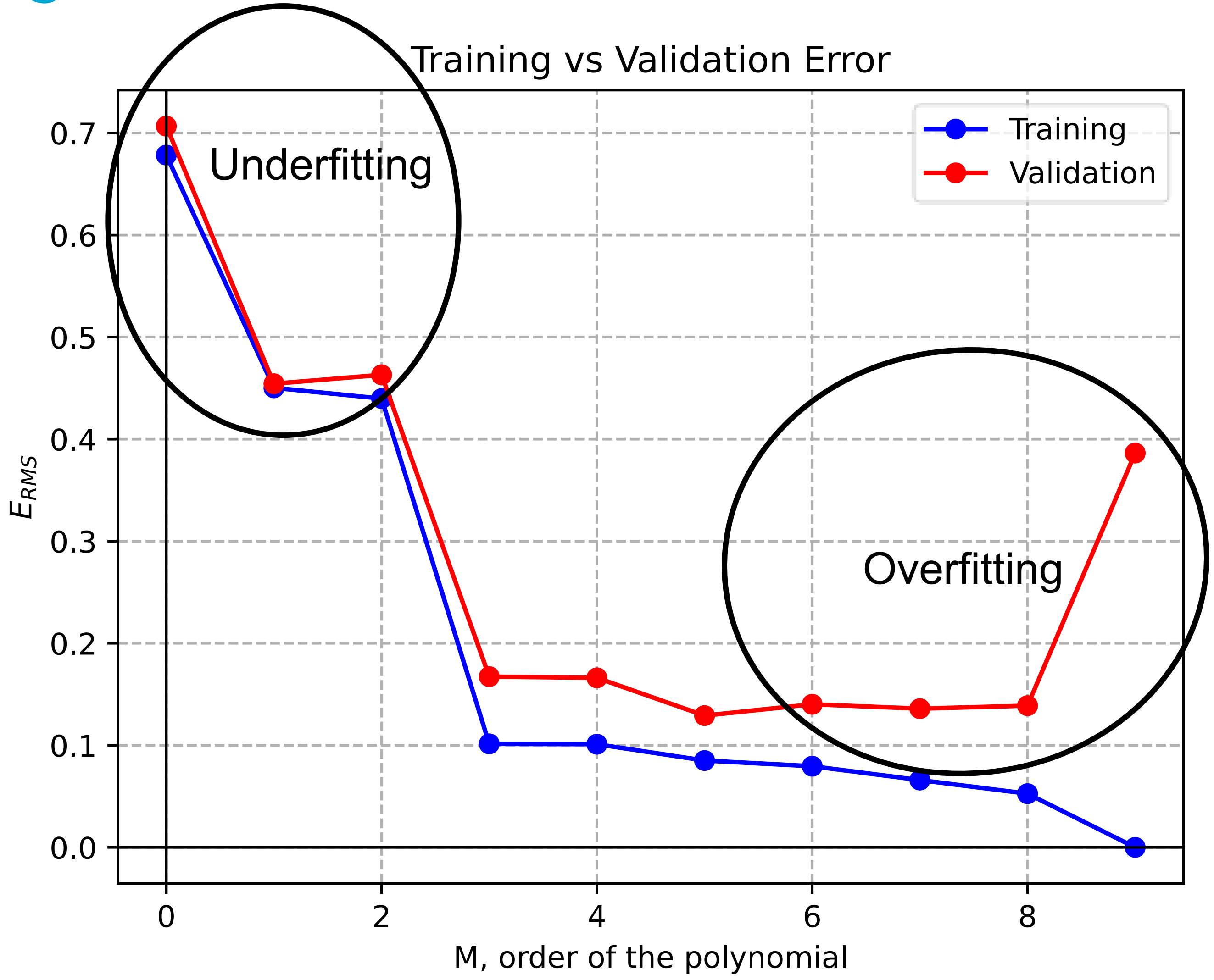
- Root mean square (RMS) error:

$$E_{RMS} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\{\hat{y}(x_n, \boldsymbol{w}) - t_n\}^2}$$



