

# Numerical interpolation

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group  
Eindhoven University of Technology

Numerical Methods (6BER03), 2024-2025

# Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines
- Tutorials

# Interpolation problem

## Definition

Given a set of points  $x_k$ ,  $k = 0, \dots, n$ ,  $x_i \neq x_j$  with associated function values  $f_k$ ,  $k = 0, \dots, n$ , or simply:  $\{x_k, f_k\}_{k=0}^n$ . The interpolation problem is defined as: find a polynomial  $p_n$  such that this interpolates the values of  $f_k$  on the points  $x_k$ :

$$p_n(x_k) = f_k, \quad k = 0, \dots, n$$

## Theorem

*The interpolation problem for  $\{x_k, f_k\}_{k=0}^n$  has a unique solution when  $x_i \neq x_j$  for  $i \neq j$ . Note that we cannot allow multiple function values  $f_k$  for the same value of  $x_k$ .*

# What is interpolation?

Interpolation means constructing additional data points within the range of, and using, a discrete set of known data points.

It is typically performed on a uniformly spread data set, but this is not strictly necessary for all methods

# Is interpolation the same as curve fitting?

NO

- Curve-fitting requires additionally some way of computing the error between function (curve) and data
- Curve-fitting does not strictly enforce the function to match the data exactly
- Curve-fitting may be done on multiple datapoints at one position
- Curve-fitting is much more expensive to do, requires optimisation

# Why do chemical engineers need interpolation?

- Comparison of two data sets which are given at different positions
  - An experimental data set may have been recorded at a constant rate, but the numerical solution is computed at irregular intervals
- Reconstruction of field values distant of computing nodes
  - A CFD simulation on a regular grid containing structures that are not grid-conformant requires interpolation to the structures
- Calculation of a physical property at a condition between those of a lookup table
  - The viscosity of a substance may have been measured at 20°C and 30°C, but not at the desired 28.5°C

# General

Several important numerical interpolation methods are discussed today:

- Piecewise constant interpolation
- Linear interpolation
  - Bilinear interpolation
- Polynomial interpolation (Newton's method)
- Spline interpolation

# Today's outline

- Introduction
- Piecwise constant
- Linear
- Polynomial
- Splines
- Tutorials



# Today's data set

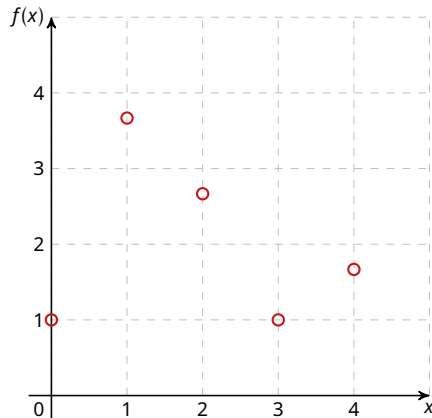
Generate the following data set:

```
1 import numpy as np
2 xdata = np.arange(0,6)
3 fun = lambda x: x**3/2 - (10*x**2)/3 + 11*x
4      /2 + 1
5 ydata = fun(xdata)
```

This yields some sample points on which we base our examples:

$x_k$	$f_k$
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$
3	1.00
4	$\frac{5}{3} = 1.67$
5	$\frac{23}{3} = 7.67$

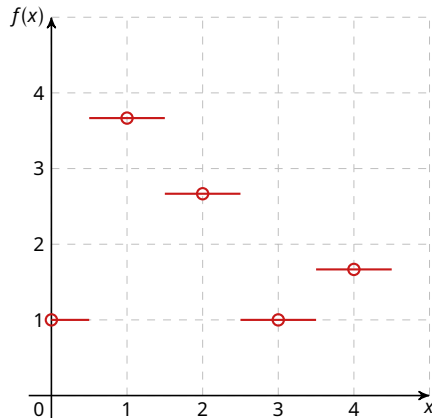
Data set  $f_n(x_n)$  represented by ○ at discrete intervals  $x_n \in \{0, 5\}$



# Piecewise constant interpolation

Data set  $f_n(x_n)$  represented by ○ at discrete intervals  $x_n \in \{0, 5\}$

- Nearest-neighbor interpolation in the continuous range  $x \in [0, 5]$
- How to treat the point halfway (e.g. at  $x = 2.5$ )?
  - $x \in [0, 0.5] \rightarrow f(x) = f(0)$
  - $x \in [0.5, 1.5] \rightarrow f(x) = f(1)$
  - $x \in [1.5, 2.5] \rightarrow f(x) = f(2)$
  - $x \in [2.5, 3.5] \rightarrow f(x) = f(3)$
  - $x \in [3.5, 4.5] \rightarrow f(x) = f(4)$
- Not often used for simple problems, but e.g. for 2D (Voronoi)



# Today's outline

- Introduction
- Piecewise constant
- **Linear**
- Polynomial
- Splines
- Tutorials

