Basics of Data Science

Peter Kandolf

2025-01-14

Table of contents

Preface

> **Warning**
>
> The notes presented here are still under construction and can change without warning.

These are the lecture notes for the *Grundlagen der Datenbasierten Methoden* class, part of the *MCI | The Entrepreneurial School* master for Mechatronik - Automation, Robotics & AI in the winter term 2024/25.

Acknowledgements

We want to thank the open source community for providing excellent tutorial and guides for Data Science topics in and with Python on the web. Individual sources are cited at the appropriate spot throughout document at hand.

We want to thank Mirjam Ziselsberger and Matthias Panny for testing, checking, suggestions and general proofreading.

These notes are built with Quarto.

> **Note**
>
> To allow acceptable formatting for multiple output formats like *html* or *pdf* some figures are numbered differently. Furthermore, the interactive content in *html* that is produced for example with `plotly` content is static in *pdf*.
> Furthermore, the numbering of some lists is in the *pdf* not consistent with regards to the *html*.

Introduction

In this class we are going to look at the basic concepts behind modern day Data Science techniques. We will always try to not only discuss the theory but also use Python to illustrate the content programmatically.

The main reference for these notes is the excellent Brunton and Kutz (2022), see citations throughout the notes. If python code is adapted form code blocks provided by Brunton and Kutz (2022), see github this is indicated and for these blocks you can also find the MATLAB equivalent in the book or on github.

These notes are intended for engineering students and therefore the mathematical concepts will rarely include rigorous proofs.

We start of by recalling the main concepts of Chapter 1 and some statistics on Chapter 2 to make sure everybody as an the same page for notation and problematically in Python.

We continue with matrix decomposition, namely Chapter 4 and Chapter 5. These basic decomposition are used to illustrate certain concepts we need in later chapters but also to show how the change of basis is influencing problems, their solution and computational properties. Furthermore, we dive into the first concepts used in machine learning where matrix computations build the foundation. As illustration we use applications for engineering and image processing.

The SVD allows us to neatly transition to Chapter 6, generalize to Chapter 7 and discuss optimization and learning properties with the help of Section 8.1.1 and Section 8.2 in Chapter 8. We mainly use toy examples but where appropriate we look at the world population or unemployment data to illustrate a concept.

In the next part we look at aspects of signal processing often found in engineering and especially mechatronics and therefore we discuss Chapter 9, Chapter 10 as well as Chapter 11 transform and how they are extended to Chapter 12. Examples range from toy examples to solving electric circuit problems or image processing.

The fifth part is based on Brunton and Kutz (2022) Chapter 3 where we look at aspects of Chapter 13 as well as the rather new topic of Chapter 14.

To round the content of these notes we look at statistics with some basis of Chapter 15 and the engineering application via Chapter 16.

Part I

Basics

In this section we are going to discuss a lot of the mathematical basics in the form of Linear Algebra as well as some topics of statistics of sets. We will always immediately show how to use the discussed content in Python.

> **Note**
>
> These notes assume that you have some basic knowledge of programming in Python and we build on this knowledge. In this sense, we use Python as a tool and only describe the inner workings if it helps us to better understand the topics at hand.
> If this is not the case have a look at MECH-M-DUAL-1-SWD, a class on software design in the same master program and from the same authors.
> Additionally, we can recommend the following books on Python:
>
> - Matthes (2023): Python Crash Course - A hands-on, project-based introduction to programming; Online Material.
> - Python Cheat Sheet provided by Matthes (2023).
> - McKinney (2022): Python for data analysis 3e; Online and Print
> - Vasiliev (2022): Python for Data Science - A Hands-On Introduction
> - Inden (2023): Python lernen – kurz & gut; German

1 Linear Algebra

The aim of this section is to discuss the basics of matrix, vector and number theory that we need for the later chapters and not introduce the whole of linear algebra. Nevertheless, we will rely on some basic definitions that can be found in any linear algebra book. For notation we refer to Golub and Van Loan (2013).

In Python the module `numpy` is used to represent vectors and matrices, we can include it like this (and give it its short hand `np`):

```
import numpy as np
```

1.1 Notation

We will refer to

$$v \in \mathbb{R}^n \quad \Leftrightarrow \quad v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix}, \quad v_i \in \mathbb{R},$$

as a *vector* v with n elements. The set $(\mathbb{R}^n, +, \cdot)$ forms a so called *vector space* with the *vector addition* $+$ and the *scalar multiplication* $\cdot$.

Definition 1.1 (Vector space)**.** For a set V over a field F with the *vectors* $u, v, w \in V$ and the *scalars* $\alpha, \beta \in F$ the following properties need to hold true to form a vector space. For the vector addition we need to have

1. associativity
$$u + (v + w) = (u + v) + w,$$
2. commutativity
$$u + v = v + u,$$
3. there needs to exists an identity element $0 \in \mathbb{R}^n$, i.e. the zero vector such that
$$v + 0 = v,$$

4. there needs to exist an inverse element

$$v + w = 0 \quad \Rightarrow w \equiv -v,$$

and this element is usually denoted by $-v$.

For the scalar multiplication we need to have

5. associativity

$$\alpha(\beta v) = (\alpha\beta)v,$$

6. distributivity with respect to the vector addition

$$\alpha(u + v) = \alpha u + \alpha v,$$

7. distributivity of the scalar addition

$$(\alpha + \beta)v = \alpha v + \beta v,$$

8. and there needs to exist a multiplicative identity element $1 \in \mathbb{R}$

$$1v = v.$$

```
v = np.array([1, 2, 3, 4])
# show the shape
print(f"{v.shape=}")
# access a single element
print(f"{v[0]=}")
# use slicing to access multiple elements
print(f"{v[0:3]=}")
print(f"{v[2:]=}")
print(f"{v[:2]=}")
print(f"{v[0::2]=}")

alpha = 0.5
w = alpha * v
print(f"{w=}")
```

```
v.shape=(4,)
v[0]=np.int64(1)
v[0:3]=array([1, 2, 3])
v[2:]=array([3, 4])
v[:2]=array([1, 2])
v[0::2]=array([1, 3])
```

```
w=array([0.5, 1. , 1.5, 2. ])
```

> **!** Important
>
> While in math we start indices with 1, Python starts with 0.

Exercise 1.1 (Vector space in Python)**.** Create some vectors and scalars with `np.array` and check the above statements with `+` and `*`.

From vectors we can move to matrices, where

$$A \in \mathbb{R}^{m\times n} \quad \Leftrightarrow \quad A = (a_{ij}) = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad a_{ij} \in \mathbb{R},$$

is called a $m \times n$ (m times n) matrix. If its values are real numbers we say it is an element of $\mathbb{R}^{m\times n}$.

> **!** Important
>
> We will use capital letters for matrices and small letters for vectors.

```python
A = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
# show the shape
print(f"{A.shape=}")
# access a single element
print(f"{A[0, 0]=}")
# use slicing to access multiple elements
print(f"{A[0, :]=}")
print(f"{A[:, 2]=}")
```

```
A.shape=(3, 4)
A[0, 0]=np.int64(1)
A[0, :]=array([1, 2, 3, 4])
A[:, 2]=array([ 3,  7, 11])
```

Consequently we can say that a vector is a $n \times 1$ matrix. It is sometimes also referred to as *column vector* and its counterpart a $1 \times n$ matrix as a *row vector*.

Exercise 1.2 (Matrix as vector space?)**.** How do we need to define $+$ and $\cdot$ to say that $(\mathbb{R}^{m\times n}, +, \cdot)$ is forming a vector space?
Does `np.array`, `+`, `*` fulfil the properties of a vector space?

If we want to refer to a row or a column of a matrix A we will use the following short hands:

- A_{i-} for *row* i,
- A_{-j} for _column j.

We can multiply a matrix with a vector, as long as the dimensions fit. Note that usually there is no $\cdot$ used to indicate multiplication:

$$Av = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & A_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = A_{-1}v_1 + A_{-2}v_2 + \cdots + A_{-n}v_n.$$

The result is a vector but this time in $\mathbb{R}^m$.

In Python the `*` operator is usually indicating multiplication. Unfortunately, in `numpy` it is interpreted as *element wise multiplication*, so we use `@` for multiplications between vector spaces.

```
w = A @ v
# show the shape
print(f"{w.shape=}")
# show the result
print(f"{w=}")
# Doing the same by hand this is tricky
w_tilde = np.zeros(A.shape[0])
for i, bb in enumerate(v):
    w_tilde += A[:, i] * bb
print(f"{w_tilde=}")
```

```
w.shape=(3,)
w=array([ 30,  70, 110])
w_tilde=array([ 30.,  70., 110.])
```

As we can see from the above equation, we can view the matrix A as a *linear mapping* or *linear function* between two vector spaces, namely from $\mathbb{R}^n$ to $\mathbb{R}^m$.

Definition 1.2 (Linear map)**.** A **linear map** between vector spaces are mappings or functions that preserve the structure of the vector space. For two vector spaces V and

W over a field F the mapping

$$T : V \to W$$

is called linear if

1. for $v, w \in V$
$$T(v + w) = T(v) + T(w),$$
2. for $v \in V$ and $\alpha \in F$
$$T(\alpha v) = \alpha T(v).$$

A linear mapping of special interest to us is the *transpose* of a matrix defined by turning rows into columns and vice versa:

$$C = A^{\mathsf{T}}, \quad \Rightarrow \quad c_{ij} = a_{ji}.$$

Consequently, the transpose of a (row) vector is a column vector.

```
print(f"{A=}")
print(f"{A.shape=}")
B = A.transpose()
print(f"{B=}")
print(f"{B.shape=}")
```

```
A=array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
A.shape=(3, 4)
B=array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
B.shape=(4, 3)
```

With this operation we can define two more mappings.

Definition 1.3 (Dot product)**.** The *dot product*, *inner product*, or *scalar product* of two vectors v and w as is defined by

$$\langle v, w \rangle = v \cdot w = v^{\mathsf{T}} w = \sum_i v_i w_i.$$

```
v = np.array([1, 2, 3, 4])
w = np.array([1, 1, 1, 1])
```

```
# alternatively we can define w with
w = np.ones(v.shape)
alpha = np.vdot(v, w)
print(f"{alpha=}")
```

```
alpha=np.float64(10.0)
```

> **Note**
>
> As $\mathbb{R}^n$ is an euclidean vector space the above function is also called the *inner product*.

Definition 1.4 (Outer product)**.** We also have the *outer product* defined as:

$$vw^\intercal = \begin{bmatrix} v_1w_1 & v_1w_2 & \dots & v_1w_n \\ v_2w_1 & v_2w_2 & \dots & v_2w_n \\ \vdots & \vdots & \ddots & \vdots \\ v_nw_1 & v_nw_2 & \dots & v_nw_n \end{bmatrix}$$

```
C = np.outer(v, w)
print(f"{C=}")
```

```
C=array([[1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.]])
```

We can also multiply matrices A and B by applying the matrix vector multiplication to each column vector of B, or a bit more elaborated:

For a $m \times p$ matrix A and a $p \times n$ matrix B the matrix-matrix multiplication ($\mathbb{R}^{m\times p} \times \mathbb{R}^{p\times n} \to \mathbb{R}^{m\times n}$)

$$C = AB \quad \Rightarrow \quad c_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj}$$

forms a $m \times n$ matrix.

Exercise 1.3 (Matrix multiplication?)**.** Show that the matrix multiplication is:

- associative
- (left and right) distributive
- but not commutative

```
C = A @ A.transpose()
print(f"{C=}")
D = A.transpose() @ A
print(f"{D=}")
```

```
C=array([[ 30,  70, 110],
       [ 70, 174, 278],
       [110, 278, 446]])
D=array([[107, 122, 137, 152],
       [122, 140, 158, 176],
       [137, 158, 179, 200],
       [152, 176, 200, 224]])
```

From the above Python snippet we can easily see that matrix-matrix multiplication is not commutative.

1.2 Norms

Definition 1.5 (Norm)**.** A **norm** is a mapping from a vector space V to the field F into the real numbers

$$\|\cdot\| : V \to \mathbb{R}_0^+, v \mapsto \|v\|$$

if it fulfils for $v, w \in V$ and $\alpha \in F$ the following

1. positivity
$$\|v\| = 0 \Rightarrow v = 0,$$
2. absolute homogeneity
$$\|\alpha v\| = |\alpha| \|v\|,$$
3. subadditivity (often called the *triangular inequality*)
$$\|v + w\| \leq \|v\| + \|w\|.$$

There are multiple norms that can be useful for vectors. The most common are:

1. the one norm
$$\|v\|_1 = \sum_i |v_i|,$$

2. the two norm (euclidean norm)

$$\|w\| = \|v\|_2 = \sqrt{\sum_i |x_i|^2} = \sqrt{\langle v, v \rangle},$$

3. more general the p-norms (for $1 \leq p \leq \infty$)

$$\|v\|_p = \left(\sum_i |v_i|^p \right)^{\frac{1}{p}},$$

4. the ∞ norm

$$\|v\|_\infty = \max_i |v_i|.$$

And for metrics:

1. the one norm (column sum norm)

$$\|A\|_1 = max_j \sum_i |a_{ij}|,$$

2. the Frobeniusnorm

$$\|A\| = \|A\|_F = \sqrt{\sum_i \sum_j |a_{ij}|^2},$$

3. the p norms are defined

$$\|A\|_p = \left(\sum_i \sum_j |a_{ij}|^p \right)^{\frac{1}{p}},$$

4. the ∞ norm (row sum norm)

$$\|A\|_1 = max_i \sum_j |a_{ij}|.$$

```
# The norms can be found in the linalg package of numpy
from numpy import linalg as LA
norm_v = LA.norm(v)
print(f"{norm_v=}")
norm_v2 = LA.norm(v, 2)
print(f"{norm_v2=}")
norm_A = LA.norm(A, 1)
print(f"{norm_A=}")
norm_Afr = LA.norm(A, "fro")
print(f"{norm_Afr=}")
```

```
norm_v=np.float64(5.477225575051661)
norm_v2=np.float64(5.477225575051661)
norm_A=np.float64(24.0)
norm_Afr=np.float64(25.495097567963924)
```

> **Note**
>
> The function `norm` from the `numpy.linalg` package can be used to compute other norms or properties as well, see docs.

1.3 Basis of vector spaces

As we will be using the notion of *basis vector* or a *basis of a vector space* we should introduce them properly.

Definition 1.6 (Basis)**.** A set of vectors $\mathcal{B} = \{b_1, \dots, b_r\}, b_i \in \mathbb{R}^n$:

1. is called linear independent if
$$\sum_{j=1}^{r} \alpha_j b_j = 0 \Rightarrow \alpha_1 = \alpha_2 = \cdots = \alpha_r = 0,$$
2. spans $\mathbb{R}^n$ if
$$v = \sum_{j=1}^{r} \alpha_j b_j, \quad \forall v \in \mathbb{R}^n, \alpha_1, \dots, \alpha_r \in \mathbb{R}.$$

The set $\mathcal{B}$ is called a *basis* of a vector space if it is linear independent and spans the entire vector space. The size of the basis, i.e. the number of vectors in the basis, is called the *dimension* of the vector space.
For a shorter notation we often associate the matrix

$$B = [b_1 | \cdots | b_n]$$

with the basis.

The standard basis of $\mathbb{R}^n$ are the vectors e_i that are zero everywhere except for index i and its associated matrix is

$$I_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & 0 \\ 0 & \dots & 0 & 1 \end{bmatrix} \in \mathbb{R}^{n \times n},$$

and called the *identity matrix*. Note, the index n is often omitted as it should be clear from the dimensions of the matrix.

The easiest way to create one of standard basis vectors, lets say $e_3 \in \mathbb{R}^3$, in Python is by calling

```
# We need to keep the index shift in mind
n = 3
e_3 = np.zeros(n)
e_3[3-1] = 1
print(f"{e_3=}")
```

```
e_3=array([0., 0., 1.])
```

and the identity matrix by

```
n = 4
I_4 = np.eye(n)
print(f"{I_4=}")
```

```
I_4=array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Example 1.1 (Standard basis)**.**

```
n = 3
e_1 = np.zeros(n)
e_1[0] = 1
e_2 = np.zeros(n)
e_2[1] = 1
e_3 = np.zeros(n)
e_3[2] = 1

x = np.random.rand(n)
print(f"{x=}")
# compute the coefficients
a = np.dot(x, e_1) / np.dot(e_1, e_1)
b = np.dot(x, e_2) / np.dot(e_2, e_2)
c = np.dot(x, e_3) / np.dot(e_3, e_3)
```

```
y = a * e_1 + b * e_2 + c * e_3
print(f"{y=}")
print(f"{np.allclose(x, y)=}")
print(f"{LA.norm(x-y)=}")
```

```
x=array([0.30403163, 0.32025007, 0.05625981])
y=array([0.30403163, 0.32025007, 0.05625981])
np.allclose(x, y)=True
LA.norm(x-y)=np.float64(0.0)
```

See `numpy.testing` for more ways of testing in `numpy`.

1.4 The inverse of a matrix

Definition 1.7 (Matrix inverse)**.** For matrices $A, X \in \mathbb{R}^{n\times n}$ that satisfy

$$AX = XA = I_n$$

we call X the inverse of A and denote it by A^{-1}.

The following holds true for the inverse of matrices:

- the inverse of a product is the product of the inverses

$$(AB)^{-1} = B^{-1}A^{-1},$$

- the inverse of the transpose is the transpose of the inverse

$$(A^{-1})^\mathsf{T} = (A^\mathsf{T})^{-1} \equiv A^{-\mathsf{T}}.$$

```
A = np.random.rand(3, 3)
print(f"{A=}")
X = LA.inv(A)
print(f"{X=}")
print(f"{A @ X=}")
print(f"{np.allclose(A @ X, np.eye(A.shape[0]))=}")
# Note that the equality is hard to achieve for floats
# Error message can currently not be displayed in pdf
# np.testing.assert_equal(A @ X, np.eye(A.shape[0]), verbose=False)
```

```
A=array([[0.17447058, 0.28783515, 0.2744076 ],
       [0.33731051, 0.0804074 , 0.07906223],
       [0.5833096 , 0.15165179, 0.94127563]])
X=array([[-0.96751291,  3.48325392, -0.01051862],
       [ 4.12223024, -0.06319366, -1.19643497],
       [-0.06457622, -2.14839518,  1.26166776]])
A @ X=array([[ 1.00000000e+00,  8.38637716e-17,  6.14025076e-17],
       [-2.77707887e-17,  1.00000000e+00,  2.58894628e-17],
       [ 5.44085126e-17,  1.00903137e-16,  1.00000000e+00]])
np.allclose(A @ X, np.eye(A.shape[0]))=True
```

Definition 1.8 (Change of basis)**.** If we associate the matrices B and C with the matrix consisting of the basis vectors of two bases $\mathcal{B}$ and $\mathcal{C}$ of a vector space we can define the transformation matrix $T_{\mathcal{C}}^{\mathcal{B}}$ from $\mathcal{B}$ to $\mathcal{C}$ as

$$T_{\mathcal{C}}^{\mathcal{B}} = C^{-1}B.$$

So if we have a vector b represented in $\mathcal{B}$ we can compute its representation in $\hat{b}$ in $\mathcal{C}$ as

$$\hat{b} = T_{\mathcal{C}}^{\mathcal{B}}b = C^{-1}Bb.$$

A special form is if we have the standard basis I and move to a basis C we get

$$\hat{b} = T_{C}^{I}b = C^{-1}b.$$

Example 1.2 (Basis Change)**.** For

$$B = \begin{bmatrix} 1 & 3 & 2 \\ 0 & 1 & 1 \\ 2 & 0 & 1 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

we get

$$T_{C}^{B} = \begin{bmatrix} \frac{3}{2} & 1 & 1 \\ \frac{1}{2} & -1 & 0 \\ -\frac{1}{2} & 2 & 1 \end{bmatrix},$$

and for a $v = 2b_1 - b_2 + 3b_3$ we can compute its representation in C as

$$\hat{v} = T_{C}^{B}v = \begin{bmatrix} \frac{3}{2} & 1 & 1 \\ \frac{1}{2} & -1 & 0 \\ -\frac{1}{2} & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 0 \end{bmatrix},$$

and therefore, $v = 5c_1 + 2c_2 + 0c_3$.
(Compare Wikipedia)

There are special basis vectors, respectively matrices that allow for easy computation of the inverse.

Definition 1.9 (Orthonormal vector)**.** We call a set of vectors $\mathcal{V} = \{u_1, u_2, \dots, u_m\}$ orthonormal if and only if

$$\forall i, j : \langle u_i, u_j \rangle = \delta_{ij}$$

where δ_{ij} is called the Kronecker delta which is 1 if and only if $i = j$ and 0 otherwise. This is true for a inner product, see Definition 1.3.

Extending this to a matrix (and to that end a basis) as follows.

Definition 1.10 (Orthogonal matrix)**.** We call a matrix $Q \in \mathbb{R}^{n \times n}$, here the real and square is important, orthogonal if its columns and rows are orthonormal vectors. This is the same as

$$Q^\mathsf{T} Q = Q Q^\mathsf{T} = I$$

and this implies that $Q^{-1} = Q^\mathsf{T}$.

For now, this concludes our introduction to linear algebra. We will come back to more in later sections.

2 Data sets

We continue our little introduction by looking at *data sets* in the sense of a list of values that we want to describe closer.

We use the `mieten3.asc` from Open Data LMU. The data set contains information about rents in Munich from the year 2003. The columns have the following meaning, see `DETAILS`:

```
Variablenbeschreibung:

nm          Nettomiete in EUR
nmqm        Nettomiete pro m² in EUR
wfl         Wohnfläche in m²
rooms       Anzahl der Zimmer in der Wohnung
bj          Baujahr der Wohnung
bez         Stadtbezirk
wohngut     Gute Wohnlage? (J=1,N=0)
wohnbest    Beste Wohnlage? (J=1,N=0)
ww0         Warmwasserversorgung vorhanden? (J=0,N=1)
zh0         Zentralheizung vorhanden? (J=0,N=1)
badkach0    Gekacheltes Badezimmer? (J=0,N=1)
badextra    Besondere Zusatzausstattung im Bad? (J=1,N=0)
kueche      Gehobene Küche? (J=1,N=0)
```

For now, we'll just show the code without much explanation because we want to jump right in and do not want to delve into how it works. We use a structured array of `numpy` for it.

```
import numpy as np
import requests
import io
response = requests.get("https://data.ub.uni-muenchen.de/2/1/miete03.asc")

# Transform the content of the file into a numpy.ndarray
data = np.genfromtxt(io.BytesIO(response.content), names=True)
# Access the data types and names
print(f"{data.dtype=}")
# Access the second element (row)
print(f"{data[1]=}")
```

```
# Access a name
print(f"{data['rooms']=}")
```

```
data.dtype=dtype([('nm', '<f8'), ('nmqm', '<f8'), ('wfl', '<f8'), ('rooms', '<f8'), ('bj', '<
data[1]=np.void((715.82, 11.01, 65.0, 2.0, 1995.0, 2.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0), d
data['rooms']=array([2., 2., 3., ..., 3., 1., 3.])
```

Now that we have some data we can look at it more closely, for this we interpret a row as a vector.

2.1 Basic properties of a data set

First we are looking at the total net rent, i.e. the row `nm`.

For a vector $v \in \mathbb{R}^n$ we have:

- the maximal value, i.e. the maximum

$$v^{max} = \max_i v_i,$$

- the minimal value, i.e. the minimum

$$v^{min} = \min_i v_i,$$

- the *mean* of all values (often called the *arithmetic mean*)

$$\overline{v} = \frac{1}{n}\sum_{i=1}^{n} v_i = \frac{v_1 + v_2 + \cdots + v_n}{n},$$

- the median, i.e. the value where half of all the other values are bigger and the other half is smaller, for a sorted v this is

$$\tilde{v} = \begin{cases} v_{(n+1)/2} & n \quad \text{odd} \\ \frac{v_{n/2}+v_{n/2+1}}{2} & n \quad \text{even} \end{cases},$$

- more general, we have quantiles. For a sorted v and $p \in (0,1)$

$$\overline{v}_p = \begin{cases} \frac{1}{2}\left(v_{np} + v_{np+1}\right) & pn \in \mathbb{N} \\ v_{\lfloor np+1 \rfloor} & pn \notin \mathbb{N} \end{cases}.$$

  Some quantiles have special names, like the median for $p = 0.5$, the lower and upper quartile for $p = 0.25$ and $p = 0.75$ (or first, second (median) and third quartile), respectively.

```
nm_max = np.max(data['nm'])
print(f"{nm_max=}")

nm_min = np.min(data['nm'])
print(f"{nm_min=}")

nm_mean = np.mean(data['nm'])
# round to 2 digits
nm_mean_r = np.around(nm_mean, 2)
print(f"{nm_mean_r=}")

nm_median = np.median(data['nm'])
print(f"{nm_median=}")

nm_quartiles = np.quantile(data['nm'], [1/4, 1/2, 3/4])
print(f"{nm_quartiles=}")
```

```
nm_max=np.float64(1789.55)
nm_min=np.float64(77.31)
nm_mean_r=np.float64(570.09)
nm_median=np.float64(534.3)
nm_quartiles=array([389.95, 534.3 , 700.48])
```

From this Python snippet we know that for tenants the rent varied between 77.31 and 1789.55, with an average of 570.09 and a median of 534.3. Of course there are tricky questions that require us to dig a bit deeper into these functions, e.g. how many rooms does the most expensive flat have? The surprising answer is 3 and it was built in 1994, but how do we obtain these results?

We can use `numpy.argwhere` or a function which returns the index directly like `numpy.argmax`.

```
max_index = np.argmax(data['nm'])
rooms = int(data['rooms'][max_index])
year = int(data['bj'][max_index])
print(f"{rooms=}, {year=}")
```

```
rooms=3, year=1994
```

2.1.1 Visualization

> **Tip**
>
> There are various ways of visualizing data in Python. Two widely used packages are `matplotlib` and `plotly`.

It often helps to visualize the values to see differences and get an idea of their use.

```
import matplotlib.pyplot as plt
nm_sort = np.sort(data["nm"])
x = np.linspace(0, 1, len(nm_sort), endpoint=True,)

plt.plot(x, nm_sort, label="net rent")
plt.axis((0, 1, np.round(nm_min/100)*100, np.round(nm_max/100)*100))
plt.xlabel('Scaled index')
plt.ylabel('Net rent - nm')

plt.plot([0, 0.25, 0.25], [nm_quartiles[0], nm_quartiles[0], nm_min],
         label='1st quartile')
plt.plot([0, 0.5, 0.5], [nm_quartiles[1], nm_quartiles[1], nm_min],
         label='2st quartile')
plt.plot([0, 0.75, 0.75], [nm_quartiles[2], nm_quartiles[2], nm_min],
         label='3st quartile')
plt.plot([0, 1], [nm_mean, nm_mean],
         label='mean')
plt.legend()
plt.show()
```

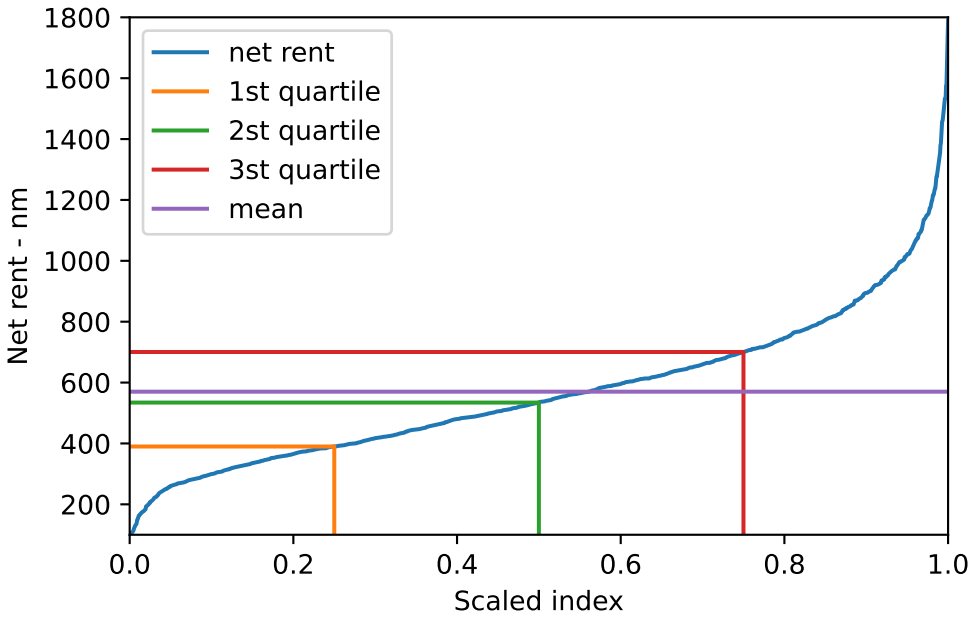


Figure 2.1: Visualization of the different measurements.

What is shown in Figure 2.1 is often combined into a single boxplot (see Figure 2.2) that provides way more information at once.

```
import plotly.graph_objects as go

fig = go.Figure()
fig.add_trace(go.Box(y=data["nm"], name="Standard"))
fig.add_trace(go.Box(y=data["nm"], name="With points", boxpoints="all"))
fig.show()
```

The plot contains the box which is defined by the 1st quantile Q_1 and the 3rd quantile Q_3, with the median as line in between these two. Furthermore, we can see the whiskers which help us identify so called outliers. By default they are defined as $\pm 1.5(Q_3 - Q_1)$, where $(Q_3 - Q_1)$ is often called the *interquartile range* (IQR).

> **Note**
>
> Figure 2.2 is an interactive plot in the html version.

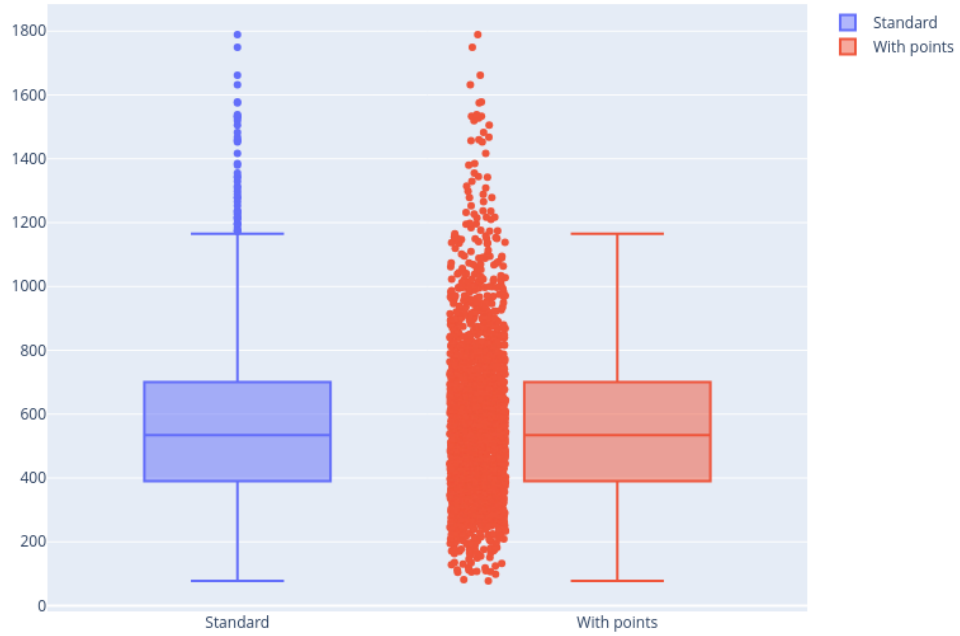


Figure 2.2: Boxplot done in plotly with whiskers following 3/2 IQR.

2.2 Spread

The *spread* (or *dispersion*, *variability*, *scatter*) are measures used in statistics to classify how data is distributed. Common examples are *variance*, *standard deviation*, and the *interquartile range* that we have already seen above.

Definition 2.1 (Variance)**.** For a finite set represented by a vector $v \in \mathbb{R}^n$ the **variance** is defined as

$$\mathrm{Var}(v) = \frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2, \quad \mu = \overline{v} \quad \text{(the mean)}$$

or directly

$$\mathrm{Var}(v) = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j>i}(v_i - v_j)^2.$$

Definition 2.2 (Standard deviation)**.** For a finite set represented by a vector $v \in \mathbb{R}^n$ the

standard deviation is defined as

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2}, \quad \mu = \overline{v} \quad \text{(the mean)}.$$

If we interpret v as a sample this is often also called *uncorrected sample standard deviation*.

Definition 2.3 (Interquartile range (IQR))**.** For a finite set represented by a vector $v \in \mathbb{R}^n$ the **interquartile range** is defined as the difference of the first and third quartile, i.e.

$$IQR = \overline{v}_{0.75} - \overline{v}_{0.25}.$$

With `numpy` they are computed as follows

```
nm_var = np.var(data["nm"])
print(f"{nm_var=}")

nm_std = np.std(data["nm"])
print(f"{nm_std=}")

nm_IQR = nm_quartiles[2] - nm_quartiles[0]
print(f"{nm_IQR=}")
```

```
nm_var=np.float64(60208.75551600402)
nm_std=np.float64(245.37472468859548)
nm_IQR=np.float64(310.53000000000003)
```

2.3 Histogram

When exploring data it is also quite useful to draw histograms. For the *net rent* this makes not much sense but for *rooms* this is useful.

```
index = np.array(range(0, len(data['rooms'])))

plt.hist(data['rooms'])
plt.xlabel('rooms')
plt.ylabel('# of rooms')
plt.show()
```

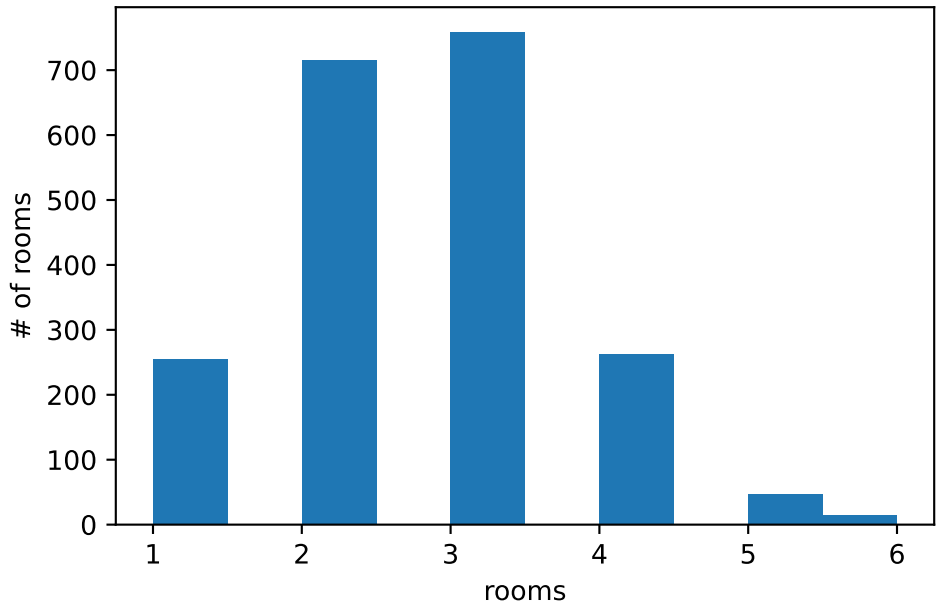


Figure 2.3: Histogram of the number of rooms in our dataset.

What we see in Figure 2.3 is simply the amount of occurrences of 1 to 6 in our dataset. Already we can see something rather interesting, there are flats with 5.5 rooms in our dataset.

Another helpful histogram is Figure 2.4 showing the amount of buildings built per year.

```
index = np.array(range(0, len(data['rooms'])))

plt.hist(data['bj'])
plt.xlabel('year of building')
plt.ylabel('# of buildings')
plt.show()
```

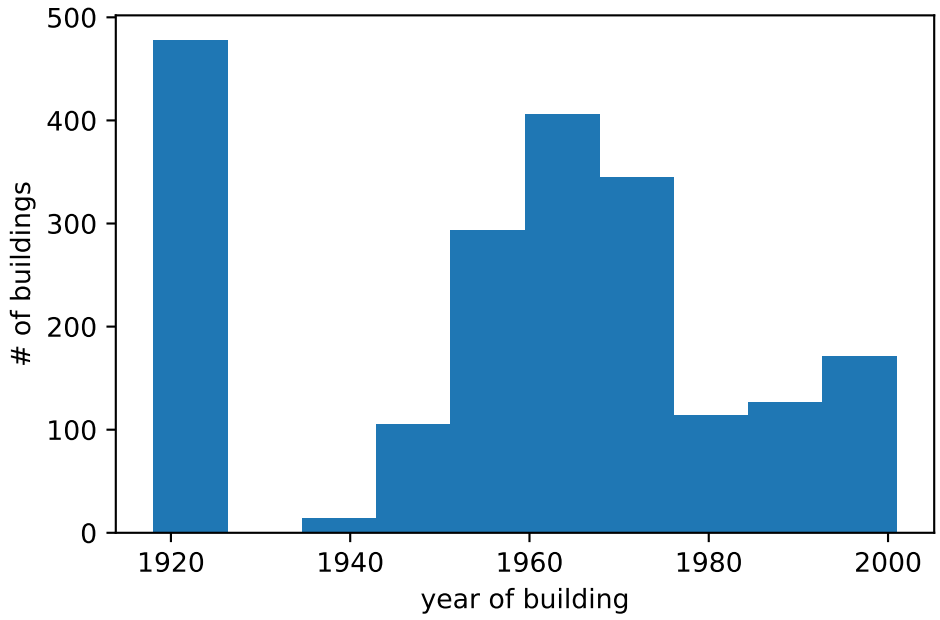


Figure 2.4: Histogram of buildings built per year.

2.4 Correlation

In statistics, the terms *correlation* or *dependence* describe any statistical relationship between *bivariate data* (data that is paired) or *random variables*.

For our dataset we can, for example, check:

1. the *living area in* m^2 - `wfl` vs. the *net rent* - `nm`
2. the *year of construction* - `bj` vs. if *central heating* - `zh0` is available
3. the *year of construction* - `bj` vs. the *city district* - `bez`

```
from plotly.subplots import make_subplots

fig = make_subplots(rows=3, cols=1)

fig.add_trace(go.Scatter(x=data["wfl"], y=data["nm"], mode="markers"),
              row=1, col=1)
fig.update_xaxes(title_text="living area in m^2", row=1, col=1)
fig.update_yaxes(title_text="net rent", row=1, col=1)

fig.add_trace(go.Scatter(x=data["bj"], y=data["zh0"], mode="markers"),
```

```
                row=2, col=1)
fig.update_xaxes(title_text="year of construction", row=2, col=1)
fig.update_yaxes(title_text="central heating", row=2, col=1)

fig.add_trace(go.Scatter(x=data["bj"], y=data["bez"], mode="markers"),
                row=3, col=1)
fig.update_xaxes(title_text="year of construction", row=3, col=1)
fig.update_yaxes(title_text="city district", row=3, col=1)
```

```
fig.show()
```

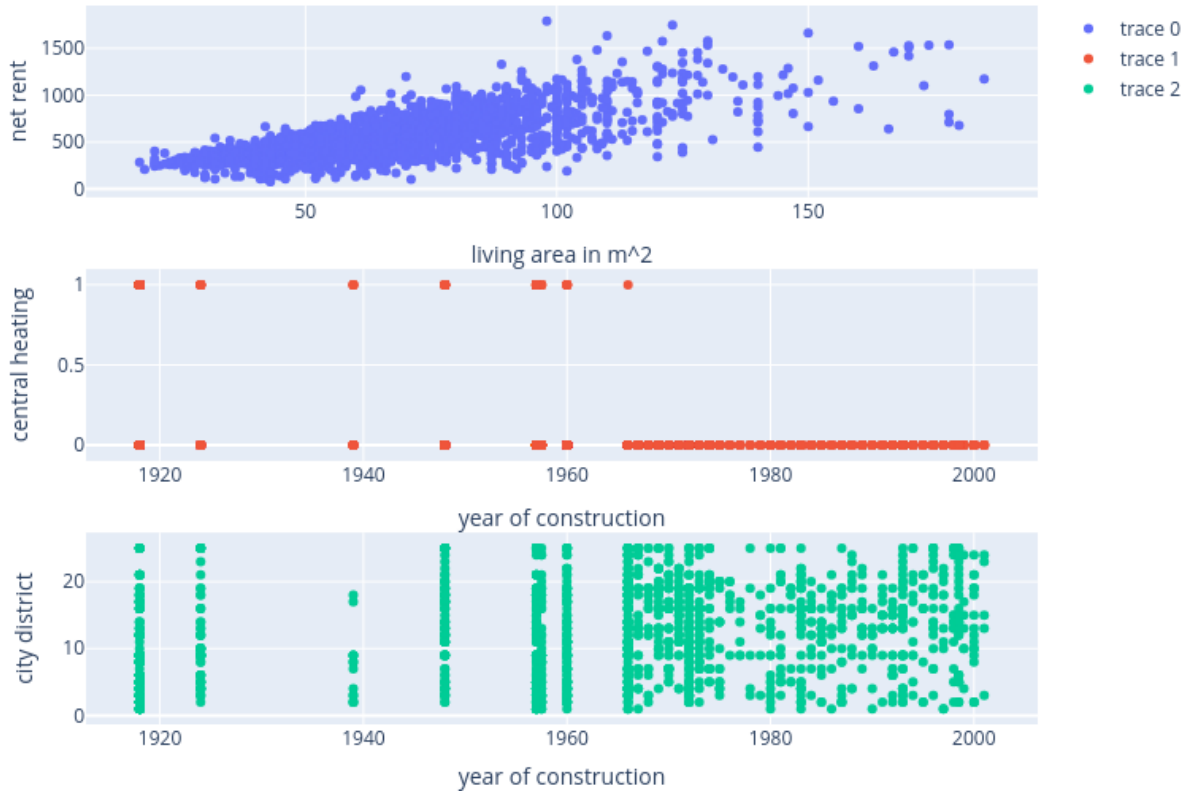


Figure 2.5: Scatterplot to investigate correlations in the data set.

In the first plot of Figure 2.5 we see that the rent tends to go up with the size of the flat but there are for sure some rather cheap options in terms of space.

The second plot of Figure 2.5 tells us that central heating became a constant around 1966. Of course we can also guess that the older buildings with central heating were renovated, but we have no data to support this claim.

The third plot of Figure 2.5 does not yield an immediate correlation.

More formally, we can describe possible correlations using the *covariance*. The covariance is a measure of the joint variability of two random variables.

Definition 2.4 (Covariance)**.** For two finite sets represented by vectors $v, w \in \mathbb{R}^n$ the **covariance** is defined as

$$\operatorname{cov}(v, w) = \frac{1}{n}\langle v - \overline{v}, w - \overline{w}\rangle.$$

The covariance is tricky to interpret, e.g. the unities of the two must not make sense. In the example below, we have rent per square meter, which makes some sense.

From the covariance we can compute the correlation.

Definition 2.5 (Correlation)**.** For two finite sets represented by vectors $v, w \in \mathbb{R}^n$ the **correlation** is defined as

$$\rho_{v,w} = \operatorname{corr}(v, w) = \frac{\operatorname{cov}(v, w)}{\sigma_v \sigma_w},$$

where σ_v and σ_w are the standard deviation of these vectors, see Definition 2.2.

In `numpy` the function `numpy.cov` computes a matrix where the diagonal is the variance of the values and the off-diagonals are the covariances of the i and j samples. Consequently, `numpy.corrcoef` is a matrix as well.