Performance and overfitting

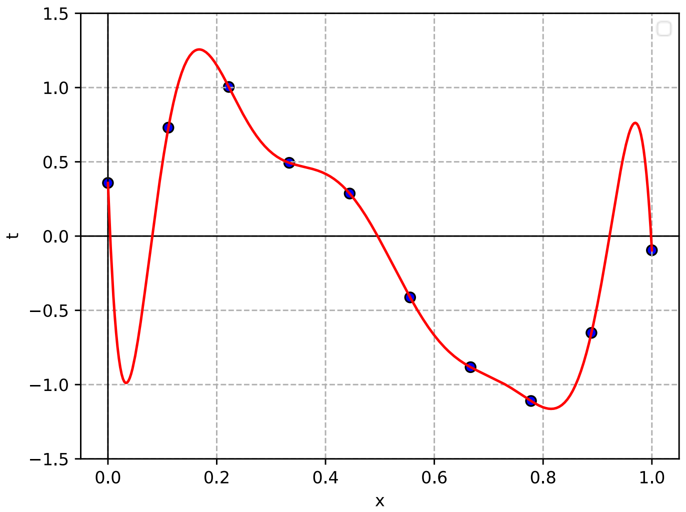
- Small M values lead to high test errors, indicating underfitting.
- $M = 5$ results in the lowest test error, making it an optimal choice.
- Model overfits for $M \geq 6$, as the test errors increase.



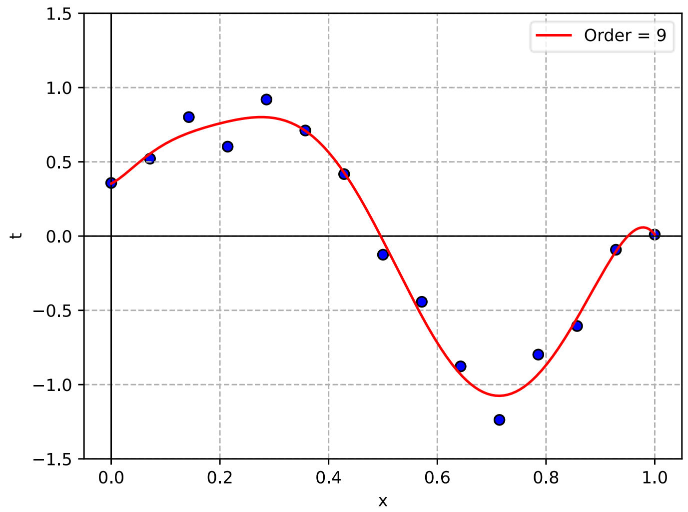
The effect of dataset size

- Polynomial of order 9 and N is the number of data points

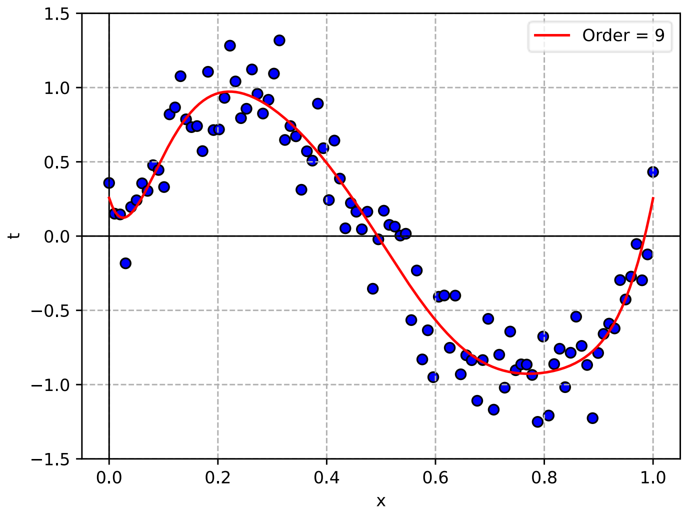
N =10



N =15



N =100



- For a given model complexity, the over-fitting problem becomes less severe as the size of the data set increases.

Agenda

- Data loading and saving in Python
- Introduction
- Interpolation
- **Regression**
  - Linear regression
  - Polynomial regression
  - **Nonlinear regression**

Reasons to use nonlinear regression

1. Real-world relationships can rarely accurately be expressed as polynomials.
2. Nonlinear equations, in principle, allow to fit complex relationships with fewer parameters than polynomials.

   → Increasing the degree of the fitting polynomial of a D-dimensional problem increases the number of parameters in the order of $\mathcal{O}(D^3)$. This increases computational complexity and the risk of overfitting.
3. Extrapolation: Fitting parameters of known, universally valid physical relations often enables a more reasonable extrapolation outside the given data range than polynomial regression.

Nonlinear regression

Consider a set of observations $(\boldsymbol{X_i}, y_i)$ for $i = 1, \dots, N$. We aim to find a nonlinear function $f(\boldsymbol{X}, \boldsymbol{\beta})$ that is parametrized by a set of parameters $\boldsymbol{\beta} \in \mathbb{R}^M$ and that takes $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ as input such that

$$y_i = f(\boldsymbol{X_i}, \boldsymbol{\beta}) + \epsilon = \hat{y}_i + \epsilon.$$

ϵ is the error between the true value y_i and the prediction $\hat{y}_i$.