# Linear interpolation

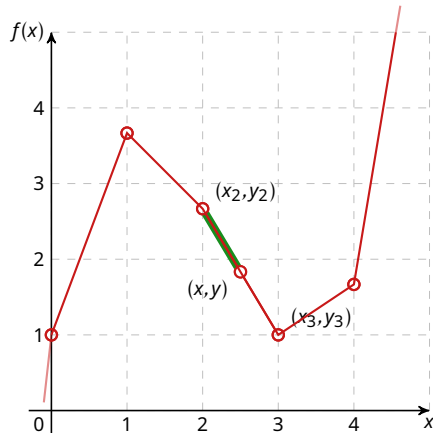
- Linear interpolation to  $(x, y)$  between 2 data points  $(x_2, y_2)$  and  $(x_3, y_3)$ :

$$\frac{y - y_2}{x - x_2} = \frac{y_3 - y_2}{x_3 - x_2}$$

- Reordered, and more formally:

$$y = y_n + (y_{n+1} - y_n) \frac{x - x_n}{x_{n+1} - x_n}$$

Data set  $f_n(x_n)$  represented by ○ at discrete intervals  $x_n \in \{0, 5\}$



# Linear interpolation

- While linear interpolation is fast, and relatively easy to program, it is not very accurate
- At the nodes, the derivatives are discontinuous i.e. not differentiable
- Error is proportional to the square of the distance between nodes

# Interpolation in Python

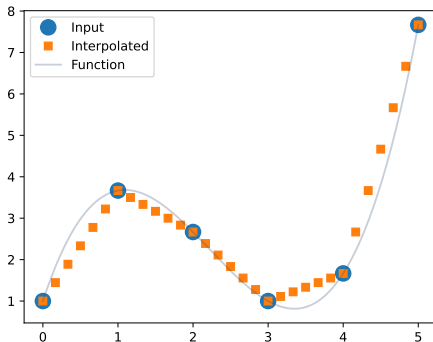
Interpolation can be done using the SciPy interpolation submodule, e.g.:

```
1 from scipy.interpolate import interp1d
2 f = interp1d(xdata, ydata, kind='linear')
```

This creates a function object `f` based on the given data.

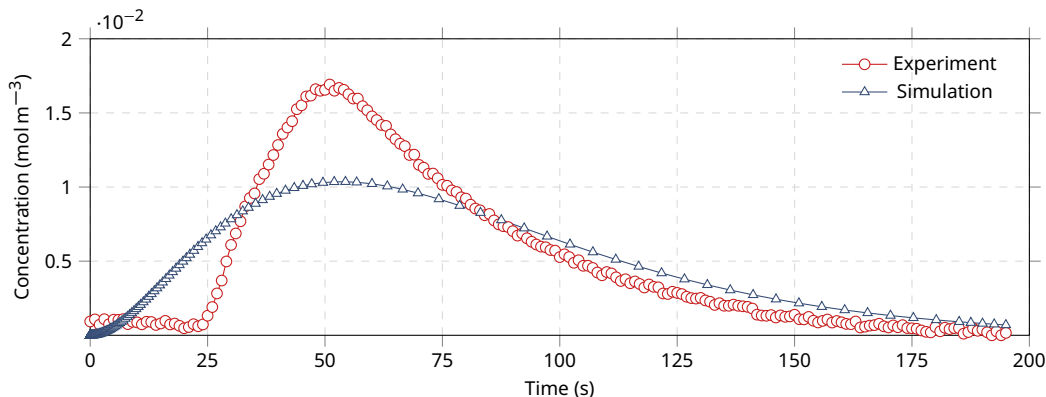
```
1 from scipy.interpolate import interp1d
2 import numpy as np
3
4 fun = lambda x: x**3/2 - (10*x**2)/3 + 11*x/2 + 1
5 xdata = np.arange(0,6)
6 ydata = fun(xdata)
7
8 f = interp1d(xdata,ydata)
9 xint = np.linspace(0,5,31)
10 yint = f(xint)
```

```
1 import matplotlib.pyplot as plt
2 plt.plot(xdata,ydata, 'o', markersize=12, label='
   Input')
3 plt.plot(xint,yint, 's', label='Interpolated')
```



## Example: Linear interpolation in Python

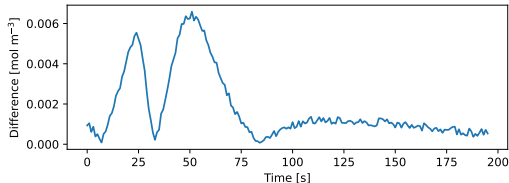
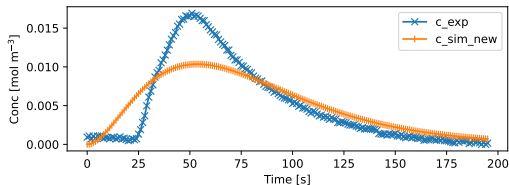
Consider the data sets in `exp_data.txt` and `sim_data.txt`, containing a normalized concentration and time vector for an experiment and a simulation. The simulation was performed with adaptive node distance to save computation time, thus the concentration is not known at the same times. We are not able to compare yet.