```
cov_nm_wfl = np.cov(data["nm"], data["wfl"])
print(f"{cov_nm_wfl[0, 1]=}")

corr_nm_wfl = np.corrcoef(data["nm"], data["wfl"])
print(f"{corr_nm_wfl[0, 1]=}")
```

```
cov_nm_wfl[0, 1]=np.float64(4369.1195844122)
corr_nm_wfl[0, 1]=np.float64(0.7074626685750687)
```

The above results, particularly $\rho_{\text{nm,wfl}} = 0.707$ suggest that the higher the rent, the more space you get.

> **Tip**
>
> Correlation and causation are not the same thing!

Tip

We showed some basic tests for correlation, there are more elaborate methods but they are subject to a later chapter.

3 Epilogue

This sums up our basic introduction. We introduced the basic mathematical constructs to use in further sections and learned how to work with them in Python.

Part II

Matrix decompositions

There are a lot of different matrix decompositions and they can be used to fulfil several tasks. We are going to look into the *eigendecomposition* as well as the *singular value decomposition*. Both of these can, for example, be used for picture compression and recognition.

For notation we are following again Golub and Van Loan (2013), which has plenty more to offer than we cover in these notes.

4 Eigendecomposition

We start off with the so called *eigendecomposition*. The main idea is to compute a decomposition (or factorization) of a square matrix into a canonical form. We will represent a matrix by its *eigenvalues* and *eigenvectors*. In order to do so we need to recall some more definitions from our favourite linear algebra book, such as Golub and Van Loan (2013).

Definition 4.1 (Determinant)**.** If $A = (a) \in \mathbb{R}^{1\times 1}$ (a single number), then its *determinant* is given by $\det(A) = a$. Now, the determinant of $A \in \mathbb{R}^{n\times n}$ is defined in terms of order-$(n-1)$ determinants:

$$\det(A) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} \det(A_1 j),$$

where $A_1 j$ is a $(n-1) \times (n-1)$ matrix obtained by deleting the first row and the jth column of A.
(Compare Golub and Van Loan 2013, 66–67)

Some important properties of the determinant are:

1. for two matrices A and B in $\mathbb{R}^{n\times n}$

$$\det(AB) = \det(A)\det(B),$$

2. for the transpose A^T of a matrix A

$$\det(A^\mathsf{T}) = \det(A),$$

3. for a scalar $\alpha \in \mathbb{R}$ and a matrix A

$$\det(\alpha A) = \alpha^n \det(A),$$

4. we call a matrix A *nonsingular* (i.e. invertible) if and only if

$$\det(A) \neq 0,$$

5. for an invertible matrix A^{-1}

$$\det\left(A^{-1}\right) = \frac{1}{\det(A)},$$

So let us check in `numpy`:

```python
import numpy as np
from numpy import linalg as LA

A = np.array([[2, 0, 0],
              [0, 3, 4],
              [0, 4, 9]])
det_A = LA.det(A)
print(f"{det_A=}")

A_T = np.transpose(A)
det_A_T = LA.det(A_T)
print(f"{det_A_T=}")

X = LA.inv(A)
det_X = LA.det(X)
print(f"{det_X=}")

print(f"{det_X*det_A=}")

det_Am = LA.det(-A)
print(f"{det_Am=}")
```

```
det_A=np.float64(21.999999999999996)
det_A_T=np.float64(21.999999999999996)
det_X=np.float64(0.04545454545454546)
det_X*det_A=np.float64(1.0)
det_Am=np.float64(-21.999999999999996)
```

Definition 4.2 (Eigenvalues)**.** For a matrix $A \in \mathbb{C}^{n\times n}$ the *eigenvalues* are the roots of the *characteristic polynomial*

$$p(\lambda) = \det(A - \lambda I).$$

Consequently, every $n \times n$ matrix has exactly n eigenvalues. Note that for a real matrix the eigenvalues might still be in $\mathbb{C}$.
The set of all eigenvalues of A is denoted by

$$\lambda(A) = \{\lambda : \det(A - \lambda I) = 0\}.$$

If all of the eigenvalues are real (in $\mathbb{R}$) we can sort them from largest to smallest

$$\lambda_n(A) \leq \cdots \leq \lambda_2(A) \leq \lambda_1(A),$$

and we denote the largest by $\lambda_{max}(A) = \lambda_1(A)$ and the smallest by $\lambda_{min}(A) = \lambda_n(A)$, respectively.

(Compare Golub and Van Loan 2013, 66–67)

From the above properties of the determinant we can conclude that, if $X \in \mathbb{C}^{n\times n}$ is nonsingular and $B = X^{-1}AX$, then A and B are called *similar* and two similar matrices have exactly the same eigenvalues.

```
lam = LA.eigvals(A)
lam_sort = np.sort(lam)
```

```
print(f"{lam_sort=}")
```

```
lam_sort=array([ 1.,  2., 11.])
```

Definition 4.3 (Eigenvector)**.** If $\lambda \in \lambda(A)$, then there exists a nonzero vector v so that

$$Av = \lambda v \iff (A - \lambda I)v = 0. \tag{4.1}$$

Such a vector is called *eigenvector* of A associated with λ.

Definition 4.4 (Eigendecomposition)**.** If $A \in \mathbb{C}^{n\times n}$ has n (linear) independent eigenvectors $v_1, \dots, v_n$ and $Av_i = \lambda_i v_i$, for $i = 1 : n$, than A is *diagonalizable*.
If we combine the eigenvectors to a matrix

$$V = [v_1|\cdots|v_n],$$

then

$$V^{-1}AV = \operatorname{diag}(\lambda_1, \dots, \lambda_n) = \Lambda = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \lambda_n \end{bmatrix}.$$

This is called the *eigendecomposition* of the matrix A.
(Compare Golub and Van Loan 2013, 66–67)

Not all matrices $A \in \mathbb{R}^{n\times n}$ are diagonalizable.

```
lam, V = LA.eig(A)
A_eigen = np.diag(lam)
print(f"{A_eigen=}")
print(f"{V=}")

v_1 = (A @ V[:, 0]) / lam[0]
```

```
print(f"{v_1=}")

A_eigen2 = LA.inv(V) @ A @ V
print(f"{A_eigen2=}")

print(f"{np.allclose(A_eigen, A_eigen2)=}")
```

```
A_eigen=array([[11.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  2.]])
V=array([[ 0.        ,  0.        ,  1.        ],
       [ 0.4472136 ,  0.89442719,  0.        ],
       [ 0.89442719, -0.4472136 ,  0.        ]])
v_1=array([0.        , 0.4472136 , 0.89442719])
A_eigen2=array([[1.10000000e+01, 0.00000000e+00, 0.00000000e+00],
       [5.55111512e-17, 1.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 2.00000000e+00]])
np.allclose(A_eigen, A_eigen2)=True
```

We can use what we have learned about the basis of a vector space and the transformation between two bases, see Section 1.3 to get a different interpretation of the eigendecomposition.

We recall, if x is represented in the standard basis with matrix I we can change to the basis represented by V and therefore $\hat{x} = V^{-1}x$.

Consequently, the equation

$$A = V\Lambda V^{-1}$$

means that in the basis created by V the matrix A is represented by a diagonal matrix.

4.1 Examples for the application

To get a better idea what the eigendecomposition can do we look into some examples.

4.1.1 Solving system of linear equations

For a system of linear equations $Ax = b$ we get

$$\begin{array}{lll} Ax & = b & \iff \\ V\Lambda V^{-1}x & = b & \iff \\ \Lambda V^{-1}x & = V^{-1}b & \iff \\ V^{-1}x & = \Lambda^{-1}V^{-1}b & \iff \\ x & = V\Lambda^{-1}V^{-1}b & \iff \end{array}$$

As $\Lambda^{-1} = \operatorname{diag}(\lambda_1^{-1}, \dots, \lambda_n^{-1})$ this is easy to compute once we have the eigenvalue decomposition.

> **Note**
>
> The computation of the eigendecomposition is not cheap, therefore this is not always worth the effort and there are other ways of solving linear systems.

4.1.2 Linear Ordinary Differential Equations

In this example we use the eigendecomposition to efficiently solve a system of differential equations

$$\dot{x} = Ax, \quad x(0) = x_0$$

By changing to the basis V and using the notation $\hat{x} = z$ we have the equivalent formulations

$$z = V^{-1}x \iff x = Vz,$$

and if follows

$$\begin{aligned} \dot{x} = Ax &\iff V\dot{z} &&= AVz \\ &\iff \dot{z} &&= V^{-1}AVz \\ &\iff \dot{z} &&= \Lambda z \end{aligned} .$$

So for an initial value z_0 the solution in t is

$$z(t) = \operatorname{diag}(e^{t\lambda_1}, \dots, e^{t\lambda_n})\, z_0.$$

We often say that it is now a *decoupled* differential equation.

4.1.3 Higher Order Linear Differential Equations

If we have a higher order linear ODE such as

$$x^{(n)} + a_{n-1}x^{(n-1)} + \cdots + a_2\ddot{x} + a_1\dot{x} + a_0 x = 0. \tag{4.2}$$

we can stack the derivatives into a vector

$$\begin{array}{rcl} x_1 &=& x \\ x_2 &=& \dot{x} \\ x_3 &=& \ddot{x} \\ \vdots &=& \vdots \\ x_{n-1} &=& x^{(n-2)} \\ x_n &=& x^{(n-1)} \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \\ \vdots \\ x^{(n-2)} \\ x^{(n-1)} \end{bmatrix},$$

and taking the derivative of this vector yields the following system

$$\underbrace{\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix}}_{\dot{x}} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-2} & -a_{n-1} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix}}_{x}.$$

We transformed it into a system of coupled 1st order ODEs $\dot{x} = Ax$ and we can solve this as seen above. More importantly, the characteristic polynomial of Equation 4.2 is equal to the characteristic polynomial of Definition 4.2 and the eigenvalues are the roots of this polynomial.

4.1.4 Generalized eigenvalue problem

Let us motivate this by the example of *modal analysis*. If we consider the free vibrations of a undamped system we get the equation

$$M\ddot{u} + Ku = 0, \quad u(0) = u_0,$$

with the mass matrix M and the stiffness matrix K and $u(t)$ being the displacement. As we know, the solution of this linear differential equation has the form $u(t) = e^{i\omega t}u_0$ and thus we get

$$(-\omega^2 M + K)u_0 = 0. \tag{4.3}$$

Definition 4.5 (Generalized eigenvalue problem)**.** If $A, B \in \mathbb{C}^{n\times n}$, then the set of all matrices of the form $A-\lambda B$ with $\lambda \in \mathbb{C}$ is a *pencil*. The *generalized eigenvalues* of $A-\lambda B$ are elements of the set $\lambda(A, B)$ defined by

$$\lambda(A, B) = \{z \in \mathbb{C} : \det(A - zB) = 0\}.$$

If $\lambda \in \lambda(A, B)$ and $0 \neq v \in \mathbb{C}^n$ satisfies

$$Av = \lambda Bv, \tag{4.4}$$

then v is an *eigenvector* of $A - \lambda B$. The problem of finding a nontrivial solution to Equation 4.4 is called the *generalized eigenvalue problem*.
(Compare Golub and Van Loan 2013, chap. 7.7)

In our example Equation 4.3 the eigenvalues are $\lambda = \omega^2$ and correspond to the square of the natural frequencies and the eigenvectors $v = u$ correspond to the modes of vibration.

If M is invertible we can write

$$M^{-1}(K - \omega^2 M)u_0 = (M^{-1}K - \omega^2 I)u_0 = 0.$$

> **Warning**
>
> In most cases inverting the matrix is not recommended, to directly solve the generalized eigenvalue problem, see (Golub and Van Loan 2013, chap. 7.7) for details.

Let us walk through this with an example from (Downey and Micheli 2024, sec. 8.3).

Example 4.1 (Two Story Building)**.**

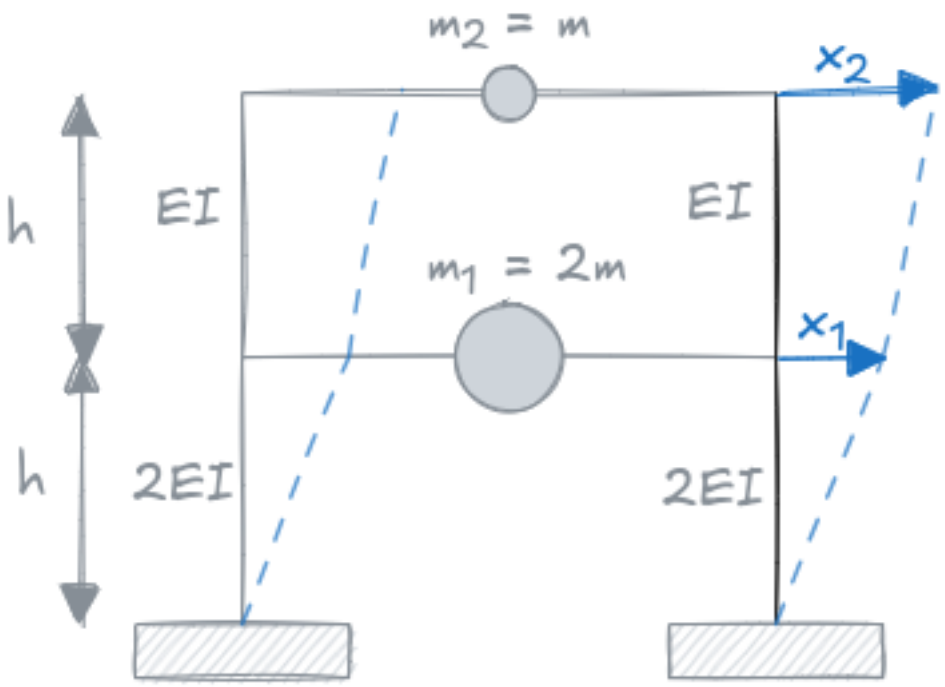


Figure 4.1: Two story frame where the floors have different dynamic properties.

In Figure 4.1 we can see a two story building consisting of a two column frame with a floor. The first floor columns are fixed at the base and have a height h, the second floor is fixed to the first and has the same height. The columns of each frame are modelled as beams with flexural rigidity EI and $2EI$, respectively Where E is called *Young's modulus* and I the *second moment of area*. The mass is centred at the floor level.
This allows us to model such a building as a system with two degrees of freedom x_1

and x_2 as the displacement indicated in blue. The dashed blue line would be such a displacement for the frame.
The resulting equations of motion become

$$\begin{aligned} m_1\ddot{x_1} + k_1 x_1 + k_2(x_2 - x_1) = 0, \\ m_2\ddot{x_1} + k_2(x_2 - x_1) = 0, \end{aligned}$$

resulting in the matrices

$$M = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}, \quad K = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix}.$$

For the derivation of the stiffness coefficients we refer to (Downey and Micheli 2024, 188) and recall the result here as follows

$$k_1 = \frac{48EI}{h^3}, \quad k_2 = \frac{24EI}{h^3}.$$

This results in the stiffness matrix

$$K = \underbrace{\frac{24EI}{h^3}}_{=k} \begin{bmatrix} 3 & -1 \\ -1 & 1 \end{bmatrix}.$$

Now we can manually compute

$$\det(-\omega^2 M + K) = 0 \Leftrightarrow 2m^2\omega^4 - 5mk\omega^2 + 2k^2 = 0,$$

and solving this equation for ω^2 results in

$$\omega_1^2 = \frac{k}{2m}, \quad \omega_2^2 = \frac{2k}{m}.$$

To compute the eigenvectors v_1 and v_2 we use Equation 4.1, i.e. for v_1 this reads as

$$-\frac{k}{2m}\begin{bmatrix} 2m & 0 \\ 0 & m \end{bmatrix} + k\begin{bmatrix} 3 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix} = \begin{bmatrix} 2k & -k \\ -k & \frac{k}{2} \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix} \overset{!}{=} \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

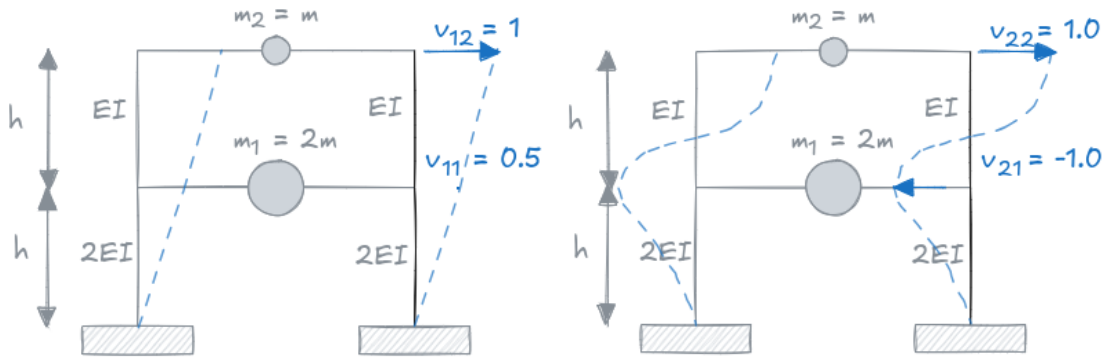
and results in the relation $2v_{11} = v_{21}$. We can select a solution as

$$v_1 = \begin{bmatrix} \frac{1}{2} \\ 1 \end{bmatrix}.$$

For v_2 we proceed similarly and derive a solution as:

$$v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

The eigenvectors illustrate how the displacement functions and are not just some theoretical value. The following figure visualizes the two modes.



(a) First mode (b) Second mode

Figure 4.2: Model of a two story building with the shape of the modes according to the modal analysis.

Two wrap up the example our overall temporal response consists of the time invariant part defined by our eigenvectors v_1, and v_2, as well as the time dependent part with our eigenfrequencies ω_1 and ω_2 as well as the constants A_1, A_2, ϕ_1, ϕ_2 depending on the initial condition (see Downey and Micheli 2024, chap. 5).

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = [v_1, v_2] \begin{bmatrix} A_1 \sin(\omega_1 t + \phi_1) \\ A_2 \sin(\omega_2 t + \phi_2) \end{bmatrix}$$

(Compare Downey and Micheli 2024, chap. 8.3, pp. 189-191)

4.1.5 Low-rank approximation of a square matrix

We can use the eigenvalue decomposition to *approximate* a square matrix.

Let us sort the eigenvalues in Λ and let us call $U^\mathsf{T} = V^{-1}$ then we can write

$$A = V \Lambda U^\mathsf{T} = \sum_{i=1}^{n} \lambda_i v_i u_i^\mathsf{T}$$

where v_i and u_i correspond to the rows of the matrices. Now we can define the *rank* r approximation of A as

$$A \approx A_r = V \Lambda U^\mathsf{T} = \sum_{i=1}^{r} \lambda_i v_i u_i^\mathsf{T}.$$

To make this a bit easier to understand the following illustration is helpful:

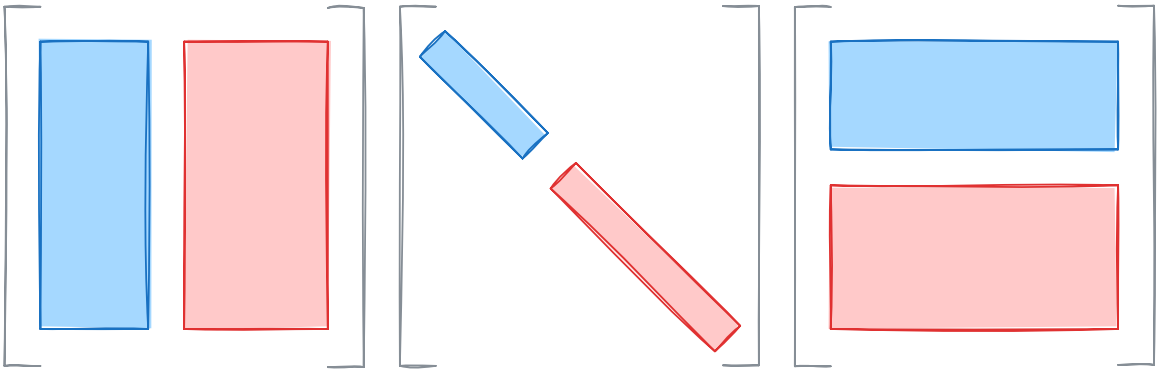


Figure 4.3: Low Rank Approximation

This approximation can be used to reduce the storage demand of an image.

```
import matplotlib.pyplot as plt
import imageio.v3 as iio
import numpy as np
import numpy.linalg as LA
%config InlineBackend.figure_formats = ["svg"]

im = np.asarray(iio.imread(
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

im_gray = rgb2gray(im)
im_cut = im_gray[1500:3001,1500:3001] / 255

lam_, V_ = LA.eig(im_cut)
order = np.argsort(np.abs(lam_))
lam = lam_[order[::-1]]
V = V_[:, order[::-1]]

rec = [1/1000, 10/100, 25/100, 50/100, 75/100]
Vinv = LA.inv(V)

plt.figure()
```

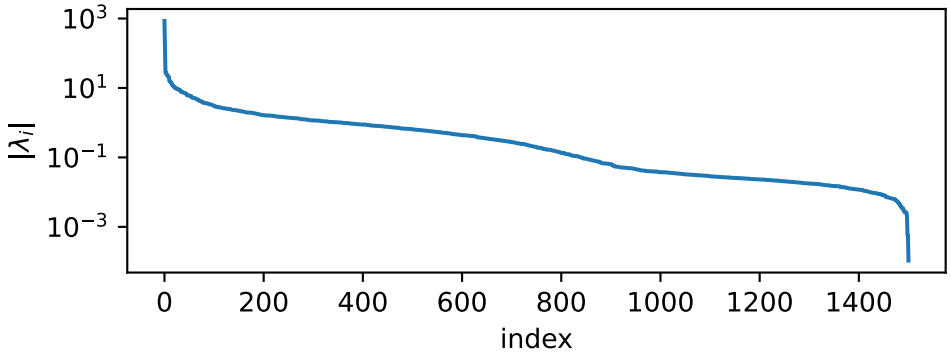
```
plt.plot((np.abs(lam)))
plt.yscale("log")
plt.ylabel(r"$|\lambda_i|$")
plt.xlabel("index")
plt.gca().set_aspect(7e1)

for p in rec:
    plt.figure()
    r = int(np.ceil(len(lam) * p))
    A_r = np.real(V[:, 0:r] @ np.diag(lam[0:r], 0) @ Vinv[0:r, :])
    plt.imshow(A_r, cmap=plt.get_cmap("gray"))
    plt.gca().set_axis_off()

plt.figure()
plt.imshow(im_cut, cmap=plt.get_cmap("gray"))
plt.gca().set_axis_off()
plt.show()
```

4.2 Summary

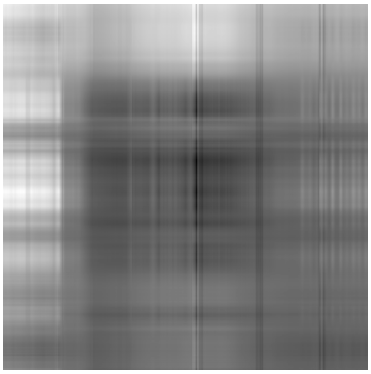
The eigenvalue decomposition is an important tool but it has its limitations:

1. the matrices involved need to be square
2. eigenvalues might be complex, even if the problem at hand is real
3. we only get a diagonal matrix Λ if all eigenvectors are linear independent
4. the computation of V^{-1} is non-trivial unless A is symmetric and V becomes unitary ($V^{-1} = V^{\mathsf{T}}$).

Therefore, we will look into a generalized decomposition called the *singular value decomposition* in the next section.



(a) Absolute value of Eigenvalues from largest to smallest



(b) 0.1% of eigenvalue

(c) 10% of eigenvalue

(d) 25% of eigenvalue

(e) 50% of eigenvalue

(f) 75% of eigenvalue

(g) original image

Figure 4.4: Image of MCI I and the reconstruction with approximated matrix.

5 Singular Value Decomposition

The Singular Value Decomposition (SVD) is an orthogonal matrix reduction. In contrast to the previously discussed Eigendecomposition it can be applied to rectangular matrices. As we will see the interpretation of the SVD can be linked to the eigendecomposition but first, lets provide a proper definition.

Theorem 5.1 (Singular Value Decomposition)**.** *If $A \in \mathbb{R}^{m\times n}$ (a real $m \times n$ matrix), there exists orthogonal matrices*

$$U = [u_1|\cdots|u_m] \in \mathbb{R}^{m\times m} \quad \textit{and} \quad V = [v_1|\cdots|v_n] \in \mathbb{R}^{n\times n}$$

such that

$$U^\mathsf{T} AV = \Sigma = \operatorname{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m\times n}, \quad p = \min\{m, n\}, \tag{5.1}$$

where $\sigma_1 \geq \sigma_2 \geq \dots \sigma_p \geq 0$.
(Compare Golub and Van Loan 2013, 76–80)

> **Note**
>
> We call σ_i a *singular value* of A, the u_i are called the *left singular vectors* of A and v_i the *right singular vectors* of A. Furthermore, $\sigma_{max}(A)$ is the largest singular value of A and $\sigma_{min}(A)$ is the smallest singular value of A.

Instead of providing a concise proof (if you are interested see (Golub and Van Loan 2013, 76)) we show a possible motivation of the definition.

If we compute the eigendecompositions of AA^T and $A^\mathsf{T}A$. The two matrices have the same positive eigenvalues - the squares of the eigenvalues of A and we get

$$\begin{aligned} (AA^\mathsf{T})U &= U(\Lambda\Lambda^\mathsf{T}), \\ (A^\mathsf{T}A)V &= V(\Lambda^\mathsf{T}\Lambda). \end{aligned}$$

For $A \in \mathbb{R}^{m\times n}$ with $m > n$ we get

$$\Lambda = \begin{bmatrix} \tilde{\Lambda} \\ 0 \end{bmatrix}$$

with the diagonal matrix $\tilde{\Lambda} \in \mathbb{R}^{n \times n}$ and

$$\begin{aligned} \Lambda\Lambda^\mathsf{T} &= \begin{bmatrix} \tilde{\Lambda}^2 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} \tilde{\Sigma} & 0 \\ 0 & 0 \end{bmatrix}, \\ \Lambda^\mathsf{T}\Lambda &= \tilde{\Lambda}^2 = \tilde{\Sigma}. \end{aligned}$$

If we expand the matrices with zeros to match the correct dimensions this corresponds to our singular value decomposition

$$A = U\Sigma V^\mathsf{T}.$$

> **!** Important
>
> The singular value decomposition always exists, is real and all singular vectors are positive.

Again, to visualize the composition helps to better understand what is happening

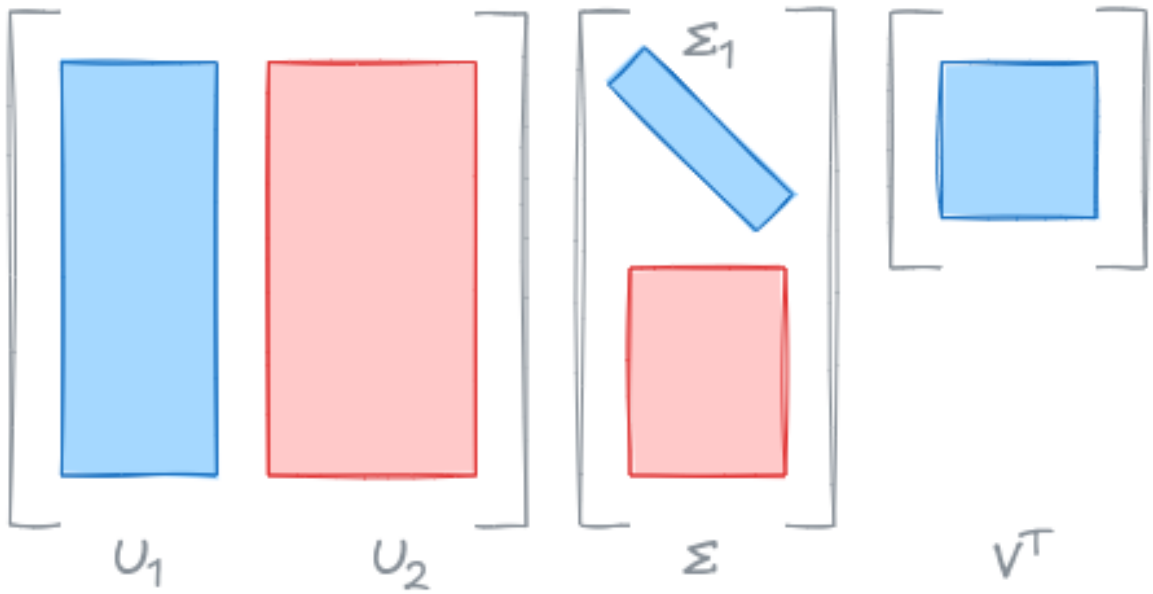


Figure 5.1: Singular Value Decomposition

In the case that of $m \geq n$ we can save storage by reducing the matrices U and Σ, to their counterpart U_1 and Σ_1 by removing the zeros.

Definition 5.1 (Thin SVD)**.** If $A \in \mathbb{R}^{m \times n}$ for $m \geq n$, then

$$A = \tilde{U}\tilde{\Sigma}V^\mathsf{T}$$

where

$$U_1 = U(:, 1:n) = [u_1|\cdots|u_n] \in \mathbb{R}^{m\times n}$$

and

$$\tilde{\Sigma} = \Sigma(1:n, 1:n) = \text{diag}(\sigma_1, \ldots, \sigma_n) \in \mathbb{R}^{n\times n}.$$

(Compare Golub and Van Loan 2013, 80)

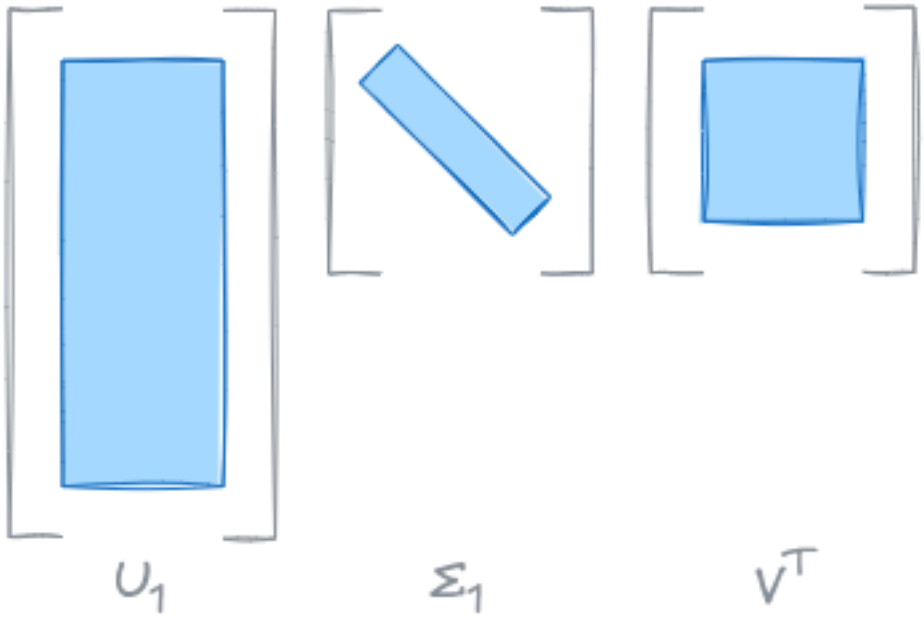


Figure 5.2: Thin SVD

In Python we can compute the (Thin)SVD as follows

```
import numpy as np
import numpy.linalg as LA

A = np.array([[3, 2, 2],
              [2, 3, -2]
              ])

U, s, Vh = LA.svd(A, full_matrices=True)
print(f"{U.shape=}, {s.shape=}, {Vh.shape=}")
print(f"{U=}")
print(f"{U @ U.transpose()=}\n")
```

```
print(f"{np.diag(s)=}\n")

print(f"{Vh=}")
print(f"{Vh.transpose() @ Vh=}\n")

A1 = U[:, :len(s)] @ np.diag(s) @ Vh[:len(s),:]
print(f"{np.allclose(A, A1)=}")
print(f"{A-A1=}\n")

S = np.zeros(A.shape)
S[:len(s), :len(s)] = np.diag(s)
A2 = U @ S @ Vh
np.allclose(A, A2)
print(f"{A-A2=}\n")

print("\nThin SVD:")
U, s, Vh = LA.svd(A, full_matrices=False)
print(f"{U.shape=}, {s.shape=}, {Vh.shape=}")
print(f"{np.allclose(A, U @ np.diag(s) @ Vh)=}")

print("\nTransposed matrix:")
B = A.transpose()
U, s, Vh = LA.svd(B, full_matrices=False)
print(f"{U.shape=}, {s.shape=}, {Vh.shape=}")
print(f"{np.allclose(B, U @ np.diag(s) @ Vh)=}")
```

```
U.shape=(2, 2), s.shape=(2,), Vh.shape=(3, 3)
U=array([[-0.70710678, -0.70710678],
       [-0.70710678,  0.70710678]])
U @ U.transpose()=array([[1.00000000e+00, 1.53686518e-16],
       [1.53686518e-16, 1.00000000e+00]])

np.diag(s)=array([[5., 0.],
       [0., 3.]])

Vh=array([[-7.07106781e-01, -7.07106781e-01, -6.47932334e-17],
       [-2.35702260e-01,  2.35702260e-01, -9.42809042e-01],
       [-6.66666667e-01,  6.66666667e-01,  3.33333333e-01]])
Vh.transpose() @ Vh=array([[ 1.00000000e+00,  1.35693925e-16,  1.32609972e-16],
       [ 1.35693925e-16,  1.00000000e+00, -4.93432455e-17],
       [ 1.32609972e-16, -4.93432455e-17,  1.00000000e+00]])
```

```
np.allclose(A, A1)=True
A-A1=array([[-8.88178420e-16, -4.44089210e-16, -8.88178420e-16],
       [-4.44089210e-16,  0.00000000e+00, -6.66133815e-16]])

A-A2=array([[-8.88178420e-16, -4.44089210e-16, -8.88178420e-16],
       [-4.44089210e-16,  0.00000000e+00, -6.66133815e-16]])


Thin SVD:
U.shape=(2, 2), s.shape=(2,), Vh.shape=(2, 3)
np.allclose(A, U @ np.diag(s) @ Vh)=True

Transposed matrix:
U.shape=(3, 2), s.shape=(2,), Vh.shape=(2, 2)
np.allclose(B, U @ np.diag(s) @ Vh)=True
```

5.1 Low rank approximation

Again, we can cut of the reconstruction at a certain point and create an approximation. More formally this is defined in the next definition.

Definition 5.2 (Low-Rank Approximation)**.** If $A \in \mathbb{R}^{m \times n}$ and has the SVD $A = U\Sigma V^{\mathsf{T}}$ than

$$A_k = U(:, 1:k)\,\Sigma(1:k, 1:k)\,V^{\mathsf{T}}(1:k, :)$$

is the *optimal* low-rank approximation of A with rank k. This is often called the *truncated SVD*.
(See Golub and Van Loan 2013, Corollary 2.4.7 p. 79)

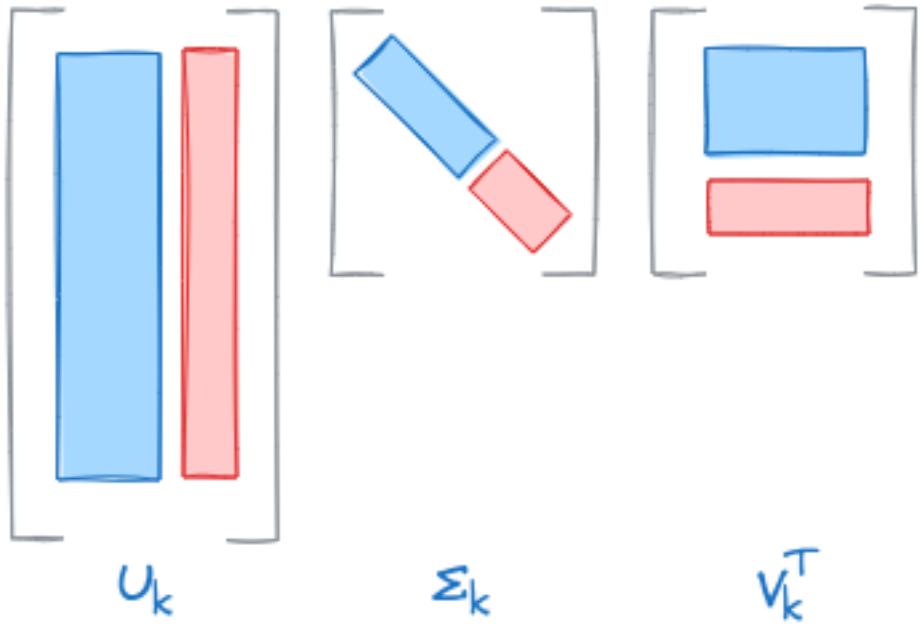


Figure 5.3: Truncated SVD

We can use this for image compression.

```
import matplotlib.pyplot as plt
import imageio.v3 as iio
import numpy as np
import numpy.linalg as LA
%config InlineBackend.figure_formats = ['svg']

im = np.asarray(iio.imread(
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[..., :3], [0.2989, 0.5870, 0.1140])

im_gray = rgb2gray(im)
im_scale = im_gray[1500:3001, 1500:3001] / 255

U, s, Vh = LA.svd(im_scale, full_matrices=False)

rec = [1/1000, 10/100, 25/100, 50/100, 75/100]
```

```
plt.figure()
plt.plot(s)
plt.yscale("log")
plt.ylabel(r"$|\sigma_i|$")
plt.xlabel("index")
plt.gca().set_aspect(5e1)

for p in rec:
    plt.figure()
    r = int(np.ceil(len(s) * p))
    A_r = U[:, :r] @ np.diag(s[:r]) @ Vh[:r, :]
    plt.imshow(A_r, cmap=plt.get_cmap("gray"))
    plt.gca().set_axis_off()

plt.figure()
plt.imshow(im_scale, cmap=plt.get_cmap("gray"))
plt.gca().set_axis_off()
plt.show()
```

> **Note**
>
> If we compare this to Figure 4.4 we can see that we get a much better result for smaller r. Let us have a look why.

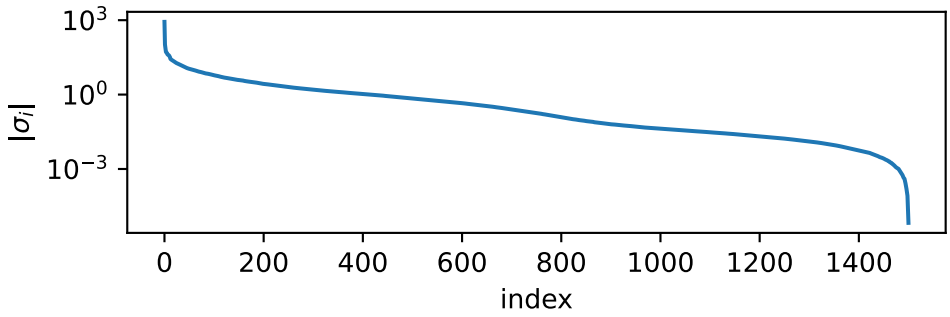
As the matrices U and V are orthogonal, they also define a basis of the corresponding (sub) vector spaces. As mentioned before, the SVD automatically selects these and they are optimal.

Consequently, the matrices U and V can be understood as reflecting patterns in the image. We can think of the columns of U and V as the vertical respectively horizontal patterns of A.

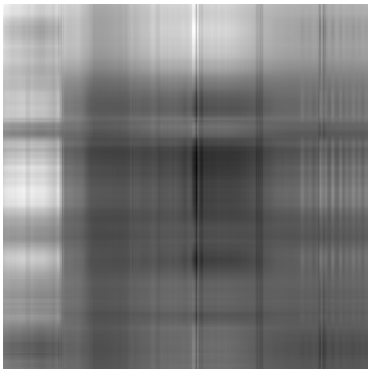
We can illustrate this by looking at the *modes* of our decomposition

$$M_k = U(:,k)V^{\mathsf{T}}(k,:).$$

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as LA
%config InlineBackend.figure_formats = ['svg']

rec = [0, 1, 2, 3, 4, 5]
```



(a) Singular values from largest to smallest



(b) 0.1% of eigenvalue

(c) 10% of eigenvalue

(d) 25% of eigenvalue

(e) 50% of eigenvalue

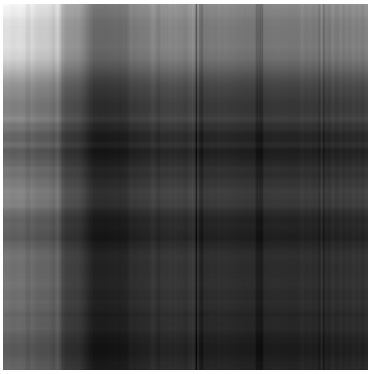
(f) 75% of eigenvalue

(g) original image

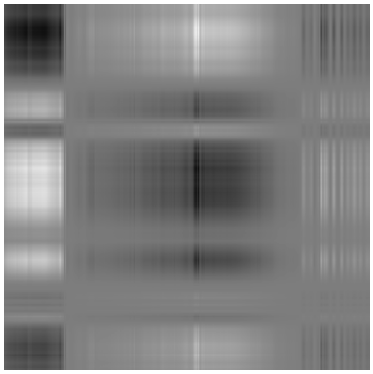
Figure 5.4: Image of MCI I and the reconstruction with reduced rank matrices.

```
for r in rec:
    plt.figure()
    M_r = np.outer(U[:, r], Vh[r, :])
    plt.imshow(M_r, cmap=plt.get_cmap("gray"))
    plt.gca().set_axis_off()
```

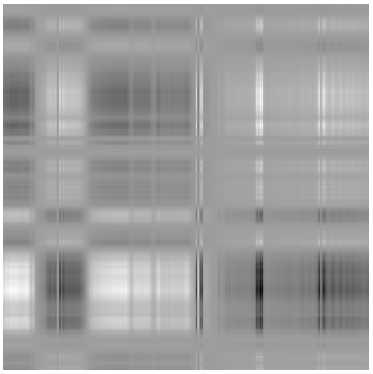
```
plt.show()
```



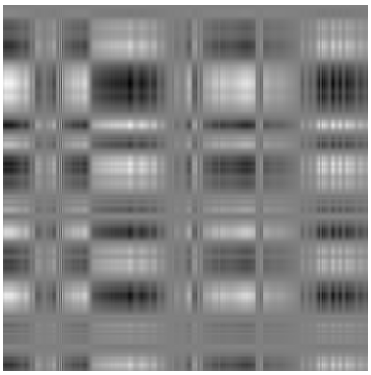
(a) First mode



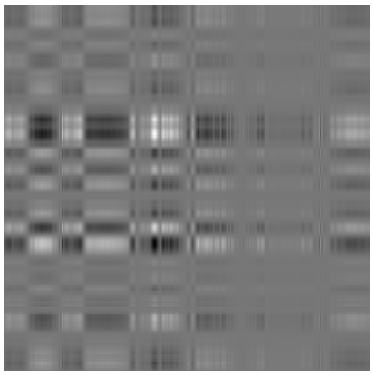
(b) Second mode



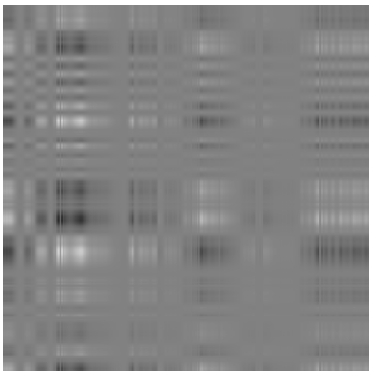
(c) Third mode



(d) Fourth mode



(e) Fifth mode



(f) Sixth mode

Figure 5.5: Modes of the SVD decomposition of the MCI I image.

> **Note**
>
> The big advantage here is, that the selection is optimal. A disadvantage is that the need to store the basis separately and this increases the necessary storage. We will see in later sections about *wavelets* and *Fourier* decomposition how a common basis can be used to reduce the storage by still keeping good reconstructive properties.

5.2 Principal Component Analysis

On of the most important applications of SVD is in the stable computation of the so called *principal component analysis* (PCA). It is a common technique in data exploration, analysis, visualization, and preprocessing.

The main idea of PCM is to transform the data in such a way that the main directions (principal components) capture the largest variation. In short we perform a change of the basis, see Definition 1.8.

Let us investigate this in terms of a (artificial) data set.

> **!** Important
>
> This example is adapted from (Brunton and Kutz 2022, Example: Noisy Gaussian Data, pp. 25-27).

We generate a noisy cloud (see Figure 5.6) that consists of 10000 points in 2D, generated from a normal distribution with zero mean and unit variance. The data is than:

1. scaled by 2 in the first direction and by $\frac{1}{2}$ in second,
2. rotated by $\frac{\pi}{3}$
3. translation in the direction $[2\ 1]^{\mathsf{T}}$.

The resulting matrix X is a long and skinny matrix with each measurement (or experiment) stacked next to each other. This means, each column represents a new set, e.g. a time step, and each row corresponds to the same sensor.