- Example
  - Langmuir-isotherm: $\theta_A = \frac{K_{eq}^A p_A}{1+K_{eq}^A p_A}$, θ_A: fraction of surface sites covered with component A, p_A: partial pressure of component A, K_{eq}^A: equilibrium constant

→ $\boldsymbol{\beta} = [K_{eq}^A], \boldsymbol{X_i} = [p_A], y_i = [\theta_A]$

Nonlinear curve fit with SciPy (1/3)

Let's generate some random example data first

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def func(x, a, b, c):
    return a * np.exp(-b * x) + c

xdata = np.linspace(0, 4, 50)
y = func(xdata, 2.5, 1.3, 0.5)
rng = np.random.default_rng()
y_noise = 0.2 * rng.normal(size=xdata.size)
ydata = y + y_noise
plt.scatter(xdata, ydata, label='data')
```

This example is obtained from scipy.optimize.curve_fit examples section.

Nonlinear curve fit with SciPy (2/3)

scipy.optimize.curve_fit

- Parameters:
  - *f* (Callable): model function *f(x,*params).* **Takes independent variable as first input argument and parameters to fit as separate remaining arguments.**
  - *xdata* (array): measured datapoints of independent variable
  - *ydata* (array): dependent data corresponding to measurements *xdata*
- Returns:
  - *popt* (array): Optimal values for parameters**params*
  - *pcov* (2-D array): estimated approximate covariance of popt

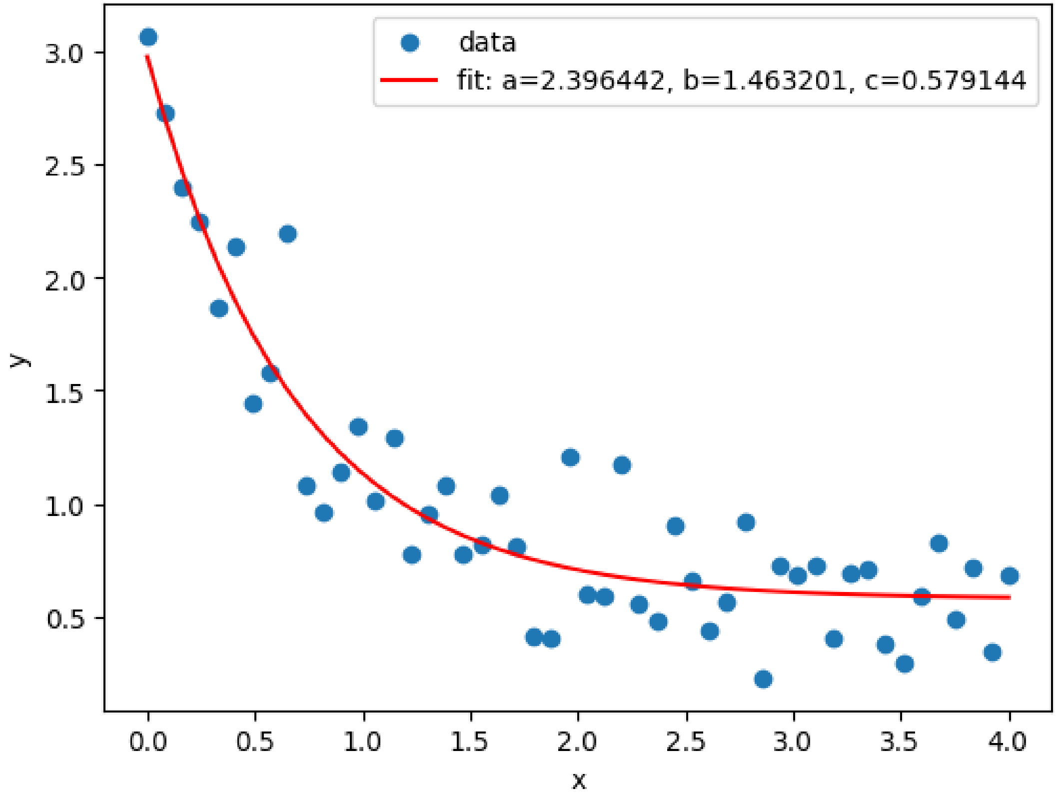
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit


popt, pcov = curve_fit(func, xdata, ydata)


plt.plot(xdata, func(xdata, *popt), 'r-',
         label='fit: a=%f, b=%f, c=%f' %
tuple(popt))
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

This example is obtained from scipy.optimize.curve_fit examples section.