# Example: Linear interpolation in Python

Consider the data sets in `exp_data.txt` and `sim_data.txt`, containing a normalized concentration and time vector for an experiment and a simulation. The simulation was performed with adaptive node distance to save computation time, thus the concentration is not known at the same times. We are not able to compare yet.

```
1 import numpy as np
2 from scipy.interpolate import interp1d
3 import matplotlib.pyplot as plt
4
5 t_sim, c_sim = np.loadtxt("scripts/interpolation/sim_data.txt").T
6 t_exp, c_exp = np.loadtxt("scripts/interpolation/exp_data.txt").T
7
8 # Linear interpolation
9 f = interp1d(t_sim, c_sim)
10 diff = np.abs(c_exp - f(t_exp))
11
12 # Plot the solution
13 plt.subplot(2, 1, 1)
14 plt.plot(t_exp, c_exp, '-x', label='c_exp')
15 plt.plot(t_exp, f(t_exp), '-|', label='c_sim_new')
16 plt.xlabel('Time [s]'); plt.ylabel('Conc [mol m$^{-3}$]')
17 plt.legend()
18
19 plt.subplot(2, 1, 2)
20 plt.plot(t_exp, diff)
21 plt.xlabel('Time [s]'); plt.ylabel('Difference [mol m$^{-3}$]')
22 plt.tight_layout()
23 # plt.show()
24 plt.savefig('figures/sim_exp_data_interp.pdf')
```





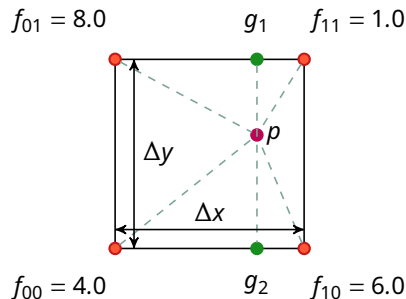
# Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain  $p(x,y)$  using 4 field values  $f_{00}, f_{10}, f_{01}$  and  $f_{11}$ .

$$\begin{aligned} g_1 &= f_{01} \frac{x_1 - x}{x_1 - x_0} + f_{11} \frac{x - x_0}{x_1 - x_0} \\ &= f_{01} \frac{x_1 - x}{\Delta x} + f_{11} \frac{x - x_0}{\Delta x} \end{aligned}$$

$$g_2 = f_{00} \frac{x_1 - x}{\Delta x} + f_{10} \frac{x - x_0}{\Delta x}$$

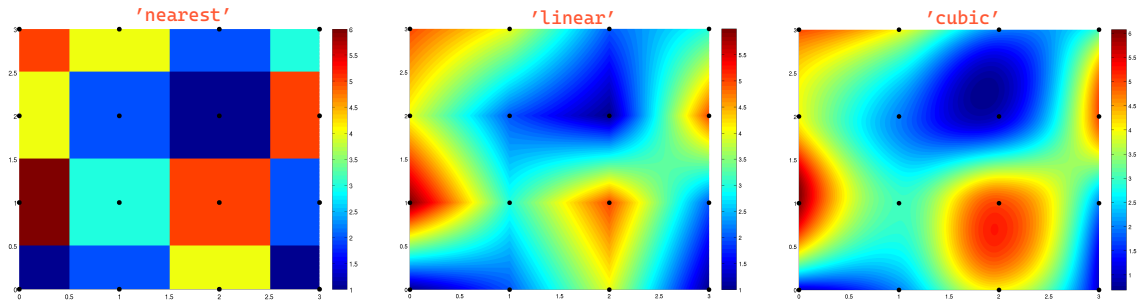
$$p = g_2 \frac{y_1 - y}{\Delta y} + g_1 \frac{y - y_0}{\Delta y}$$



- The order of interpolation (x or y direction first) does not matter; the results are equal

# Higher-dimensional field interpolation in Python

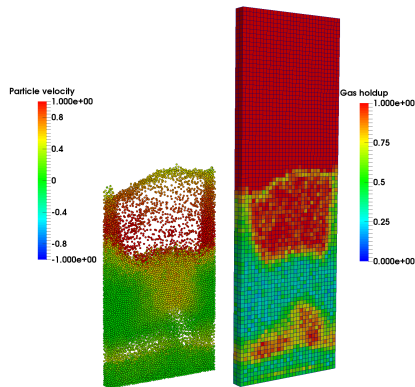
2D or higher-dimensional fields of data can be interpolated in Python using the `scipy.interpolate.interp2d`, `scipy.interpolate.interp3d`, or even `scipy.interpolate.RegularGridInterpolator` functions. The method can be adjusted:



- Also consider tri-linear interpolation (for 3D fields) with `scipy.interpolate.LinearNDInterpolator`, or bicubic interpolation (2D, but third order) with `scipy.interpolate.interp2d`.

# A practical example

Field interpolation is used in e.g. CFD simulations, e.g. a fluidized bed simulation using a *discrete particle model*, where particles are found in between the grid nodes used for velocity computation.