> **!** Important
>
> The following code is a slight adaptation (for nicer presentation in these notes) of the (Brunton and Kutz 2022, Code 1.4) also see notebook on github.

```
import matplotlib.pyplot as plt
import numpy as np
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)        # Make sure to stay reproducible

xC = np.array([2, 1])       # Center of data (mean)
sig = np.array([2, 0.5])    # Principal axes

theta = np.pi / 3             # Rotate cloud by pi/3

R = np.array([[np.cos(theta), -np.sin(theta)],     # Rotation matrix
```

```
              [np.sin(theta), np.cos(theta)]])

nPoints = 10000            # Create 10,000 points
X = R @ np.diag(sig) @ np.random.randn(2, nPoints) + \
        np.diag(xC) @ np.ones((2, nPoints))

fig = plt.figure()
ax1 = fig.add_subplot(121)
ax1.plot(X[0, :], X[1, :], ".", color="k")
ax1.grid()
ax1.set_aspect("equal")
plt.xlim((-6, 8))
plt.ylim((-6, 8))

## f_ch01_ex03_1b

Xavg = np.mean(X, axis=1)                  # Compute mean
B = X - np.tile(Xavg, (nPoints, 1)).T       # Mean-subtracted data

# Find principal components (SVD)
U, S, VT = np.linalg.svd(B, full_matrices=False)
S = S / np.sqrt(nPoints - 1)

ax2 = fig.add_subplot(122)
ax2.plot(X[0, :], X[1, :], ".", color="k")   # Plot data to overlay PCA
ax2.grid()
ax2.set_aspect("equal")
plt.xlim((-6, 8))
plt.ylim((-6, 8))

theta = 2 * np.pi * np.arange(0, 1, 0.01)

# 1-std confidence interval
Xstd = U @ np.diag(S) @ np.array([np.cos(theta), np.sin(theta)])

ax2.plot(Xavg[0] + Xstd[0, :], Xavg[1] + Xstd[1, :],
        "-", color="r", linewidth=3)
ax2.plot(Xavg[0] + 2 * Xstd[0, :], Xavg[1] + 2 * Xstd[1, :],
        "-", color="r", linewidth=3)
ax2.plot(Xavg[0] + 3 * Xstd[0, :], Xavg[1] + 3 * Xstd[1, :],
        "-", color="r", linewidth=3)

# Plot principal components U[:,0]S[0] and U[:,1]S[1]
```

```
ax2.plot(np.array([Xavg[0], Xavg[0] + U[0,0] * S[0]]),
         np.array([Xavg[1], Xavg[1] + U[1,0] * S[0]]),
         "-", color="cyan", linewidth=5)
ax2.plot(np.array([Xavg[0], Xavg[0] + U[0,1] * S[1]]),
         np.array([Xavg[1], Xavg[1] + U[1,1] * S[1]]),
         "-", color="cyan", linewidth=5)
```

```
plt.show()
```

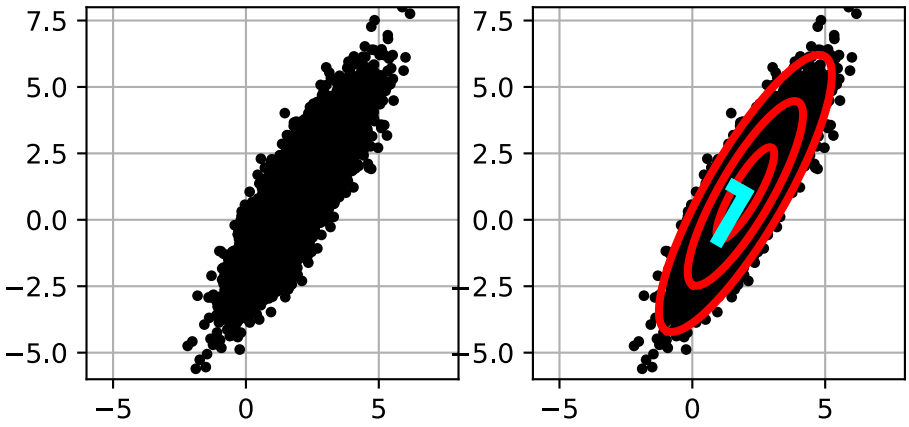


Figure 5.6: Principal components of the mean-subtracted Gaussian data on the left as, as well as the first three standard deviation ellisoids and the two scaled left singular vectors.

5.2.1 Computation

For the computation we follow the outline given in (Brunton and Kutz 2022, chap. 1.5). First we need to center our matrix X according to the mean per feature, in our case per row.

$$\overline{x}_j = \frac{1}{n} \sum_{i=1}^{n} X_{ij}$$

and our *mean matrix* is the outer product with the one vector

$$\overline{X} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \overline{x}$$

which can be used to compute the centred matrix $B = X - \overline{X}$.

The PCA is the eigendecomposition of the covariance matrix

$$C = \frac{1}{n-1} B^\mathsf{T} B \tag{5.2}$$

> **Note**
>
> The normalization factor of $n-1$ in Equation 5.2 an not n is called *Bassel's correction* and compensates for the bias in the estimation of the population variance.

As C is symmetric and positive semi-definite, therefore it has non-negative real eigenvalues and the matrix V of the eigendecomposition satisfies $V^{-1} = V^\mathsf{T}$ (i.e. it is orthogonal Definition 1.10). The principal components are the eigenvectors and the eigenvalue are the variance along these components.

If we instead compute the SVD of $B = U\Sigma V^\mathsf{T}$ we get

$$C = \frac{1}{n-1} B^\mathsf{T} B = \frac{1}{n-1} V\Sigma V^\mathsf{T} = \frac{1}{n-1} V(\Lambda^\mathsf{T}\Lambda)V^\mathsf{T}$$

leading to a way of computing the principal components in a robust way as

$$\lambda_k = \frac{\sigma_k^2}{n-1}.$$

> **Tip**
>
> If the sensor ranges of our matrix are very different in magnitude the correlation matrix is scaled by the *row wise standard deviation* of B similar as for the mean.

In our example we get our scaled $\sigma_1 = 1.988 \approx 2$ and $\sigma_2 = 0.496 \approx \frac{1}{2}$. These results recover our given parameters very well. Additionally we can see that our rotation matrix is closely matched by U (up to signs) from our SVD:

$$R_{\frac{\pi}{3}} = \begin{bmatrix} 0.5 & 0.866 \\ -0.866 & 0.5 \end{bmatrix}, \quad U = \begin{bmatrix} -0.501 & -0.865 \\ -0.865 & 0.501 \end{bmatrix}$$

5.2.2 Example Eigenfaces

We combine SVD/PCA in a illustrative example called *eigenfaces* as introduced in (Brunton and Kutz 2022, Sec 1.6, pp. 28-34).

The idea is to apply the PCA techniques to a large set of faces to extract the dominate correlations between the images and create a *face basis* that can be used to represent an image

in these coordinates. For example you can reconstruct a face in this space by projecting onto the eigen vectors or it can be used for face recognition as similar faces usually cluster under this projection.

The images are taken from the Yale Face Dataset B, in our case we use a GitHub that provides Julia Pluto notebooks for Chapter 1 to 4 of Brunton and Kutz (2022).

Our training set, so to speak, consists of the first 36 people in the dataset. We compute the *average face* and subtract it from our dataset to get our matrix B. From here a SVD provides us with our basis U. To test our basis we use individual 37 and a portion of the image of the MCI Headquarter (to see how well it performs on objects). For this we use the projection

$$\tilde{x} = U_r U_r^\mathsf{T} x.$$

If we split this up, we first project onto our found patterns (encode) and than reconstruct from them (decode).

> **Note**
>
> We can understand this as *encoding* and *decoding* our test image, which is the general setup of an *autoencoder* (a topic for another lecture).
> The correlation coefficients $x_r = U_r^\mathsf{T} x$ might reveal patterns for different x. In the case of faces, we can use this for face recognition, i.e. if the coefficients of x_r are in the same cluster as other images, they are probably from the same person.

> **Important**
>
> The following code is an adaptation of the (Brunton and Kutz 2022, Code 1.7 and 1.9).

```
import numpy as np
import numpy.linalg as LA
import scipy
import requests
import io
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

response = requests.get(
    "https://github.com/frankhuettner/Data_Driven_Science_Julia_Demos"
    "/raw/refs/heads/main/DATA/allFaces.mat")

data = scipy.io.loadmat(io.BytesIO(response.content))
faces = data["faces"]
m = int(data["m"][0,0])
```

```
n = int(data["n"][0,0])
nfaces = np.ndarray.flatten(data['nfaces'])

trainingFaces = faces[:, : np.sum(nfaces[:36])]
avgFace = np.mean(trainingFaces, axis=1)

B = trainingFaces - np.tile(avgFace, (trainingFaces.shape[1], 1)).T
U, _, _ = LA.svd(B, 'econ')

testFace = faces[:, np.sum(nfaces[:36])]
testFaceMS = testFace - avgFace
rec = [25, 100, 400]

fig = plt.figure()
axs = []
axs.append(fig.add_subplot(2, 4, 1))
axs.append(fig.add_subplot(2, 4, 2))
axs.append(fig.add_subplot(2, 4, 3))
axs.append(fig.add_subplot(2, 4, 4))
axs.append(fig.add_subplot(2, 4, 5))
axs.append(fig.add_subplot(2, 4, 6))
axs.append(fig.add_subplot(2, 4, 7))
axs.append(fig.add_subplot(2, 4, 8))

for i, p in enumerate(rec):
    r = p
    A_r = avgFace + U[:, :r] @ U[:, :r].T @ testFaceMS
    axs[i].imshow(np.reshape(A_r, (m, n)).T, cmap=plt.get_cmap("gray"))
    axs[i].set_axis_off()
    axs[i].set_title(f"${r=}$")

axs[3].imshow(np.reshape(testFace, (m, n)).T, cmap=plt.get_cmap("gray"))
axs[3].set_axis_off()
axs[3].set_title(f"Original image")

shift = 1500
testFaceMS = np.reshape(im_gray[shift:shift+n, shift:shift+m].T, n*m) - \
             avgFace
rec = [100, 400, 1600]
for i, p in enumerate(rec):
    r = p
    A_r = avgFace + U[:, :r] @ U[:, :r].T @ testFaceMS
    axs[4 + i].imshow(np.reshape(A_r, (m, n)).T, cmap=plt.get_cmap("gray"))
```

```
    axs[4 + i].set_axis_off()
    axs[4 + i].set_title(f"${r=}$")
axs[7].imshow(im_gray[shift:shift+n, shift:shift+m], cmap=plt.get_cmap("gray"))
axs[7].set_axis_off()
axs[7].set_title(f"Original image")
plt.show()
```

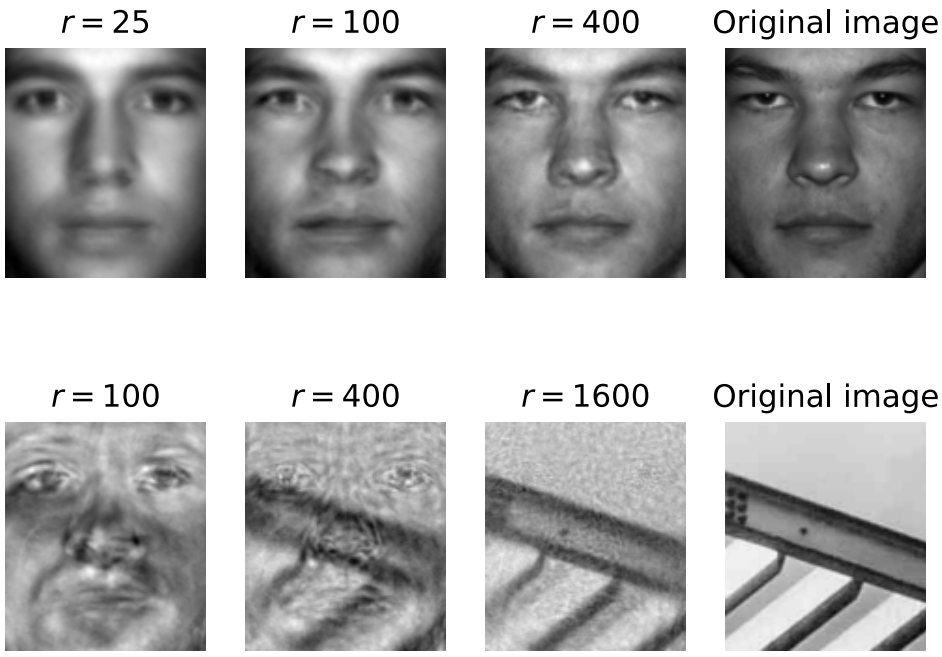


Figure 5.7: Approximate reconstruction of a test face and an object using the eigenfaces basis for different order r.

> **Note**
>
> Due to resource limitations the above computation can not be done for each build. We try to make sure that the code matches the image but if something is different if you try it yourself we apologise for that.

5.3 Further applications of the SVD

There are many more applications of the SVD but we want to highlight some regarding systems of linear equations,

$$Ax = b \tag{5.3}$$

where the matrix A, as well as the vector b is known an x is unknown.

Depending on the structure of A and the specific b we have no, one, or infinitely many solutions. For now the interesting case is where A is rectangular and therefore we have either an

- under-determined system $m \ll n$, so more unknowns than equations,
- over-determined system $m \gg n$, so more equations than unknowns.

For the second case (more equations than unknowns) we often switch to solving the optimization problem that minimizes

$$\|Ax - b\|_2^2. \tag{5.4}$$

This is called the *least square* solution. The least square solution will also minimize $\|Ax - b\|_2$. For an under-determined system we might seek the solution which minimizes $\|x\|_2$ called the *minimum norm* solution.

If we us the SVD decomposition for $A = U\Sigma V^\mathsf{T}$ we can define the following

Definition 5.3 (Pseudo-inverse)**.** We define the matrix $A^\dagger \in \mathbb{R}^{m\times n}$ by $A^\dagger = V\Sigma^\dagger U^\mathsf{T}$ where

$$\Sigma^\dagger = \operatorname{diag}\left(\frac{1}{\sigma_1}.\frac{1}{\sigma_2}, \dots, \frac{1}{\sigma_r}, 0, \dots, 0\right) \in \mathbb{R}^{m\times n}, \quad r = \operatorname{rank}(A).$$

The matrix $A^\dagger$ is often called the *Moore-Penrose left pseudo-inverse* as it fulfils the *Moore-Penrose conditions* conditions. It is also the matrix to provides the minimal Frobenius norm solution to

$$\min_{X\in\mathbb{R}^{m\times n}} \|AX - I_n\|_F.$$

(Compare Golub and Van Loan 2013, 290)

If we only use the truncated version, i.e. where we only use non-zero singular values, we can use it to find good solutions to Equation 5.4.

In `numpy` it can be computed by `numpy.linalg.pinv`.

Definition 5.4 (Condition number)**.** The condition number of a matrix provides a measure how sensitive the solution of Equation 5.3 is to perturbations in A and b. For a square matrix A the condition number is defined as

$$\kappa(A) = \|A\| \left\|A^{-1}\right\|,$$

for an appropriate underlying norm. For the 2-norm κ_2 is

$$\kappa_2(A) = \|A\|_2 \left\|A^{-1}\right\|_2 = \frac{\sigma_{max}}{\sigma_{min}}.$$

To get a better idea on what this means think of it in this way. For the perturbed linear system

$$A(x + \epsilon_x) = b + \epsilon_b,$$

we can outline the worst case, where ϵ_x aligns with the singular vector of the largest singular vector and x with the smallest singular value, i.e.

$$A(x + \epsilon_x) = \sigma_{min} x + \sigma_{max} \epsilon_x.$$

Consequently, the output signal-to-noise $\|b\|/\|\epsilon_b$ is equivalent with the input signal-to-noise $\|x\|/\|\epsilon_x$ and the factor between those two is $\kappa_2(A)$.
In this sense κ_2 can be extended for more general matrices.
(Compare Golub and Van Loan 2013, 87; and Brunton and Kutz 2022, 18–19)

5.3.1 Linear regression with SVD

Before we go into more details about regression in the next section we give a brief outlook in terms of how to solve such a problem with SVD.

! Important

This example is adapted from (Brunton and Kutz 2022, Example: One-Dimensional Linear Regression, Example: Cement Heat Generation Data, pp. 19-22).

5.3.1.1 Linear Regression (see Brunton and Kutz 2022, 19–21)

First we just take a linear correlation that we augment with some Gaußian Noise. So our matrix A is simple a vector with our x-coordinates and b is the augmented image under our linear correlation.

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad \Leftrightarrow \quad U\Sigma V^\mathsf{T} x = b \quad \Leftrightarrow \quad x = A^\dagger b$$

For this example $\Sigma = \|a\|_2$, $V = 1$, and $U = \frac{a}{\|a\|_2^2}$. This is basically just the projection of b

along our basis a and this is

$$x = \frac{a^\mathsf{T} b}{a^\mathsf{T} a}.$$

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as LA
%config InlineBackend.figure_formats = ['svg']
np.random.seed(6020)       # Make sure to stay reproducible

k = 3
A = np.arange(-2, 2, 0.25).reshape(-1, 1)
b = k*A + np.random.randn(*A.shape) * 0.5

U, s, VT = LA.svd(A, full_matrices=False)

x = VT.T @ np.diag(1/s) @ U.T @ b

plt.plot(A, k*A, color="k", label="Target")
plt.plot(A, b, 'x', color="r", label="Noisy data")
plt.plot(A, A*x, '--', color="b", label="Regression line")
plt.legend()
plt.xlabel("$a$")
plt.ylabel("$b$")
plt.show()
```

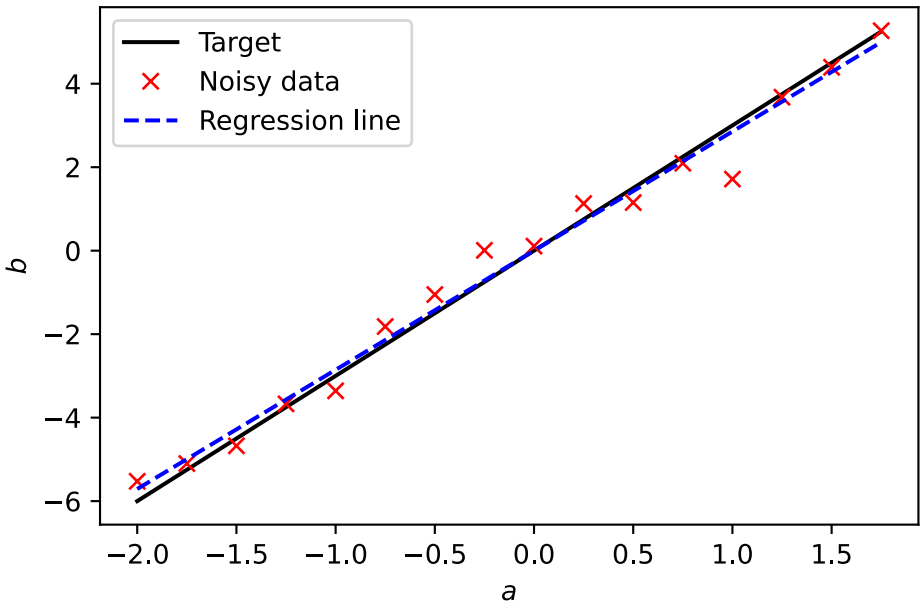


Figure 5.8: Linear regression with SVD.

Our reconstructed unknown $x = 2.855$ and is a reasonable good match for $k = 3.0$

5.3.1.2 Multi-Linear Regression (see Brunton and Kutz 2022, 21–23)

The second example is based on the *Portland Cement Data* build in with MATLAB. In Python we again use the dataset provided on GitHub. The data set contains the heat generation during the hardening of 12 cement mixtures comprised of 4 basic ingredients, i.e. $A \in \mathbb{R}^{13\times4}$. The aim is to determine the weights x that relate the proportion of the ingredients to the heat generation in the mixture.

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as LA
import requests
import io
%config InlineBackend.figure_formats = ['svg']

# Transform the content of the file into a numpy.ndarray
response = requests.get(
    "https://github.com/frankhuettner/Data_Driven_Science_Julia_Demos"
    "/raw/refs/heads/main/DATA/hald_ingredients.csv")
```

```
# Transform the content of the file into a numpy.ndarray
A = np.genfromtxt(io.BytesIO(response.content), delimiter=",")

response = requests.get(
    "https://github.com/frankhuettner/Data_Driven_Science_Julia_Demos"
    "/raw/refs/heads/main/DATA/hald_heat.csv")
b = np.genfromtxt(io.BytesIO(response.content), delimiter=",")

U, s, VT = LA.svd(A, full_matrices=False)

x = VT.T @ np.diag(1/s) @ U.T @ b

plt.plot(b, color="k", label="Target - Heat data")
plt.plot(A@x, '--', color="b", label="Regression")
plt.legend()
plt.xlabel("mixture")
plt.ylabel("Heat[cal/g]")
plt.show()
```

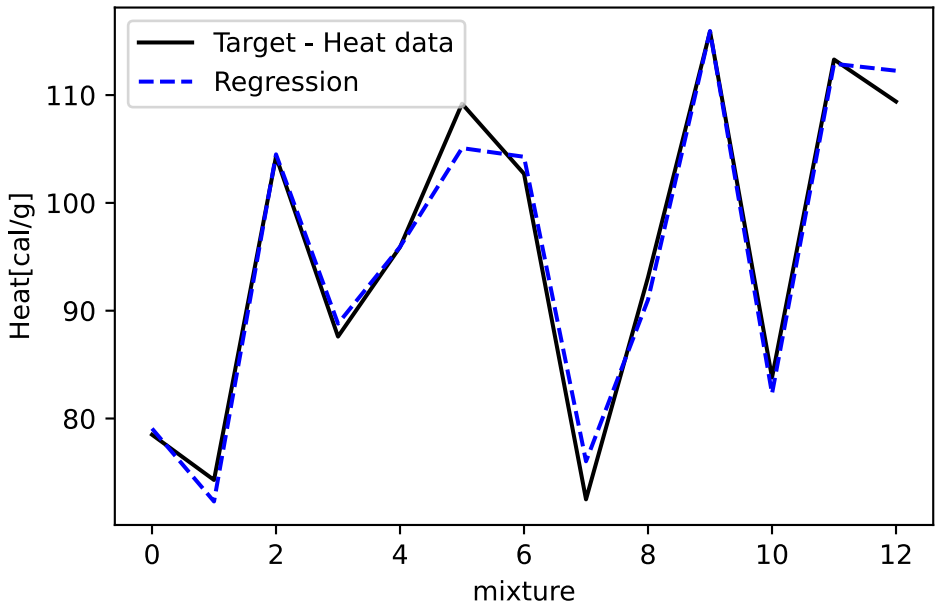


Figure 5.9: Estimate for hardening in cement mixtures.

This concludes our investigation of matrix decompositions, we will investigate further decompositions of signals later, but for now we dive deeper into regression.

Part III

Regression analysis

In general, *regression analysis* can be understood as a set of tools that is used to estimate or establish a relationship between a dependent variable Y (also called outcome or response variable, label) and the independent variable X (also called regressor, predictors, covariates, explanatory variable or feature). If we add a regression function f and some unknown parameters c to the mix the problem can be written mathematically as

$$Y = f(X, c) \tag{5.5}$$

where c is found by optimizing for *a good fit* of f to the data.

We split up the discussion along the well known topics:

- Chapter 6
- Chapter 7
  - Section 7.1
- Chapter 8
  - Section 8.2

Parts of this section are based on (Brunton and Kutz 2022, sec. 4).

6 Linear Regression

The general idea of linear regression is to approximate a *point cloud* by a mathematical function, this is often called *curve fitting* and it is closely related to optimization techniques, (compare Brunton and Kutz 2022, sec. 4.1).

Let us assume that we want to establish a relationship between our n observations with m independent variables. In order to get the notation down correctly we should describe the variables more closely.

We have n observations (the k-th representation is denoted by y_k) resulting in a vector:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}.$$

We have m independent variables and consequently for each of this n observations this results in a matrix

$$X = \begin{bmatrix} X_{1-} \\ X_{2-} \\ X_{3-} \\ \vdots \\ X_{n-} \end{bmatrix} = [X_{-1}, \dots, X_{-m}],$$

where the second representation more closely fits to our idea of the m independent variables.

Let us model the relation in a linear fashion with the parameters c as

$$y_k \overset{!}{=} X_{k1}c_1 + X_{k2}c_2 + \cdots + X_{km}c_m, \quad 1 \leq k \leq n,$$

or in short

$$y = Xc = f(X, c).$$

> **Note**
>
> In literature the parameters are most of the time called β but for the sake of keeping notation consistent we opt for c (and not b to not confuse it with $Ax = b$).

> **Tip**
>
> Usually the convention is to define $X_{-1} = 1$ as constant 1 at each position. The corresponding c_1 is called the *intercept*. We even find this convention if theory tells us c_1 is 0 as some procedures assume this term to exist.

> **Note**
>
> We talk of a linear model, as long as it is linear in the parameters c. It might happen that regressors have a non linear relation.

Let us assume for a moment we already have a realisation of f, i.e. we know the parameters c, we get the error of the approximation as

$$\mathbf{e} = y - f(X, c) \quad \Leftrightarrow \quad \mathbf{e}_k = y_k - f(X_{k-}, c). \tag{6.1}$$

There are various possibilities to select the metric for minimizing $\mathbf{e}$ and therefore characterizing the quality of the *fit*. This is done by selecting the underlying norm. Most commonly we use the 1-norm, the 2-norm and the ∞-norm, i.e.

$$E_1(f) = \frac{1}{n}\sum_{k=1}^{n} |f(X_{k-}, c) - y_k|, \tag{6.2}$$

for the 1-norm or mean absolute error,

$$E_2(f) = \sqrt{\frac{1}{n}\sum_{k=1}^{n} |f(X_{k-}, c) - y_k|^2}, \tag{6.3}$$

for the 2-norm or least-square error,

$$E_\infty(f) = \max_k |f(X_{k-}, c) - y_k| \tag{6.4}$$

for the ∞-norm or maximum error. Of course p-norms work as well

$$E_p(f) = \left(\frac{1}{n}\sum_{k=1}^{n} |f(X_{k-}, c) - y_k|^p\right)^{1/p}.$$

If we go back and we want to solve Equation 6.1 for c we have different realisations, depending on the selected norm.

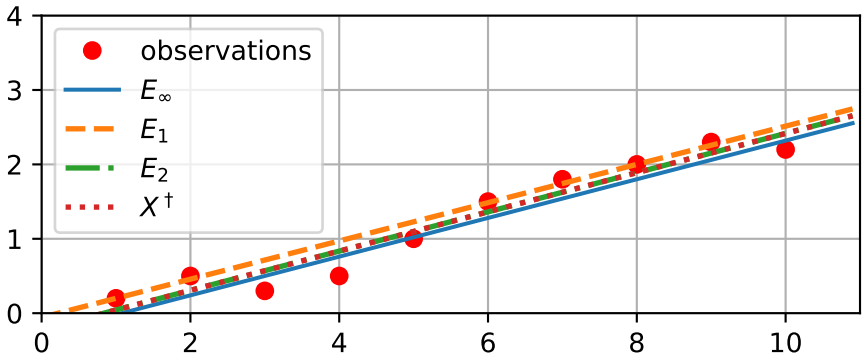
To illustrate this we use the example from (Brunton and Kutz 2022, 136–67).

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.optimize
%config InlineBackend.figure_formats = ["svg"]

# Function definitions
def fit1(x0, t):
    x, y = t
    return 1/len(y)*np.max(np.abs(x0[0] * x + x0[1] - y))
def fit2(x0, t):
    x, y = t
    return 1/len(y)*np.sum(np.abs(x0[0] * x + x0[1] - y))
def fit3(x0, t):
    x, y = t
    return 1/len(y)*np.sum(np.power(np.abs(x0[0] * x + x0[1] - y), 2))

# The data
x = np.arange(1, 11)
y = np.array([0.2, 0.5, 0.3, 0.5, 1.0, 1.5, 1.8, 2.0, 2.3, 2.2])
z = np.array([0.2, 0.5, 0.3, 3.5, 1.0, 1.5, 1.8, 2.0, 2.3, 2.2])
t = (x, y)
t2 = (x, z)

x0 = np.array([1, 1])

p1 = scipy.optimize.fmin(fit1, x0, args=(t,), disp=False)
p2 = scipy.optimize.fmin(fit2, x0, args=(t,), disp=False)
p3 = scipy.optimize.fmin(fit3, x0, args=(t,), disp=False)

p12 = scipy.optimize.fmin(fit1, x0, args=(t2,), disp=False)
p22 = scipy.optimize.fmin(fit2, x0, args=(t2,), disp=False)
p32 = scipy.optimize.fmin(fit3, x0, args=(t2,), disp=False)

xf = np.arange(0, 11, 0.1)
y1 = np.polyval(p1, xf)
y2 = np.polyval(p2, xf)
y3 = np.polyval(p3, xf)

y12 = np.polyval(p12, xf)
y22 = np.polyval(p22, xf)
y32 = np.polyval(p32, xf)

X = np.array([x, np.ones(x.shape)]).T
```

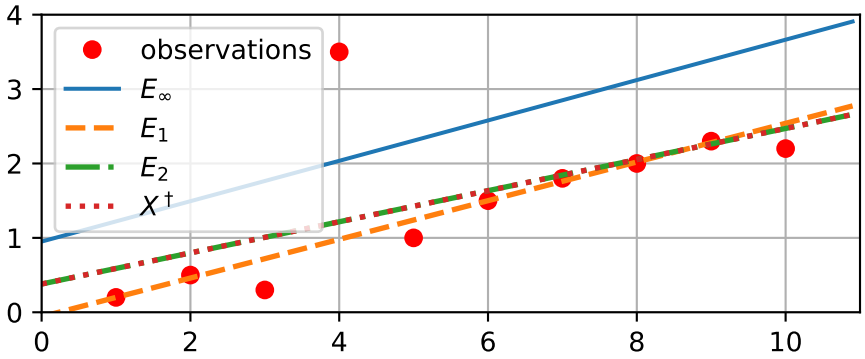
```
p4 = np.linalg.pinv(X) @ y
p42 = np.linalg.pinv(X) @ z

y4 = np.polyval(p4, xf)
y42 = np.polyval(p42, xf)

fig = plt.figure()

plt.plot(x, y, "o", color="r", label="observations")
plt.plot(xf, y1, label=r"$E_\infty$")
plt.plot(xf, y2, "--", linewidth=2, label=r"$E_1$")
plt.plot(xf, y3, "-.", linewidth=2, label=r"$E_2$")
plt.plot(xf, y4, ":", linewidth=2, label=r"$X^\dagger$")
plt.ylim(0, 4)
plt.xlim(0, 11)
plt.legend(loc="upper left")
plt.gca().set_aspect(1)
plt.grid(visible=True)
plt.show()

plt.plot(x, z, "o", color="r", label="observations")
plt.plot(xf, y12, label=r"$E_\infty$")
plt.plot(xf, y22, "--", linewidth=2, label=r"$E_1$")
plt.plot(xf, y32, "-.", linewidth=2, label=r"$E_2$")
plt.plot(xf, y42, ":", linewidth=2, label=r"$X^\dagger$")
plt.ylim(0, 4)
plt.xlim(0, 11)
plt.legend(loc="upper left")
plt.gca().set_aspect(1)
plt.grid(visible=True)
plt.show()
```

If we use Equation 6.3 the regression fit is called the *least square fit*.

> **Tip**
>
> For the optimization algorithm to work fast it is often useful to rewrite the above error functions, e.g. the square-root in E_2 and the absolute value are not required.



(a) Observations with no outliers.



(b) Observations with outliers.

Figure 6.1: Line fit with different norms. Top without outliers, bottom with one outlier.

6.1 Ordinary Least Square

It is worth looking into the least square solution

$$E_2(f) = \sqrt{\frac{1}{n}\sum_{k=1}^{n} | \underbrace{f(X_{k-}, c) - y_k}_{\mathbf{e}_k} |^2},$$

more closely. We can interpret it as the optimization problem

$$c = \underset{v}{\operatorname{argmin}} \|y - Xv\|_2$$

and with some linear algebra we get

$$\begin{aligned} c &= \underset{v}{\operatorname{argmin}} \|y - Xv\|_2, \\ &= \underset{v}{\operatorname{argmin}} \langle y - Xv, y - Xv \rangle, \\ &= \underset{v}{\operatorname{argmin}} (y - Xv)^\mathsf{T}(y - Xv), \\ &= \underset{v}{\operatorname{argmin}} y^\mathsf{T}y - y^\mathsf{T}Xv - v^\mathsf{T}X^\mathsf{T}y + v^\mathsf{T}X^\mathsf{T}Xv. \end{aligned}$$

In order to find a solution we compute the derivative with respect to v set it to 0 and simplify, i.e.

$$\frac{\mathrm{d}}{\mathrm{d}\, v} y^\mathsf{T}y - y^\mathsf{T}Xv - v^\mathsf{T}X^\mathsf{T}y + v^\mathsf{T}X^\mathsf{T}Xv = -2X^\mathsf{T}y + 2X^\mathsf{T}Xv \tag{6.5}$$

and

$$v = (X^\mathsf{T}X)^{-1}X^\mathsf{T}y \equiv X^\dagger y.$$

We recall, that $X^\dagger$ is called the Moore-Penrose pseudo-inverse, see Definition 5.3.

See Figure 6.1 for the result when using the pseudo-inverse.

6.1.1 Alternative computation

The pseudo-inverse provides us with the optimal solution but for large systems the computation can be inefficient, or more precisely, there are more efficient ways to get the same results.

Following (Brunton and Kutz 2022, 137–38) we can find an alternative for the above example in Figure 6.1.

We want to fit the data points (x_i, y_i) with the function $f(x) = c_2x + c_1$ resulting in the error

$$E_2(f) = \sqrt{\frac{1}{n}\sum_{k=1}^{n}(c_2x_k + c_1 - y_k)^2}.$$

A solution that minimizes the above equation also minimizes

$$E_2 = \sum_{k=1}^{n} (c_2 x_k + c_1 - y_k)^2$$

and we find the solution by partial differentiation

$$\frac{\mathrm{d}E_2}{\mathrm{d}\,c_1} = 0 \Leftrightarrow \sum_{k=1}^{n} 2(c_2 x_k + c_1 - y_k) = 0,$$

$$\frac{\mathrm{d}E_2}{\mathrm{d}\,c_2} = 0 \Leftrightarrow \sum_{k=1}^{n} 2(c_2 x_k + c_1 - y_k) x_k = 0,$$

and this results in the system

$$\begin{bmatrix} n & \sum_k x_k \\ \sum_k x_k & \sum_k x_k^2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \sum_k y_k \\ \sum_k x_k y_k \end{bmatrix}. \tag{6.6}$$

This ansatz can be extended to polynomials of degree k, where the result is always a $(k+1) \times (k+1)$ matrix.

6.2 Polynomial Regression

Polynomial regression, despite its name, is linear regression with a special function f where the relation is polynomial in $x = [a_1, \ldots, x_m]^\mathsf{T}$

$$y_k = x_k^0 + x_k^1 c_1 + x_k^2 c_2 + \cdots + x_k^m c_m, \quad 1 \leq k \leq n,$$

With the matrix form

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \tag{6.7}$$

> **Note**
>
> The matrix Equation 6.7 is called the Vandermonde matrix.

This can be solved in the same ways as the before with $X^\dagger$ or the direct system, but it should not as Equation 6.7 is badly conditioned. There are other methods like *divided differences*, Lagrange interpolation for this task.

Example 6.1 (Parameter estimation of a falling object)**.** Just because we deal with linear regression this does not means that the model needs to be linear too. As long as we are linear in the parameters c we can apply our findings, even for non linear *independent variables*.
To illustrate this, let us consider an object falling without aerodynamic drag, described by the differential equation

$$m\ddot{y}(t) = -mg,$$

for the gravitational constant g. Integration with respect to t results in

$$y(t) = y(0) + v(0)t - \frac{g}{2}t^2.$$

So we get

$$Xk- = \begin{bmatrix} 1 & t_k & -\frac{1}{2}t_k^2 \end{bmatrix}, \quad \text{and} \quad y = \begin{bmatrix} y^{(0)} \\ v^{(0)} \\ g \end{bmatrix}$$

or in the long form, for $t_{k+1} - t_k = 0.1$

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0.1 & -0.005 \\ 1 & 0.2 & -0.020 \\ 1 & 0.3 & -0.045 \\ 1 & 0.4 & -0.080 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

and we can, for example, estimate our unknowns $y^{(0)}$, $v^{(0)}$, and g by

$$c = X^{\dagger}y.$$

Example 6.2 (Polynomial regression)**.** In the following example we generate an artificial sample of $n = 100$ points resulting in the samples

$$y_k = \frac{1}{2}x_k^2 + x_k + 2 + \epsilon_k$$

where ϵ_k is a random number that simulates the error. We perform the interpolation with $X^{\dagger}$.

```
import matplotlib.pyplot as plt
import numpy as np
import warnings
warnings.simplefilter('ignore', np.exceptions.RankWarning)
```

```
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)

m = 100
x = 6 * np.random.rand(m) - 3
y = 1/2 * x ** 2 + x + 2 + np.random.randn(m)

X1 = np.vander(x, 2)
X2 = np.vander(x, 3)
X3 = np.vander(x, 16)

p1 = np.linalg.pinv(X1) @ y
p2 = np.linalg.pinv(X2) @ y
p3 = np.linalg.pinv(X3) @ y
p4 = np.polyfit(x, y, 300)

xf = np.arange(-3, 3, 0.1)
y1 = np.polyval(p1, xf)
y2 = np.polyval(p2, xf)
y3 = np.polyval(p3, xf)
y4 = np.polyval(p4, xf)

fig = plt.figure()
plt.plot(x, y, "o", color="r", label="observations", linewidth=3)
plt.plot(xf, y1, label=r"$m=1$")
plt.plot(xf, y2, label=r"$m=2$")
plt.plot(xf, y3, label=r"$m=16$")
plt.plot(xf, y4, label=r"$m=300$")

plt.ylim(0, 10)
plt.xlim(-3, 3)
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.legend(loc="upper left")
#plt.gca().set_aspect(0.25)
plt.grid(visible=True)
plt.show()
```

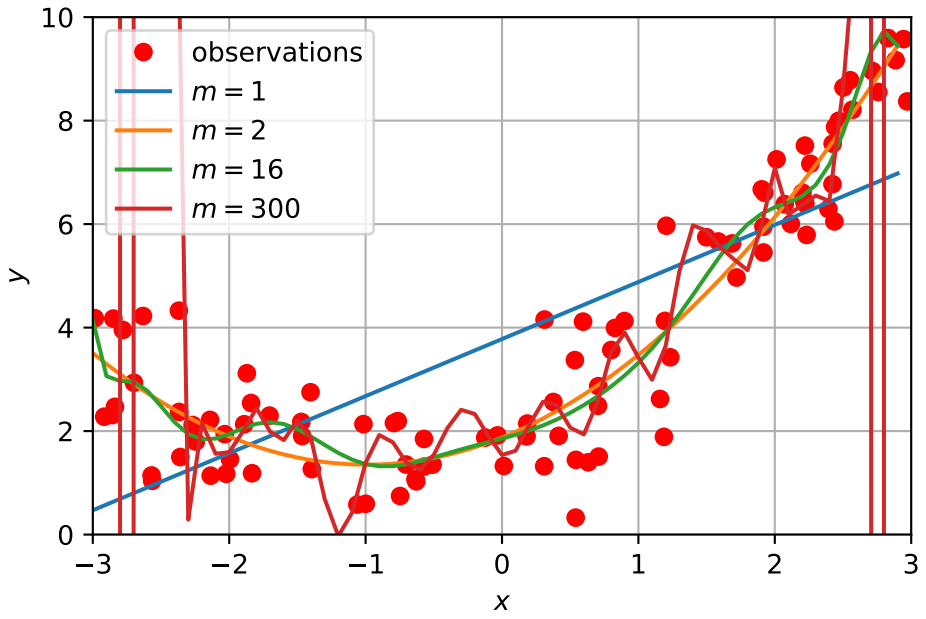


Figure 6.2: Fitting for different degrees of polynomial m

Caution

The condition of the Vandermonde matrix increases rapidly:

degree	$m = 2$	$m = 3$	$m = 5$	$m = 10$	$m = 15$	$m = 20$
κ_2	6.07e0	1.63e1	1.89e2	154e5	1.24e8	1.41e11

The result of the $m = 300$ is unstable and we can not compute it via $X^{\dagger}$.

As can be seen in Figure 6.2 we do not necessarily get a good result if we use a higher degree polynomial. This is especially true if we extrapolate and not interpolate.

6.3 Data Linearization

Quite often it is possible to linearize our model at hand. For example if we want to fit for

$$f(x, c) = c_2 \exp(c_1 x), \tag{6.8}$$

and use the same derivation as for Equation 6.6 we end up with the corresponding system as

$$c_2 \sum_k x_k \exp(2c_1 x_k) - \sum_k x_k y_k \exp(c_1 x_k) = 0,$$
$$c_2 \sum_k \exp(2c_1 x_k) - \sum_k y_k \exp(c_1 x_k) = 0.$$

This non-linear system can not be solved in a straight forward fashion but we can avoid it by linearization with the simple transformation

$$\begin{aligned} \hat{y} &= \ln(y), \\ \hat{x} &= x, \\ c_3 &= \ln c_2, \end{aligned}$$

and taking the natural logarithm of both sides of Equation 6.8 and simplifying

$$\ln y = \ln(c_2 \exp(c_1 x)) = \ln(c_2) + c_1 x.$$

Now all that is left to apply ln to the data y and solve the linear problem. In order to apply it to the original function the parameters transform needs to be reversed.

Example 6.3 (World population)**.** We take a look at the population growth were the data is kindly provided by Ritchie et al. (2023). Have a look at there excellent work on ourworldindata.org.