Nonlinear curve fit with SciPy

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit


popt, pcov = curve_fit(func, xdata, ydata)

plt.plot(xdata, func(xdata, *popt), 'r-',
         label='fit: a=%f, b=%f, c=%f' %
tuple(popt))
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

`y = func(xdata, 2.5, 1.3, 0.5)`



This example is obtained from scipy.optimize.curve_fit examples section.

Live coding: Nonlinear regression

- Open Colab: [Nonlinear regression]
- Using scipy.optimize.curve_fit

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Learning objectives

After successfully completing this lecture, you are able to…

- explain interpolation and regression.
- discuss the advantages and limitations of different interpolation and regression methods.
- use Python libraries' built-in functions for constructing Lagrange polynomials and Splines.
- derive the nominal equation for linear regression problems.
- implement the three ways to solve linear (or polynomial) regression problems from scratch.
- apply Python libraries' built-in functions to nonlinear regression problems.

Thank you very much for your attention!

Live coding: Interpolation

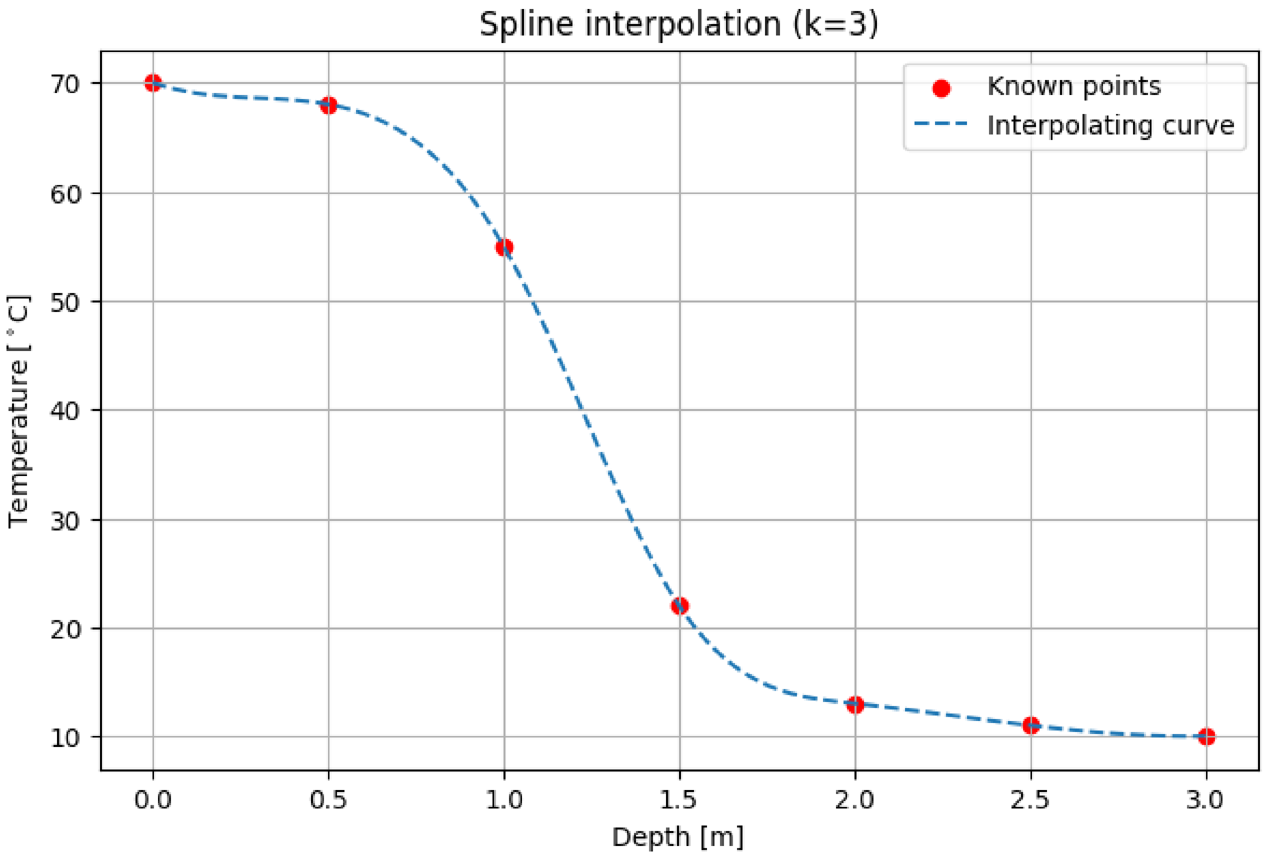
- Open Colab: Interpolation

  Package scipy.interpolate:

  scipy.interpolate.interp1d

  Input arguments:

  - **x**: x-values of known points
  - **y**: y-values of known points
  - **Kind** (str): ‘linear’, ‘nearest’, ‘nearest-up’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘previous’, or ‘next’



Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main