# Today's outline

- Introduction
- Piecewise constant
- Linear
- **Polynomial**
- Splines
- Tutorials

# Polynomial interpolation

The examples that we have seen, are simplified forms of *Newton polynomials*. We can interpolate our data with a polynomial of degree  $n$ :

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$$

# Polynomial interpolation via Vandermonde matrix

Consider the data points  $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ , the Vandermonde matrix  $V$ , coefficient vector  $a$  and function value vector  $y$ :

$$V_{m,n} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix} \quad a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

The coefficients of a polynomial through the data are obtained by solving the linear system  $Va = y$ .

```
1 import numpy as np
2 x = np.array([0, 1, 2])
3 y = np.array([1.0000, 3.6667, 2.6667])
4 V = np.vander(x, increasing=True)
5 print(V)
```

```
[ 1.  4.50005 -1.83335]
```

So we found the equation:

$$p_2(x) = -1.8333x^2 + 4.5x + 1$$

These Vandermonde-systems are often *ill-conditioned*, so we need another, more stable, method!

# Construction of Newton polynomials

Formally, the polynomials  $p_n(x)$  are described using prefactors  $f[x_0, \dots, x_k]$  and polynomial terms  $w_m(x)$ :

$$p_n(x) = \sum_{k=0}^n f[x_0, \dots, x_k] w_k(x)$$

The polynomial terms are computed via:

$$w_0(x) = 1, \quad w_1(x) = (x - x_0), \quad w_2(x) = (x - x_0) \cdot (x - x_1),$$

$$w_m(x) = (x - x_0) \cdot (x - x_1) \cdots (x - x_{m-1}) = w_{m-1} \cdot (x - x_{m-1})$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j), \quad m = 0, \dots, n$$

The prefactors are *forward divided differences*, which can be computed as:

$$f[x_{x-k}, \dots, x_r] \equiv \frac{f[x_{r-k+1}, \dots, x_r] - f[x_{r-k}, \dots, x_{r-1}]}{x_r - x_{r-k}}$$

# Construction of Newton polynomials: example

Sample data

$x_k$	$f_k$
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^n f[x_0, \dots, x_k] w_k(x)$$

$$f[x_{r-k}, \dots, x_r] \equiv \frac{f[x_{r-k+1}, \dots, x_r] - f[x_{r-k}, \dots, x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

$x_k$	$f_k$
$x_0$	$f[x_0] = f_0$
$x_1$	$f[x_1] = f_1 \quad f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_0}$
$x_2$	$f[x_2] = f_2 \quad f[x_1, x_2] = \frac{f_2 - f_1}{x_2 - x_1} \quad f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$

$x_k$	$f_k$
0	1
1	3.67 $\frac{\frac{11}{3} - 1}{1 - 0} = \frac{8}{3}$
2	2.67 $\frac{\frac{8}{3} - \frac{11}{3}}{2 - 1} = \frac{-1}{1} = -1 \quad \frac{(-1) - \frac{8}{3}}{2 - 0} = -\frac{11}{6}$



# Construction of Newton polynomials: example

Sample data

$x_k$	$f_k$
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^n f[x_0, \dots, x_k] w_k(x)$$

$$f[x_{r-k}, \dots, x_r] \equiv \frac{f[x_{r-k+1}, \dots, x_r] - f[x_{r-k}, \dots, x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

$x_k$	$f_k$
0	1
1	3.67 $\frac{\frac{11}{3}-1}{1-0} = \frac{8}{3}$
2	2.67 $\frac{\frac{8}{3}-\frac{11}{3}}{2-1} = \frac{-1}{1} = -1$ $\frac{(-1)-\frac{8}{3}}{2-0} = -\frac{11}{6}$

$$p_2(x) = 1 \cdot w_m(0) + \frac{8}{3} \cdot w_m(1) + \left(-\frac{11}{6}\right) \cdot w_m(2)$$

$$= 1 \cdot 1 + \frac{8}{3} \cdot (x - 0) + \left(-\frac{11}{6}\right) \cdot (x - 0)(x - 1) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

# Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

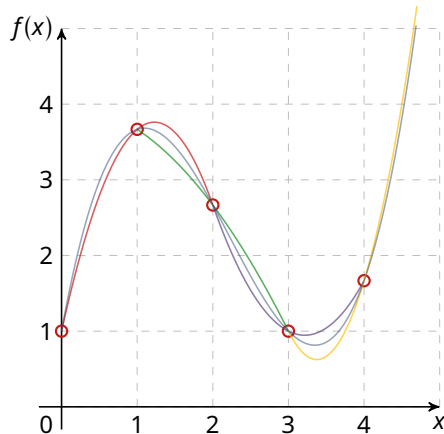
$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

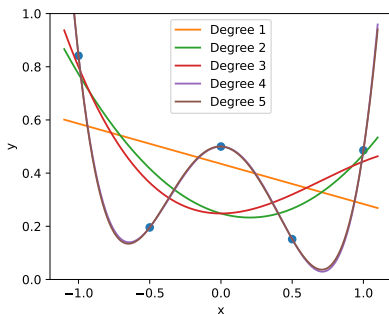
$$f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$$