```
from owid.catalog import charts
import numpy as np
import scipy.optimize
import plotly.graph_objects as go
from plotly.subplots import make_subplots

df = charts.get_data(
    "https://ourworldindata.org/grapher/population?country=~OWID_WRL")
data = df[df["entities"] == "World"]
x = data["years"].to_numpy()
y = data["population"].to_numpy()
ylog = np.log(y)

def fit3(x0, t):
    x, y = t
    return np.sum(np.power(np.abs(x0[0] * x + x0[1] - y), 2))

start = [-np.inf, 0, 1700, 1900, 1980]
```

```
yest = []
```

```
for s in start:
    filter = x >= s
    t = (x[filter], ylog[filter])
    x0 = np.array([1, 1])
    b = scipy.optimize.fmin(fit3, x0, args=(t,), disp=False)
    yest.append(np.exp(b[1]) * np.exp(b[0] * x))

fig = go.Figure()
fig2 = go.Figure()
fig.add_trace(go.Scatter(mode="markers", x=data["years"],
                y=data["population"], name="data"))
fig2.add_trace(go.Scatter(mode="markers", x=data["years"],
                y=data["population"], name="data"))

for i, ye in enumerate(yest):
    fig.add_trace(go.Scatter(x=x, y=ye, name=f"fit from {start[i]}"))
    fig2.add_trace(go.Scatter(x=x, y=ye, name=f"fit from {start[i]}"))


fig.update_xaxes(title_text="year", range=[1700, 2023])
fig.update_yaxes(title_text="population")
fig.update_layout(legend=dict(
    yanchor="top",
    y=0.99,
    xanchor="left",
    x=0.01
))
fig.show()

fig2.update_xaxes(title_text="year", range=[1700, 2023])
fig2.update_yaxes(title_text="population", type="log", range=[8.5,10])
fig2.update_layout(legend=dict(
    yanchor="top",
    y=0.99,
    xanchor="left",
    x=0.01
))
fig2.show()
```

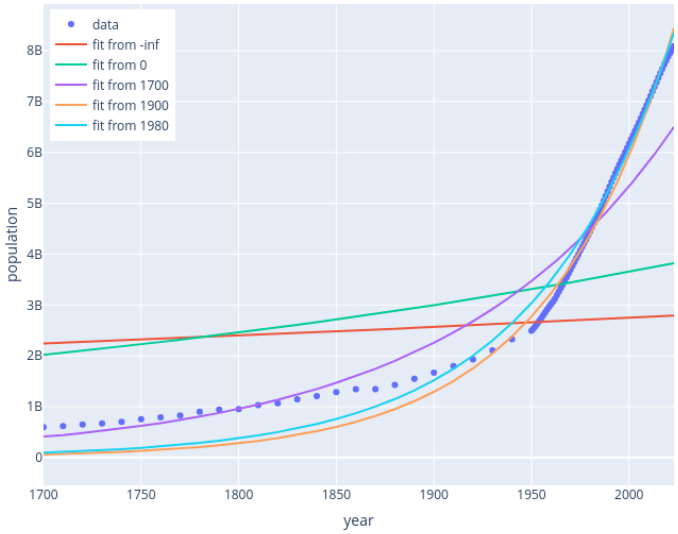


Figure 6.3: World population with regression lines normal scale.

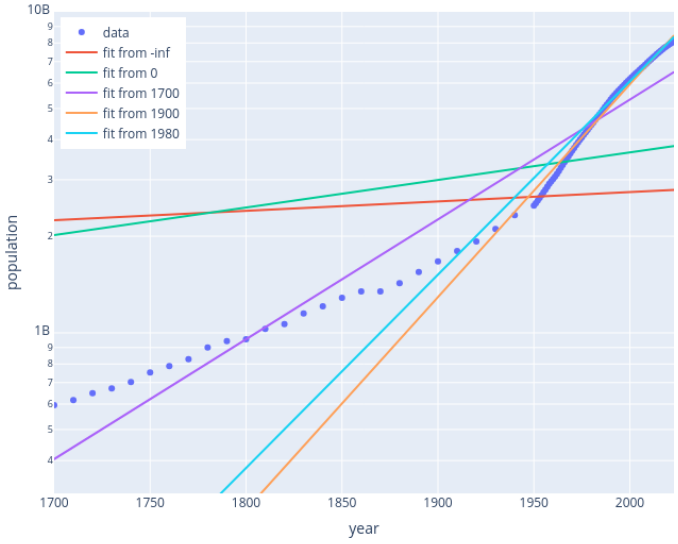


Figure 6.4: World population with regression lines logarithmic scale.

Next, we are going to look into actual non-linear regression.

7 Non-linear Regression

> **!** Important
>
> This section is mainly based on Brunton and Kutz (2022), Section 4.2.

We extend our theory of *curve fitting* to a non-linear function $f(X, c)$ with coefficients $c \in \mathbb{R}^m$ and our n X_{k-}. We assume that $m < n$.

If we define our root-mean-square error depending on c as

$$E_2(c) = \sum_{k=1}^{n} (f(X_{k-}, c) - y_k)^2$$

and we can minimize with respect to each c_j resulting in a $m \times m$ system

$$\sum_{k} (f(X_{k-}, c) - y_k) \frac{\partial f}{\partial c_j} = 0 \quad \text{for} \quad j = 1, 2, \dots, m.$$

Depending on the properties of the function at hand it can be guaranteed to find an extrema or not. For example convex functions have guarantees of convergence while non-convex functions can have chelating features that make it hard to work with optimization algorithms.

To solve such a system we employ iterative solvers that use an initial guess. Let us look at the most common, the gradient descent method.

7.1 Gradient Descent

For a higher dimensional system or function f the gradient must be zero

$$\nabla f(x) = 0$$

to know that we are in an extrema. Since we can have saddle points this is not the sole criteria but a necessary one. Gradient descent, as the name suggest uses the gradient as *direction* in an iterative algorithm to find a minimum.

The idea is basically, if you are lost on a mountain in the fog and you can not see the path, the fastest and a reliable way that only uses local information is to follow the steepest slope down.

> **Warning**
>
> A function does not necessarily experience gravity in the same way as we do, so please do not try this in real live, i.e. cliffs tend to be hard to walk down.

We express this algorithm in terms of the iterations x^k for guesses of the minimum with the updates

$$x^{(k+1)} = x^{(k)} - \delta \nabla f(x^{(k)})$$

where the parameter δ defines how far along the gradient descent curve we move. This formula is an update for a Newton method where we use the derivative as the update function. This leaves us with the problem to find an algorithm to determine δ.

Again, we can view this as an optimization problem for a new function

$$F(\delta) = f(x^{(k+1)}(\delta))$$

and

$$\partial_\delta F = -\nabla f(x^{(k+1)}) \nabla f(x^{(k)}) = 0. \tag{7.1}$$

Now the interpretation of Equation 7.1 is that we want that the gradient of the current step is orthogonal to the gradient of the next step.

In order to make it clearer we follow the example given in (Brunton and Kutz 2022, sec. 4.2,pp. 141-144).

Example 7.1 (Gradient descent)**.** For the function

$$f(x) = x_1^2 + 3x_2^2$$

we can compute the gradient as

$$\nabla f(x) = \begin{bmatrix} \partial_{x_1} f(x) \\ \partial_{x_2} f(x) \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 6x_2 \end{bmatrix}$$

Resulting in

$$x^{k+1} = x^{(k)} - \delta \nabla f(x^{(k)}) = \begin{bmatrix} (1-2\delta)x_1^{(k)} \\ (1-6\delta)x_2^{(k)} \end{bmatrix}.$$

Consequently

$$F(\delta) = (1-2\delta)^2 x_1^2 + (1-6\delta)^2 x_2^2,$$
$$\partial_\delta F = -2^2(1-2\delta)x_1^2 - 6^2(1-6\delta)x_2^2,$$

and

$$\partial_\delta F(\delta) = 0 \Leftrightarrow \delta = \frac{x_1^2 + 9x_2^2}{2x_1^2 + 54x_2^2}.$$

```
import plotly.graph_objects as go
import numpy as np

x_ = np.linspace(-3, 3, 20)
y_ = np.linspace(-3, 3, 20)
X, Y = np.meshgrid(x_, y_)

f = lambda x, y: np.pow(x, 2) + 3 * np.pow(y, 2)
grad_f = lambda x: x * np.array([2, 6]).reshape(x.shape)
delta = lambda x: (x[0]**2 + 9 * x[1]**2)/(2 * x[0]**2 + 54 * x[1]**2)

Z = f(X, Y)

fig = go.Figure()
fig.add_trace(go.Surface(z=Z, x=X, y=Y,
                         colorscale='greys', name="Function"))
fig.update_traces(contours_z=dict(show=True, usecolormap=True,
                                  highlightcolor="limegreen",
                                  project_z=True))
fig.update_scenes(xaxis_title_text=r"x_1",
                  yaxis_title_text=r"x_2",
                  zaxis_title_text=r"f(x)")
x = np.array([3, 2]).reshape((1, 2))
z = np.array(f(x[0, 0], x[0, 1]))
diff = 1

while diff > 1e-10:
    x_new = x[-1, :] - delta(x[-1, :]) * grad_f(x[-1, :])
    z = np.hstack((z, f(x_new[0], x_new[1])))
    diff = np.linalg.norm(z[-1] - z[-2])
    x = np.vstack((x, x_new))

fig.add_scatter3d(x=x[:, 0], y=x[:, 1], z=z,
                  line_color='red', name="Descent path")

fig.show()
```

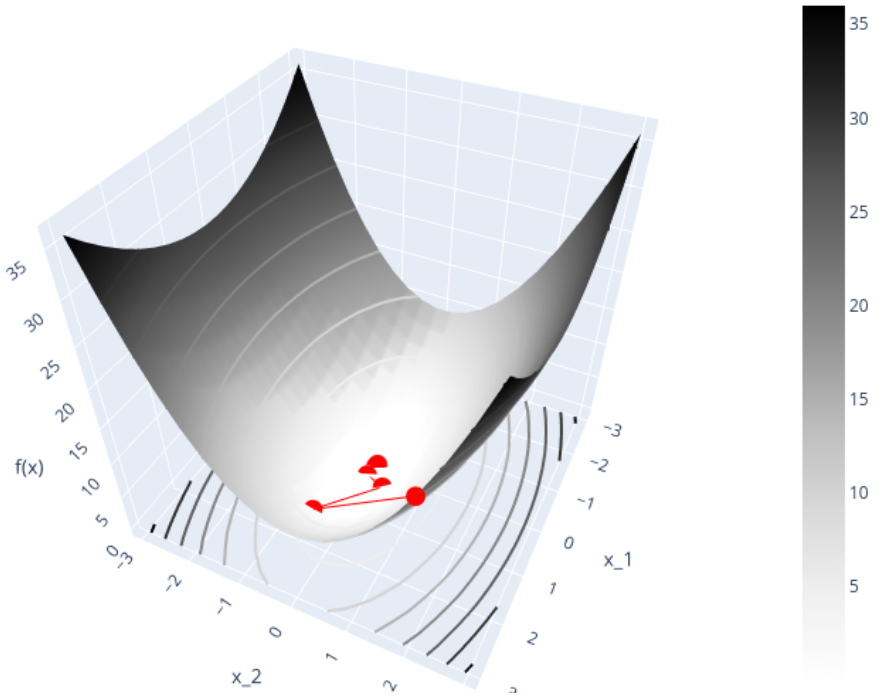


Figure 7.1: Gradient descent applied for the function $f(x) = x_1^2 + 3x_2^2$.

Note

If you can not compute the gradient analytically there are numerical methods to help do the computation.

In order to get a better idea on how this is working for *curve fitting* we apply the gradient descent method to our curve fitting from Section 6.1.

In Equation 6.5 we computed the gradient and instead of computing $X^\dagger$ with high cost we get the low cost iterative solver:

$$c^{(k+1)} = c^{(k)} - \delta(2X^\mathsf{T}Xc^{(k)} - 2X^\mathsf{T}y)$$

As δ is tricky to compute we go ahead and introduce we do not update it but prescribe it. This will not grant us the optimal convergence (if there is convergence) but if we choose it right we still get convergence.

> **Note**
>
> In machine learning the parameter δ is often called the *learning rate*.

So lets try it with our example from Figure 6.1.

```
import matplotlib.pyplot as plt
import numpy as np
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)

grad = lambda c, X, y: 2 * X.T @ (X @ c - y)
update = lambda c, delta, X, y: c - delta * grad(c, X, y)

def gd(c, delta, X, y, n, stop=1e-10):
    diff = 1
    for _ in range(1, n):
        cnew = update(c, delta, X, y)
        diff = np.linalg.norm(cnew - c)
        c = cnew
        if diff < stop: break
    return c

# The data
x = np.arange(1, 11)
y = np.array([0.2, 0.5, 0.3, 0.5, 1.0,
              1.5, 1.8, 2.0, 2.3, 2.2]).reshape((-1, 1))

X = np.array([x, np.ones(x.shape)]).T
delta = 0.002
c = np.random.random((2, 1))

c_10 = gd(c, delta, X, y, 50)
c_20 = gd(c_10, delta, X, y, 50)
c_30 = gd(c_20, delta, X, y, 200)
p4 = np.linalg.pinv(X) @ y

xf = np.arange(0, 11, 0.1)
y1 = np.polyval(c_10, xf)
y2 = np.polyval(c_20, xf)
y3 = np.polyval(c_30, xf)
y4 = np.polyval(p4, xf)
```

```
fig = plt.figure()
plt.plot(x, y, "o", color="r", label="observations")
plt.plot(xf, y1, label=r"$n=50$")
plt.plot(xf, y2, label=r"$n=100$")
plt.plot(xf, y3, label=r"$n=300$")
plt.plot(xf, y4, label=r"$E_2$")
plt.ylim(0, 4)
plt.xlim(0, 11)
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc="upper left")
plt.gca().set_aspect(1)
plt.grid(visible=True)
plt.show()
```

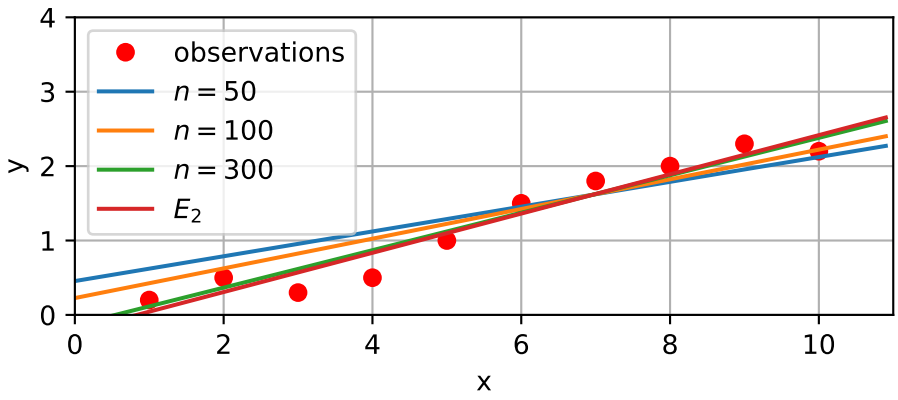


Figure 7.2: Line fit with gradient descent for different number of iterations and learning rate 2e-3.

The above algorithm uses the entire set X for the computation. For a large enough set X this is quite cost intense, even if it is still cheaper than computing $X^\dagger$.

7.2 Stochastic Gradient Descent

In order to reduce cost we can randomly select some points of our training set and only train with those. Obviously the computation of the gradient becomes much faster. We call this method *Stochastic Gradient descent* (SGD).

In Figure 7.3 we see the convergence for randomly selecting 1, 3, and 6 indices of our possible 10.

The downside of the SGD algorithm is that the algorithm does not settle down for a long time and will *jump*. In the other side it might get less stuck in local minima.

One possibility to try to get the strength of both is to use SDG to get a good guess for your initial value and SD for the fine tuning.

```
import matplotlib.pyplot as plt
import numpy as np
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)

grad = lambda c, X, y: 2 * X.T @ (X @ c - y)
update = lambda c, delta, X, y: c - delta * grad(c, X, y)

def sgd(c, delta, X, y, n, indices=-1, stop=1e-10):
    if indices == -1:
        indices = X.shape[0]
    diff = 1
    for _ in range(1, n):
        I = np.random.choice(X.shape[0], size=indices, replace=False)
        I.sort()
        cnew = update(c, delta, X[I, :], y[I])
        diff = np.linalg.norm(cnew - c)
        c = cnew
        if diff < stop: break
    return c

# The data
x = np.arange(1, 11)
y = np.array([0.2, 0.5, 0.3, 0.5, 1.0,
              1.5, 1.8, 2.0, 2.3, 2.2]).reshape((-1, 1))

X = np.array([x, np.ones(x.shape)]).T
delta = 0.002
c = np.random.random((2, 1))

c_10 = sgd(c, delta, X, y, 200, 1)
c_20 = sgd(c, delta, X, y, 200, 3)
c_30 = sgd(c, delta, X, y, 200, 5)
c_ft = gd(c_20, delta, X, y, 150, -1)
p4 = np.linalg.pinv(X) @ y

xf = np.arange(0, 11, 0.1)
```

```
y1 = np.polyval(c_10, xf)
y2 = np.polyval(c_20, xf)
y3 = np.polyval(c_30, xf)
yft = np.polyval(c_ft, xf)
y4 = np.polyval(p4, xf)


fig = plt.figure()
plt.plot(x, y, "o", color="r", label="observations")
plt.plot(xf, y1, label=r"#I=$1$")
plt.plot(xf, y2, label=r"#I=$3$")
#plt.plot(xf, y3, label=r"#I=$5$")
plt.plot(xf, yft, label=r"#I=$3$ GD $n=100$")
plt.plot(xf, y4, label=r"$E_2$")

plt.ylim(0, 4)
plt.xlim(0, 11)
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc="upper left")
plt.gca().set_aspect(1)
plt.grid(visible=True)
plt.show()
```

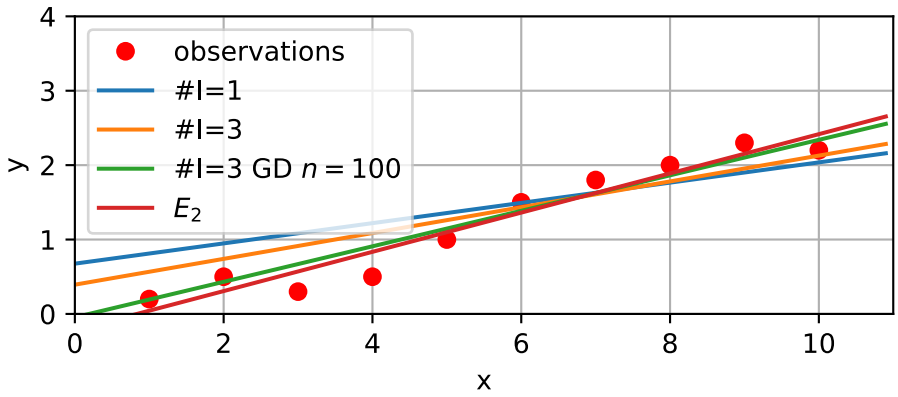


Figure 7.3: Line fit with stochastic gradient descent with 1 or 3 samples and 200 iterations as well as the 3 sample version as initial guess for GD with 100 iterations.

7.3 Categorical Variables

Even with our excursion to non-linear regression we still had somewhat regular data to work with. This is not always the case. Sometimes there are trends in the data, like per month, or day. The inclusion of categorical variables can help to control for trends in the data.

We can integrate such variables to the regressor by adding columns to the matrix X for each of the categories. Note, they can be interpreted as to correspond to the offset (the constant 1) so this column can be omitted and each category gets a separate offset.

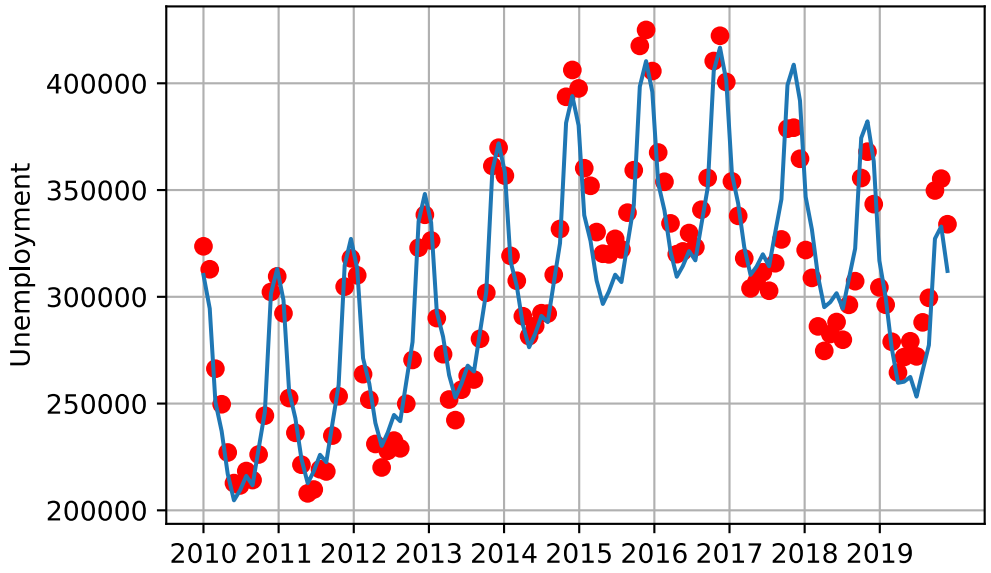
We can see this in action in the following example. We investigate the unemployment data in Austria. There is a strong seasonality Figure 7.4b in the data. This is largely due to the fact that the Austrian job market has a large touristic sector with its season and the construction industry employs less people during summer.

For the regression Figure 7.4b we can see that this captures the seasonal change quite well.

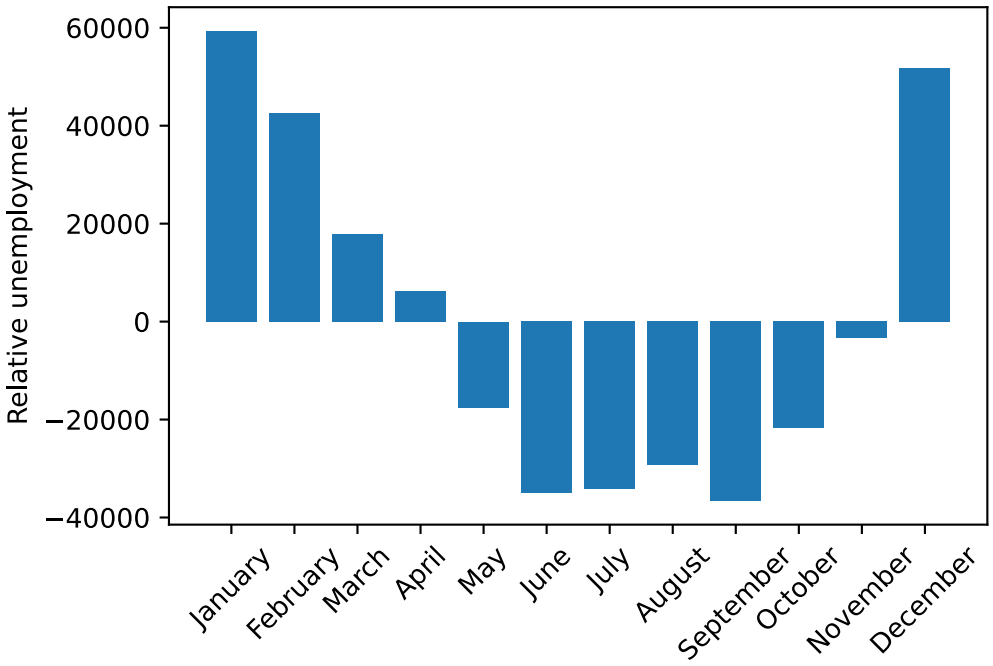
The data is taken from Arbeitsmarktdaten online.

> **Note**
>
> There is also quite a difference between man and woman that could be categorized separately.

We wrap up this section about regression by talking more abstract about the regression of linear systems and some general thoughts about the selection of the model and consequences.



(a) Regression with categorical variables per month.



(b) Seasonality of the unemployment average over the years.

Figure 7.4: Unemployment data from Austria for the years 2010 to 2017.

8 Optimizers

As we have seen in the previous section the task of regression usually results in an optimization problem. It is worth investigating this further by looking closely on the

$$Ax = b \tag{8.1}$$

problem for different dimensions of A.

We investigate the impact of restricting our solution not just by Equation 8.1 but with the help of the ℓ_1 and ℓ_2 norm imposed on the solution x. As we have seen in Figure 6.1 the choice of norm has an implication on the result and the same is true here.

> **!** Important
>
> This section is mainly based on Brunton and Kutz (2022), Section 4.3.

8.1 Over-Determined Systems

We speak of an over-determined system if we have more rows than columns, i.e. A is tall and skinny and in general there is no solution to Equation 8.1 but rather we minimize the error according to a norm, see Section 6.1. If we further impose a restriction on x we can select a more specific solution.

The generalized form is

$$x = \underset{v}{\operatorname{argmin}} \|Av - b\|_2 + \lambda_1 \|v\|_1 + \lambda_2 \|v\|_2 \tag{8.2}$$

where the parameters λ_1 and λ_2 are called the *penalization coefficients*, with respect to the norm. Selecting these coefficients is the first step towards *model selection*.

Let us have a look at this in action for solving a *random* system with different parameters λ_1 and setting λ_2.

8.1.1 LASSO

The *least absolute shrinkage and selection operator* LASSO solves Equation 8.2 with $\lambda_1 > 0$ and $\lambda_2 = 0$, i.e. only optimizing with the ℓ_1 norm. The theory tells us that for increasing λ_1 we should get more and more zeros in our solution x.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)

m = 500
n = 100
A = np.random.rand(m, n)
b = np.random.rand(m)
x0 = np.linalg.pinv(A) @ b

optimize = lambda x, A, b, lam, norm: np.linalg.norm(A @ x - b, ord=norm[0]) +\
    lam * np.linalg.norm(x, ord=norm[1])

fig = plt.figure()
axs = []
axs.append(fig.add_subplot(4, 1, 1))
axs.append(fig.add_subplot(4, 3, 10))
axs.append(fig.add_subplot(4, 1, 2))
axs.append(fig.add_subplot(4, 3, 11))
axs.append(fig.add_subplot(4, 1, 3))
axs.append(fig.add_subplot(4, 3, 12))

for i, lam in enumerate([0, 0.1, 0.5]):
    res = minimize(optimize, args=(A, b, lam, [2, 1]), x0=x0)
    axs[i * 2].bar(range(n), res.x)
    axs[i * 2].text(5, 0.05, rf"$\lambda_1={lam}$")
    axs[i * 2].set_xlim(0, 100)
    axs[i * 2].set_ylim(-0.1, 0.1)
    axs[i * 2 + 1].hist(res.x, 20)
    axs[i * 2 + 1].text(-0.08, 50, rf"$\lambda_1={lam}$")
    axs[i * 2 + 1].set_xlim(-0.1, 0.1)
    axs[i * 2 + 1].set_ylim(0, 70)
axs[0].set_xticks([])
axs[2].set_xticks([])
axs[3].set_yticks([])
```

```
axs[5].set_yticks([])


plt.subplots_adjust(top = 0.99, bottom=0.1, hspace=0.35, wspace=0.2)
plt.show()
```

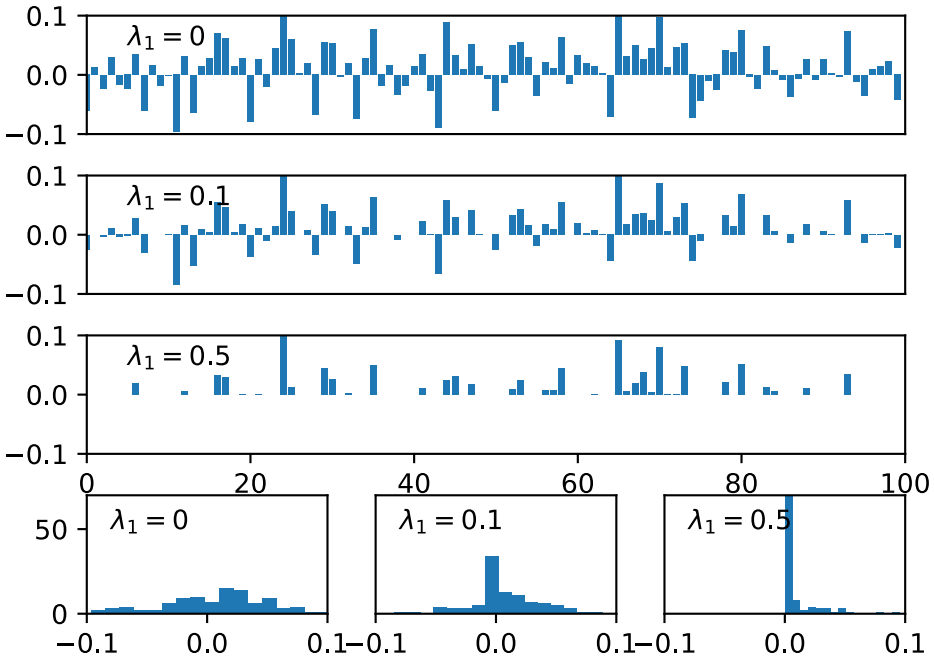


Figure 8.1: LASSO regression coefficients of an over-determined system with 500 constraints and 100 unknowns. Top three show the values for the solution and the bottom three a histogram of this solution. The label with lambda maps the two.

The last row of Figure 8.1 confirms this quite impressively, interesting enough the solution also becomes positive.

8.1.2 RIDGE

The Ridge Regression solves Equation 8.2 with $\lambda_1 = 0$ and $\lambda_2 > 0$, i.e. only optimizing with the ℓ_2 norm. The theory tells us that for for increasing λ_1 we should get more and more zeros in our solution x.

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

```
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)

m = 500
n = 100
A = np.random.rand(m, n)
b = np.random.rand(m)
x0 = np.linalg.pinv(A) @ b

optimize = lambda x, A, b, lam, norm: np.linalg.norm(A @ x - b, ord=norm[0]) +\
    lam * np.linalg.norm(x, ord=norm[1])

fig = plt.figure()
axs = []
axs.append(fig.add_subplot(4, 1, 1))
axs.append(fig.add_subplot(4, 3, 10))
axs.append(fig.add_subplot(4, 1, 2))
axs.append(fig.add_subplot(4, 3, 11))
axs.append(fig.add_subplot(4, 1, 3))
axs.append(fig.add_subplot(4, 3, 12))

for i, lam in enumerate([0, 0.1, 0.5]):
    res = minimize(optimize, args=(A, b, lam, [2, 2]), x0=x0)
    axs[i * 2].bar(range(n), res.x)
    axs[i * 2].text(5, 0.05, rf"$\lambda_2={lam}$")
    axs[i * 2].set_xlim(0, 100)
    axs[i * 2].set_ylim(-0.1, 0.1)
    axs[i * 2 + 1].hist(res.x, 20)
    axs[i * 2 + 1].text(-0.08, 15, rf"$\lambda_2={lam}$")
    axs[i * 2 + 1].set_xlim(-0.1, 0.1)
    axs[i * 2 + 1].set_ylim(0, 20)
axs[0].set_xticks([])
axs[2].set_xticks([])
axs[3].set_yticks([])
axs[5].set_yticks([])


plt.subplots_adjust(top = 0.99, bottom=0.1, hspace=0.35, wspace=0.2)
plt.show()
```

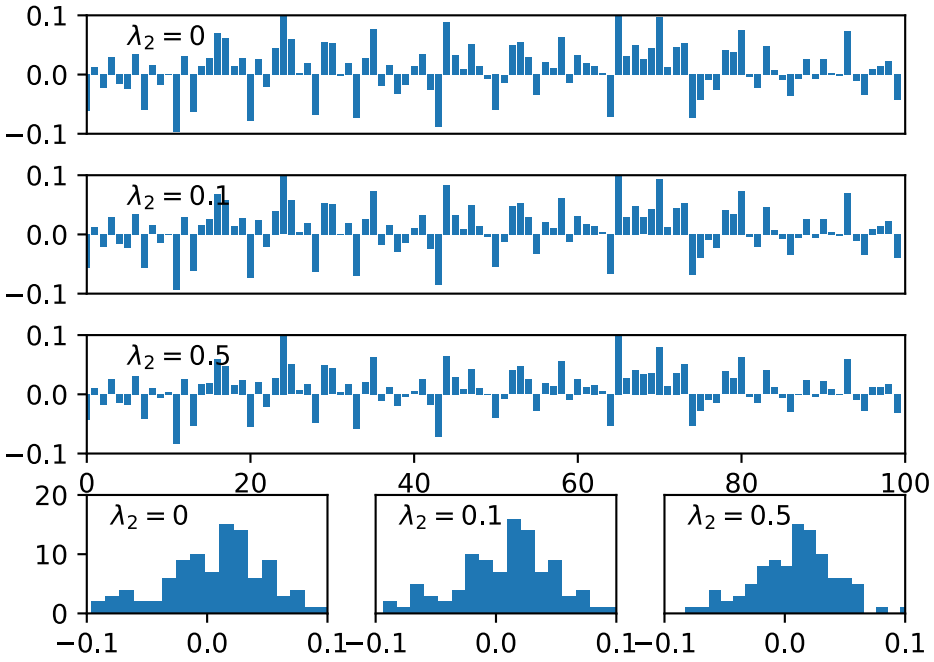


Figure 8.2: Ridge regression coefficients of an over-determined system with 500 constraints and 100 unknowns. Top three show the values for the solution and the bottom three a histogram of this solution. The label with lambda maps the two.

8.2 Model Selection/Identification and over-/underfitting

Let us use the results we have obtain so far for a discussion on *model selection*.

So far, we have mostly explicitly proposed a model that we think will fit our data and we have seen that even it this case we can still choose multiple parameters to fin tune our selection.

Now consider the other possibility, we have data where the model is unknown. For example, in Example 6.1 we stopped with degree 2 for our polynomial because we know about Newton's principles, if we don't know it, we might extend the model for a higher degree.

One of the leading assumptions to use in such a case is:

> *Among competing hypotheses, the one with the fewest assumptions should be selected, or when you have two competing theories that make exactly the same predictions, the simpler one is the more likely.* - Occam's razor

This plays an intimate role in over- and underfitting of models. To illustrate this we recall Example 6.2 with Figure 6.2 as seen below once more.

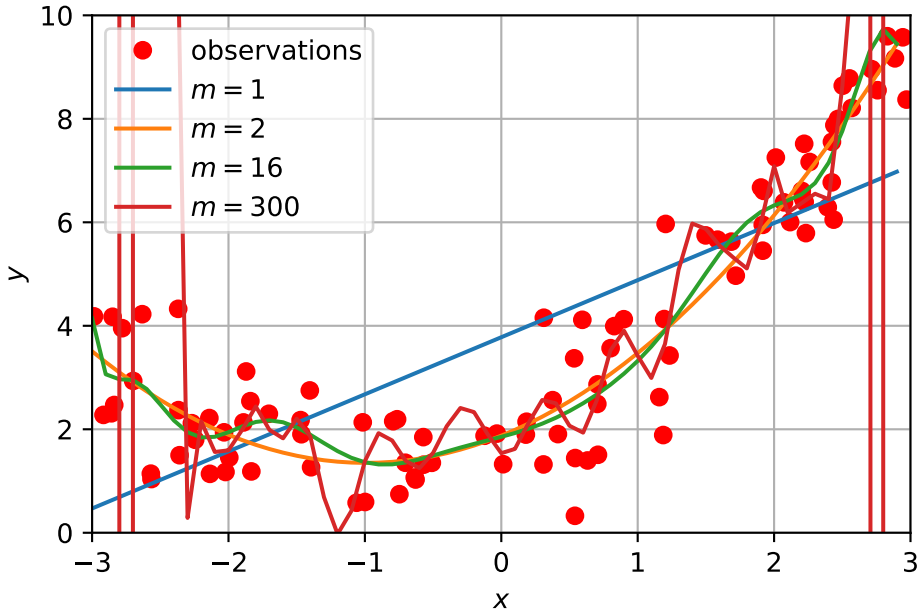


Figure 8.3: Fitting for different degrees of polynomial m

For $m = 1$, a straight line, we have an underfitted model. We can not adequately capture the underlying model, at least not in the entire region.

If we move to $m = 16$ and the extreme $m = 300$ we see an overfitted system. The $m = 16$ curve follows clusters of points *too close*, e.g. in the region around $x = -2$, this is more pronounced for $m = 300$ where we quite often closely follow our observations but between them we clearly overshoot.

In this way we can also say that an overfitted system follows the *training* set to closely and will not generalize good for another *testing/evaluation* set.

As a consequence model selection should always be followed by a cross-validation. Meaning we need to check if our model is any good.

A classic method is the *k-fold cross validation*:

> *Take random portions of your data and build a model. Do this k times and average the parameter scores (regression loadings) to produce a cross-validated model. Test the model predictions against withheld (extrapolation) data and evaluate whether the model is actually any good.* - (see Brunton and Kutz 2022, 159)

As we can see, there are a lot of further paths to investigate but for now this concludes our excursion into regression.

Part IV

Signal Processing

In previous parts we have already discussed how changing a coordinate system or basis (Definition 1.8) can simplify expression or computation, see Chapter 4 and Chapter 5, among others.

One of the most fundamental coordinate transformations was introduced by J.-B. Joseph Fourier in the early 1800s. While investigating *heat* he discovered that sine and cosine functions with increasing frequency form an orthogonal basis (Definition 1.6 & Definition 1.9). In fact, the sine and cosine functions for an eigenbasis for the heat equation

$$\frac{\partial u}{\partial t} = \Delta u$$

and solving it becomes trivial once you determine the corresponding eigenvalues that are connected to the geometry, amplitudes, and boundary conditions.

In the 200+ years since, this discovery has not only founded new corners of mathematics but also allows via the *fast fourier transform* or FFT the real-time image and audio compression that make our global communication networks work.

In the same area the related wavelets where developed to for advanced signal processing and compression.

In this section we are going to discuss basics of signal processing in terms of these and other signal transformations, see

- Chapter 9
- Chapter 11
- Chapter 12

Parts of this section are based on (Brunton and Kutz 2022, chap. 2).

9 Fourier Transform

The fourier transform helps us convert a signal from the time domain to the frequency domain. In this section our main concern is going to be one dimensional signals, but the concepts can be applied to multiple dimensions.

Before we can start defining the *Fourier Series* we need to extend our notion of vector space to functions space. This is done with *Hilbert* spaces. The computational rules follow the same principal as in Definition 1.1, what we want to investigate is the inner product.

Definition 9.1 (Hilbert inner product)**.** The *Hilbert inner product* of two functions $f(x)$ and $g(x)$ is defined for $x \in [a, b]$ as:

$$\langle f(x), g(x) \rangle = \int_a^b f(x)\overline{g}(x)\,\mathrm{d}x,$$

where $\overline{g}(x)$ denotes the complex conjugate.

At first this looks strange but it is closely related to our already known Definition 1.3.

As a first step, if we move from real to complex vector spaces the transpose is replaced by the conjugate transposed or hermit transpose, in notation the $^{\mathbf{T}}$ becomes $^{\mathbf{H}}$.

Now consider a discrete version of f and g at regular intervals $\Delta x = \frac{b-a}{n-1}$ where

$$f_i = f(x_i) = f(a + (i-1)\Delta x), \quad i = 1, \ldots n,$$

same for g_i and accordingly $x_1 = a + 0\Delta x = a$ and $x_n = a + (n-1)\Delta x = b$.

The inner product is than

$$\langle f, g \rangle = \langle \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}, \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} \rangle = \sum_{i=1}^{n} f_i \overline{g}_i = \sum_{i=1}^{n} f(x_i)\overline{g}(x_i).$$

As this sum will increase by increasing n we should normalize it by the factor Δx.

$$\frac{b-a}{n-1}\langle g, f \rangle = \sum_{i=1}^{n} f(x_i)\overline{g}(x_i)\Delta x.$$

If we now increase $n \to \infty$ we get $\Delta x \to 0$ and the sum transforms into the integral.

Definition 9.2 (Hilbert two norm)**.** The *Hilbert two norm* of the function $f(x)$ is defined for $x \in [a, b]$ as:

$$\|f\|_2 = \sqrt{\langle f(x), g(x) \rangle} = \left(\int_a^b f(x)\overline{f}(x)\,\mathrm{d}x \right)^{\frac{1}{2}},$$

where $\overline{f}(x)$ denotes the complex conjugate.
The set of all functions with bounded norm defines the Hilbert space $L^2(a, b)$, i.e. the set of all square integrable functions. This space is also called the space of *Lebesgue* integrable functions.

Similar as we saw projection in vector spaces related to the inner product this is true here as well.

Definition 9.3 (Periodic function)**.** We call a function $f : \mathbb{R} \to \mathbb{R}$ periodic with a period of $L > 0$, L-periodic for short, if

$$f(g + L) = f(t), \quad \forall t \in \mathbb{R}.$$

The following holds true for L-periodic functions:

1. If L is a period than nL for $n = 1, 2, 3, ...$ is a period as well.
2. If f and g are L-periodic, than $\alpha f + \beta g$ are L-periodic, for $\alpha, \beta \in \mathbb{C}$.
3. I f is L-periodic it follows that $\forall a \in \mathbb{R}$

$$\int_a^{a+T} f(t)\,\mathrm{d}t = \int_0^T f(t)\,\mathrm{d}t.$$

4. If f is L-periodic than $F(t) = f(\frac{t}{\omega})$ with $\omega = \frac{2\pi}{L}$ is 2π-periodic.

The Fourier series is nothing else as the projection of a function with an integer period on the domain $[a, b]$ onto the orthogonal basis defined by the sine and cosine functions.

9.1 Fourier Series

In Fourier analysis the first result is stated for a periodic and piecewise smooth function $f(t)$.

Definition 9.4 (Fourier Series)**.** For a L-periodic function $f(t)$ we can write

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos\left(\omega k t\right) + b_k \sin\left(\omega k t\right) \right), \tag{9.1}$$

for

$$a_k = \frac{2}{L} \int_0^L f(t) \cos\left(\omega k t\right) \, \mathrm{d}t,$$
$$b_k = \frac{2}{L} \int_0^L f(t) \sin\left(\omega k t\right) \, \mathrm{d}t.$$

where we can view the last two equations as the projection onto the orthogonal basis $\{\cos(kt), \sin(kt)\}_{k=0}^{\infty}$, i.e.

$$a_k = \frac{1}{\| \cos\left(\omega k t\right) \|_2^2} \langle f(t), \cos\left(\omega k t\right) \rangle,$$
$$b_k = \frac{1}{\| \sin\left(\omega k t\right) \|_2^2} \langle f(t), \sin\left(\omega k t\right) \rangle.$$

If we perform a partial reconstruction by truncating the series at M we get

$$\hat{f}_M(t) = \frac{a_0}{2} + \sum_{k=1}^{M} \left(a_k \cos\left(\omega k t\right) + b_k \sin\left(\omega k t\right) \right).$$

With the help of Euler's formula:

$$\mathrm{e}^{\mathrm{i}kt} = \cos(kt) + \mathrm{i}\sin(kt) \tag{9.2}$$

we can rewrite Equation 9.1 as

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \mathrm{e}^{\omega \mathrm{i} k t}$$

with

$$c_k = \frac{1}{L} \int_0^L f(t) \mathrm{e}^{-\omega \mathrm{i} k t} \, \mathrm{d}t.$$

and for $n = 1, 2, 3, ...$

$$c_0 = \tfrac{1}{2} a_0, \quad c_n = \tfrac{1}{2}(a_n - \mathrm{i} b_n), \quad c_{-n} = \tfrac{1}{2}(a_n + \mathrm{i} b_n).$$

Note

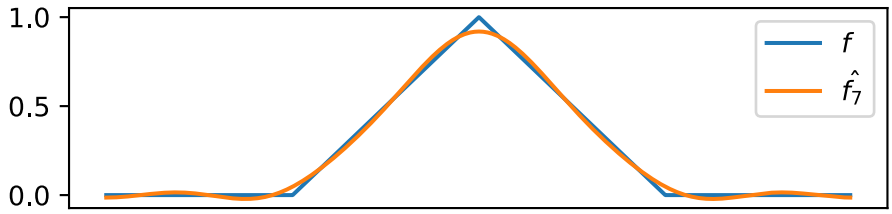
If $f(t)$ is real valued than $c_k = \overline{c}_{-k}$.

Example 9.1 (Fourier Series of Hat functions)**.** We test the Fourier Series with two different hat functions. The first represents a triangle with constant slope up and down, the second a rectangle with infinite slope in the corners.

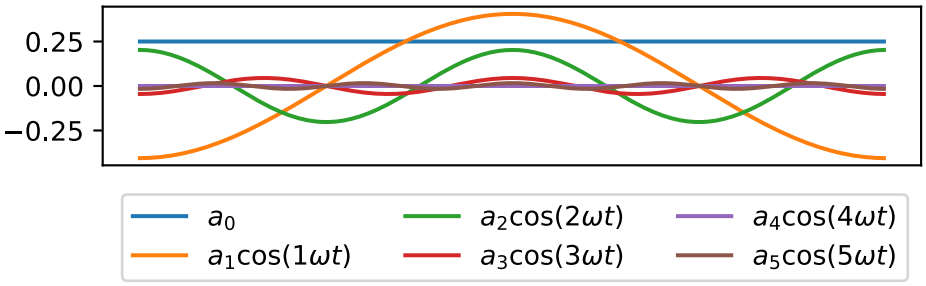
```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

# Parameters
L = 2 * np.pi
M = 5
M2 = 50
N = 512
# Hat functions
fun = lambda t, L: 0 if abs(t) > L / 4 else (1 - np.sign(t) * t * 4 / L)
fun2 = lambda t, L: 0 if abs(t) > L / 4 else 1

# t
t = np.linspace(-L/2, L/2, N, endpoint=False)                    ①
dt = t[1] - t[0]
w = np.pi * 2 / L

f = np.fromiter(map(lambda t: fun(t, L), t), t.dtype)
f2 = np.fromiter(map(lambda t: fun2(t, L), t), t.dtype)

# Necessary functions
scalarproduct = lambda f, g, dt: dt * np.vecdot(f, g)
a_coeff = lambda n, f: 2 / L * scalarproduct(f, np.cos(w * n * t), dt)
b_coeff = lambda n, f: 2 / L * scalarproduct(f, np.sin(w * n * t), dt)

# f_hat_0
f_hat = np.zeros((M + 1, N))
f_hat[0, :] = 1/2 * a_coeff(0, f)
f2_hat = np.zeros((M2 + 1, N))
f2_hat[0, :] = 1/2 * a_coeff(0, f2)

# Computation of the approximation
a = np.zeros(M)
b = np.zeros(M)
for i in range(M):
    a[i] = a_coeff(i + 1, f)
```

```
    b[i] = b_coeff(i + 1, f)
    f_hat[i + 1, :] = f_hat[i, :] + \
        a[i] * np.cos(w * (i + 1) * t) + \
        b[i] * np.sin(w * (i + 1) * t)

for i in range(M2):
    f2_hat[i + 1, :] = f2_hat[i, :] + \
        a_coeff(i + 1, f2) * np.cos(w * (i + 1) * t) + \
        b_coeff(i + 1, f2) * np.sin(w * (i + 1) * t)

# Figures
plt.figure(0)
plt.plot(t, f, label=r"$f$")
plt.plot(t, f_hat[-1, :], label=r"$\hat{f_7}$")
plt.xticks([])
plt.legend()
plt.gca().set_aspect(1.5)

plt.figure(1)
plt.plot(t, f_hat[0, :], label=rf"$a_{0}$")
for i in range(M):
    plt.plot(t, a[i] * np.cos(w * (i+1) * t),
             label=rf"$a_{i+1}\cos({i+1}\omega t)$")
plt.legend(ncol=np.ceil((M + 1) / 2), bbox_to_anchor=(1, -0.1))
plt.xticks([])
plt.gca().set_aspect(1.5)

plt.figure(2)
plt.plot(t, f2, label=r"$f$")
plt.plot(t, f2_hat[7, :], label=r"$\hat{f}_7$")
plt.plot(t, f2_hat[20, :], label=r"$\hat{f}_{20}$")
plt.plot(t, f2_hat[50, :], label=r"$\hat{f}_{50}$")
plt.xlabel(r"$x$")
plt.legend()
plt.gca().set_aspect(1.5)

plt.show()
```

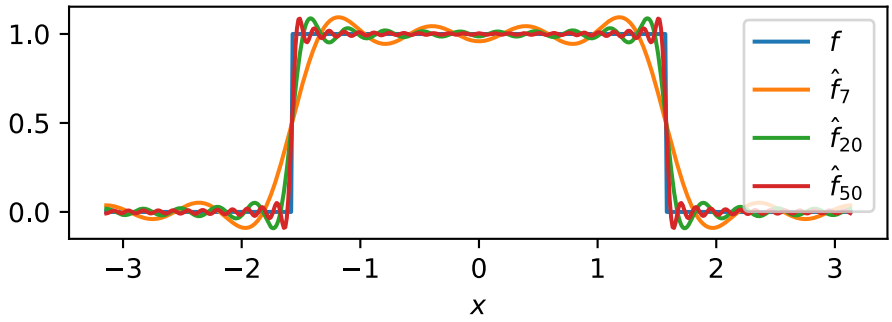
① Not including the endpoint is important, as this is part of the periodicity of the function.



(a) Sawtooth function and the reconstruction with 7 nodes



(b) Nodes of the reconstruction



(c) Step function and the reconstruction with various nodes

Figure 9.1: Fourier transform of a two hat functions.

Note

The phenomenon that the truncated Fourier series oscillates in Figure 9.1c due to the discontinuity of the function is called the Gibbs phenomenon.

9.2 Fourier Transform

The Fourier Series is defined for L-periodic functions. The Fourier transform extends this to functions with the domain extended to $\pm\infty$.

Let us start of with the series representation we already know:

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \mathrm{e}^{\mathrm{i}\omega k t}$$

with the coefficients

$$c_k = \frac{1}{2L} \int_{-L}^{L} f(t) \mathrm{e}^{-\mathrm{i}\omega_k t} \, \mathrm{d}t,$$

with $\omega_k = \frac{k\pi}{L} = k\Delta\omega$.

If we now perform the transition for $L \to \infty$ resulting in $\Delta\omega \to 0$ and basically moving from discrete frequencies to a continuous set of frequencies. This results in

$$f(t) = \lim_{\Delta\omega \to 0} \sum_{k=-\infty}^{\infty} \frac{\Delta\omega}{2\pi} \int_{-\frac{\pi}{\Delta\omega}}^{\frac{\pi}{\Delta\omega}} f(\xi) \mathrm{e}^{-\mathrm{i}k\Delta\omega\xi} \, \mathrm{d}\xi \; \mathrm{e}^{\Delta\omega \mathrm{i} k t}$$

which is a Riemann integral and the kernel becomes the Fourier Transform of our function.

Definition 9.5 (Fourier Transform)**.** A function $f : \mathbb{R} \to \mathbb{R}$ is called fourier transposable if

$$\hat{f}(\omega) = \mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t) \mathrm{e}^{-\mathrm{i}\omega t} \, \mathrm{d}t$$

exists for all $\omega \in \mathbb{R}$. In this case we call $\hat{f}(\omega) \equiv \mathcal{F}\{f(t)\}$ the **Fourier transform** of $f(t)$. The **inverse Fourier transform** is defined as

$$\mathcal{F}^{-1}\{\hat{f}(\omega)\} = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) \mathrm{e}^{\mathrm{i}\omega x} \, \mathrm{d}\omega$$

i Note

The pair $(f, \hat{f})$ is often called the *Fourier transform pair*.
The two integrals converge, as long as both functions are Lebesgue integrable, i.e.

$$\int_{-\infty}^{\infty} |f(t)| \, \mathrm{d}t \leq \infty,$$

or $f, \hat{f} \in L^1[(-\infty, \infty)]$.

As could be expected, the Fourier transform has properties that lead to computational advantages.

For tow functions $f, g \in L^1[(-\infty, \infty)]$ and $\alpha, \beta \in \mathbb{C}$ the following properties hold:

1. **Linearity**

$$\mathcal{F}\{\alpha f(t) + \beta g(t)\} = \alpha\mathcal{F}\{f(t)\} + \beta\mathcal{F}\{g(t)\} = \alpha\hat{f}(\omega) + \beta\hat{g}(\omega),$$

and

$$\mathcal{F}^{-1}\{\alpha\hat{f}(\omega) + \beta\hat{g}(\omega)\} = \alpha\mathcal{F}^{-1}\{\hat{f}(\omega)\} + \beta\mathcal{F}^{-1}\{\hat{g}(\omega)\} = \alpha f(t) + \beta g(t).$$

2. **Conjugation**

$$\mathcal{F}\{\overline{f(t)}\} = \overline{\hat{f}(-\omega)}.$$

3. **Scaling**, for $\alpha \neq 0$

$$\mathcal{F}\{f(\alpha t)\} = \frac{1}{|\alpha|}\hat{f}\left(\frac{\omega}{\alpha}\right).$$

4. **Drift in time**, for $a \in \mathbb{R}$

$$\mathcal{F}\{f(t-a)\} = \mathrm{e}^{-\mathrm{i}\omega a}\hat{f}(\omega).$$

5. **Drift in frequency**, for $a \in \mathbb{R}$

$$\mathrm{e}^{\mathrm{i}at}\mathcal{F}\{f(t)\} = \hat{f}(\omega - a).$$

6. If f is **even** or **odd**, than $\hat{f}$ is even or odd, respectively.

7. **Derivative in time**

$$\mathcal{F}\{\partial_t f(t)\} = \mathrm{i}\omega\hat{f}(\omega)$$

We are going to prove this by going through the lines

$$\begin{aligned}
\mathcal{F}\left\{\frac{d}{d\,t}f(t)\right\} &= \int_{-\infty}^{\infty} f'(t)\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t \\
&= \left[f(t)\mathrm{e}^{-\mathrm{i}\omega t}\right]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} -\mathrm{i}\omega f(t)\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t \\
&= \mathrm{i}\omega\int_{-\infty}^{\infty} f(t)\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t \\
&= \mathrm{i}\omega\mathcal{F}\{f(t)\}
\end{aligned}$$

For higher derivatives we get

$$\mathcal{F}\{\partial_t^n f(t)\} = \mathrm{i}^n\omega^n\hat{f}(\omega)$$

8. **Derivative in frequency**
$$\mathcal{F}\{t^n f(t)\} = \mathrm{i}^n \partial_\omega^n \hat{f}(\omega)$$

9. The **convolution** of two functions is defined as
$$(f * g)(t) = \int_{-\infty}^{\infty} f(t-\xi) g(\xi)\, \mathrm{d}\xi,$$
and for the Fourier transform
$$\mathcal{F}\{(f * g)(t)\} = \hat{f} \cdot \hat{g}.$$

10. **Parseval's Theorem**
$$\|f\|_2^2 = \int_{-\infty}^{\infty} |f(t)|^2\, \mathrm{d}t = \frac{1}{2\pi} \int_{-\infty}^{\infty} |\hat{f}(\omega)|^2\, \mathrm{d}\omega$$
stating that the Fourier Transform preserves the 2-norm up to a scaling factor.

9.3 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is a way of approximating the Fourier transform on discrete vectors of data and it essentially a discretized version of the Fourier transform by sampling the function and numerical integration.

Definition 9.6 (Discrete-Fourier Transform)**.** For equally spaced values $t_k = k\Delta t$, for $k \in \mathbb{Z}$ and $\Delta t > 0$ and the discrete values of the function evaluations $f_k = f(t_k)$. If the function is periodic with $L = N\Delta t$ than the discrete Fourier transform is given as
$$\hat{f}_k = \sum_{j=0}^{N-1} f_j\, \mathrm{e}^{-\mathrm{i}jk\frac{2\pi}{N}}, \tag{9.3}$$
and its inverse (iDFT) as
$$f_k = \frac{1}{N} \sum_{j=0}^{N-1} \hat{f}_j\, \mathrm{e}^{\mathrm{i}jk\frac{2\pi}{N}}.$$

As we can see, the DFT is a linear operator and therefore it can be written as a matrix vector product
$$\begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_N \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_N & \omega_N^2 & \dots & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & \dots & \omega_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \dots & \omega_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{bmatrix} \tag{9.4}$$

with $\omega_N = \exp(-\mathrm{i}\frac{2\pi}{N})$.

> **Note**
>
> The matrix of the DFT is a unitary Vandermonde matrix.

As we can transfer the properties of the Fourier transform to the DFT we get the nice properties for sampled signals.

The downside of the DFT is that it does not scale well for large N as the matrix-vector multiplication is $\mathcal{O}(N^2)$ and becomes slow.

> **Tip**
>
> Code for the computation of the DFT matrix.

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

# Parameters
N = 256
w = np.exp(-1j * 2 * np.pi / N )

J, K = np.meshgrid(np.arange(N), np.arange(N))
DFT = np.power(w, J*K)
```

9.4 Fast Fourier Transform

In 1965, James W. Cooley (IBM) and John W. Tukey (Princeton) developed the so called *fast Fourier transform* (FFT) that scales with $\mathcal{O}(N\log(N))$. which becomes almost linear for large enough N, see Cooley and Tukey (1965).

> **Note**
>
> To give an idea of what this change means and why this algorithm was a game changer. Audio is most of the time sampled with 44.1kHz, i.e. 44 100 samples per second. For a 10s audio clip the vector f will have the length $N = 4.41 \times 10^5$. The DFT computation (without generating the matrix) results ins approximately 2×10^{11} multiplications. The FFT on the other hand requires 6×10^6 leading to a speed-up of about 30000.
> How this influenced our world we know from the use in our daily communication networks.

(Compare Brunton and Kutz 2022, 65–66)

Note

We should note that Cooley and Tukey where not the first to propose a FFT but the provided the formulation used today. Gauss already formulated the FFT 150 years earlier in 1805 for orbital approximations. Apparently, he did the necessary computations in his head and needed a fast algorithm so he developed the FFT. Gauss being Gauss did not see this as something important and it did not get published until 1866 in his compiled notes, Gauß (1866).

The main idea of the FFT is to exploit symmetries in the Fourier transform and to relate the N-dimensional DFT to a lower dimensional DFT by reordering the coefficients.

Definition 9.7 (Fast-Fourier Transform)**.** If we assume that $N = 2^n$, i.e. a power of 2, in particular $N = 2M$, and F_N denotes the matrix of Equation 9.4 for dimension N and we have $\hat{f} = F_N f$ and $f = \frac{1}{N}\overline{F_N}\hat{f}$. By splitting f in the even and odd indices as

$$e = [f_0, f_2, \dots, f_{N-2}]^\mathsf{T} \in \mathbb{C}^M$$

and

$$o = [f_1, f_3, \dots, f_{N-1}]^\mathsf{T} \in \mathbb{C}^M$$

and for Equation 9.4 we get

$$\begin{aligned}\hat{f}_k &= \sum_{j=0}^{N-1} f_j\, \omega^{jk} = \sum_{j=0}^{M-1} f_{2j}\, \omega^{(2j)k} + \sum_{j=0}^{M-1} f_{2j+1}\, \omega^{(2j+1)k} \\ &= \sum_{j=0}^{M-1} e_j\, (\omega^2)^{jk} + \omega^k \sum_{j=0}^{M-1} o_j\, (\omega^2)^{jk}.\end{aligned}$$

If we further split $\hat{f}$ in an upper and lower part

$$u = [\hat{f}_0, \hat{f}_2, \dots, \hat{f}_{M-1}]^\mathsf{T} \in \mathbb{C}^M$$

and

$$l = [\hat{f}_M, \hat{f}_{M+1}, \dots, \hat{f}_{N-1}]^\mathsf{T} \in \mathbb{C}^M$$

and with the property $\omega^{k+M} = \omega^k \omega^M = -\omega^k$ we get

$$\begin{aligned}u_k &= \sum_{j=0}^{M-1} e_j\, (\omega^2)^{jk} + \omega^k \sum_{j=0}^{M-1} o_j\, (\omega^2)^{jk}, \\ l_k &= \sum_{j=0}^{M-1} e_j\, (\omega^2)^{jk} - \omega^k \sum_{j=0}^{M-1} o_j\, (\omega^2)^{jk}.\end{aligned}$$

This results in the more visual matrix representation

$$\hat{f} = F_N f = \begin{bmatrix} I_M & D_M \\ I_M & -D_M \end{bmatrix} \begin{bmatrix} F_M & 0 \\ 0 & F_M \end{bmatrix} \begin{bmatrix} f_{even} \\ f_{odd} \end{bmatrix},$$

for I_M being the identity matrix in dimension M and

$$D_M = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & \omega & 0 & \dots & 0 \\ 0 & 0 & \omega^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \omega^{(M-1)} \end{bmatrix}$$

Now repeat this n times.

Note

If N is not a power of 2 padding is used to make the size fit by extending the vector with zeros.

Note

The original FFT paper (Cooley and Tukey 1965) uses bit flipping and similar techniques to boost performance even more. It can even be implemented to allow for in place computation to save storage.
IMAGE
(Compare Meyberg and Vachenauer 1992, 331)

9.4.1 Examples for the FFT in action

In order to give an idea how FFT works in an application we follow the examples given in (Brunton and Kutz 2022, 66–76).

Example 9.2 (FFT for de-noising)**.** For a signal consisting of two main frequencies $f_1 = 50$ and $f_2 = 120$ we construct a signal

$$f(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$$

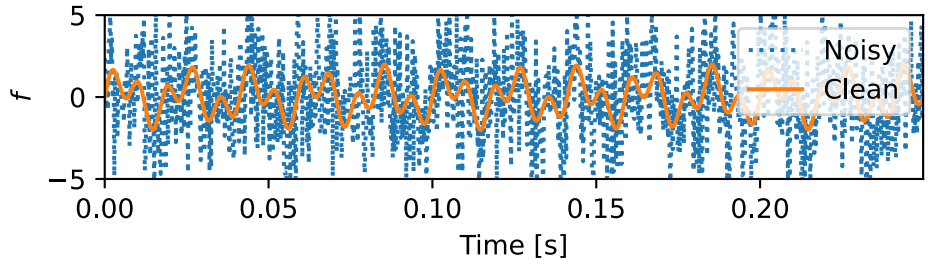
and add some Gaussian white noise `np.random.randn`.
We compute the FFT from the two signals and their power spectral density (PSD), i.e.

$$PSD(\hat{f}) = \frac{1}{N}\|\hat{f}\|^2.$$

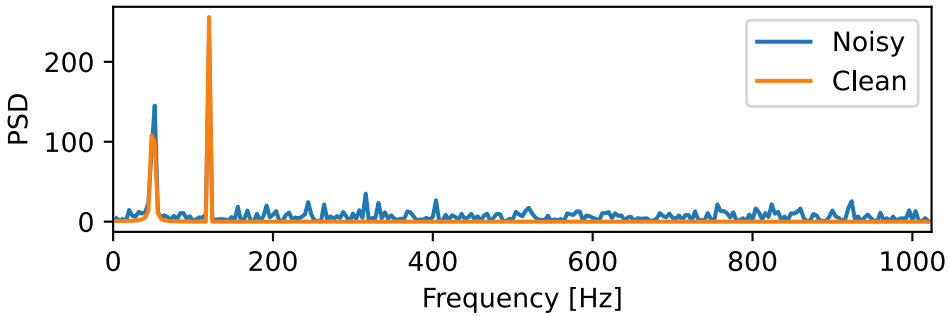
We use the PSD to take all frequencies with a $PSD < 100$ out of our reconstruction as a filter. This removes noise from the signal.

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

np.random.seed(6020)
# Parameters
N = 1024
a, b = 0, 1/4
t = np.linspace(a, b, N, endpoint=False)
dt = t[1] - t[0]
f1 = 50
f2 = 120
fun = lambda t: np.sin(2 * np.pi * f1 * t) + np.sin(2 * np.pi * f2 * t)

f_clean = fun(t)
f_noise = fun(t) + 2.5 * np.random.randn(len(t))              # Add some noise

fhat_noise = np.fft.fft(f_noise)
fhat_clean = np.fft.fft(f_clean)

PSD_noise = np.abs(fhat_noise)**2 / N
PSD_clean = np.abs(fhat_clean)**2 / N

freq = (1 / (dt * N)) * np.arange(N)
L = np.arange(1, np.floor(N/4), dtype='int')

# Apply filter in spectral space
filter = PSD_noise > 100
PSDclean = PSD_noise * filter
fhat_filtered = filter * fhat_noise
f_filtered = np.fft.ifft(fhat_filtered)

# Figures
plt.figure(0)
plt.plot(t, f_noise, ":", label=r"Noisy")
plt.plot(t, f_clean, label=r"Clean")
plt.xlabel("Time [s]")
plt.ylabel(r"$f$")
```

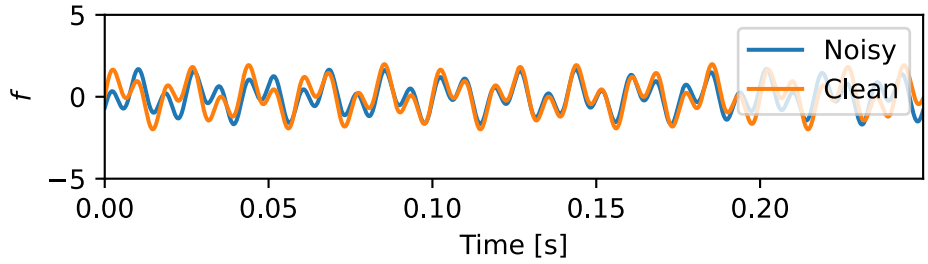
```
plt.xlim(t[0], t[-1])
plt.ylim(-5, 5)
plt.legend(loc=1)
plt.gca().set_aspect(5e-3)

plt.figure(1)
plt.plot(freq[L], PSD_noise[L], label=r"Noisy")
plt.plot(freq[L], PSD_clean[L], label=r"Clean")
plt.xlabel("Frequency [Hz]")
plt.ylabel("PSD")
plt.xlim(0, int(freq[L[-1] + 1]))
plt.legend(loc=1)
plt.gca().set_aspect(1)

plt.figure(3)
plt.plot(t, np.real(f_filtered), label=r"Noisy")
plt.plot(t, f_clean, label=r"Clean")
plt.xlabel("Time [s]")
plt.ylabel(r"$f$")
plt.xlim(t[0], t[-1])
plt.ylim(-5, 5)
plt.legend(loc=1)
plt.gca().set_aspect(5e-3)
plt.show()
```



(a) Original clean signal and noisy signal.



(b) Scaled square norm of of the Fourier coefficients (PSD), only parts are shown.



(c) Original signal and de-noised signal.

Figure 9.2: Signal noise filter with FFT.

As can be seen in the Figure 9.2c, the reconstruction is not exact. This is due to the fact that the reconstructed frequencies are not matched exactly plus we have some multiples that show up as well. In particular:

	Frequency	PSD
0	52.0	145.259
1	120.0	230.273
2	3976.0	230.273

3 4044.0 145.259

Note: For Figure 9.2c we discarded the imaginary part of the reconstruction.

Exercise DFT vs. FFT

Exercise 9.1 (DFT vs. FFT)**.** Implement Example 9.2 with DFT and FFT such that you can evaluate the runtime and create a plot showing the different runtime as well as check if the two produce the same result

For the Fourier transform we stated that the multiplication with $\mathcal{F}\{\partial f\} = \mathrm{i}\omega\mathcal{F}\{f\}$, similarly we can derive a formula for the numerical derivative of a sampled function by multiplying each entry of the transformed vector by $\mathrm{i}\kappa$ for $\kappa = \frac{2\pi k}{N}$. κ is called the discrete wavenumber associated with the component k.

Let us explore this with the example stated in (Brunton and Kutz 2022, 68–69).

Example 9.3 (Spectral derivative)**.** We compute the so called spectral derivative for the function

$$f(t) = \cos(t)\exp\left(-\frac{t^2}{25}\right)$$
$$\partial_t f(t) = -\sin(t)\exp\left(-\frac{t^2}{25}\right) - \frac{2}{25}tf(t)$$

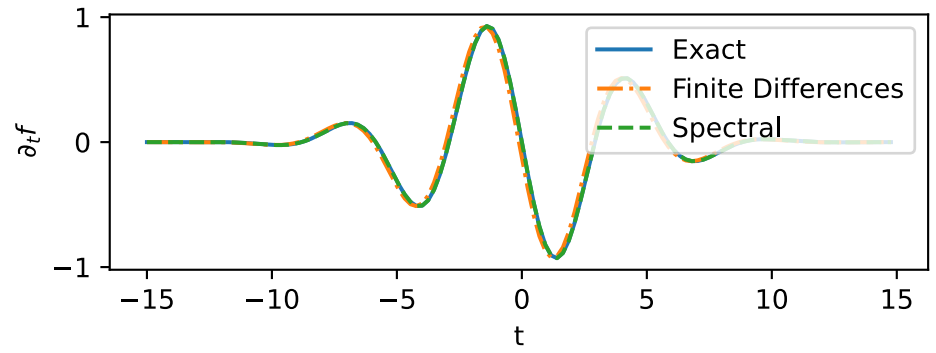
In order to porovide something to compare our results to we also compute the forward Euler finite-differences for the derivative

$$\partial_t f(t_k) \approx \frac{f(t_{k+1}) - f(t_k)}{t_{k+1} - t_k}.$$

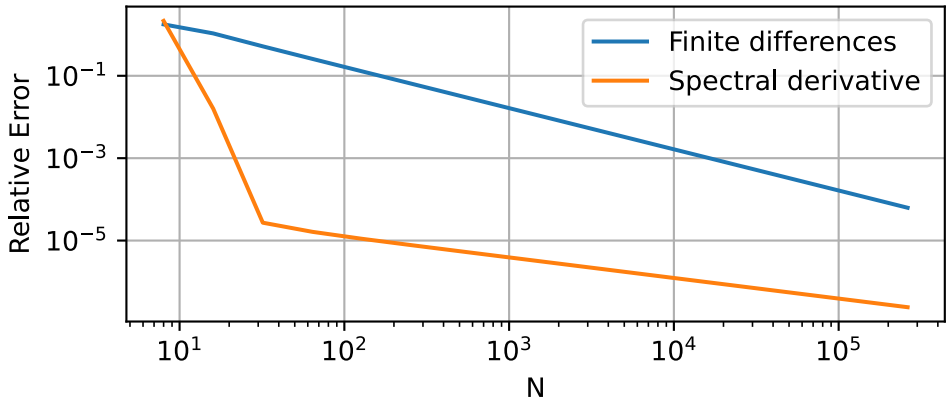
```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

# Parameters
N = 128
a, b = -15, 15
L = b - a

fun = lambda t: np.cos(t) * np.exp(-np.power(t, 2) / 25)
dfun = lambda t: -(np.sin(t) * np.exp(-np.power(t, 2) / 25) + \
                   (2 / 25) * t * fun(t))
```

```
def fD(N, fun, dfun, a, b):
    t = np.linspace(a, b, N, endpoint=False)
    dt = t[1] - t[0]
    f = fun(t)
    df_DD = np.diff(f) / dt
    df_DD = np.append(df_DD, (f[-1] - f[0]) / dt)
    return df_DD, np.linalg.norm(df_DD - dfun(t)) / np.linalg.norm(df_DD)

def spD(N, fun, dfun, a, b):
    t = np.linspace(a, b, N, endpoint=False)
    f = fun(t)
    fhat = np.fft.fft(f)
    kappa = np.fft.fftfreq(N, (b - a) / (N * 2 * np.pi))
    df_hat = kappa * fhat * (1j)
    df_r = np.fft.ifft(df_hat).real
    return df_r, np.linalg.norm(df_r - dfun(t)) / np.linalg.norm(df_r)


# Finite differences
df_fD, e = fD(N, fun, dfun, a, b)
# Spectral derivative
df_spD, e = spD(N, fun, dfun, a, b)

# Figures
t = np.linspace(a, b, N, endpoint=False)
plt.figure(0)
plt.plot(t, dfun(t), label="Exact")
plt.plot(t, df_fD, "-.", label="Finite Differences")
plt.plot(t, df_spD, "--", label="Spectral")
plt.xlabel("t")
plt.ylabel(r"$\partial_t f$")
plt.legend(loc=1)
plt.gca().set_aspect(5)

plt.figure(1)
n = 19
M = range(3, n)
e_spD = np.ones(len(M))
e_fD = np.ones(len(M))
for i, j in enumerate(M):
    _, e_fD[i] = fD(2**j, fun, dfun, a, b)
```

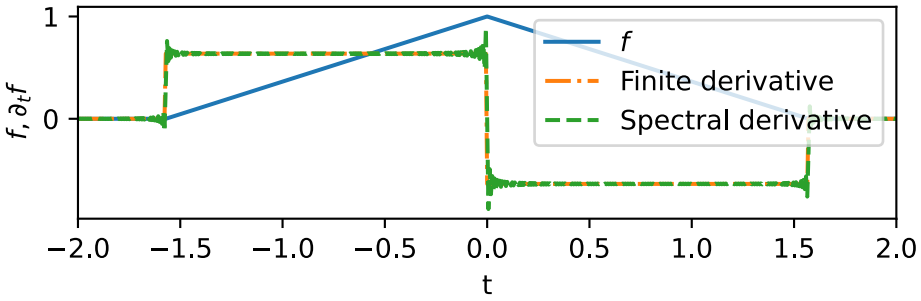
```
    _, e_spD[i] = spD(2**j, fun, dfun, a, b)

plt.loglog(np.pow(2, M), e_fD, label="Finite differences")
plt.loglog(np.pow(2, M), e_spD, label="Spectral derivative")
plt.grid()
plt.xlabel("N")
plt.ylabel("Relative Error")
plt.legend(loc=1)
plt.gca().set_aspect(2.5e-1)

fun_saw = lambda t, L: 0 if abs(t) > L / 4 else (1 - np.sign(t) * t * 4 / L)
a, b = -np.pi, np.pi
L = b - a
fun2 = lambda t: np.fromiter(map(lambda t: fun_saw(t, L), t), t.dtype)
N = 2**10

t = np.linspace(a, b, N, endpoint=False)

plt.figure(2)
df_spD, _ = spD(N, fun2, fun2, a, b)
df_fD, _ = fD(N, fun2, fun2, a, b)
plt.plot(t, fun2(t), label=r"$f$")
plt.plot(t, df_fD, "-.", label="Finite derivative")
plt.plot(t, df_spD, "--", label="Spectral derivative")
plt.xlabel("t")
plt.ylabel(r"$f, \partial_t f$")
plt.xlim(-2, 2)
plt.legend(loc=1)
plt.gca().set_aspect(0.5)
```



(a) Computation of the derivative with different methods



(b) Accuracy of the methods for computing the derivative



(c) Gibbs phenomenon for the spectral derivative for discontinuous functions

Figure 9.3: Computing the derivative of a function

As can be seen in Figure 9.3b we can reduce the error of both methods by increasing N. Nevertheless, the spectral method is more accurate and converges faster.

In Section 4.1.2 we have already seen how the eigendecomposition can be used to compute the solution of ordinary differential equations by changing the basis and transforming the equation into a basis that can be handled easy. The same is true with the Fourier transformation.

As mentioned before, Fourier introduced it to solve the heat equation and we will do the same.

! Important

Above we have always transformed the *time* component of a function into the frequency domain via FFT. In the next example we have the time evolution of a signal and therefore the function depends on time and space. We apply the FFT to the space component to compute the derivative in *frequency* domain w.r.t. the transformed variable.

Example 9.4 (Heat Equation)**.** The heat equation in 1D is given by

$$\dot{u}(t,x) = \alpha^2 \partial_x^2 u(\tau, x),$$

for $u(t,x)$ as the temperature distribution in space (x) and time (t). By applying the Fourier transform we get $\mathcal{F}\{u(t,x)\} = \hat{u}(t,\omega)$ and

$$\dot{\hat{u}}(t,\omega) = -\alpha^2\omega^2\hat{u}(t,\omega).$$

This transformed the partial differential equation into an ordinary differential equation and we can solve it for each fixed frequency ω as

$$\hat{u}(t,\omega) = \mathrm{e}^{-\alpha^2\omega^2 t}\hat{u}(0,\omega).$$

where $\hat{u}(0,\omega)$ is nothing else than the Fourier transform of the initial temperature distribution at time $t = 0$. To compute the inverse Fourier transform we can make use of the convolution property introduced above.

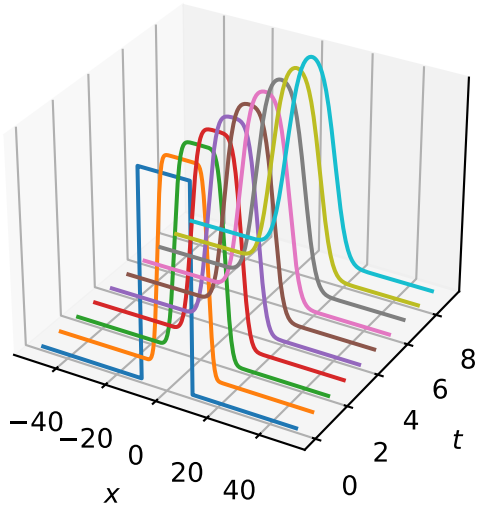
$$\begin{aligned} u(t,x) &= \mathcal{F}^{-1}\{\hat{u}(t,\omega)\} = \mathcal{F}^{-1}\{\mathrm{e}^{-\alpha^2\omega^2 t}\} * u(0,x) \\ &= \frac{1}{2\alpha\sqrt{\pi t}}\mathrm{e}^{-\frac{x^2}{4\alpha^2 t}} * u(0,x). \end{aligned}$$

For the numerical computation it is more convenient to stay in the frequency domain and use κ as before.
The majority of the following code is from (Brunton and Kutz 2022, Code 2.5 and Code 2.6).

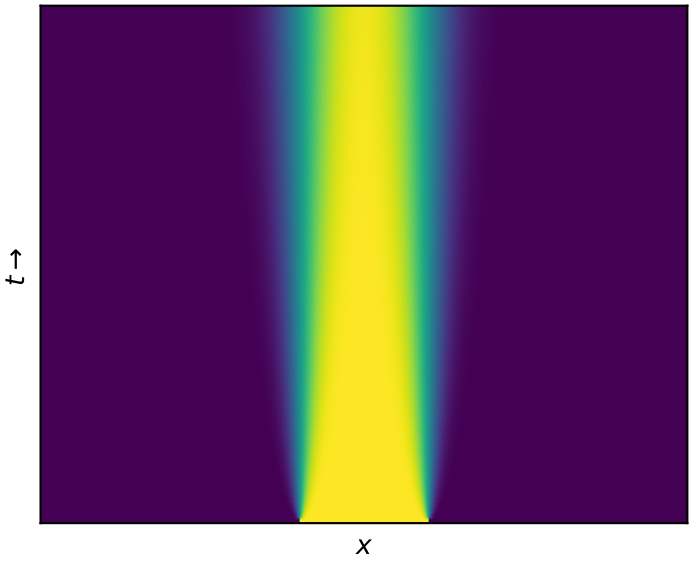
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

```
%config InlineBackend.figure_formats = ["svg"]

alpha = 1
a, b = -50, 50
L = b - a
N = 1000
x = np.linspace(a, b, N, endpoint=False)
dx = x[1] - x[0]

# Define discrete wavenumbers
#kappa = 2 * np.pi * np.fft.fftfreq(N, d=dx)
kappa = np.fft.fftfreq(N, (b - a) / (N * 2 * np.pi))

# Initial condition
u0 = np.zeros_like(x)
u0[np.abs(x) < 10] = 1
u0hat = np.fft.fft(u0)

# Simulate in Fourier frequency domain
dt = 0.001
T = np.arange(0, 10, dt)

fun = lambda t, x: - alpha**2 * (np.power(kappa, 2)) * x
euler = lambda y, dt, t, fun: y + dt * fun(t, y)

uhat = np.zeros((len(T), len(u0hat)), dtype="complex")
uhat[0, :] = u0hat
for i, t in enumerate(T[1:]):
    uhat[i + 1, :] = euler(uhat[i, :], dt, t, fun)

u = np.zeros_like(uhat)

for k in range(len(T)):
    u[k, :] = np.fft.ifft(uhat[k, :])

u = u.real

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
#ax.view_init(elev=45, azim=-20, roll=0)
ax.set_xlabel(r"$x$")
```

```
ax.set_ylabel(r"$t$")
ax.set_zlabel(r"$u(t, x)$")
ax.set_zticks([])

u_plot = u[0:-1:int(1 / dt), :]
for j in range(u_plot.shape[0]):
    ys = j * np.ones(u_plot.shape[1])
    ax.plot(x, ys, u_plot[j, :])

# Image plot
plt.figure()
plt.imshow(np.flipud(u[0:-1:100]), aspect=8)
plt.xlabel(r"$x$")
plt.ylabel(r"$t \to$")
plt.gca().axes.get_xaxis().set_ticks([])
plt.gca().axes.get_yaxis().set_ticks([])
plt.show()
```



(a) Evolution of the heat equation in time.



(b) x-t diagram of the transport equation.

Figure 9.4: Simulation of the heat equation in 1D.

Next, we look at the transport equation.

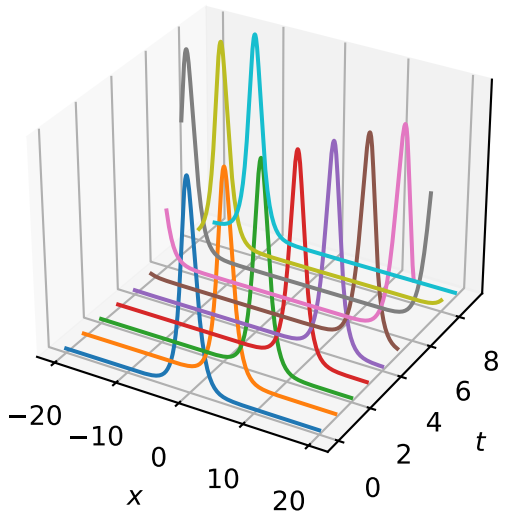
Example 9.5 (Transport or advection)**.** The transport equation in 1D is given by

$$\dot{u} = -c\partial_x u.$$

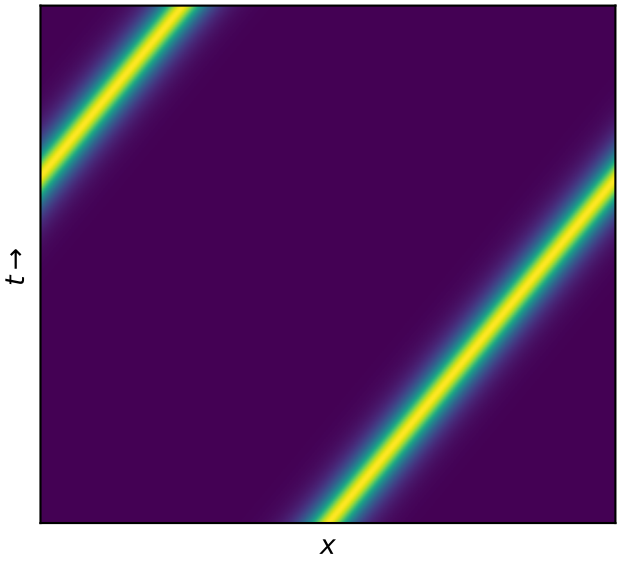
It is called the transport equation, as the initial value simply propagates in time to the right hand side of the domain with speed c, i.e. $u(t,x) = u(0, x - ct)$. The solution is simply computed by exchanging the right hand side from the above code.
The majority of the following code is from (Brunton and Kutz 2022, Code 2.5 and Code 2.6).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
%config InlineBackend.figure_formats = ["svg"]

c = 2
a, b = -20, 20
L = b - a
N = 1000
x = np.linspace(a, b, N, endpoint=False)
dx = x[1] - x[0]

# Define discrete wavenumbers
#kappa = 2 * np.pi * np.fft.fftfreq(N, d=dx)
kappa = np.fft.fftfreq(N, (b - a) / (N * 2 * np.pi))

# Initial condition
u0 = 1/np.cosh(x)
u0hat = np.fft.fft(u0)

# SciPy's odeint function doesn't play well with complex numbers,
# so we recast the state u0hat from an N-element complex vector
# to a 2N-element real vector
u0hat_ri = np.concatenate((u0hat.real, u0hat.imag))

# Simulate in Fourier frequency domain
dt = 0.025
T = np.arange(0, 600*dt, dt)

def rhsWave(uhat_ri,t,kappa,c):
    uhat = uhat_ri[:N] + (1j) * uhat_ri[N:]
    d_uhat = -c * (1j ) *kappa * uhat
```

```
    d_uhat_ri = np.concatenate((d_uhat.real, d_uhat.imag)).astype('float64')
    return d_uhat_ri

uhat_ri = odeint(rhsWave, u0hat_ri, T, args=(kappa, c))

uhat = uhat_ri[:, :N] + (1j) * uhat_ri[:, N:]

u = np.zeros_like(uhat)

for k in range(len(T)):
    u[k, :] = np.fft.ifft(uhat[k, :])

u = u.real

# Waterfall plot
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$t$")
ax.set_zlabel(r"$u(t, x)$")
ax.set_zticks([])
u_plot = u[0:-1:60, :]
for j in range(u_plot.shape[0]):
    ys = j * np.ones(u_plot.shape[1])
    ax.plot(x, ys, u_plot[j, :])

# Image plot
plt.figure()
plt.imshow(np.flipud(u), aspect=1.5)
plt.xlabel(r"$x$")
plt.ylabel(r"$t \to$")
plt.gca().axes.get_xaxis().set_ticks([])
plt.gca().axes.get_yaxis().set_ticks([])
plt.show()
```



(a) Evolution of the transport equation in time.



(b) x-t diagram of the transport equation.

Figure 9.5: Computing the derivative of a function

To increase complexity from these simple equations we move to the nonlinear Burger's equation which is a combination of the two above and an example that produces shock waves in fluids.

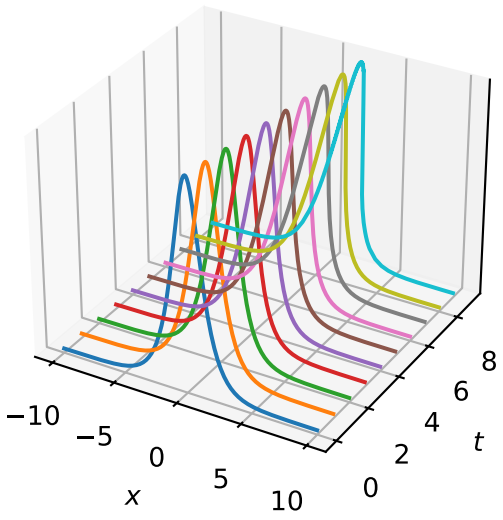
Example 9.6 (Burger's equation)**.** Burger's equation in 1D is given by

$$\dot{u} + u\partial_x u = \nu \partial_x^2 u.$$

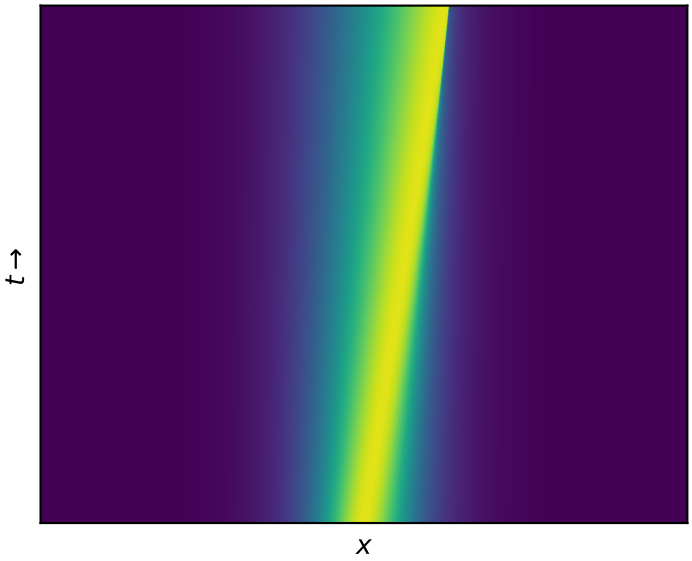
The equation consists of a nonlinear convection part $u\partial_t u$ as well as diffusion. The convection part is designed in such a way that larger amplitudes travel faster and therefore causes a shock wave to form.
Regarding the solution of this equation, the interesting part is that we need to map back and forth between the Fourier space and time space to apply the nonlinearity.
This example is also very useful to test the capabilities of your solver as by decreasing the diffusion factor further and further we can approach an infinitely steep shock and in theory it can break like an ocean wave.
The majority of the following code is from (Brunton and Kutz 2022, Code 2.5 and Code 2.6).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
%config InlineBackend.figure_formats = ["svg"]

nu = 0.001
a, b = -10, 10
L = b - a
N = 1000
x = np.linspace(a, b, N, endpoint=False)
dx = x[1] - x[0]

# Define discrete wavenumbers
#kappa = 2 * np.pi * np.fft.fftfreq(N, d=dx)
kappa = np.fft.fftfreq(N, (b - a) / (N * 2 * np.pi))

# Initial condition
u0 = 1/np.cosh(x)

# Simulate in Fourier frequency domain
dt = 0.025
T = np.arange(0,100*dt,dt)

def rhsBurgers(u, t, kappa, nu):
    uhat = np.fft.fft(u)
    d_uhat = (1j) * kappa * uhat
    dd_uhat = -np.power(kappa, 2) * uhat
```

```
    d_u = np.fft.ifft(d_uhat)
    dd_u = np.fft.ifft(dd_uhat)
    du_dt = -u * d_u + nu * dd_u
    return du_dt.real

u = odeint(rhsBurgers, u0, T, args=(kappa, nu))

# Waterfall plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$t$")
ax.set_zlabel(r"$u(t, x)$")
ax.set_zticks([])

u_plot = u[0:-1:10, :]
for j in range(u_plot.shape[0]):
    ys = j * np.ones(u_plot.shape[1])
    ax.plot(x, ys, u_plot[j, :])

# Image plot
plt.figure()
plt.imshow(np.flipud(u), aspect=8)
plt.xlabel(r"$x$")
plt.ylabel(r"$t \to$")
plt.gca().axes.get_xaxis().set_ticks([])
plt.gca().axes.get_yaxis().set_ticks([])
plt.show()
```



(a) Evolution of Burger's equation in time.



(b) x-t diagram of Burger's equation.

Figure 9.6: Time integration of the Burger's equation

The following example is taken from (Meyberg and Vachenauer 1992).

Example 9.7 (Ground Emitter Circuit)**.** We can use FFT to compute some base properties of a ground emitter circuit Figure 9.7 and how the signal is propagated through this circuit Figure 9.8.