# Polynomial fitting in Python: example

Develop the polynomials  $p_1(x)$  through  $p_5(x)$  using the following data set:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 xdata = np.arange(-1,1.5,0.5)
4 ydata = [x * np.sin(x)/np.sqrt(x+2) if x != 0 else 0.5 for x in xdata]
5 plt.plot(xdata,ydata,'o')
```



```
1 xc = np.linspace(-1.1,1.1,1001,endpoint=True)
2 for deg in range(1,6):
3     # Fit coefficients
4     p_coeffs = np.polyfit(xdata,ydata,deg)
5     # Compute function values
6     y = np.polyval(p_coeffs,xc)
7     # Plot
8     plt.plot(xc,y,label=f'Degree {deg}')
```

RankWarning: Polyfit may be poorly conditioned

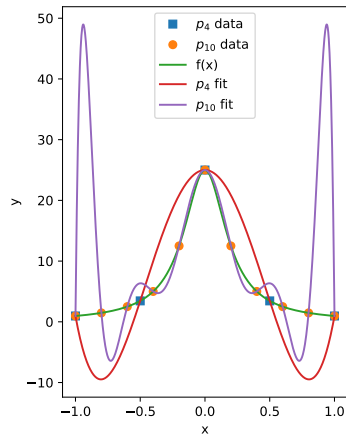
# Exercise

Develop the  $p_4(x)$  and  $p_{10}(x)$  interpolants from the following data sets:

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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 f = lambda x: 1/(x**2 + 1/25)
4 x4,x10,xinf = [np.linspace(-1, 1, n) for n in [5,11,1001]]
5 y4,y10,yinf = f(x4), f(x10), f(xinf)
```

```
6 # Get coefficients for 4th and 10th order polynomial
7 p4 = np.polyfit(x4, y4, 4)
8 p10 = np.polyfit(x10, y10, 10)
9 # Compute function values using fitted coeffs
10 yinf4 = np.polyval(p4, xinf)
11 yinf10 = np.polyval(p10, xinf)
```



# Final thoughts on polynomial interpolation

- An polynomial interpolant of order  $n$  requires  $n + 1$  data points
  - More data points: interpolant does *not always* cross the points
  - Fewer data points: interpolant is not unique
- Higher-degree polynomials at equidistant points may cause strong oscillatory behaviour (Runge's phenomenon)
  - Mitigation of the problem on Chebyshev (i.e. non uniform grid)...
  - ... or by performing piecewise interpolation (next topic)
- Python functions `np.polyfit(x,y,n)` and `np.polyval(p,x_new)` were demonstrated.

# Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- **Splines**
- Tutorials

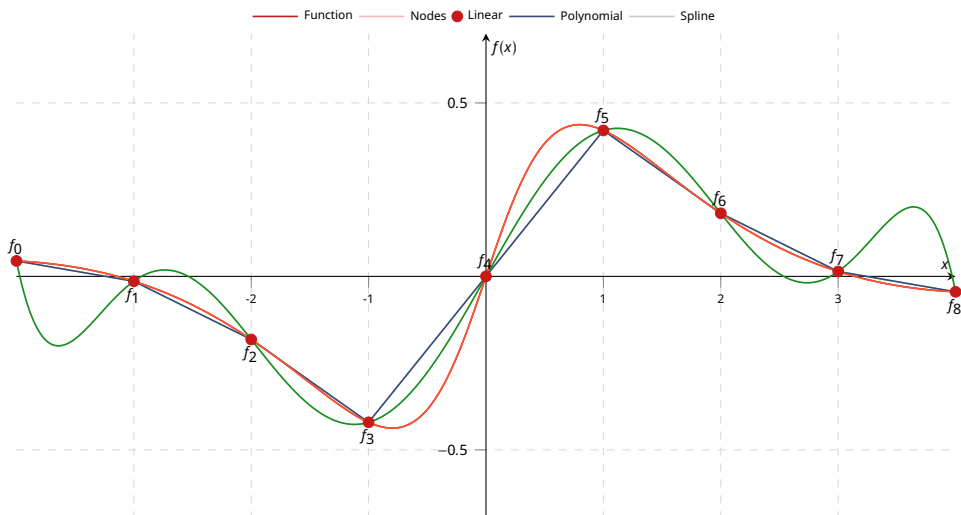
# Spline interpolation

A spline is a numerical function that represents a **smooth, higher order, piecewise polynomial** interpolants of a data set.