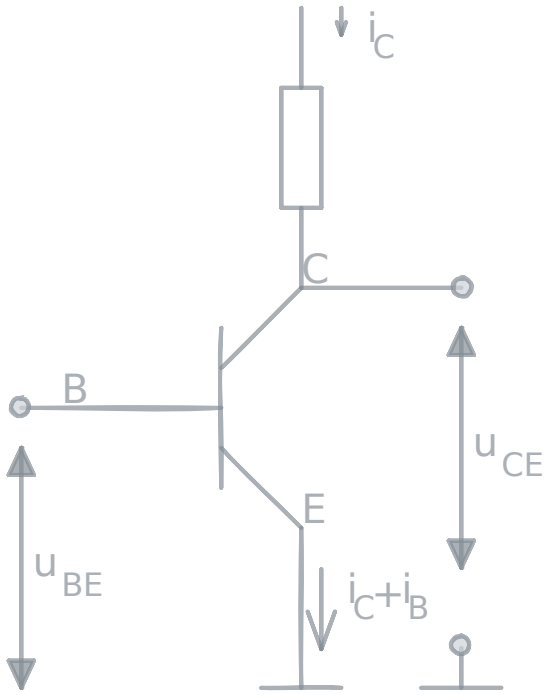


Figure 9.7: Electric circuit

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

frequ = 2 * np.pi * 50
f = lambda t: 0.8 * np.sin(t * frequ)
diod = lambda t: (np.exp(1.2 + t) - 1)

t = np.linspace(0, 2 * np.pi / frequ, 1024, endpoint=False)
x = np.linspace(np.min(f(t)) * 1.3, np.max(f(t)) * 1.1, 1024)

ic = lambda t: diod(f(t))

k = np.arange(0, 16) * 1 / 16
```

```
# The provided figures where used to create the illustation
if False:
    plt.figure()
    plt.plot(f(t), t)
    plt.plot([np.min(t),np.max(t)], [0,0], "gray")
    plt.axis("off")
    plt.savefig("sin.svg", transparent=True)
    plt.figure()
    plt.plot(x, diod(x))
    z = np.array([0, np.min(f(t)), np.max(f(t))])
    plt.plot(z, diod(z), "bx")
    plt.axis("off")
    plt.xlim([-2,5])
    plt.gcf().patch.set_visible(False)
    plt.savefig("diode.svg", transparent=True)

    plt.figure()
    plt.plot(t, ic(t))
    plt.plot(2*np.pi*k/frequ, ic(2*np.pi*k/frequ), "ro")
    plt.axis("off")
    plt.savefig("ic.svg", transparent=True, bbox_inches ="tight")

y = ic(k)
yhat = (np.fft.fft(y))
#Necessary correction factor for the FFT
factor = 1 / 16
yy = factor * yhat

ic_mean = np.mean(ic(np.linspace(0, 1/50, 2**20)))
c0 = yy[0].real
effective_value = np.linalg.norm(yy[1:])
harmonic_distortion = np.linalg.norm(yy[3:-2])/np.linalg.norm(yy[1:])
```

The transition of the transistor can be approximated by a diode and is illustrated in the u_{BE} vs. i_C diagram on the top left (orange signal in Figure 9.8). The exciting function u on the bottom left is a normal sine wave with 50Hz.
All together we get the equation

$$i_c = \mathrm{e}^{1.2+0.8\sin(2\pi 50t)} - 1$$

for the current in [mA] running through the transistor.

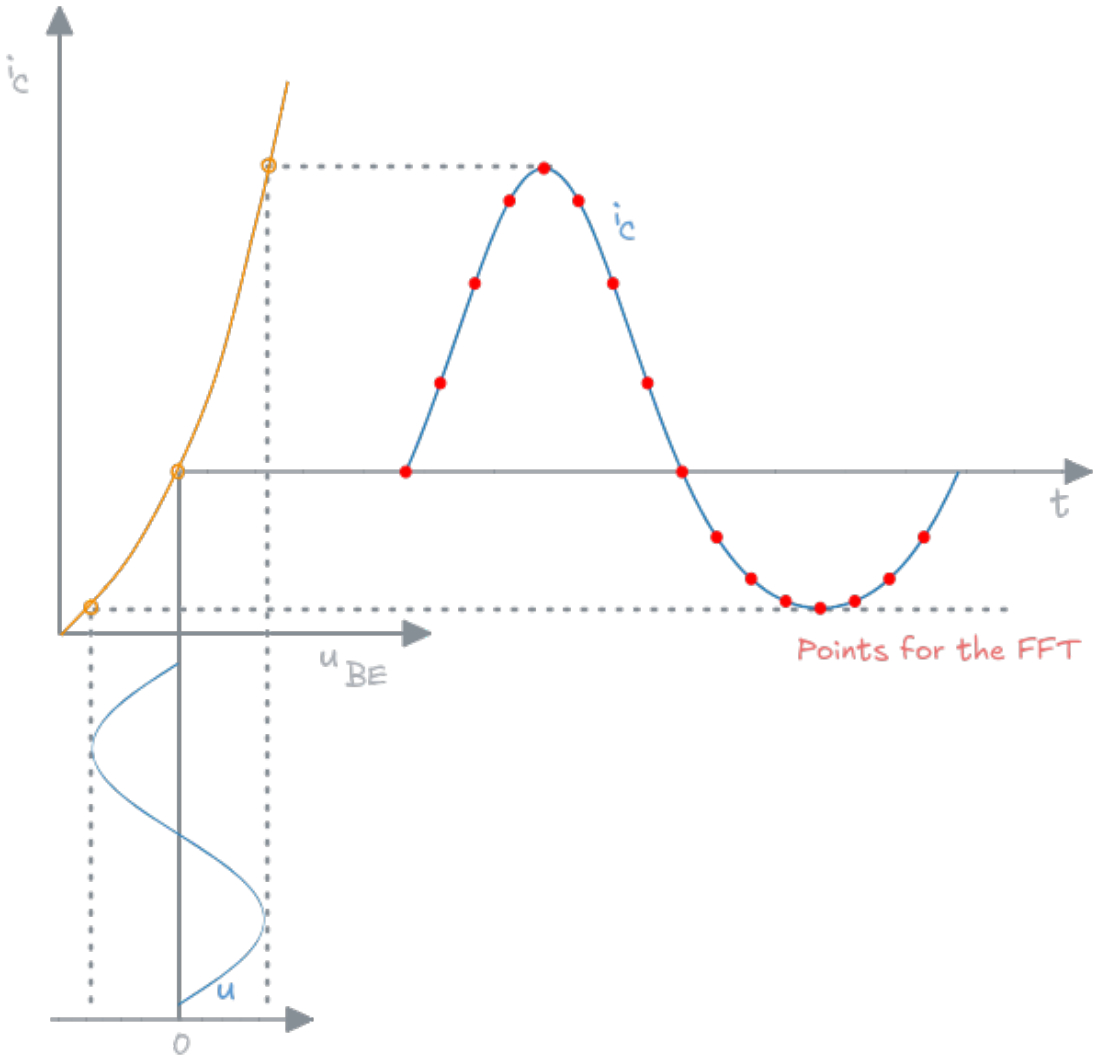


Figure 9.8: Signal transition for the electric circuit shown above

We can see, that the signal is boosted depending on the characteristic curve of the diode, i.e. the positive part of the sine more than the negative part. If we compute take the fourier transform as illustrated with the red points on the curve on the right we get from the FFT coefficients c_i:

1. the direct current component:

$$c_0 = 2.873$$

1. the effective value from Parseval's Theorem:

$$\overline{i_c} = \sqrt{\sum_{i=0}^{15} |c_i|^2} = 2.071$$

1. the total harmonic distortion (how the signal is deformed):

$$THD = \frac{\sqrt{\sum_{i=2}^{14} |c_i|^2}}{\sqrt{\sum_{i=1}^{15} |c_i|^2}} = 0.193$$

This concludes our example with the FFT.

9.5 Gabor Transform

The Fourier transform is computed as soon as all the values of a signal are sampled. It provides spectral information over the entire signal and not when in time certain frequencies occur, i.e. no temporal information. The Fourier transform can only characterize periodic and stationary signals. The time component is *used* during the integration and no longer present in the transformed signal.

If both is needed in order to generate a spectrogram (plot frequency versus time), the Gabor transform brings remedy. Technically it is the Fourier transform of the signal masked by a sliding window

$$\mathcal{G}\{f\}(t,\omega) = G_f(t,\omega) = \int_{-\infty}^{\infty} f(\tau)\mathrm{e}^{-\mathrm{i}\omega\tau}\overline{g}(\tau - t)\,\mathrm{d}\tau.$$

for the so called *kernel* $g(t)$ a Gaussian bell curve is often used

$$g(t-\tau) = \mathrm{e}^{-\frac{(t-\tau)^2}{a^2}}.$$

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

np.random.seed(6020)
# Parameters
N = 1024
a, b = 0, 1/4
```

```
t = np.linspace(a, b, N, endpoint=False)
dt = t[1] - t[0]
f1 = 50
f2 = 120
fun = lambda t: np.sin(2 * np.pi * f1 * t) + np.sin(2 * np.pi * f2 * t)

gauss = lambda x, a: np.exp(-np.pow(x, 2) / a**2)

f_clean = fun(t) + np.random.randn(len(t))
gauss = gauss(t - 0.125, 0.025) * 4.8

# Figures
plt.figure(0)
plt.plot(t, f_clean, label="Signal")
plt.plot(t, gauss, label=r"$g(t-\tau)$")
#plt.xlabel("Time [s]")
#plt.ylabel(r"$f$")
plt.xticks([0.1, 0.125, 0.15], [r"$\tau - a$", r"$\tau$", r"$\tau + a$"])
plt.xlim(t[0], t[-1])
plt.ylim(-5, 5)
plt.yticks([])
plt.legend(loc=1)
plt.gca().set_aspect(5e-3)

plt.show()
```

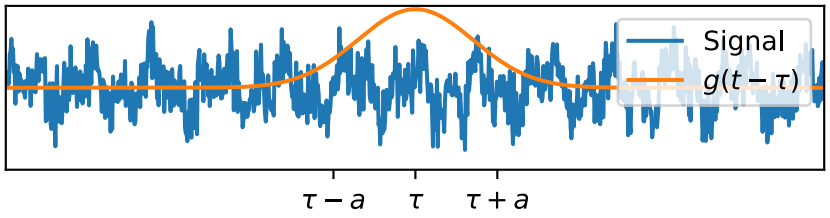


Figure 9.9: Signal with Gaussian kernel

Figure 9.9 illustrates the sliding time window with the spread as a and the center τ. The inverse is given as

$$f(t) = \mathcal{G}^{-1}\{G_f(t,\omega)\} = \frac{1}{2\pi\|g\|^2} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} G_f(\tau,\omega)\mathrm{e}^{\mathrm{i}\omega\tau}\overline{g}(t-\tau)\,\mathrm{d}\omega\,\mathrm{d}t.$$

As with the Fourier transform the application of the Gabor transform is usually done in its

discrete form.

Definition 9.8 (Discrete Gabor Transform)**.** For discrete time $\tau = k\Delta t$ and frequencies $\nu = j\Delta\omega$ the discretized kernel function has the form

$$g_{j,k} = \mathrm{e}^{\mathrm{i}2\pi j\Delta\omega t} g(t - k\Delta t)$$

and the discrete Gabor transform

$$\mathcal{G}\{f\}_{j,k} = \langle f, g_{j,k}\rangle = \int_{-\infty}^{\infty} f(\tau)\overline{g}_{j,k}(\tau)\,\mathrm{d}\tau.$$

The following example is taken from (Brunton and Kutz 2022, 77–78).

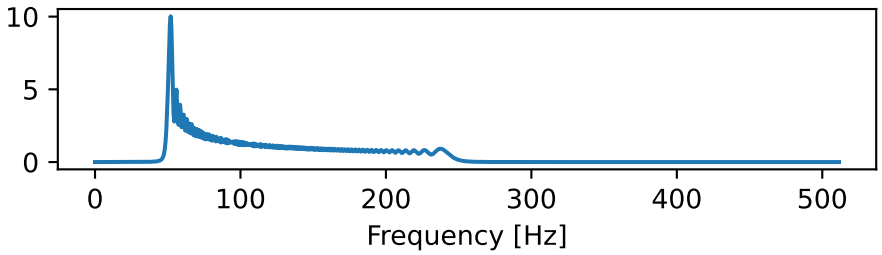
Example 9.8 (Gabor transform for a quadratic chirp)**.** As example we use an oscillating cosine function where the frequency of the oscillation increases as a quadratic function in time:

$$f(t) = \cos(2\pi\, t\, h(t)), \quad \text{with} \quad h(t) = h_0 + (h_1 - h_0)\frac{t}{3t_1^2},$$

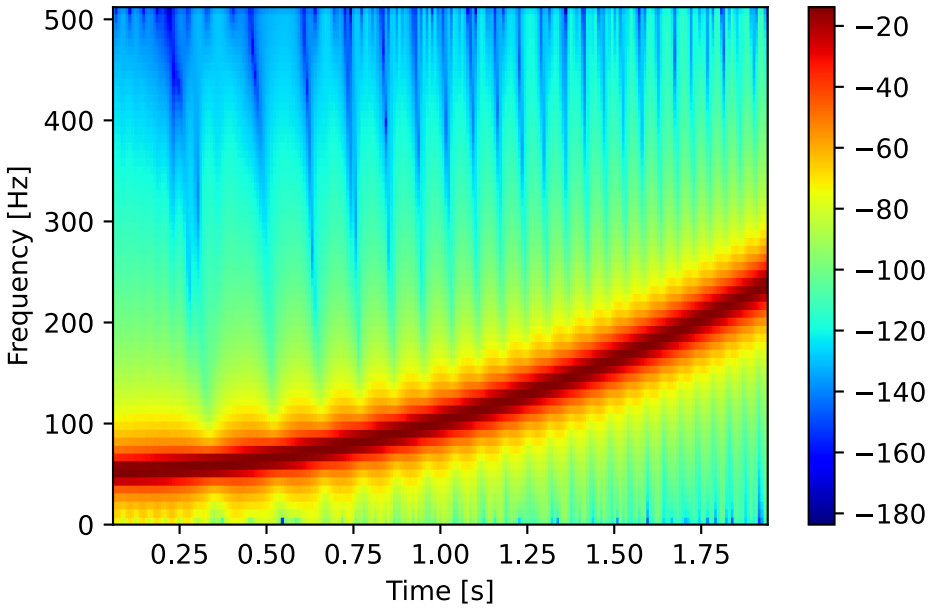
where the frequency shifts from h_0 to h_1 between $t = 0$ and $t = t_1$.
The majority of the following code is from (Brunton and Kutz 2022, Code 2.9).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
%config InlineBackend.figure_formats = ["svg"]

N = 2048
a, b = 0, 2
t = np.linspace(a, b, N, endpoint=False)
dt = t[1] - t[0]
h0 = 50
h1 = 250
t1 = 2
x = np.cos(2 * np.pi * t * (h0 + (h1 - h0) * np.power(t,2) / (3 * t1**2)))

plt.figure(0)
N = len(x)
freq = (1 / (dt * N)) * np.arange(N)
PSD_noise = np.abs(np.fft.fft(x))**2 / N
plt.plot(freq[:N//2], PSD_noise[:N//2])
plt.xlabel("Frequency [Hz]")
```

```
plt.gca().set_aspect(10)

plt.figure(1)
plt.specgram(x, NFFT=128, Fs=1/dt, noverlap=120, cmap='jet')
plt.colorbar()
plt.xlabel("Time [s]")
plt.ylabel("Frequency [Hz]")
plt.show()
```



(a) Power spectral density of the signal.



(b) Spectrogram.

Figure 9.10: Quadratic chirp signal

We can see a clear peak at 50Hz but no information of time is given, where else in the

spectrogram we see how the frequency progresses in time.

> **Note**
>
> For the analysis of time-frequency we see Heisenberg's uncertainty principle at work. We can not attain high resolution simultaneously in both time and frequency domain. In the time domain we have perfect resolution but no information about the frequencies, in the Fourier domain the time information is not present.
> The spectrogram resolves both but with reduced resolution. This effect manifests differently at different locations of the chirp signal. At the left where the frequency is almost constant, the band becomes narrower, whereas at the right it becomes wider due to the increased blur in horizontal direction. The product of the uncertainties of these quantities has a minimal value. In this context time and frequency domain respectively represent the extremes.

The Fourier transform always assumes a harmonic nature in the signal it is applied to. Therefore, it is best suited for e.g. music, rotating machines or vibrations. For other signals, especially with discontinuous parts (Gibbs phenomenon) there are more adequate bases functions. We look at these in the next section Chapter 11 but first we look at the Laplace transform Chapter 10.

10 Laplace Transform

The Fourier Transform is only possible for well-behaved functions that are Lebesgue integrable, i.e. $f(x)\ inL^1[(-\infty,\infty)]$. This excludes functions like the exponential $\mathrm{e}^{\lambda t}$ or the Heaviside function.

In order to tackle these *badly behaved* functions we can use the Laplace transform, which is basically a one sided Fourier-like transform.

Definition 10.1 (The Laplace Transform)**.** A function $f : [0,\infty) \to \mathbb{R}$ is called laplace transposable if

$$F(s) = \mathcal{L}\{f(t)\} := \int_0^\infty f(t)\mathrm{e}^{-st}\,\mathrm{d}t$$

exists for all $s \in \mathrm{R}$. In this case we call $F(s) \equiv \mathcal{L}\{f(t)\}$ the **Laplace transform** of $f(t)$. The **inverse Laplace transform** is defined as

$$f(t) = \mathcal{L}^{-1}\{F(s)\} = \frac{1}{s\pi\mathrm{i}}\int_{\gamma-\mathrm{i}\infty}^{\gamma+\mathrm{i}\infty} F(x)\mathrm{e}^{st}\,\mathrm{d}s$$

for the complex valued $s = \gamma + \mathrm{i}\omega$.
It is quite common to use the capitalized function name for the Laplace transform, as we did here.

To give a better understanding of theses formula we follow the *derivation* from (Brunton and Kutz 2022, sec. 2.5).

Let us consider the function $f(t) = \mathrm{e}^{\lambda t}$. As mentioned before we can not directly use the Fourier transform as the function is not bounded. In order to trap the function for $t \to \infty$ we multiply by $\mathrm{e}^{-\gamma t}$ where γ is damping more that f grows. In order to handle $t \to -\infty$ we multiply by the Heaviside function

$$H(t) = \begin{cases} 0, & t \le 0, \\ 1, & t > 0, \end{cases}$$

and transform it into a *one-sided* function as in definition Definition 10.1 required. We end up with the function

$$\underline{f}(t) = f(t)\mathrm{e}^{-\gamma t}H(t) = \begin{cases} 0, & t \le 0, \\ f(t)\mathrm{e}^{-\gamma t}, & t > 0. \end{cases}$$

Taking the Fourier transform we get

$$\begin{aligned}\hat{\underline{f}}(\omega) = \mathcal{F}\{\underline{f}(t)\} &= \int_{-\infty}^{\infty} \underline{f}(t)\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t = \int_{0}^{\infty} f(t)\mathrm{e}^{-\gamma t}\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t = \\ &= \int_{0}^{\infty} f(t)\mathrm{e}^{-(\gamma+\omega)t}\,\mathrm{d}t = \int_{0}^{\infty} f(t)\mathrm{e}^{-st}\,\mathrm{d}t = \\ &= \mathcal{L}\{f(t)\} = F(s)\end{aligned}$$

giving rise to the Laplace transform.

To get the inverse we start with the inverse Fourier transform of $\hat{\underline{f}}(\omega)$

$$\underline{f}(t) = \mathcal{F}^{-1}\{\hat{\underline{f}}(\omega)\} = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{\underline{f}}(\omega)\mathrm{e}^{\mathrm{i}\omega t}\,\mathrm{d}\omega.$$

Multiply both sides with $\mathrm{e}^{\gamma t}$ we get

$$\begin{aligned}f(t)H(t) = \underline{f}(t)\mathrm{e}^{\gamma t} &= \frac{1}{2\pi}\int_{-\infty}^{\infty} \mathrm{e}^{\gamma t}\hat{\underline{f}}(\omega)\mathrm{e}^{\mathrm{i}\omega t}\,\mathrm{d}\omega = \\ &= \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{\underline{f}}(\omega)\mathrm{e}^{(\gamma+\mathrm{i}\omega)t}\,\mathrm{d}\omega\end{aligned}$$

By a change of variables $s = \gamma + \mathrm{i}\omega$ we get $\mathrm{d}\omega = \frac{1}{\mathrm{i}}\mathrm{d}s$ and

$$f(t)H(t) = \frac{1}{s\pi\mathrm{i}}\int_{\gamma-\mathrm{i}\infty}^{\gamma+\mathrm{i}\infty} F(x)\mathrm{e}^{st}\,\mathrm{d}s = \mathcal{L}^{-1}\{F(s)\}.$$

> **Note**
>
> Form this derivation we see that the Laplace transform is a generalized Fourier transform for a broader spectrum of functions.
> Sometimes the Laplace transform is simpler than the Fourier transform. In particular this is the case for the Dirac delta function that has infinitely many frequencies in Fourier domain but is constant 1 in Laplace domain. We will use this to compute the impulse response of systems with *forcing*.

A lot of the properties we saw for the Fourier transform carry over to the Laplace transform.

1. **Linearity**

$$\mathcal{L}\{\alpha f(t) + \beta g(t)\} = \alpha\mathcal{L}\{f(t)\} + \beta\mathcal{L}\{g(t)\} = \alpha F(s) + \beta G(s).$$

2. **Conjugation**

$$\mathcal{L}\{\overline{f(t)}\} = \overline{\hat{f}(\overline{s})}.$$

3. **Scaling**, for $\alpha \geq 0$

$$\mathcal{L}\{f(\alpha t)\} = \frac{1}{\alpha} F\left(\frac{s}{\alpha}\right).$$

4. **Drift in time**, for $a \in \mathbb{R}$

$$\mathcal{L}\{f(t-a)H(t-a)\} = \mathrm{e}^{-as} F(s)$$

and

$$\mathcal{L}\{f(t)H(t-a)\} = \mathrm{e}^{-as}\mathcal{L}\{f(t+a)\}.$$

5. **Drift in frequency**, for $a \in \mathbb{R}$

$$\mathrm{e}^{at}\mathcal{L}\{f(t)\} = F(s-a).$$

6. **Derivative in time**

$$\mathcal{L}\{\partial_t f(t)\} = sF(s) - f(0)$$

We are going to prove this by going through the lines

$$\begin{aligned}\mathcal{L}\left\{\frac{d}{d\,t} f(t)\right\} &= \int_0^\infty f'(t)\mathrm{e}^{-st}\,\mathrm{d}t \\ &= [f(t)\mathrm{e}^{-st}]_0^\infty - \int_0^\infty f(t)(-s\mathrm{e}^{-st})\,\mathrm{d}t \\ &= -f(0) + s\mathcal{L}\{f(t)\}\end{aligned}$$

For higher derivatives we get

$$\mathcal{L}\{\partial_t^n f(t)\} = s^n F(s) - \sum_{k=1}^{n} s^{n-k} f^{(k-1)}(0).$$

7. **Integral in time**

$$\mathcal{L}\left\{\int_0^t f(\tau)\,\mathrm{d}\tau\right\} = \frac{1}{s} F(s).$$

8. **Derivative in frequency**

$$\mathcal{L}\{t^n f(t)\} = (-1)^n \partial_\omega^n F(s)$$

9. **Integral in frequency**

$$\mathcal{L}\left\{\frac{1}{t} f(t)\right\} = \int_s^\infty F(u)\,\mathrm{d}u.$$

10. The **convolution** of two functions is defined as

$$(f * g)(t) = \int_0^\infty f(\tau) g(t-\tau)\,\mathrm{d}\tau,$$

and for the Laplace transform

$$\mathcal{L}\{(f * g)(t)\} = F(s) \cdot G(s).$$

! Important

The following examples are mostly from Meyberg and Vachenauer (1992).

Example 10.1 (Laplace transform of some basic functions)**.**

1. For the constant $f(t) = 1$ function we get

$$\mathcal{L}\{f(t)\} = \frac{1}{s}, \quad s > 0.$$

2. For the exponential $f(t) = \mathrm{e}^{\lambda t}$ we get

$$\mathcal{L}\{f(t)\} = \frac{1}{s-\lambda}, \quad s > \lambda \in \mathcal{R}.$$

3. For the cosine $f(t) = \cos \omega t$ we get

$$\mathcal{L}\{f(t)\} = \frac{s}{s^2+\omega^2}, \quad s > 0.$$

4. For the sine $f(t) = \sin \omega t$ we get

$$\mathcal{L}\{f(t)\} = \frac{\omega}{s^2+\omega^2}, \quad s > 0.$$

Example 10.2 (Laplace transform of a step function)**.** For a rectangle with width $a \geq 0$ and height 1 we get

$$f(t) = H(t) - H(t-a)$$

we get

$$\mathcal{L}\{f(t)\} = \frac{1-\mathrm{e}^{-as}}{s}$$

We can *extend* this to a step function

$$g(t) = \sum_{n=0}^{\infty} H(t - na)$$

and we get

$$\mathcal{L}\{g(t)\} = \frac{1}{s(1 - \mathrm{e}^{-as})}$$

Example 10.3 (Cam drive)**.** A cam drive has an elastic and lightly damped punch S with eigenfrequency $\omega = 2$.

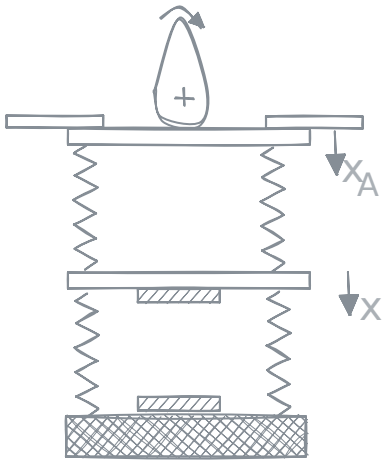


Figure 10.1: Cam drive schematics

We excite the system with a half-wave sine of half the eigenfrequency.

$$\begin{aligned} x_A(t) &= \frac{1}{2}(\sin t + |\sin t|) \\ &= \begin{cases} \sin(t), & 2n\pi \le t \le (2n+1)\pi, \\ 0, & (2n+1)\pi \le t \le (2n+2)\pi, \end{cases} \qquad \text{for} \quad n = 0, 1, 2, ... \\ &= \sum_{n=0}^{\infty} H(t - n\pi)\sin(t - n\pi). \end{aligned}$$

For the excitement of S the initial value problem

$$\ddot{x} + 4x = x_A(t), \quad x(0) = \dot{x}(0) = 0,$$

is given.

Via the Laplace transform we can solve for $X(s)$ in

$$s^2X(s) + 4X(s) = \sum_{n=0}^{\infty} \frac{1}{1+s^2} \mathrm{e}^{-n\pi s}$$

and obtain

$$X(s) = \frac{1}{(1+s^2)(s^2+4)} \sum_{n=0}^{\infty} \mathrm{e}^{-n\pi s}.$$

Via the inverse Laplace transform we can get back to $x(t)$

$$x(t) = \frac{1}{6}(\sin(t) + |\sin(t)|) - \frac{\sin(2t)}{6} \sum_{0}^{\infty} H(t - n\pi)$$

and without the heavy side function

$$x(t) = \begin{cases} \frac{1}{3}\sin(t) - \frac{2n+1}{6}\sin(2t), & 2n\pi \le t \le (2n+1)\pi, \\ -\frac{n+1}{3}\sin(2t), & (2n+1)\pi \le t \le (2n+2)\pi. \end{cases}$$

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

heavy_side = lambda t: np.ones_like(t) * (t>=0)
g = lambda n, t: (np.heaviside(t - 2 * n * np.pi, 1) - \
                  np.heaviside(t - (2 * n + 1) * np.pi, 1))
h = lambda n, t: (np.heaviside(t - (2 * n + 1) * np.pi, 1) - \
                  np.heaviside(t - (2 * n + 2) * np.pi, 1))

xa = lambda t: 1 / 2 * (np.sin(t) + np.abs(np.sin(t)))

fun1 = lambda n, t: (1 / 3 * np.sin(t) - (2 * n + 1) / 6 * \
                     np.sin(2 * t)) * g(n, t)
fun2 = lambda n, t:-1 * (n + 1) / 3 * np.sin(2 * t) * h(n, t)
funn = lambda n, t: fun1(n, t) + fun2(n, t)
fun = lambda t: funn(0, t) + funn(1, t) + funn(2, t) + funn(3, t)

t = np.linspace(0, 8 * np.pi, 1024)
i = np.argmax(funn(0, t))
j = np.argmax(funn(3, t))
k = (funn(3, t[j]) - funn(0, t[i])) / (t[j] - t[i])
line = lambda t: k * (t - t[i]) + funn(0, t[i])
```

```
plt.plot(t, xa(t), "--", label=r"$x_A$")
plt.plot(t, fun(t), label=r"$x$")
plt.plot(t, line(t), "k:")
plt.xlabel(r"t")
plt.ylabel(r"x")
plt.legend()
plt.gca().set_aspect(8*np.pi / (3 * 3.0))
plt.ylim([-1.5, 1.5])
```

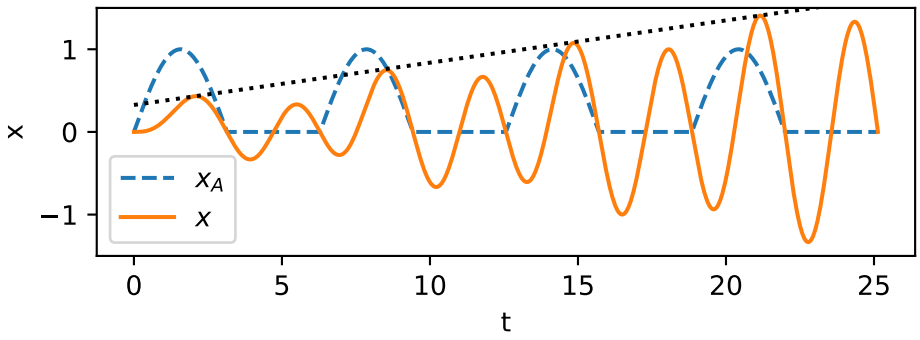


Figure 10.2: Exciting function and response, we indicate the growth rate as a dotted line.

Example 10.4 (Electric Circuit)**.** For the electric components R (resistor), C capacitor, and L coil we can use the Laplace transform to show how voltage $u(t)$ and current $i(t)$ is transformed.
We need to assume that $i(0) = 0$ than we get

Table 10.1: Laplace transform for common electrical components

component	$u(t)$ and $i(t)$	Laplace transform
R	$u_R(t) = Ri(t)$	$U_r(s) = RI(s)$
C	$u_C(t) = \frac{1}{x}\int_0^t i(\tau)\,\mathrm{d}\tau$	$U_C(s) = \frac{1}{sC}I(s)$
L	$u_L(t) = L\frac{\mathrm{d}i}{\mathrm{d}t}$	$U_L(s) = sLI(s)$

An Ohm's law applies for

$$U(s) = Z(s)I(s), \quad Z(s) \in \{R, \frac{1}{Cs}, Ls\}$$

For a RCL electric circuit

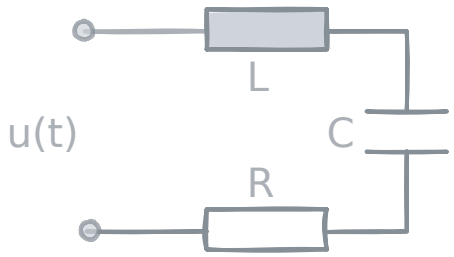


Figure 10.3: RCL oscillating circuit

we have

$$u(t) = u_R(t) + u_c(t) + u_L(t)$$

and we get for the Laplace transformed current $I(s)$

$$I(s) = \frac{U(s)}{R + \frac{1}{sC} + sL} = \frac{sCU(s)}{s^2LC + sRC + 1} = Z(s)U(s)$$

and

$$i(t) = \mathcal{L}^{-1}\{z(t)\} * u(t).$$

Example 10.5 (The Linear Time Invariant transfer function of LTI Systems)**.** In signal and and control theory the **transfer function** $H(s)$ is used the describe a system. If we assume that at $t = 0$ we have an initial state (zero state) we get the relation between input signal $x(t)$ and output signal $y(t)$ in the Laplace space as

$$Y(s) = H(s)Y(s)$$

and consequently

$$y(t) = h(t) * x(t)$$

where $h(t) = \mathcal{L}^{-1}\{H(s)\}$ is called the **impulse response**.

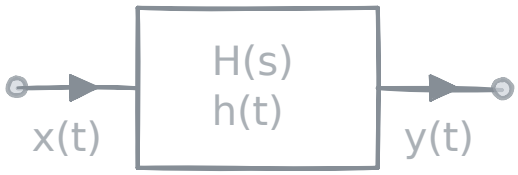


Figure 10.4: Linear Time invariant System

The relation between x and y can often be described via a linear differential equation with constant coefficients. In these cases

$$\alpha_n y^{(n)} + \alpha_{n-1} y^{(n-1)} + \cdots + \alpha_1 \dot{y} + \alpha_0 y = x(t)$$

and for $y(0) = \dot{y}(0) = \cdots = y^{(n)}(0) = 0$ we get

$$H(s) = \frac{1}{\alpha_n s^n + \alpha_{n-1} s^{n-1} + \cdots \alpha_1 s + \alpha_0}.$$

We can rewrite this as

$$\alpha_n (sH(s) - \frac{1}{\alpha_n}) + \alpha_{n-1} s^{n-1} H(s) + \cdots + \alpha_1 sH(s) + \alpha_0 H(s) = 0$$

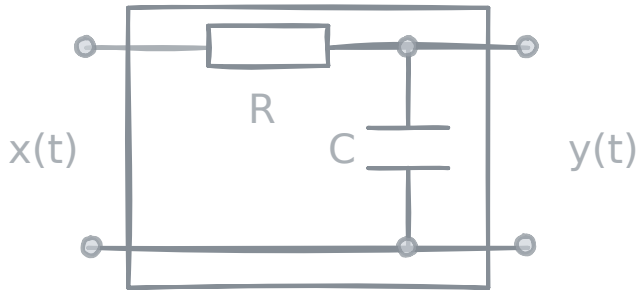
and with the properties of the Laplace transform we get

$$\alpha_n h^{(n)} + \alpha_{n-1} h^{(n-1)} + \cdots + \alpha_1 \dot{h} + \alpha_0 h = 0$$

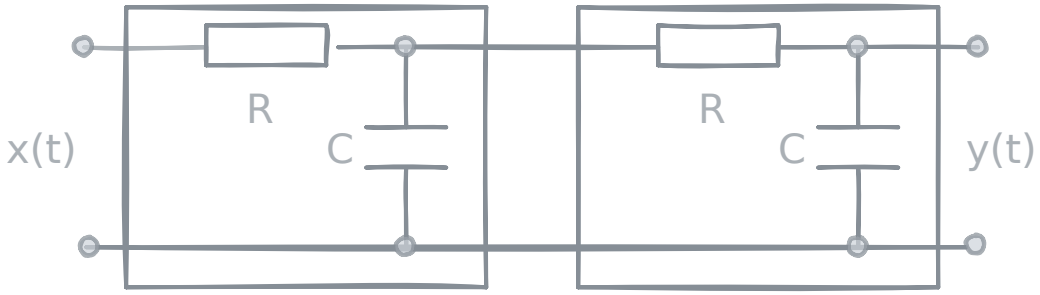
with $h(0) = \dot{h}(0) = \cdots = h^{(n-2)}(0) = 0$, and $h^{(n-1)}(0) \frac{1}{\alpha_n}$.
This tells us the *answer* to the zero input signal $x(t) = 0$ is the solution of above initial value problem with initial state as an impulse of size $\frac{1}{\alpha_n}$.
In terms of the Dirac-delta function $\delta(t)$ the function $h(t)$ is the answer to the initial value problem

$$\alpha_n h^{(n-1)} + \alpha_{n-1} h^{(n-1)} + \cdots + \alpha_1 \dot{h} + \alpha_0 h = \delta(t).$$

Example 10.6 (The RC low pass filter)**.** The RC low pass filter



(a) First order RC low pass



(b) Second order RC low pass

Figure 10.5: The electric circuit for the low pass filters.

has the transfer function

$$H(s) = \frac{1}{sRC + 1}$$

and the impulse response

$$h(t) = \frac{1}{RC} \mathrm{e}^{-\frac{t}{RC}}.$$

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

RC = 1
fun = lambda t: 1 / RC * np.exp(-t / RC) * np.heaviside(t, 1)
t = np.linspace(-0.5, 8.5, 1024)

plt.plot(t, fun(t))
plt.text(-0.4, 0.96, r"$\frac{1}{RC}$")
plt.xlabel(r"t")
```

```
plt.ylabel(r"y")
plt.gca().set_aspect(9 / (3 * 1.0))
```

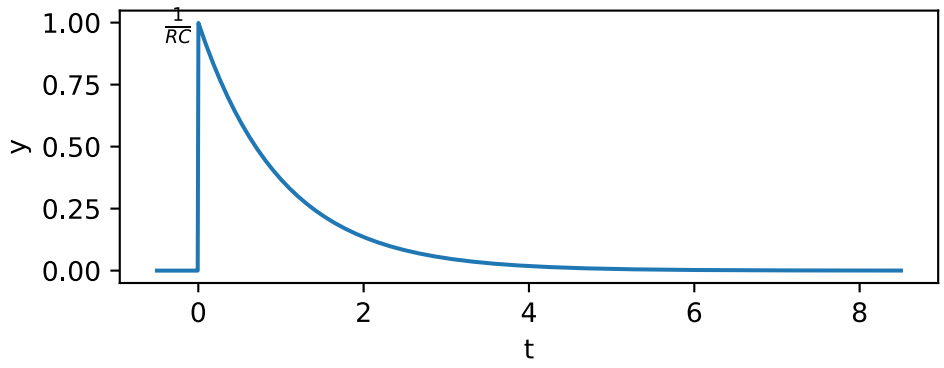


Figure 10.6: Impulse response for the first order RC low pass filter

If we extend this to a second order low pass filter Figure 10.5b we get

$$H(s) = \frac{1}{(sRC+1)^2}$$

and the impulse response

$$h(t) = \frac{1}{(RC)^2} t \mathrm{e}^{-\frac{t}{RC}}.$$

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

RC = 1
fun = lambda t: 1 / RC**2 * t * np.exp(-t / RC) * np.heaviside(t, 1)
t = np.linspace(-0.5, 8.5, 1024)

plt.plot(t, fun(t))
plt.text(-0.5, 0.95/(np.exp(1)), r"$\frac{1}{RC \mathrm{e}}$")
plt.text(0.8, -0.05, r"$RC$")
plt.plot([-0.1, 1, 1], [1/np.exp(1), 1/np.exp(1), 0], "k:")
plt.xlabel(r"t")
plt.ylabel(r"y")
plt.gca().set_aspect(9 / (3 * 0.37))
```

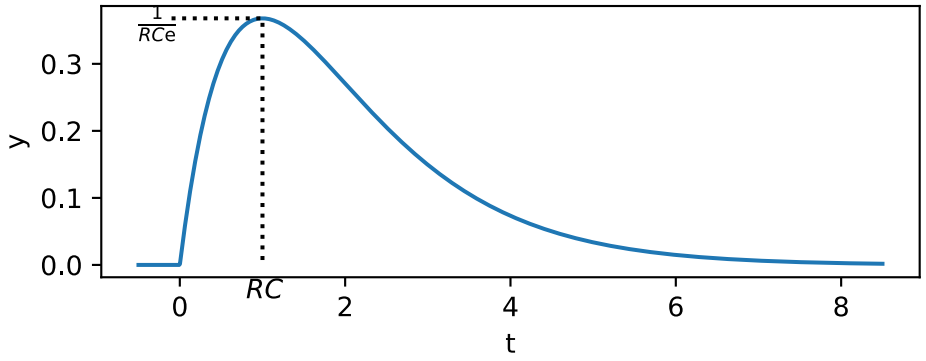


Figure 10.7: Impulse response for the first order RC low pass filter

11 Wavelet transform

With Wavelets we extend the concept of the Fourier analysis to general orthogonal bases. This extensions is done in such a way that we can do a multi-resolution decomposition and thus partially overcome the uncertainty principal discussed before.

Wavelets are both, local and orthogonal. The whole family of a wavelet are generated by scaling and translating a *mother* wavelet $\psi(t)$ as

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right),$$

where the parameters a and b are responsible for scaling and translating, respectively.

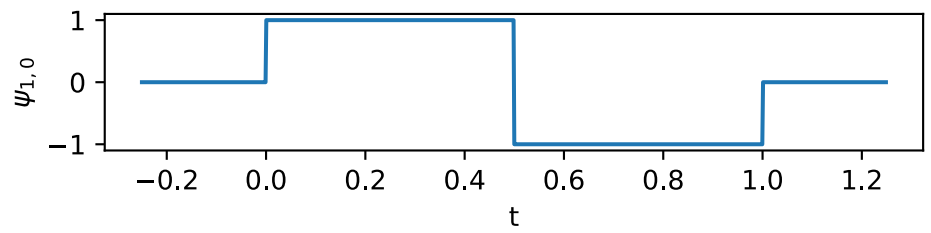
The simplest example is the so called Haar wavelet.

Example 11.1 (The Haar wavelet)**.** The *mother* wavelet is defined as the step function

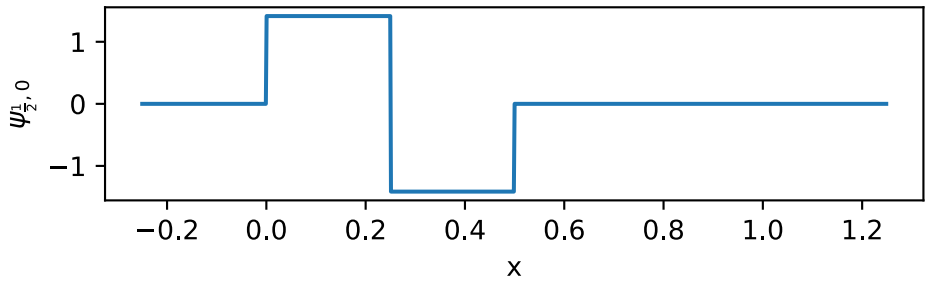
$$\psi(t) = \begin{cases} 1, & \text{for } 0 \le t \le \frac{1}{2} \\ -1, & \text{for } \frac{1}{2} \le t \le 1 \\ 0, & \text{else} \end{cases}$$

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
%config InlineBackend.figure_formats = ["svg"]

N = 1000
a, b = -0.25, 1.25
t = np.linspace(a, b, N, endpoint=False)
mother = lambda t: np.heaviside(t, 1) - np.heaviside(t - 1/2, 1) - \
                   (np.heaviside(t - 1/2, 1) - np.heaviside(t - 1, 1))
psi = lambda t, a, b: 1 / np.sqrt(a) * mother((t - b) / a)

plt.figure(0)
plt.plot(t, psi(t, 1, 0))
plt.xlabel("t")
plt.ylabel(r"$\psi_{1,0}$")
```

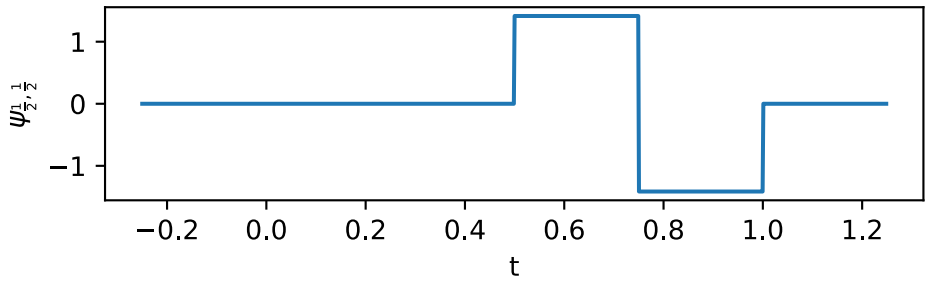
```
plt.gca().set_aspect(0.125)

plt.figure(1)
plt.plot(t, psi(t, 1/2, 0))
plt.xlabel("x")
plt.ylabel(r"$\psi_{\frac{1}{2}, 0}$")
plt.gca().set_aspect(0.125)

plt.figure(2)
plt.plot(t, psi(t, 1/2, 1/2))
plt.xlabel("t")
plt.ylabel(r"$\psi_{\frac{1}{2}, \frac{1}{2}}$")
plt.gca().set_aspect(0.125)

plt.show()
```



(a) Scaling 1 and translation 0.



(b) Scaling 1/2 and translation 0.



(c) Scaling 1/2 and translation 1/2.

Figure 11.1: Haar wavelets for the first two levels of multi resolution.

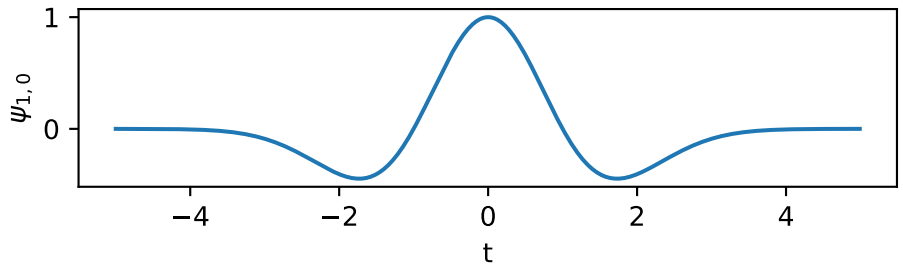
Note that the Haar wavelets are orthogonal and provide a hierarchical basis for a signal.

Example 11.2 (The Maxican hat wavelet)**.** The *mother* wavelet is defined as the is the negative normalized second derivative of a Gaussian function,

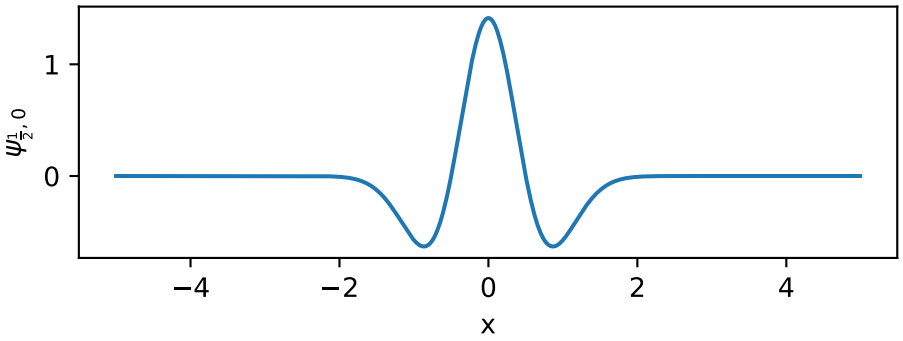
$$\psi(t) = (1-t)^2 \, \mathrm{e}^{-\frac{t}{2}}$$

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

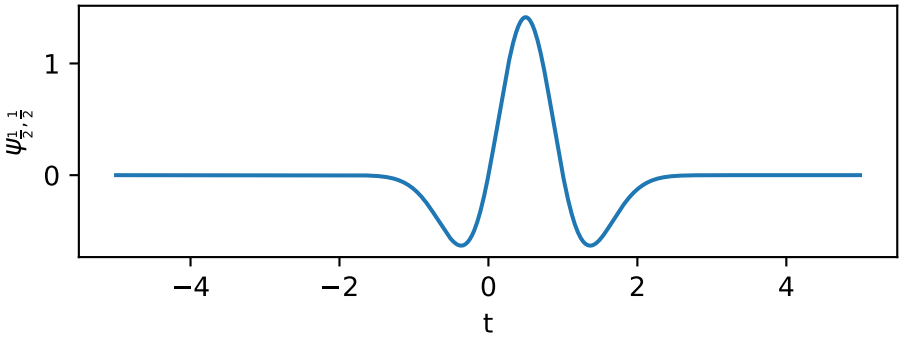
```
from scipy.integrate import odeint
%config InlineBackend.figure_formats = ["svg"]