- **Smooth:** the interpolant is continuous in the first and second derivatives
- **Higher order:** The most common type of splines uses third-order polynomials (cubic splines)
- **Piecewise polynomial:** The interpolant is constructed between each two consecutive tabulated points

# Splines: comparison to other interpolation techniques

$$\text{Interpolation of } f(x) = \frac{\sin x}{1+x^2}$$

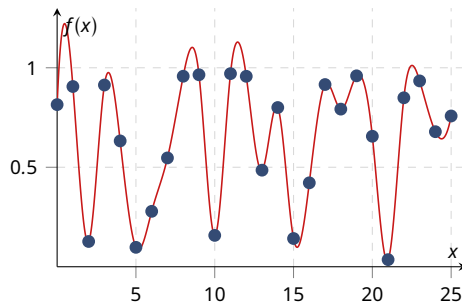




# Spline interpolation in Python

We can generate a random data set, and interpolate using `scipy.interpolate.interp1d`:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.interpolate import make_interp_spline
4
5 # Generate random data set
6 xdata = np.arange(0, 26)
7 ydata = np.random.rand(len(xdata))
8
9 # Interpolant on a fine mesh
10 xc = np.linspace(0, 25, 1001)
11 ifun = make_interp_spline(xdata, ydata)
12 yc = ifun(xc)
13
14 # Plot the data
15 plt.plot(xdata, ydata, 'o')
16 plt.plot(xc, yc, '-r')
17 plt.show()
```



Note: The **SciPy Optimize** module contains various interpolation methods with a similar interface.

# Summary

- Interpolation is used to obtain data between existing data points
  - (Bi-)Linear, polynomial and spline interpolation methods
  - Construction of Newton polynomials
  - Oscillations of high-order polynomials
- Legendre polynomials: alternative way of performing the polynomial interpolation (not discussed here)

# Interpolation tutorials

- ① In Python, generate the data:

```
1 x = np.arange(-4, 6, 1)
2 y = [0, 0, 0, 1, 1, 1, 0, 0, 0, 0]
```

Interpolate the data using polynomial interpolation (which order do you use?) and a spline. Plot the results together with the original data in a graph.

- ② Do the same exercise for the following data. Can you explain your observations?