N = 1000
a, b = -5, 5
t = np.linspace(a, b, N, endpoint=False)
mother = lambda t, d: (1 - np.pow(t, 2)) * np.exp(-1/2 * np.pow(t, 2))
psi = lambda t, a, b: 1 / np.sqrt(a) * mother((t - b) / a, 2)

plt.figure(0)
plt.plot(t, psi(t, 1, 0))
plt.xlabel("t")
plt.ylabel(r"$\psi_{1,0}$")
plt.gca().set_aspect(1.5)

plt.figure(1)
plt.plot(t, psi(t, 1/2, 0))
plt.xlabel("x")
plt.ylabel(r"$\psi_{\frac{1}{2}, 0}$")
plt.gca().set_aspect(1.5)

plt.figure(2)
plt.plot(t, psi(t, 1/2, 1/2))
plt.xlabel("t")
plt.ylabel(r"$\psi_{\frac{1}{2}, \frac{1}{2}}$")
plt.gca().set_aspect(1.5)

plt.show()
```



(a) Scaling 1 and translation 0.



(b) Scaling 1/2 and translation 0.



(c) Scaling 1/2 and translation 1/2.

Figure 11.2: Mexican hat wavelets for the first two levels of multi resolution.

If we have a wavelet ψ, we can generate a new wavelet by convolution

$$\psi * \phi$$

, for a bounded and integrable function ϕ.

Definition 11.1 (Continuous Wavelet Transform)**.** The continuous wavelet transform is given by

$$\mathcal{W}_\psi\{f\}(a,b) = \langle f, \psi_{a,b}\rangle = \int_{-\infty}^{\infty} f(t)\overline{\psi}_{a,b}\,\mathrm{d}t,$$

this is only true for a bounded wavelet ψ

$$C_\psi = \int_{-\infty}^{\infty} \frac{|\hat{\psi}(\tau)|^2}{|\tau|}\,\mathrm{d}\tau,$$

i.e. a wavelet with $C_\psi < \infty$. In this case also the inverse transform exists and is defined as

$$f(t) = \frac{1}{C_\psi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathcal{W}_\psi\{f\}(a,b)\psi_{a,b}(t)\frac{1}{a^2}\,\mathrm{d}a\,\mathrm{d}b,$$

From the continuous wavelet transform we go on to the discrete wavelet transform as similar for the Fourier transforms we have seen we will hardly ever have the entire signal at hand for the transformation.

Definition 11.2 (Discrete Wavelet Transform)**.** The discrete wavelet transform is given by

$$\mathcal{W}_\psi\{f\}(j,k) = \langle f, \psi_{j,k}\rangle = \int_{-\infty}^{\infty} f(t)\overline{\psi}_{j,k}\,\mathrm{d}t,$$

where $\psi_{j,k}$ is a discrete family of wavelets

$$\psi_{j,k}(t) = \frac{1}{a^j}\psi\left(\frac{t-kb}{a^j}\right).$$

The inverse is than given by

$$f(t) = \sum_{j,k=-\infty}^{\infty} \mathcal{W}_\psi\{f\}(j,k)\psi_{j,k}(t).$$

Which is nothing else than expressing the function in the wavelet family. If this family of wavelets is orthogonal (as e.g. the Haar wavelet) it is possible to expand the function f uniquely as they form a basis.

i Note

There also exists a *fast* wavelet transform that reduces the computational complexity from $\mathcal{O}(N\log N)$ to $\mathcal{O}(N)$ by cleverly reusing parts of the inner product computation.

Example 11.3 (Signal analysis with the Haar wavelet)**.** To start and get an idea how the analysis works we use an instructive example. We have a piecewise constant function as $v = [3, 1, 0, 4, 0, 6, 9, 9]^\mathsf{T}$

$$f(x) = \sum_{i=1}^{8} v_i \chi_{[i-1,i)},$$

where $\chi_{[c,d)}$ is the indicator function that is 1 on the interval $[c, d)$ and zero everywhere else.
Now let us proceed through the levels of the transform to see what is happening, we select $a = 2$ and $b = 1$:

$$\mathcal{W}\{f\}(1,:) = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \begin{bmatrix} 3 \\ 1 \\ 0 \\ 4 \\ 0 \\ 6 \\ 9 \\ 9 \end{bmatrix} = \begin{bmatrix} 2\sqrt{2} \\ 2\sqrt{2} \\ 3\sqrt{2} \\ 9\sqrt{2} \\ \sqrt{2} \\ -2\sqrt{2} \\ -3\sqrt{2} \\ 0 \end{bmatrix}$$

We than apply the next level just for the first half

$$\mathcal{W}\{f\}(2,:) = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} 2\sqrt{2} \\ 2\sqrt{2} \\ 3\sqrt{2} \\ 9\sqrt{2} \\ \sqrt{2} \\ -2\sqrt{2} \\ -3\sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 12 \\ 0 \\ -6 \\ \sqrt{2} \\ -2\sqrt{2} \\ -3\sqrt{2} \\ 0 \end{bmatrix}$$

and our final step

$$\mathcal{W}\{f\}(3,:) = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} 4 \\ 12 \\ 0 \\ -6 \\ \sqrt{2} \\ -2\sqrt{2} \\ -3\sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 8\sqrt{2} \\ -4\sqrt{2} \\ 0 \\ -6 \\ \sqrt{2} \\ -2\sqrt{2} \\ -3\sqrt{2} \\ 0 \end{bmatrix}$$

```
import numpy as np
import matplotlib.pyplot as plt
import pywt
%config InlineBackend.figure_formats = ["svg"]


N = 1000
a, b = 0, 8
t = np.linspace(a, b, N, endpoint=False)
v = np.array([3, 1, 0, 4, 0, 6, 9, 9])
chi = lambda t, a, b: np.heaviside(t - a, 1) - np.heaviside(t - b, 1)
mother = lambda t: chi(t, 0, 1/2) - chi(1/2, 1)
psi = lambda t, a, b: 1 / np.sqrt(a) * mother((t - b) / a, 2)

def f(t, v, spread=1):
    y = np.zeros(t.shape)
    for i, x in enumerate(v):
        y += x * chi(t, i * spread, (i + 1) * spread)
    return y

X = np.zeros((4, len(v)))
for i in range(0, 4):
    x = pywt.wavedec(v, wavelet="Haar", level=i)
    X[i, :] = np.concatenate(x)

plt.figure(0)
plt.plot(t, f(t, v), label="Signal")
#plt.plot(t, f(t, X[0, :], 1))
plt.plot(t, f(t, X[1, 0:5], 2), label="Level 1")
plt.plot(t, f(t, X[2, 0:3], 4), label="Level 2")
plt.plot(t, f(t, X[3, 0:1], 8), label="Level 3")
plt.xlabel("t")
plt.legend()
plt.show()
```

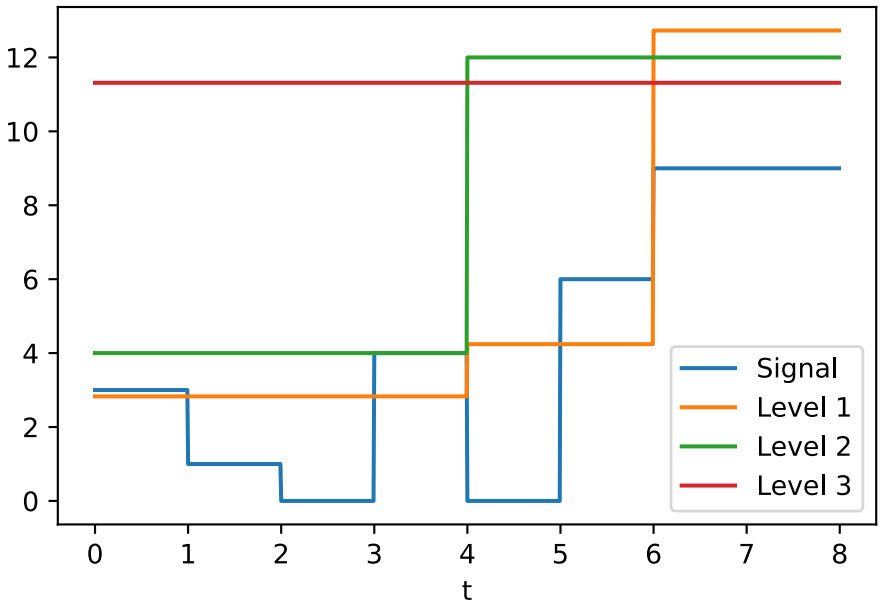


Figure 11.3: Haar wavelets for the first three levels of multi resolution analysis.

12 Two-Dimensional Transform

There is nothing preventing us from extending the transforms we discussed before to 2D. One of the most common applications are image processing.

12.1 Fourier

If we apply FFT to a matrix $X \in \mathrm{R}^{m \times n}$ we can simply apply the 1D version to every row and than to every column of the resulting matrix. The other way round will produce the same final result.

This is shown in the code below but note we should use the more efficient `np.fft.fft2`.

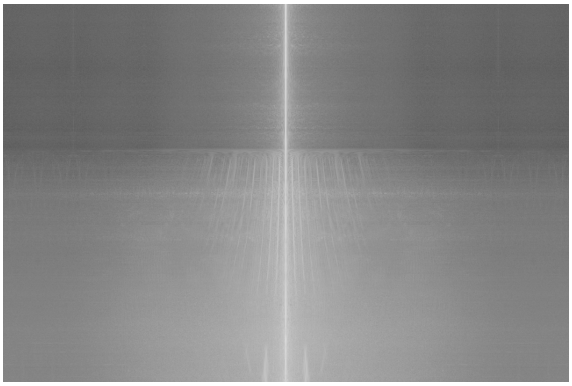
```
import matplotlib.pyplot as plt
import imageio.v3 as iio
import numpy as np
%config InlineBackend.figure_formats = ['svg']

im = np.asarray(iio.imread(
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

def myplot(A):
    plt.figure()
    plt.imshow(A, cmap="gray")
    plt.axis("off")
    plt.gca().set_aspect(1)

A = rgb2gray(im)

Cshift = np.zeros_like(A, dtype='complex')
C = np.zeros_like(A, dtype='complex')
for j in range(A.shape[0]):
```

```
    C[j, :] = np.fft.fft(A[j, :])
    Cshift[j,:] = np.fft.fftshift(np.copy(C[j, :]))

Rshift = np.zeros_like(A, dtype='complex')
R = np.zeros_like(A, dtype='complex')
D = np.zeros_like(C)
for j in range(A.shape[1]):
    R[:, j] = np.fft.fft(A[:, j])
    Rshift[:, j] = np.fft.fftshift(np.copy(R[:, j]))
    D[:, j] = np.fft.fft(C[:, j])

myplot(A)
myplot(np.log(np.abs(Cshift)))
myplot(np.log(np.abs(Rshift)))
myplot(np.fft.fftshift(np.log(np.abs(D))))
```

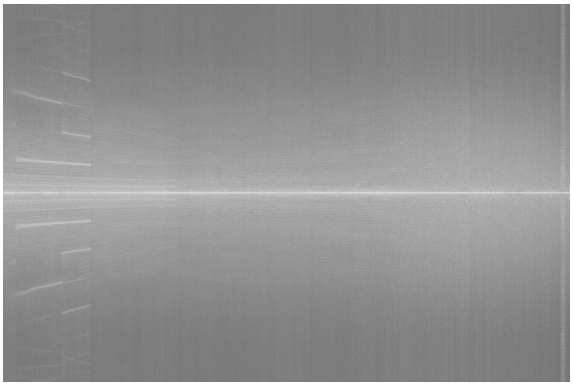
Of course we can use this to compress the image by removing small values from the transform.

```
import matplotlib.pyplot as plt
import imageio.v3 as iio
import numpy as np
%config InlineBackend.figure_formats = ['svg']

im = np.asarray(iio.imread(
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

def myplot(A):
    plt.figure()
    plt.imshow(A, cmap="gray")
    plt.axis("off")
    plt.gca().set_aspect(1)

A = rgb2gray(im)
A_fft = np.fft.fft2(A)
A_fft_sort = np.sort(np.abs(A_fft.reshape(-1)))
myplot(A)

for c in (0.05, 0.01, 0.002):
    thresh = A_fft_sort[int(np.floor((1 - c) * len(A_fft_sort)))]
```

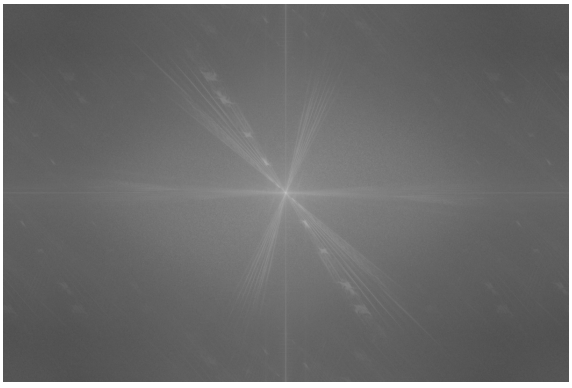
(a) Original image.



(b) Image after applying FFT on each individual row



(c) Image after applying FFT on each individual column



(d) Row and column wise FFT (order does not matter)

Figure 12.1: Image of MCI I and the row/column wise FFT.

```
    A_fft_th = A_fft * (np.abs(A_fft) > thresh)
    A_th = np.fft.ifft2(A_fft_th).real
    myplot(A_th)
```

(a) Original image.

(b) 5% of FFT coefficients

(c) 1% of FFT coefficients

(d) 0.2% of FFT coefficients

Figure 12.2: Image of MCI I and the reconstruction with various amounts of FFT coefficients left.

We can also use the FFT for de-noising and filtering of signals. It is rather simple to just eliminate certain frequency bands in the frequency domain.

```
import matplotlib.pyplot as plt
import imageio.v3 as iio
import numpy as np
%config InlineBackend.figure_formats = ['svg']
np.random.seed(6020)

im = np.asarray(iio.imread(
```

```
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

def myplot(A):
    plt.figure()
    plt.imshow(A, cmap="gray")
    plt.axis("off")
    plt.gca().set_aspect(1)

A = rgb2gray(im)
A_noise = A + (200 * np.random.randn(*A.shape)).astype('uint8')
A_noise_fft = np.fft.fft2(A_noise)
A_noise_fft_shift = np.fft.fftshift(A_noise_fft)
F = np.log(np.abs(A_noise_fft_shift) + 1)

myplot(A_noise)
myplot(F)

nx, ny = A.shape
X, Y = np.meshgrid(np.arange(-ny/2 + 1, ny / 2 + 1),
                   np.arange(-nx / 2 + 1, nx / 2 + 1))
R2 = np.power(X, 2) + np.power(Y, 2)
ind = R2 < 150**2
A_noise_fft_shift_filter = A_noise_fft_shift * ind
F_filter = np.log(np.abs(A_noise_fft_shift_filter) + 1)

A_filter = np.fft.ifft2(np.fft.fftshift(A_noise_fft_shift_filter)).real
myplot(A_filter)
myplot(F_filter)
```

12.2 Wavelet

Similar to the FFT also the Wavelet transform is used in much the same situations.

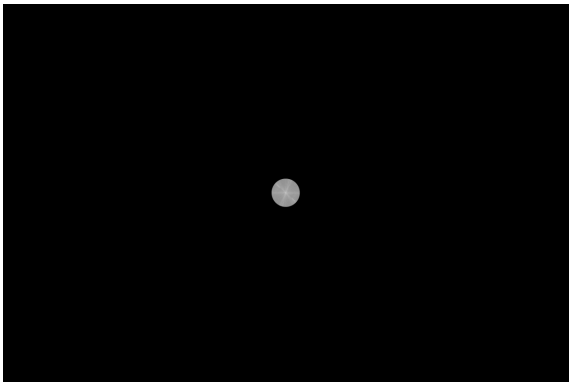
Before we go on and apply the wavelet transform in the same situations we show how the multi level approach looks like for an image. For the image we use the *Daubechies 1* wavelets.

```
import matplotlib.pyplot as plt
import imageio.v3 as iio
```

(a) Noisy image.

(b) Noisy FFT coefficients

(c) Filtered/De-noised image



(d) Filtered/De-noised FFT coefficients

Figure 12.3: Image of MCI I and the reconstruction with various amounts of FFT coefficients left.

```
import numpy as np
import pywt
%config InlineBackend.figure_formats = ['svg']

im = np.asarray(iio.imread(
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

n = 2
A = rgb2gray(im)
coeffs = pywt.wavedec2(A, wavelet="db1", level=n)

coeffs[0] /= np.abs(coeffs[0]).max()
arr, coeff_slices = pywt.coeffs_to_array(coeffs)

plt.imshow(arr, cmap='gray', vmin=-0.25, vmax=0.75)
plt.axis("off")
plt.gca().set_aspect(1)
plt.show()
```

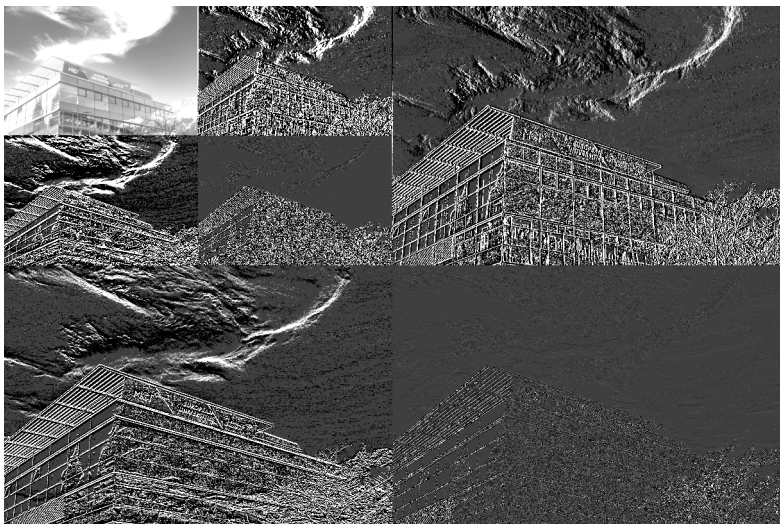
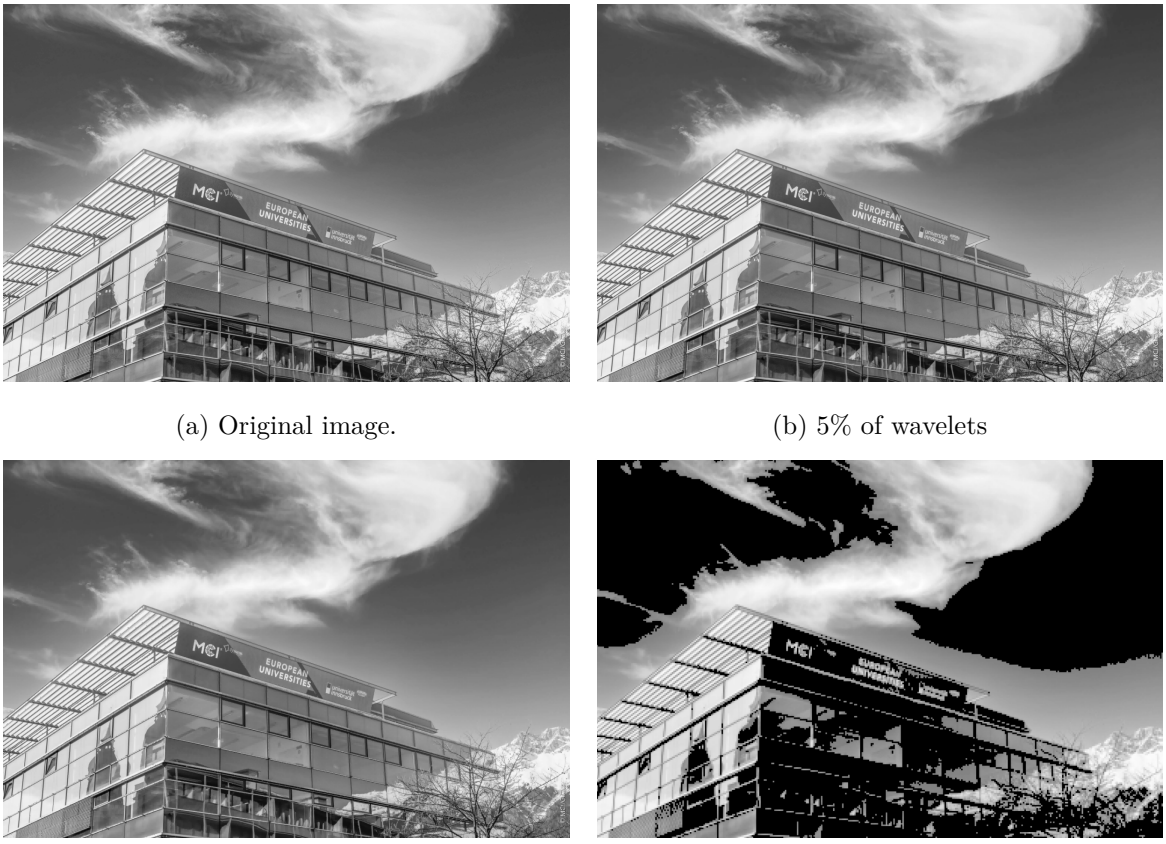


Figure 12.4: First three levels of the discrete wavelet transform.

Of course we can use this to compress the image by removing small values from the transform.

```
import matplotlib.pyplot as plt
import imageio.v3 as iio
import numpy as np
import pywt
%config InlineBackend.figure_formats = ['svg']

im = np.asarray(iio.imread(
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

def myplot(A):
    plt.figure()
    plt.imshow(A, cmap="gray")
    plt.axis("off")
    plt.gca().set_aspect(1)

w = "db1"
A = rgb2gray(im)
coeffs = pywt.wavedec2(A, wavelet=w, level=4)

coeff_arr, coeff_slices = pywt.coeffs_to_array(coeffs)
Csort = np.sort(np.abs(coeff_arr.reshape(-1)))
myplot(A)
for c in (0.05, 0.01, 0.002):
    thresh = Csort[int(np.floor((1 - c) * len(Csort)))]
    Cfilt = coeff_arr * (np.abs(coeff_arr) > thresh)

    coeffs_filt = pywt.array_to_coeffs(Cfilt, coeff_slices, output_format='wavedec2')
    A_r = pywt.waverec2(coeffs_filt, wavelet=w)
    myplot(A_r.astype('uint8'))
```

For ne-noising filters are applied or we and this is going to be the subject for a lecture more focused on image processing.



(a) Original image.

(b) 5% of wavelets

(c) 1% of wavelets

(d) 0.2% of wavelets

Figure 12.5: Image of MCI I and the reconstruction with various amounts of wavelets.

Part V

Sparsity and compression

We have seen in the sections before that our signals and data can often be expressed in an optimal way by changing basis. Moreover, this often results in sparse data and therefore gives us the opportunity for compression. By expressing our data in this basis most coefficients are zero or small.

The zero coefficients give rise to sparsity and the almost zero coefficients allow us to compress the data further without loosing too much information. We have seen this in the Eigendecomposition, the singular value decomposition, within regression choices, the Fourier transform, the Wavelet transform, and in other such transformations not covered here.

Recent development in mathematics have given rise to the field of *compressed sensing*, where not high dimensional signals are collected and transformed or compressed but we start by a acquiring *compressed* signals and solve for the sparsest high-dimensional signal that is *consistent* with the collected data.

Here we will discuss sparsity and compression and give an outlook on compressed sensing. We have already seen multiple examples and we will use this chapter to contextualize these results and combine them to give rise to new ideas and further aspects.

It is worth mentioning that quite often sparsity gives rise to so called *parsimonious* models that avoid overfitting and remain interpretable because they have a low number of terms. This fits neatly into the discussion of Occam's razor: the simplest explanation is in general the correct one. Simple can also mean the least coefficients and this means sparsity. Furthermore, it can also help to create more robust algorithms are outliers have less influence.

> **!** Important
>
> Parts of this section are based on (Brunton and Kutz 2022, chap. 3).

13 Sparsity and Compression

As we have seen before and maybe know from our own experience, most image and audio signals are highly compressible. Here compressible means we can find a basis that allows for a sparse reprehension of our signal. Let us put this into a small definition.

Definition 13.1 (K-sparse data)**.** A compressible signal $x \in \mathbb{R}^n$ may be written in a sparse vector $s \in \mathbb{R}^n$ with a basis transformation (see Definition 1.8 for the formal definition) expressed by the matrix $B \in \mathbb{R}^{n \times n}$ and

$$x = Bs.$$

The vector s is called K-sparse if it contains exactly K non-zero elements.

> **Important**
>
> It is important to note that this does not imply that B is sparse.

If the basis is generic such as the Fourier or Wavelet basis, i.e. we do not need to store the matrix B but can generate it on the spot, we only need to store the K non-zero elements of s to reconstruct the original signal.

> **Note**
>
> As we have seen in Chapter 12 images (and audio) signals are highly compressible in the Fourier and Wavelet basis with view most entries small or zero. By setting the small value to zero we reduce the density further without a high loss of quality.
> We see this in JPEG compression for images and MP3 compression for audio signals. If we stream an audio signal on view an image on the web we only need to transfer s and not x, saving bandwidth and storage as we go.

We have seen in Figure 5.4 that we can use the SVD to reduce the size as well. The downside here is that we need to store U and V (Definition 5.2) even if we reduce the rank. This is rather inefficient. On the other hand, we have used SVD in Section 5.2.2 with the Eigenfaces example how we can create a basis with SVD that can be used to classify an entire class of images - human faces. Storing the basis matrices in this case is comparable cheap and it allows us to use certain aspects of the downsampling for learning purposes.

We also need to stress that SVD and Fourier are unitary transformations which make the move into and from the basis cheep. This is the basis for a lot of computation seen in the field of compressed sensing and compression in general.

> **Note**
>
> The driving factors for compression are audio, image and video, but also raw data compression as seen with `zip`, `7z` and all the other available algorithms.
> It is wrong to assume that we do not see this in engineering applications. High dimensional differential equations usually have a solution on a low dimensional manifolds and therefore imply that sparsity can be seen here to.

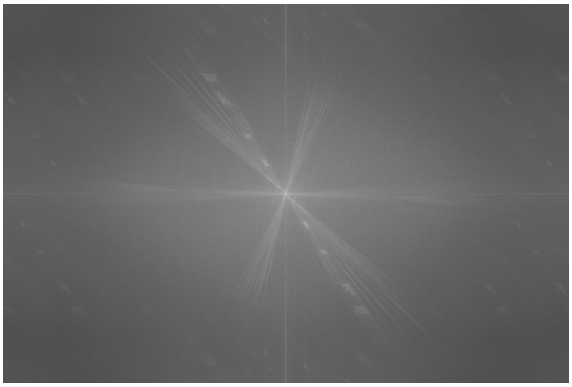
Let us return to image compression and follow along with the examples given in (Brunton and Kutz 2022, chaps. 3, pp98–101).

We can use the code provided earlier. We move from Figure 13.1a to Figure 13.1b via $\mathcal{F}$. From Figure 13.1b to Figure 13.1d we only keep the highest 5% of our values and move to Figure 13.1c via $\mathcal{F}^{-1}$.

```
import matplotlib.pyplot as plt
import imageio.v3 as iio
import numpy as np
%config InlineBackend.figure_formats = ['svg']

im = np.asarray(iio.imread(
    "https://www.mci.edu/en/download/27-logos-bilder?"
    "download=618:mci-eu-web"))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

def myplot(A):
    plt.figure()
    plt.imshow(A, cmap="gray")
    plt.axis("off")
    plt.gca().set_aspect(1)

A = rgb2gray(im)
A_fft = np.fft.fft2(A)
A_fft_sort = np.sort(np.abs(A_fft.reshape(-1)))
myplot(A)

myplot(np.log(np.abs(np.fft.fftshift(A_fft)) + 1))
c = 0.05
```

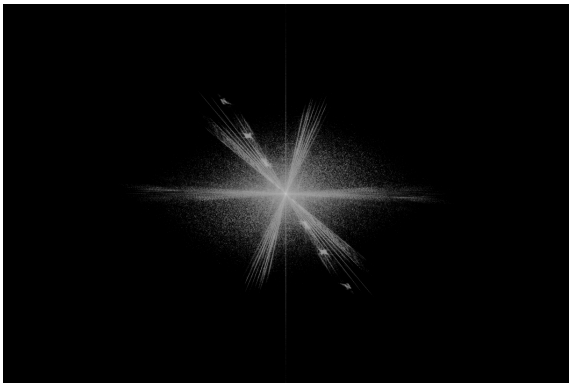
```
thresh = A_fft_sort[int(np.floor((1 - c) * len(A_fft_sort)))]
A_fft_th = A_fft * (np.abs(A_fft) > thresh)
A_th = np.fft.ifft2(A_fft_th).real
myplot(A_th)
myplot(np.log(np.abs(np.fft.fftshift(A_fft_th)) + 1))
```

(a) Original image.



(b) Fourier coefficients

(c) Compressed image



(d) Truncated FFT coefficients (5%)

Figure 13.1: Image of MCI I and the reconstruction with various amounts of FFT coefficients left.

In order to get an idea why the Fourier transform is useful in this scenario we look at the image as a surface.

> **Note**
>
> In order to make this feasible for interactive rendering we use only the upper left quarter of the image.

```
import plotly.graph_objects as go
import numpy as np
%config InlineBackend.figure_formats = ['svg']

B = np.transpose(A[:int(A.shape[0]/2):5, :int(A.shape[1]/2):5])
y = np.arange(B.shape[0])
x = np.arange(B.shape[1])

X,Y = np.meshgrid(x,y)
fig = go.Figure()
fig.add_trace(go.Surface(z=B, x=X, y=Y))
fig.show()
```

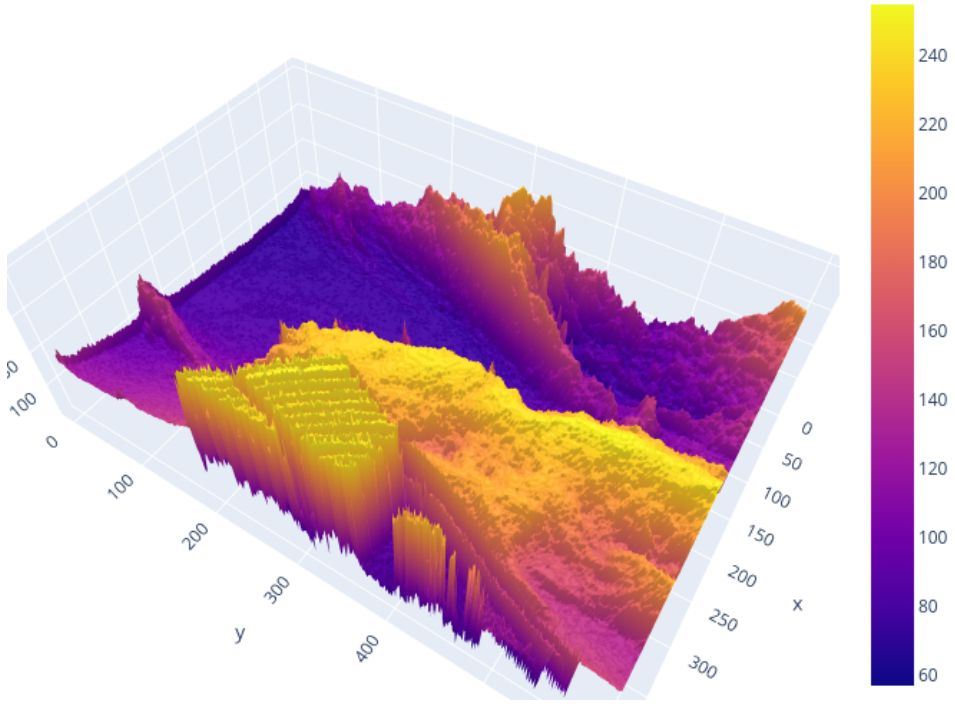


Figure 13.2: Upper left quarter of the MCI I image as a 3D surface.

As can be seen in Figure 13.2 we can express the clouds with view modes and even the raster of the building seams to fit this model nicely.

It is not very surprising to have such structure in a so called *natural image*. The image or pixel space is big, very big. For an $n \times n$ black and white image there are 2^{n^2} possible images. So for a 20×20 image we already have 2^{400} possible images which a number with a number with 121 digits and it is assumed that there are (only) about 10^{82} atoms in the universe.

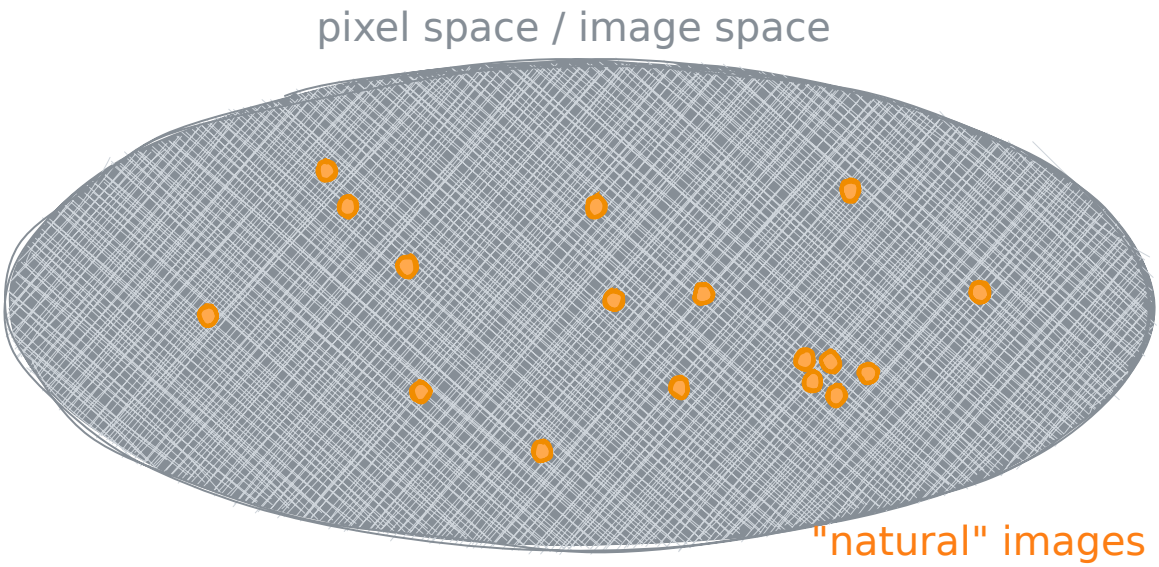


Figure 13.3: Illustration to show the fastness of pixel (image) space in comparison to images we can make sense of aka *natural* images.

So finding structure in images, especially in images with high resolution is not surprising. The rest is basically random noise. Most of the *dimensions* of pixel space are only necessary if we want to encode this random noise images but for a cloud with some mountains and a building only need some dimensions and are therefore highly compressible.

14 Compressed Sensing

We use compression in everyday life extensively. Nevertheless, it almost always requires us to have the high resolution data/measurement and than compress it down. Here we discuss the reverse, where we collect relatively few *compressed* or *random* measurements and then infer a sparse representation.

Until recent developments the computation was non-polynomial (NP) hard problem but now there are algorithms to reconstruct the full signal with high probability using convex algorithms.

Definition 14.1 (Compressed Sensing)**.** For a signal x that is K-sparse in $\mathcal{B}$ with matrix B we need n measurements and can compress them to K.
In compressed sensing we collect p randomly chosen or *compressed* measurements and solve for the K elements of s that are not zero, with $K < p \ll n$.
We can represent p linear measurements of x as the matrix $C \in \mathbb{R}^{p \times n}$ and thus

$$y = Cx.$$

Relating this back to s and $\mathcal{B}$ we get

$$y = Cx = CBs = Ds,$$

as an under-determined system. To get the optimal solution, i.e. the smallest K, we optimize

$$\check{s} = \operatorname{argmin}_s \|s\|_0, \quad \text{subject to} \quad y = CBs.$$

Here $\|\cdot\|_0$ is ℓ_0 pseudo norm that measures the number of non-zero elements.

As we will see, the selection of the matrix C is important to allow for the optimization to be such that we do not need brute-force to look for $\check{s}$. This brute-force search is combinatorial in n and K, for a fixed (known) K but significantly larger for unknown K.

Under certain condition on C we can relax the conditions to a convex ℓ_1-minimization

$$\check{s} = \operatorname{argmin}_s \|s\|_1, \quad \text{subject to} \quad y = CBs. \tag{14.1}$$

In order to have Equation 14.1 converge with high properbility we need to following assumptions to be met (precise description will follow):

1. The measurement matrix C must be *incoherent* with respect to the sparsifying basis $\mathcal{B}$, i.e. the rows of C do not correlate with the columns of B.
2. The number of measurements p must be sufficiently large, on the order of

$$p \approx \mathcal{O}\left(K \log\left(\frac{n}{K}\right)\right) \approx k_1 K \log\left(\frac{n}{K}\right),$$

where k_1 depends on how incoherent C and B are.

> **Note**
>
> In Figure 13.1 we illustrated compressed sensing with a 4016×6016 image. With our 5% of Fourier coefficients this puts the sparsity $K = 0.05 \times 4016 \times 6016 \approx 1208013$ and with $k_1 = 3$ we would stay with p 3.6 million measurements and about 15% of the original pixels.
> The trick part is how to select them, which is why compressed sensing is not used much in imaging, except for some cases like magnetic resonance imaging (MRI) for patience that can not stay still or sedation is risky.

The idea of the two conditions is to make CB act more or less like a unitary transformation on K-sparse vectors s, preserving relative distance between vectors and allowing for the ℓ_1 convex optimization. We will see this in therms of the *restricted isometry property* (RIP).

We know of the Shannon-Nyquist sampling theorem:

> *If a function $f(t)$ contains no frequencies higher than b hertz, then it can be completely determined from its ordinates at a sequence of points spaced less than $\frac{1}{2b}$ seconds apart.* - see Wikipedia

telling us that we need a sampling rate of at least double the highest frequency to recover the signal properly. However, this is only true for signals with broadband frequency content and this is hardly ever the case for uncompressed signals. As we expect an uncompressed signal can be expressed as a sparse signal in the correct basis we can relax the Shannon-Nyquist theorem. Nevertheless, we need precise timing for our measurements and the recovery is not guaranteed, it is possible with high probability.

> **Note**
>
> There are alternative formulations based on so called *greedy algorithms* that determine the sparsity through an iterative matching pursuit problem, e.g. the *compressed sensing matching pursuit* algorithm or CoSaMP for short.
> A related convex formulation we have already seen and can be applied here is
>
> $$\check{s} = \text{argmin}_s \, \|CBs - y\|_2 + \lambda_1 \|s\|_1$$
>
> compare Chapter 8 where λ is a weight for the importance of sparsity, see Figure 8.1.

Chapter 8 actually serve as our first example and shows neatly how the ℓ_1 norm promotes sparsity whereas the ℓ_2 solution stays dense. We can also use this example to give an insight to what *with high probability* means. We created a random matrix in this example with 5 times more rows than columns, unless we are very unlucky and we have a lot of linear dependency between the rows we will have infinitely many solution with *high probability*.

Example 14.1 (Recovering an Audio Signal from Sparse Measurements)**.** Let us consider a audio signal consisting of a two-tone audio signal

$$f(t) = \cos(97t\,2\pi) + \cos(777t\,2\pi)$$

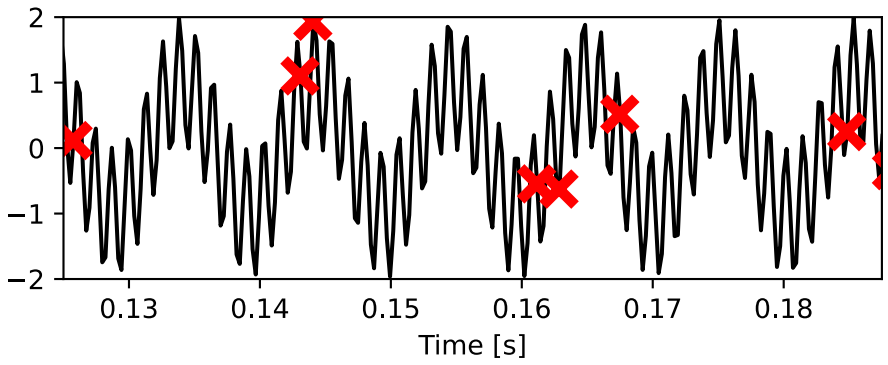
which is sparse in the frequency domain (transformation via FFT). The highest frequency is 777Hz and therefore the Shannon-Nyquist theorem tells us we should sample with at least 1554Hz.
The sparsity of the sample allows us to reconstruct the signal by sampling randomly with an average sampling rate of 128Hz, well below the Shannon-Nyquist threshold.
The code relies on the CoSaMP implementation Needell and Tropp (2009) provided on gitHub and included below so we can work with it.
The code is adapted from (Brunton and Kutz 2022, Code 3.3)

```
import numpy as np

def cosamp(phi, u, s, epsilon=1e-10, max_iter=1000):
    """
    Return an `s`-sparse approximation of the target signal
    Input:
        - phi, sampling matrix
        - u, noisy sample vector
        - s, sparsity
    """
    a = np.zeros(phi.shape[1])
    v = u
    it = 0 # count
    halt = False
    while not halt:
        it += 1

        y = np.dot(np.transpose(phi), v)
        # large components
        omega = np.argsort(y)[-(2*s):]
        omega = np.union1d(omega, a.nonzero()[0])
        phiT = phi[:, omega]
```

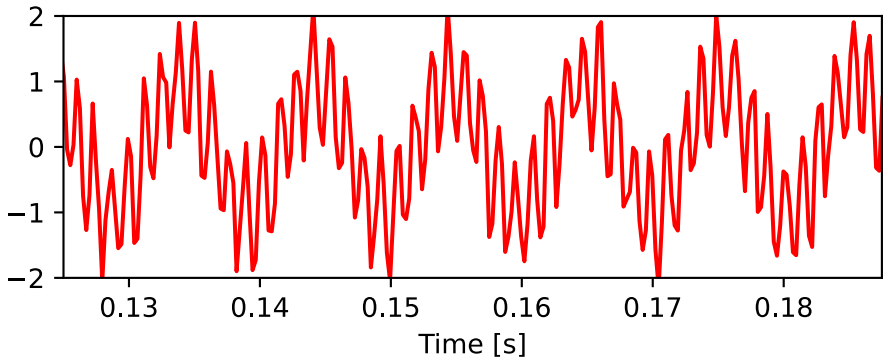
```
        b = np.zeros(phi.shape[1])
        # Solve Least Square
        b[omega], _, _, _ = np.linalg.lstsq(phiT, u)

        # Get new estimate
        b[np.argsort(b)[:-s]] = 0
        a = b

        # Halt criterion
        v_old = v
        v = u - np.dot(phi, a)

        halt = (np.linalg.norm(v - v_old) < epsilon) or \
            np.linalg.norm(v) < epsilon or \
            it > max_iter

    return a
```

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import dct, idct
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)

N = 4096
t = np.linspace(0, 1, N)
fun = lambda t: np.cos(97 * t * 2 * np.pi) + np.cos(777 * t * 2 * np.pi)
f = fun(t)
f_hat = np.fft.fft(f)
PSD = np.abs(f_hat)**2 / N

## Randomly sample signal
# num. random samples
p = N // 32
perm = np.floor(np.random.rand(p) * N).astype(int)
y = f[perm]

## Solve compressed sensing problem
# Build Psi
Psi = dct(np.identity(N))
# Measure rows of Psi
```

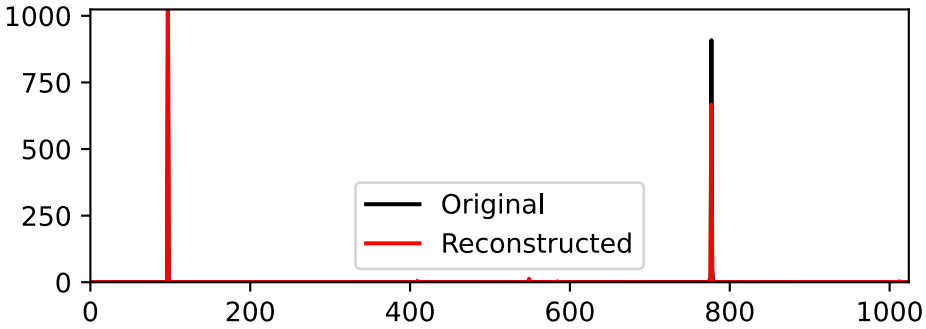
```
Theta = Psi[perm, :]

# CS via matching pursuit
s = cosamp(Theta, y, s=10, epsilon=1.e-10, max_iter=10)
frec = idct(s)
# computes the (fast) discrete fourier transform
frec_hat = np.fft.fft(frec, N)
# Power spectrum (how much power in each freq)
PSDrec = np.abs(frec_hat)**2 / N

time_window = np.array([512, 768]) / N
freq = np.arange(N)
L = int(np.floor(N / 2))

plt.figure()
plt.plot(t, f, "k")
plt.plot(perm/N, y, "rx", ms=12, mew=4)
plt.xlabel("Time [s]")
plt.xlim(time_window[0], time_window[1])
plt.ylim(-2, 2)
plt.gca().set_aspect(5e-3)

plt.figure()
plt.plot(t, frec, "r")
plt.xlim(time_window[0], time_window[1])
plt.xlabel("Time [s]")
plt.ylim(-2, 2)
plt.gca().set_aspect(5e-3)

plt.figure()
plt.plot(freq[:L], PSD[:L], "k", label="Original")
plt.plot(freq[:L], PSDrec[:L], "r", label="Reconstructed")
plt.xlim(0, N/4)
plt.ylim(0, N/4)
plt.legend()
plt.gca().set_aspect(1/3)

plt.show()
```



(a) Time window of the signal, the x marks sampling points in the domain.