```
1 t = [0, 0.1, 0.499, 0.5, 0.6, 1.0, 1.4, 1.5, 1.899, 1.9, 2.0]
2 y = [0, 0.06, 0.17, 0.19, 0.21, 0.26, 0.29, 0.29, 0.30, 0.31, 0.31]
```

**Hint:** Use `scipy.interpolate.interpld(..., kind="...")` to use different splines.

# Numerical integration

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group  
Eindhoven University of Technology

Numerical Methods (6BER03), 2024-2025

# Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials

# What is numerical integration?

To determine the integral  $I(x)$  of an integrand  $f(x)$ , which can be used to compute the area underneath the integrand between  $x = a$  and  $x = b$ .

$$I(x) = \int_a^b f(x) dx$$

Today we will outline different numerical integration methods.

- Riemann integrals
- Trapezoidal rule
- Simpson's rule

# Why do chemical engineers need integration?

- Obtaining the cumulative particle size distribution from a particle size distribution
- The concentration outflow over time may be integrated to yield the residence time distribution
- Integration of a varying product outflow yields the total product outflow
- Quantitative analysis of mixture components via e.g. GC/MS
- Not all function have an explicit antiderivative, e.g.  $\int e^{x^2} dx$  or  $\int \frac{1}{\ln x} dx$

# Today's outline

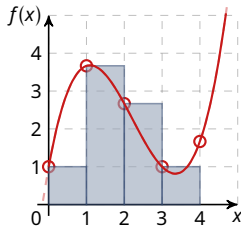
- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials



# Riemann integrals

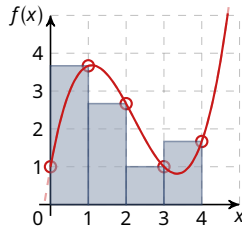
Basic idea: Subdivide the interval  $[a, b]$  into  $n$  subintervals of equal length  $\Delta x = \frac{b-a}{n}$  and use the sum of area to approximate the integral.

Left endpoint rule



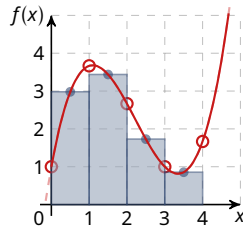
$$L_n = \sum_{i=0}^{n-1} f(x_i) \Delta x_i$$

Right endpoint rule



$$R_n = \sum_{i=0}^{n-1} f(x_{i+1}) \Delta x_i$$

Midpoint rule



$$M_n = \sum_{i=0}^{n-1} f(\bar{x}_i) \Delta x_i$$

$$\text{with } \bar{x}_i = \frac{x_i + x_{i+1}}{2}$$

# Errors in Riemann integrals

We define the exact integral as  $I = \int_a^b f(x)dx$ , and  $L_n$ ,  $R_n$  and  $M_n$  represent the left, right and midpoint rule approximations of  $I$  based on  $n$  intervals.

Writing  $f_{\max}^{(k)}$  for the maximum value of the  $k$ -th derivative, the upper-bounds of the errors by Riemann integrals are:

- $|I - L_n| \leq \frac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - R_n| \leq \frac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - M_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{24n^2}$

Note that while  $|I - L_n|$  and  $|I - R_n|$  give the same *upper-bounds* of the error, this does not mean the same error. Rather, the error is of opposite sign!

# Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials

# Trapezoid rule

Since the sign of the approximation error of the left and right endpoint rules is opposite, we can take the average of these approximations:

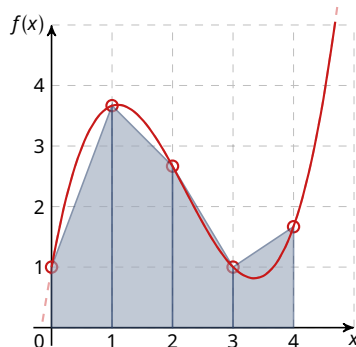
$$T_n = \frac{L_n + R_n}{2}$$

The total area is obtained by geometric reconstruction of trapezoids:

$$T_n = \sum_{i=0}^{n-1} \frac{f(x_{i+1}) + f(x_i)}{2} \Delta x_i$$

Note that this can be rewritten for equidistant intervals:

$$T_n = \frac{b-a}{2n} (f(x_0) + 2f(x_1) + \dots + 2f(x_{n-1}) + f(x_n))$$



# Error in trapezoid integration

The trapezoid rule result over  $n$  intervals  $T_n$  approximates the exact integral  $I = \int_a^b f(x)dx$ . The upper-bounds of the error is given as:

$$|I - T_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{12n^2}$$

Recall that the midpoint rule approximates with an upper-bound error of

$$|I - M_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{24n^2}$$

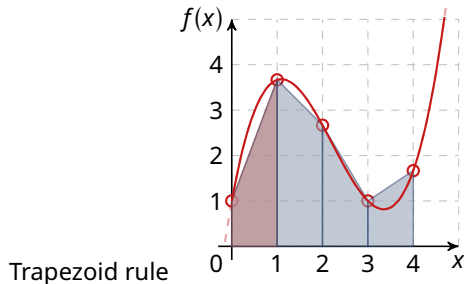
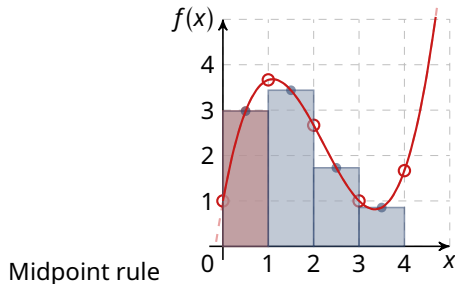
The midpoint rule approximation has lower error bounds than the trapezoid rule. A linear function is, however, better approximated by the trapezoid rule.

# Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- **Simpson's rule**
- Conclusion
- Tutorials

# Towards higher-order integration

Compare how the midpoint and trapezoid functions behave on convex and concave parts of a graph.



In convex parts (bending down), the midpoint rule tends to overestimate the integral (trapezoid underestimates). In concave parts (bending up), the midpoint rule tends to underestimate the integral (trapezoid overestimates).

## Towards higher-order integration

The errors of the midpoint rule and trapezoid rule behave in a similar way, but have opposite signs.

- Midpoint:  $|I - M_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{24n^2}$
- Trapezoid:  $|I - T_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{12n^2}$

For a quadratic function, the errors relate as:

$$|I - M_n| = \frac{1}{2}|I - T_n|$$

Taking the weighted average of these two yields the Simpson's rule:

$$S_{2n} = \frac{2}{3}M_n + \frac{1}{3}T_n$$

The  $2n$  means we have  $2n$  subintervals: the  $n$  trapezoid intervals are subdivided by the midpoint rule.



# Simpson's rule

Consider the interval  $i \in [x_0, x_2]$ , subdivided in three equidistant interpolation points:  $x_0, x_1, x_2$ .

- Midpoint:  $M_i = f\left(\frac{x_0 + x_2}{2}\right)2\Delta x = f(x_1)2\Delta x$
- Trapezoid:  $T_i = \frac{f(x_0) + f(x_2)}{2}2\Delta x$
- Simpson:  $S_i = \frac{2}{3}M_i + \frac{1}{3}T_i$

Note that  $M_i$  and  $T_i$  were computed on interval  $x_2 - x_0 = 2\Delta x$ .

Now we have:

$$\begin{aligned} S_i &= \frac{2}{3} [f(x_1)2\Delta x] + \frac{1}{3} \left[ \frac{f(x_0) + f(x_2)}{2} 2\Delta x \right] \\ &= \frac{4\Delta x}{3} f(x_1) + \frac{\Delta x}{3} f(x_0) + f(x_2) = \frac{\Delta x}{3} (f(x_0) + 4f(x_1) + f(x_2)) \end{aligned}$$

# Simpson's rule

We write  $f(x_k) = f_k$ . The integral of an interval  $i \in [x_0, x_2]$  is approximated as:

$$S_i = \frac{\Delta x}{3} (f_0 + 4f_1 + f_2)$$

The next interval,  $S_j$  with  $j \in [x_2, x_4]$  with midpoint  $x_3 = \frac{x_2+x_4}{2}$  is approximated as:

$$S_j = \frac{\Delta x}{3} (f_2 + 4f_3 + f_4)$$

If we sum these two intervals we obtain:

$$\begin{aligned} I \approx S_i + S_j &= \left[ \frac{\Delta x}{3} (f_0 + 4f_1 + f_2) \right] + \left[ \frac{\Delta x}{3} (f_2 + 4f_3 + f_4) \right] \\ &= \frac{\Delta x}{3} (f_0 + 4f_1 + 2f_2 + 4f_3 + f_4) \end{aligned}$$

# Simpson's rule

In general, Simpson's rule can be written as:

$$\begin{aligned}\int_a^b f(x)dx &\approx \sum_{\substack{k=2 \\ k \text{ even}}}^n \frac{\Delta x}{3} (f_{k-2} + 4f_{k-1} + f_k) \\ &= \frac{\Delta x}{3} (f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \dots + 2f_{n-2} + 4f_{n-1} + f_n)\end{aligned}$$

The error is given by:

$$|I - S_n| \leq \frac{f_{\max}^{(4)}(b-a)^5}{180n^4}$$

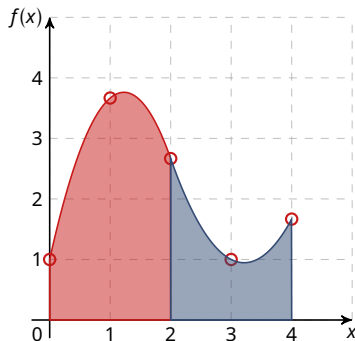
if integrand  $f$  is differentiable on  $[a, b]$ .

# Simpson's rule: example

Recall our example data, described by  $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\ldots$$

- Interpolating  $x_0, x_1$  and  $x_2$ :  $p_{2a}(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$   
 $\int_0^2 p_{2a} = \frac{55}{9} \approx 6.1111$
- Interpolating  $x_2, x_3$  and  $x_4$ :  $p_{2b}(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$   
 $\int_2^4 p_{2b} = \frac{25}{9} \approx 2.777\ldots$
- Adding the separate integrals:  
 $\int_0^2 p_{2a} + \int_2^4 p_{2b} = \frac{80}{9}$



Using Simpson's rule:

$$I \approx \frac{\Delta x}{3} (f_0 + 4f_1 + 2f_2 + 4f_3 + f_4) = \frac{1}{3} (1 + 4 \cdot 3.6667 + 2 \cdot 2.6667 + 4 \cdot 1.0000 + 1.6667) = 8.88888 = \frac{80}{9}$$

Simpson's method is of fourth order, and it gives exact approximations of third order polynomials!

# Integration in Python

Integration can be done numerically in Python.

- `np.trapz(y, x)` uses the trapezoid rule to integrate the data. Make sure you use the `x` variable if your data is not spaced with  $\Delta x = 1$ . Can handle non-equidistant data.

```
1 import numpy as np
2 x = np.linspace(-2, 2, 2001)
3 y = 1 / (x**2 + 1)
4 I = np.trapz(y, x) # Or: scipy.integrate.trapezoid
5 print(I)
```

2.214297328921525

- Integration of functions can be done using the `quad(func, a, b)` function:

```
1 import numpy as np
2 from scipy.integrate import quad
3 f = lambda x: np.exp(-x**2)
4 I, err = quad(f, 0, 10)
5 print(I, err)
```

0.886226925452758 1.8483380528941764e-13

# Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials

# What hasn't been discussed?

This course is by no means complete, and further reading is possible.

- Gaussian quadrature: A third-order integration method that requires only two base points (in contrast to the third order Simpson's method, which requires three points)
- Adaptive techniques: Parts of a function that are relatively steady (no wild oscillations) and differentiable can be integrated with much larger step sizes than other parts of the function.
- Simpson's 3/8-rule: Yet another integration technique, requiring an additional data point

# Summary

- Several techniques for numerical integration were discussed:
  - Riemann sums, trapezoid rule, Simpson's rule
  - Upper-bound errors were given for each technique
  - Built-in Python functions were illustrated
- Continue with characterization of convergence of the integration methods in the tutorials!



# Integration tutorials

- 1 Implement a function to integrate a mathematical function for a specific number of integration intervals. Implement it as a function, which can be called with arguments:

- Function (handle) to integrate
- Integration boundaries (as separate arguments or as a  $2 \times 1$  numpy array)
- Number of integration intervals

For instance: `def leftrule(func, x0, x1, N):.`

- 2 Set up a function to integrate:

```
1 def myfunction(x):  
2     return x**2 - 4*x + 6 + np.sin(5*x)
```

- 3 Integrate the function, e.g. `int_left = leftrule(myfunction, 0, 10, 25)`
- 4 Assess how the number of intervals affects the deviation from the true integral value.
- 5 Create a log-log plot of the deviation vs. number of intervals used.
- 6 Do this for all methods discussed<sup>1</sup> and compare their performance in a graph

---

<sup>1</sup> Riemann left, right, midpoint, trapezoid, and Simpson