(b) Time window of the reconstructed signal from the above measurements.



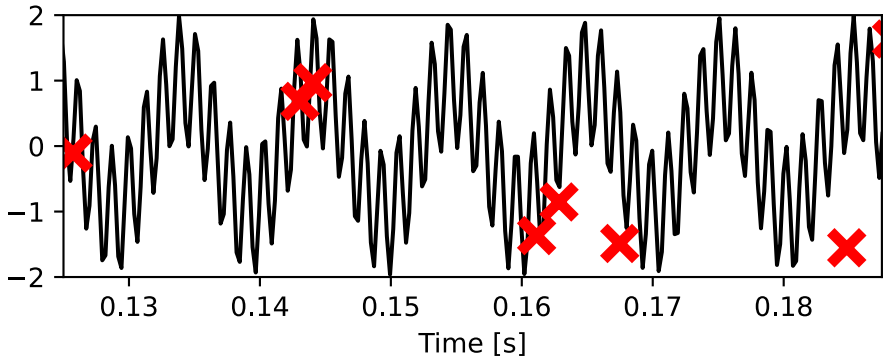
(c) Power spectral density of the original and reconstructed signal.

Figure 14.1: Compressed sensing reconstruction of a two frequency audio signal, entire time from [0, 1].

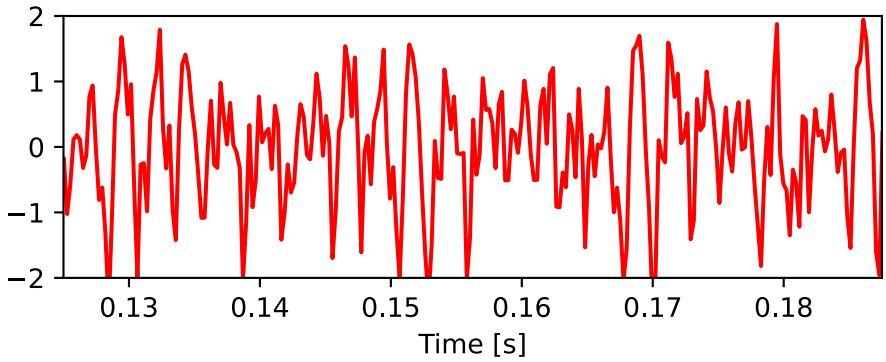
The CoSaMP algorithm works in the discrete cosine transform basis, which can be derived from the FFT. For this algorithm we sample $p = 128$ points in time from the $N = 4096$

high resolution sampling. We therefore know exactly the timing of these samples. If we would sample uniformly in time we fail, see Figure 14.2. This happens because we see an aliasing effect for the high-frequency domain resulting in erroneous frequency peaks. If we compare the samples in Figure 14.1a and Figure 14.2a it is hard to see the difference.

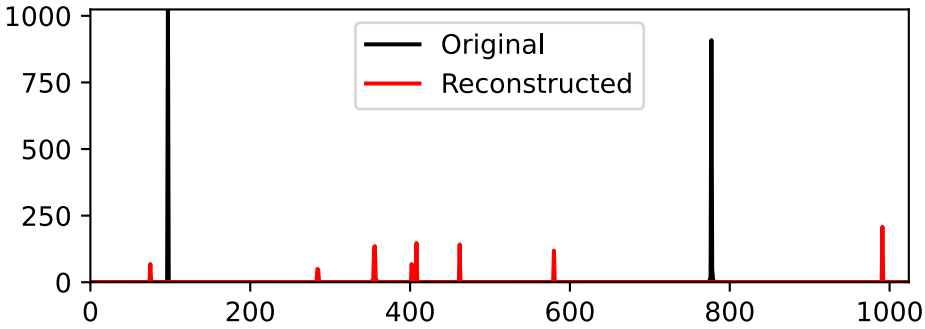
```
ttt = np.random.uniform(0, 1, p)
y = fun(ttt)

## Solve compressed sensing problem
Psi = dct(np.identity(N))
Theta = Psi[perm, :]

# CS via matching pursuit
s = cosamp(Theta, y, s=10, epsilon=1.e-10, max_iter=10)
frec = idct(s)
frec_hat = np.fft.fft(frec, N)
PSDrec = np.abs(frec_hat)**2 / N

plt.figure()
plt.plot(t, f, "k")
plt.plot(perm/N, y, "rx", ms=12, mew=4)
plt.xlabel("Time [s]")
plt.xlim(time_window[0], time_window[1])
plt.ylim(-2, 2)
plt.gca().set_aspect(5e-3)

plt.figure()
plt.plot(t, frec, "r")
plt.xlim(time_window[0], time_window[1])
plt.xlabel("Time [s]")
plt.ylim(-2, 2)
plt.gca().set_aspect(5e-3)

plt.figure()
plt.plot(freq[:L], PSD[:L], "k", label="Original")
plt.plot(freq[:L], PSDrec[:L], "r", label="Reconstructed")
plt.xlim(0, N/4)
plt.ylim(0, N/4)
plt.legend()
plt.gca().set_aspect(1/3)
```



(a) Time window of the signal, the x marks sampling points in the domain.



(b) Time window of the reconstructed signal from the above measurements.



(c) Power spectral density of the original and reconstructed signal.

Figure 14.2: Compressed sensing reconstruction of a two frequency audio signal, entire time from [0, 1] - sampling uniform in time.

In the algorithm CoSaMP the desired level of sparsity can be selected. For the above plots we used $s = 10$ to simulate an unknown factor. Convergence to the sparsest solution

relies on p, which in return is related to the sparsity K.
(Compare Brunton and Kutz 2022, 107)

14.1 The theoretic basics of compressed sensing

Now that we have seen compressed sensing in action we need to bring in the theoretical basics that form this theory.

The key feature is to look into the geometry of sparse vectors and how these vectors are transformed via random measurements. More precisely, for large enough p (amount of measurements) our matrix $D = CB$ of Definition 14.1 preserves the distance and inner product structure of sparse vectors. In turn this means, we need to find a matrix C such that D is a near-isometry map on sparse vectors.

Note

An isometry map is a map that is distance preserving, i.e. the distance between to points is not changed under this map.

$$\|a - b\| = \|f(a) - f(b)\|$$

A unitary map is a map that preserves distance and angle between vectors, i.e.

$$\langle a, b\rangle = \langle f(a), f(b)\rangle$$

For a linear map $f(x) = Ux$ this results in $U^{\mathrm{H}}U = UU^{\mathrm{H}} = I$.

If D behaves as a near isometry, it is possible to solve

$$y = Ds$$

for the *sparsest* vector s using convex ℓ_1 minimization.

Furthermore, a general rule is, the more incoherent the measurements are the smaller we can choose p.

Definition 14.2 (Restricted Isometry Property (RIP))**.** For p incoherent measurements the matrix $D = CB$ satisfies a **restricted isometry property** (RIP) for sparse vectors s

$$(1 - \delta_K)\|s\|_2^2 \leq \|CBs\|_2^2 \leq (1 + \delta_K)\|s\|_2^2,$$

with the restricted isometry constant δ_K. For a small enough δ_K CB acts as a near-isometry on K-sparse vectors.

In particular, for $\delta_K < 1$ and therefore $(1 - \delta_K) > 0$ and $(1 + \delta_K) > 0$ this means the norm induced by CB is equivalent to the two norm on the K-sparse vectors.

It is difficult to compute the δ_K in the RIP and as C may be selected at random there is more information included in the statistical properties of δ_K for a family of matrices C. In general, increasing p ill decrease δ_K, same as having incoherence vectors and both improve the properties of CB by bringing it closer to an isometry.

Luckily, there exist generic sampling matrices C that are sufficiently incoherent with resect to nearly all transform bases. In particular, Gaussian and Bernoulli random measurement matrices satisfy Definition 14.2 for a generic B (with high probability).

14.2 Sparse Regression

Let us return to regression for a moment an in particular to the LASSO method introduced in Section 8.1.1. We have seen that ℓ_1, i.e. the $\|\cdot\|_1$, promotes sparsity and we can use this to create a more robust method that rejects outliers.

We split up our points into a training and test set (the classic 80:20 split). For the test set an varying the parameter λ_1 through a range of values we create a *fit* with the training set and test against the test set.

Example 14.2 (LASSO fit for a regression problem)**.** For our

$$Ax = b$$

problem, we consider a problem the dimensions as 200 observations with 10 candidate predictions. This results in a matrix $A \in \mathcal{R}^{200\times 10}$ and we select the vector $b \in \mathcal{R}^{200}$ as the linear combination of exactly two of theses 10 candidates. As a result, the vector x is 2-sparse by construction, and the aim is to recover this. In order to give the algorithm something to work on we add noise to b resulting in no zero element in b.
To perform a 10 fold cross validation [1] for the LASSO method we use the features of `sklearn` [2].

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import model_selection
import pandas as pd
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)
```

```
m = 200
n = 10

A = np.random.randn(m, n)
x = np.array([0, 0, 1, 0, 0, 0, -1, 0, 0, 0])
b = A @ x + 2 * np.random.randn(m)

# k-cross validation for the Lasso method
cv = 10
lassoCV = linear_model.LassoCV(cv=cv, random_state=6020, tol=1e-8)
lassoCV.fit(A, b)
# Recompute the lasso method for the optimal lambda selected
lasso_best = linear_model.Lasso(alpha=lassoCV.alpha_, random_state=6020)
lasso_best.fit(A, b)

# Plotting
plt.figure()
Lmean = lassoCV.mse_path_.mean(axis=-1)
error = [np.min(lassoCV.mse_path_, axis=-1),
         np.max(lassoCV.mse_path_, axis=-1)] / np.sqrt(cv)
plt.errorbar(lassoCV.alphas_, Lmean, yerr=error, ecolor="lightgray")
plt.plot(lassoCV.alpha_,
         Lmean[lassoCV.alphas_==lassoCV.alpha_],
         "go", mfc='none')
plt.xscale("log")
plt.ylabel("Means Square Error")
plt.xlabel(r"$\lambda_1$")
plt.gca().invert_xaxis()
plt.show()
```

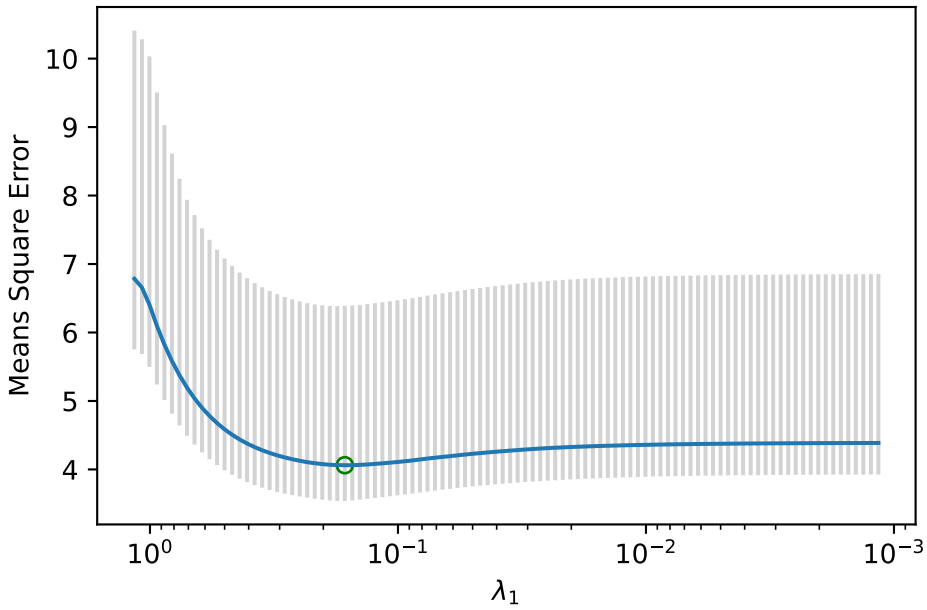


Figure 14.3: Cross-validation mean square error of Lasso fit, the green circle marks the optimal choice, the blue line the mean error of the k-folds and the error bar the maximal and minimal error form the k-folds.

The resulting coefficients for the best fit with lasso are

```
array([ 0.        ,  0.        ,  1.0978465 ,  0.        , -0.        ,
       -0.        , -1.06495486,  0.        ,  0.        ,  0.        ])
```

and a appropriate fit for our toy example.
(Compare Brunton and Kutz 2022, 115)

Note

It is also possible to build a dictionary and and look for a sparse representation in this dictionary.
An example how this can be done with the Eigenfaces example can be found in (Brunton and Kutz 2022, 117)

[2]We split our data into 10 sets, use 9 for the training and 1 for test and average over the results.
[2]Install the package via `pdm add scikit-learn`.

14.3 Robust Principal Component Analysis (RPCA)

We discussed the principal component analysis in Section 5.2 as an application for the SVD. Similar as regression, also the PCA is sensitive to outliers and *corrupt data*. In Candès et al. (2011) an algorithm to make it more robust was developed.

The main idea of the paper is to decompose a matrix X into a structured low-rank matrix L and a sparse matrix S containing the outliers and corrupt data

$$X = L + S.$$

If we can recover the principal components of L from X we have a robust method as the perturbation of S has little influence.

It might not be immediately obvious but applications of this split are video surveillance where the background is represented by L and the foreground objects in L, face recognition with the eigenfaces in L and shadows, occlusions (like glasses or masks) are in S.

The task boils down to an optimization problem of the form

$$\min_{L,S} \operatorname{rank}(L) + \|S\|_0 \quad \text{subject to} \quad L + S = X, \tag{14.2}$$

where unfortunately both parts are not convex but we can search for it with *high probability* using

$$\min_{L,S} \|L\|_\star + \lambda \|S\|_1 \quad \text{subject to} \quad L + S = X. \tag{14.3}$$

In Equation 14.3 $\|\cdot\|_\star$ is called the *nuclear norm* consisting of the sum of all singular values and we use it as our proxy for the rank.

Equation 14.3 is called *principal component pursuit* (PCP) and in Candès et al. (2011) the authors show that Equation 14.3 converges to Equation 14.2 for

1. $\lambda = 1/\sqrt{\max(n, m)}$ for $X^{m \times n}$,
2. L is not sparse,
3. S is not low-rank, where we assume that the entries do not span a low-dimensional column space.

We can solve the PCP with an augmented Lagrange multiplier algorithm such as

$$\mathcal{L}(L, S, Y) = \|L\|_\star + \lambda \|S\|_1 + \langle Y, X - L - S \rangle + \frac{\mu}{2} \|X - L - S\|_F^2.$$

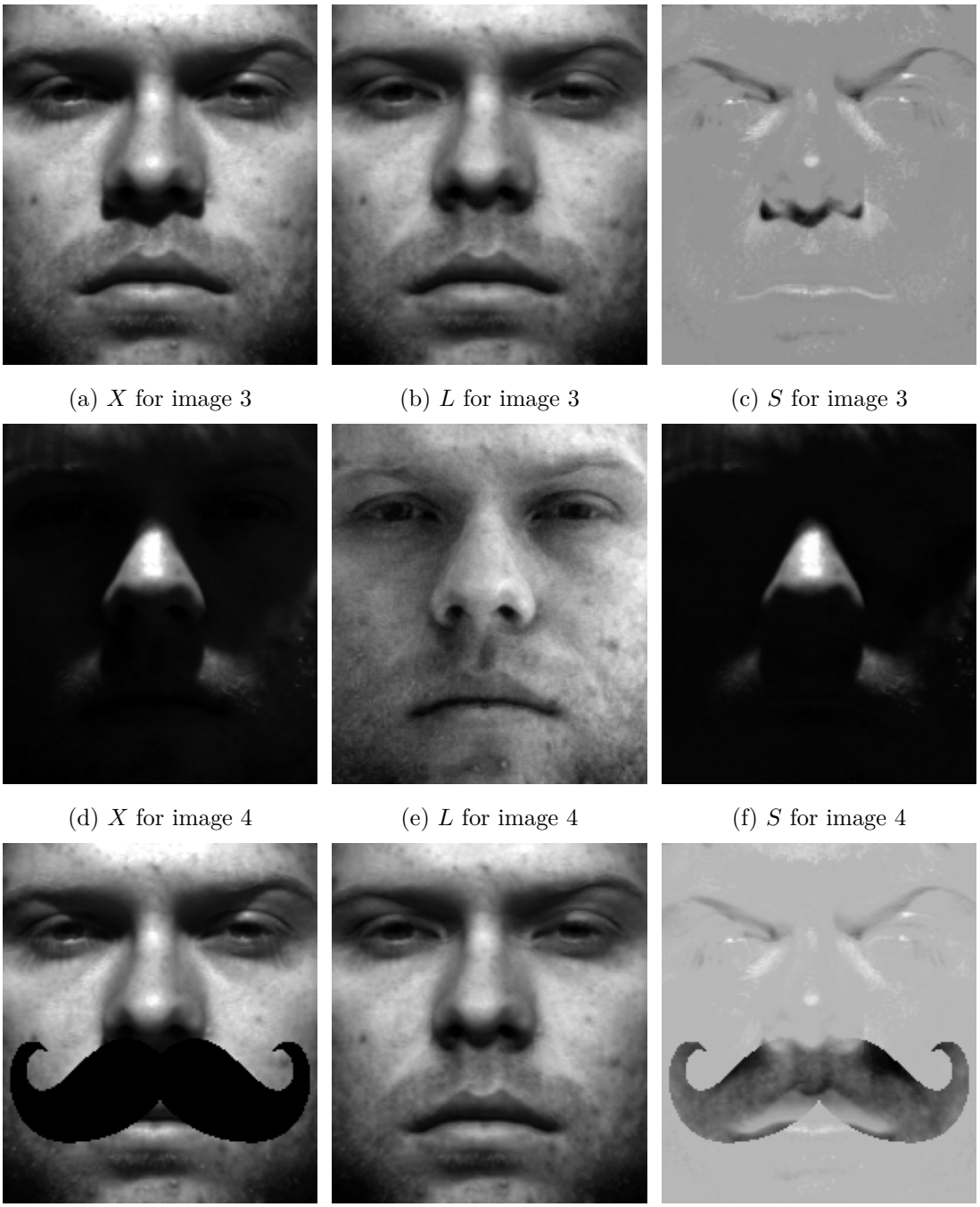
This is an iterative method where we solve for L_k and S_k, update $Y_{k+1} = Y_k + \mu(X - L_k - S_k)$ and check for convergence. For this specific problem the alternations method (ADM) provides a simple procedure to solve for L and S.

Example 14.3 (Robust principal component analysis with Yale B dataset)**.** The following implementation of RPCA can be found in (Brunton and Kutz 2022, 121) with the same application to the eigenfaces dataset.

```
def shrink(X, tau):
    Y = np.abs(X) - tau
    return np.sign(X) * np.maximum(Y, np.zeros_like(Y))


def SVT(X, tau):
    U, S, VT = np.linalg.svd(X, full_matrices=False)
    out = U @ np.diag(shrink(S, tau)) @ VT
    return out


def RPCA(X):
    n1, n2 = X.shape
    mu = n1 * n2 / (4 * np.sum(np.abs(X.reshape(-1))))
    lambd = 1 / np.sqrt(np.maximum(n1, n2))
    thresh = 10**(-7) * np.linalg.norm(X)

    S = np.zeros_like(X)
    Y = np.zeros_like(X)
    L = np.zeros_like(X)
    count = 0
    while (np.linalg.norm(X - L - S) > thresh) and (count < 1000):
        L = SVT(X - S + (1 / mu) * Y, 1 / mu)
        S = shrink(X - L + (1 / mu) * Y, lambd / mu)
        Y = Y + mu*(X - L - S)
        count += 1
    return L, S
```

```
import numpy as np
import numpy.linalg as LA
import scipy
import requests
import io
import imageio.v3 as iio
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ['svg']
```

```
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

response = requests.get(
    "https://github.com/frankhuettner/Data_Driven_Science_Julia_Demos"
    "/raw/refs/heads/main/DATA/allFaces.mat")

data = scipy.io.loadmat(io.BytesIO(response.content))
faces = data["faces"]
m = int(data["m"][0,0])
n = int(data["n"][0,0])
nfaces = np.ndarray.flatten(data['nfaces'])

XX = faces[:,:nfaces[0]]
im = np.asarray(iio.imread(
    "https://raw.githubusercontent.com/dynamicslab/databook_python"
    "/refs/heads/master/DATA/mustache.jpg"))
A = np.round(rgb2gray(im)/255).astype("uint8")
X = np.append(XX, (XX[:, 2] * A.T.flatten()).reshape((-1, 1)), axis=1)

L, S = RPCA(X)

for index in [2, 3, -1]:
    plt.figure()
    plt.imshow(np.reshape(X[:, index], (m, n)).T, cmap="gray")
    plt.gca().axis("off")
    plt.figure()
    plt.imshow(np.reshape(L[:, index], (m, n)).T, cmap="gray")
    plt.gca().axis("off")
    plt.figure()
    plt.imshow(np.reshape(S[:, index], (m, n)).T, cmap="gray")
    plt.gca().axis("off")
```



(a) X for image 3 (b) L for image 3 (c) S for image 3

(d) X for image 4 (e) L for image 4 (f) S for image 4

(g) X for image 3 with comic disguise (h) L for image 3 with comic disguise (i) S for image 3 with comic disguise

Figure 14.4: RPCA decomposition for the Yale B dataset.

In Figure 14.4 we can see that RPCA effectively filters out occluded regions and shadows form X in L. The missing part is filled in with the most consistent low-rank feature from the provided dataset. In our case we include all faces from the first person plus the third image with a fake moustache.
We need to stress, that we can not use this setup to reconstruct a different face obscured by a moustache.

14.4 Sparse Sensor Placement

So far we have looked how to reconstruct a signal with random measurements in a generic basis. But how about placing sensors at the correct points to reconstruct the signal with high probability. This can dramatically reduce the amount of data to measure.

We can set *tailored sensors* for a particular library (our basis) instead of *random sensors* in a generic library.

Amongst other things, this can be used to reconstruct faces, or classify signals, see (Brunton and Kutz 2022, chap. 3.8) and references within.

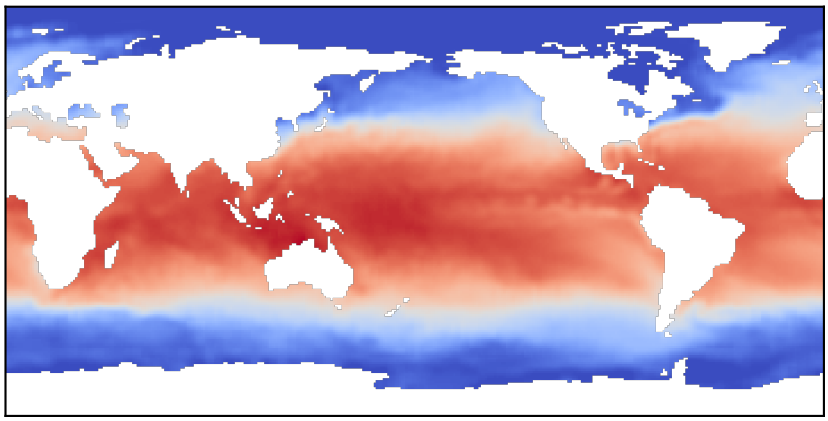
To see this in action we use the Python package `PySensors` and the example *Sea Surface Temperature (SST) sensors* from their documentation, see the docs accessed on the 28th of November 2024.

Example 14.4 (Sea Surface Temperature (SST) sensors)**.** For a given dataset of sea surface temperature as training data we would like to place (sparse) sensors optimal that allow us to reconstruct the temperature at any other location. This is achieved with the `SSPOR` algorithm from Manohar et al. (2018).

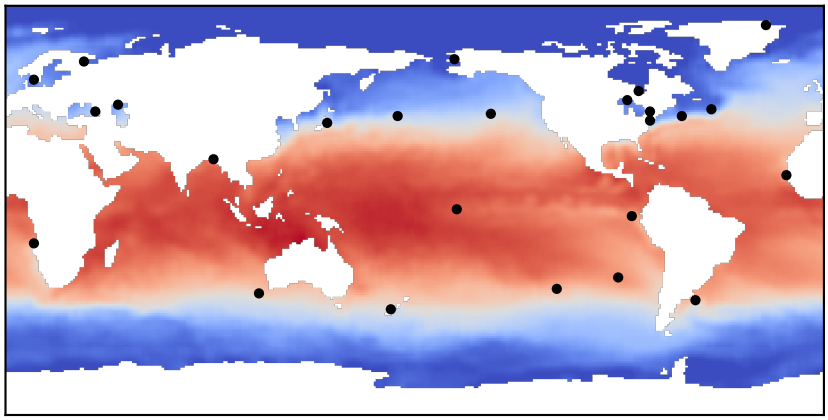
```
from ftplib import FTP
import numpy as np
import matplotlib.pyplot as plt
import netCDF4
import pysensors as ps
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)

# Import and save data locally
ftp = FTP('ftp.cdc.noaa.gov')
ftp.login()
ftp.cwd('/Datasets/noaa.oisst.v2/')
```

```
filenames = ['sst.wkmean.1990-present.nc', 'lsmask.nc']

for filename in filenames:
    localfile = open(filename, 'wb')
    ftp.retrbinary('RETR ' + filename, localfile.write, 1024)
    localfile.close()

ftp.quit()

f = netCDF4.Dataset('sst.wkmean.1990-present.nc')
lat,lon = f.variables['lat'], f.variables['lon']
SST = f.variables['sst']
sst = SST[:]

f = netCDF4.Dataset('lsmask.nc')
mask = f.variables['mask']

masks = np.bool_(np.squeeze(mask))
snapshot = float("nan")*np.ones((180,360))
snapshot[masks] = sst[0,masks]

plt.figure()
plt.imshow(snapshot, cmap=plt.cm.coolwarm)
plt.xticks([])
plt.yticks([])
X = sst[:,masks]
X = np.reshape(X.compressed(), X.shape)

# Compute optimal sensor placement
model = ps.SSPOR(
    basis=ps.basis.SVD(n_basis_modes=25),
    n_sensors=25
)
model.fit(X)
sensors = model.get_selected_sensors()

# Plot sensor locations
temp = np.transpose(0 * X[1,:])
temp[sensors] = 1
img = 0 * snapshot
img[masks] = temp
```

```
plt.figure()
plt.imshow(snapshot, cmap=plt.cm.coolwarm)
indx = np.where(img==1)
plt.scatter(indx[1], indx[0], 8, color='black')
plt.xticks([])
plt.yticks([])
```



(a) Sea surface temperature.



(b) Optimal learned sensor placements to recover sea surface temperature

Figure 14.5: Finding optimal sensor placements to recover the sea surface temperature.

(Compare docs of `PySensors` accessed on the 28th of November 2024)

This concludes our excursion into compressed sensing.

Part VI

Statistics

Data science as we know it does not work without the solid foundations given via statistics and stochastic.

We have already seen some of these topics in our section about the basic properties of sets, Chapter 2 and we will build on this here to recall the most important results.

15 Bayesian Statistics

In the middle of the eighteenth century Joshua Bayes, a Presbyterian minister, set the ground work and as it so often happens was first not acknowledged by his peers.

The basic rule is often illustrated via a Venn diagram for sets.

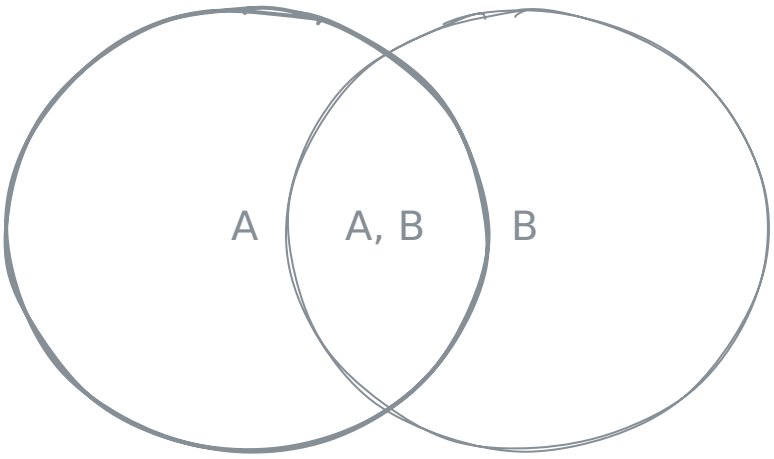


Figure 15.1: Venn diagram illustrating the principal of joint probabilities.

Translated into a statistic problem this becomes: What is the probability, that a number is both, in set A and set B.

Before we can write this down proper we need to introduce some notation. With $P(A)$ we denote the probability of the event A and $P(A, B, ...)$ the probability of all events listed combined. Furthermore we call $P(\neg A)$ the probability of *not* A, i.e. $P(\neg A) = 1 - P(A)$.

Definition 15.1 (Independent events - unconditional probability)**.** We call two events A and B independent if the fact that we know A happened does not give us any insight the probability that B happens or vice versa.
In this case the probability of both events happening is

$$P(A, B) = P(A)P(B).$$

An example is the classic coin toss. After throwing the first coin that lands on heads we have no information what the second toss will likely be.

So if we say A represents *first flip heads*, B *second flip heads* we get

$$P(A,B) = P(A)P(B) = \frac{1}{2}\frac{1}{2} = \frac{1}{4},$$

or if B represents *both flips tail* we get:

$$P(A,B) = P(A)P(B) = \frac{1}{2}0 = 0.$$

Now what happens when the two events are not independent.

Definition 15.2 (Dependent events - conditional probability)**.** If two events A and B are not independent and the probability of event B is not zero, i.e. $P(B) \neq 0$ we define the conditional probability of A under the condition B as $P(A|B)$ and we get the relation

$$P(A|B) = \frac{P(A,B)}{P(B)}.$$

Therefore, if we return to our Venn diagram in Figure 15.1 we can see this relation played out. As $P(A|B)$ can be understood as the fraction of probability B that intersects with A.

Definition 15.3 (Law of joint probabilities)**.** For two events A and B the following relation

$$P(A,B) = P(B|A)P(A) = P(A|B)P(B)$$

is called the **law of joint probabilities**.

Definition 15.4 (Bayes' theorem)**.** For two events A and B as well as there dependent probabilities the following relation is called Bayes' theorem

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

The term $P(A|B)$ is usually called the **posterior probability density** (posterior for short), $P(A)$ the **prior**, $P(B|A)$ the **likelihood**, and finally the term $P(B)$ the **denominator**.

i Note

We have some further properties that follow from the above definitions for two events A and B.

If they are independent we get

$$P(A|B) = P(A).$$

Furthermore, if $\neg B$ denotes the negative of B we have

$$P(A) = P(A, B) + P(A, \neg B)$$

and therefore we get

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|\neg A)P(\neg A)}$$

Let us look at this with an example.

Example 15.1 (Cancer Screening)**.** Let us assume we look at a medical procedure that tests for something like prostate or breast cancer.
This test for *cancer* is quite reliable with a true positive rate of 80% and a true negative rate with 90%. This translates into the probabilities

$$P(+|cancer) = 0.8,$$
$$P(+|\neg cancer) = 0.1.$$

furthermore we assume that only 1% of all the people taking the exam actually have cancer, i.e.

$$P(cancer) = 0.01.$$

From Beyes' theorem we can compute the probability (or likelihood) to have cancer when receiving a positive test result as

$$P(cancer|+) = \frac{P(+|cancer)P(cancer)}{P(+)}$$

The part we do not know in this equation is $P(+)$ but we can compute this with Definition 15.3 as

$$\begin{aligned} P(+) &= P(+, cancer) + P(+, \neg cancer) \\ &= P(+|cancer)P(cancer) + P(+|\neg cancer)P(\neg cancer) \\ &= 0.8 \cdot 0.01 + 0.1 \cdot 0.99 = 0.107. \end{aligned}$$

Overall, this allow us to compute

$$P(cancer|+) = \frac{P(+|cancer)P(cancer)}{P(+)} = \frac{0.8 \cdot 0.01}{0.107} = 0.07477.$$

(Compare Lambert 2018, sec. 3.6.2)

Exercise 15.1 (Bayes' theorem)**.**

1. A person is known to lie one third the time. This person throws a die and reports it is a 4 (four).

   What is the probability that it is actually a four?

2. There are 3 urns containing 3 white and 2 black, 2 white and 3 black, as well as 4 white and 1 black ball respectively. Each urn can be chosen with equal probability.

   What is the likelihood that a white ball is drawn?

15.1 Random variable

A random variable X (often also called random quantity or stochastic variable) is a variable that is dependent on *random events* and is a function defined on a probability space $(\Omega, \mathcal{F}, P)$. Ω is called the sample space of all possible outcomes, $\mathcal{F}$ the event space (a set of outcomes in the sample space), and P is the probability function which assigns each event in the event space a probability as a number between 0 and 1.

A very simple random variable is the one describing a coin toss. It is one for heads and zero for tails. To increase the complexity we can search for a random variable that tells us how often tails was the result for n coin tosses.

To get a better idea what $(\Omega, \mathcal{F}, P)$ and X is we should ask a the following question: *How likely is the value of* $X = 2$*?*

To answer this question we need to find the probability of the event $\{ ,:, X( ) = 2\}$ that can be expressed via P as $P(X = 2) = p_X(2)$.

If we record all this probabilities of outcomes of X results in the **probability distribution** of X. If this probability distribution is real valued we get the **cumulative distribution function** (CDF)

$$F_X(x) = P(X \leq x).$$

15.1.1 Discrete random variables

We have already seen a discrete random variable as *the number of heads for* n *coin tosses*, other possibilities are *the number of children for a person.*

Example 15.2 (Coin toss)**.** For the coin toss we have $\Omega = \{heads, tails\}$ and

$$X(\omega) = \begin{cases} 1, & \text{if } \omega = heads, \\ 0, & \text{if } \omega = tails. \end{cases}$$

for a *fair coin* the function

$$F_X(x) = \begin{cases} \frac{1}{2}, & \text{if } y = 1, \\ \frac{1}{2}, & \text{if } y = 0. \end{cases}$$

Of course this does not mean we will get exactly $\frac{1}{2}$ for n tosses but with the rule of large numbers we see that it will end up there for a lot of flips

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]
np.random.seed(6020)
import scipy.special

x = range(1, 1_001, 2)
y = []
for n in x:
    y.append(np.sum(np.random.rand(n) < 0.5) / n)

plt.plot(x, y)
plt.plot([0, max(x)], [1/2,  1/2])
plt.gca().set_aspect(1000 / (3))
plt.show()
```

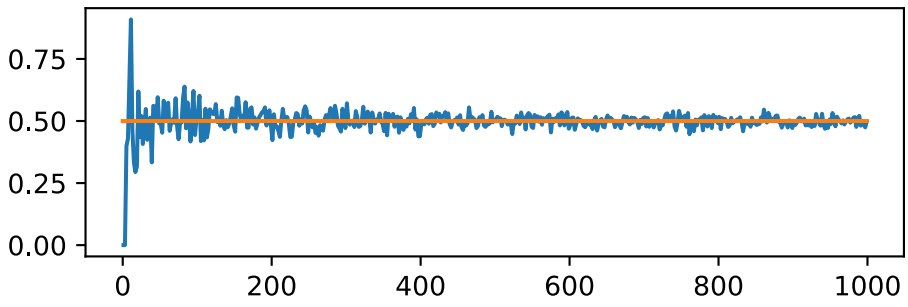


Figure 15.2: Rule of large numbers for coin toss, coin toss simulated by a random numbers smaller or grater than 0.5.

15.1.2 Continuous random variable

If the cumulative distribution function is continuous everywhere we get have a continuous random variable. For such variables it makes more sense to not look for an exact equality with a number but rather, what is the probability that X lies between $[a, b]$.

Example could be the height of a person (if we can measure continuously). In this example it the probability that a person is exactly $185m$ is zero but the question if somebodies hight is in $[180, 190)$ makes sense.

> **Note**
>
> We introduced the Laplace transform in Chapter 10. It finds an application in probability theory where the Laplace transform of a random variable X is defined as the expected value of X with probability density function f, i.e.
>
> $$\mathcal{L}\{f\}(s) = \mathrm{E}\left[\mathrm{e}^{-sX}\right].$$
>
> Furthermore, we can recover the copulative distribution function of a continuous random variable X as
>
> $$F_X(x) = \mathcal{L}^{-1}\left\{\frac{1}{s}\mathrm{E}\left[\mathrm{e}^{-sX}\right]\right\}(x) = \mathcal{L}^{-1}\left\{\frac{1}{s}\mathcal{L}\{f\}(s)\right\}(x).$$

15.2 Probability distributions

It is instructive to look at some of the most common probability distributions so that we know what we can describe by these them.

Example 15.3 (Binomial distribution)**.** The binomial distribution is a discrete probability function with parameters n and p. It describes the number of successes in a sequence of n independent experiments, where p describes the probability of the experiment being a success.
The probability density function is

$$f(x) = \begin{pmatrix} n \\ x \end{pmatrix} p^k (1-p)^{n-k},$$

with

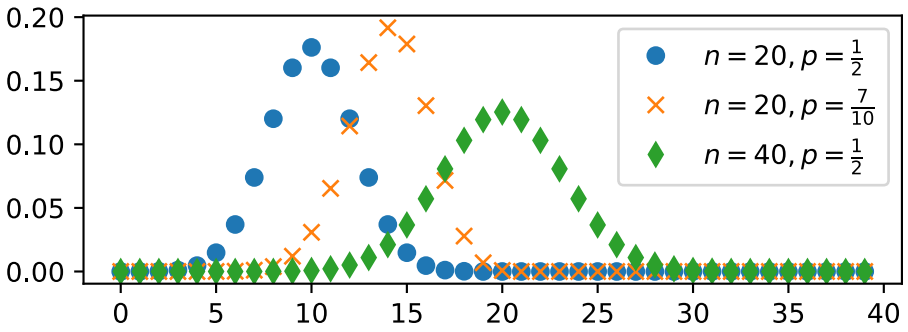
$$\begin{pmatrix} n \\ x \end{pmatrix} = \frac{n!}{k!(n-k)!}.$$

It measures the probability that we get exactly x successes in n independent trials.

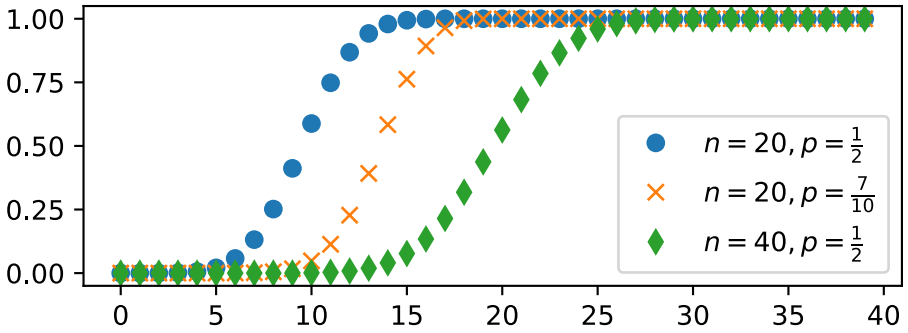
The corresponding CDF is

$$F(x) = \sum_{i=0}^{\lfloor x \rfloor} \left(\begin{array}{c} n \\ i \end{array} \right) p^i (1-p)^{n-i}.$$

```
import numpy as np
import matplotlib.pyplot as plt
import math
%config InlineBackend.figure_formats = ["svg"]


f = lambda n, p, x: np.vectorize(lambda x: math.comb(n, x))(x) * \
                    np.pow(p, x) * np.pow(1 - p, n - x)

helper = lambda n, p, x: np.sum(f(n, p, np.arange(0, x + 1)))
cfd = lambda n, p, x: np.vectorize(lambda x: helper(n, p, x))(x)

x = np.arange(0, 40)
plt.figure()
plt.plot(x, f(20, 0.5, x), "o", label=r"$n=20, p=\frac{1}{2}$")
plt.plot(x, f(20, 0.7, x), "x", label=r"$n=20, p=\frac{7}{10}$")
plt.plot(x, f(40, 0.5, x), "d", label=r"$n=40, p=\frac{1}{2}$")
plt.gca().set_aspect(40/(3 * 0.2))
plt.legend()
plt.figure()
plt.plot(x, cfd(20, 0.5, x), "o", label=r"$n=20, p=\frac{1}{2}$")
plt.plot(x, cfd(20, 0.7, x), "x", label=r"$n=20, p=\frac{7}{10}$")
plt.plot(x, cfd(40, 0.5, x), "d", label=r"$n=40, p=\frac{1}{2}$")
plt.gca().set_aspect(40/(3 * 1))
plt.legend()
plt.show()
```



(a) Probability density function.



(b) Cumulative distribution function.

Figure 15.3: Binomial distribution - basic functions.

Example 15.4 (Bernoulli distribution)**.** The Bernoulli distribution is a discrete probability function which takes the value 1 with probability p and the value 0 with probability $q = 1 - p$. It is how we can model a coin toss or anything that is expressed in a *yes-no question*.
It is the special case of a Binomial distribution Example 15.3 with $n = 1$.
The probability density function becomes

$$f(x) = p^x(1-p)^{1-x} \quad \text{for} \quad x \in \{0, 1\}$$

and the corresponding CDF

$$F(x) = \begin{cases} 0 & \text{if } x \leq 0, \\ 1-p & \text{if } 0 \leq x \leq 1, \\ 1 & \text{if } x \geq 1. \end{cases}$$

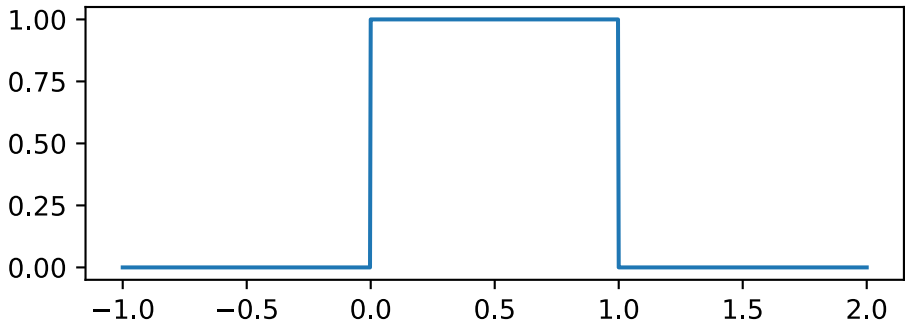
Example 15.5 (Continuous uniform distribution)**.** The continuous uniform distribution is also called the rectangular distribution and describes an outcome that lies between certain bounds.
The probability density function is

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if\_} a \leq x \leq b, \\ 0 & \text{else.} \end{cases}$$

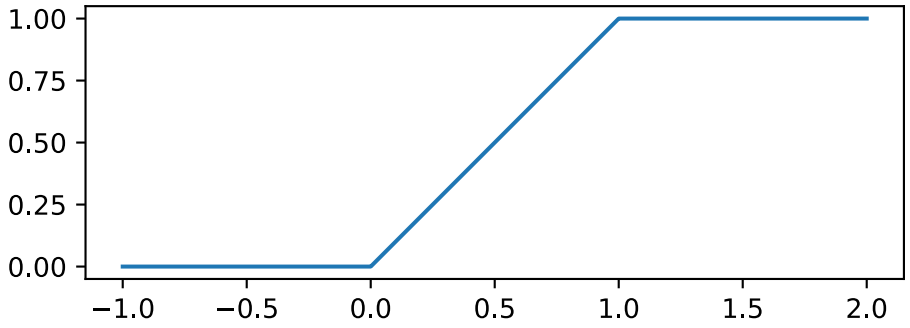
and the corresponding CDF

$$F(x) = \begin{cases} \frac{x-a}{b-a} & \text{if\_} a \leq x \leq b, \\ 0 & \text{else.} \end{cases}$$

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]
a = 0
b = 1
f = lambda x: np.astype((x >= a) & (x < b), float) / (b - a)
cfd = lambda x: (x - a) / (b - a) * np.astype((x >= a) & (x < b) , float) + \
                np.astype((x >= b) , float)
x = np.linspace(-1, 2, 1024)
plt.figure()
plt.plot(x, f(x))
plt.gca().set_aspect(3/(3 * 1))
plt.figure()
plt.plot(x, cfd(x))
plt.gca().set_aspect(3/(3 * 1))
plt.show()
```



(a) Probability density function.



(b) Cumulative distribution function.

Figure 15.4: Continuous uniform distribution - basic functions.

Example 15.6 (Normal distribution)**.** The normal distribution or Gaussian distribution is a continuous probability function for a real valued random variable.
The probability density function is

$$f(x) = \mathcal{N}[\mu, \sigma^2](x) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is called the mean and the parameter δ^2 is the called the variance. The standard deviation of the distribution is σ. For $\mu = 0$ and $\sigma^2 = 1$ it is called the **standard normal distribution**.
The corresponding CDF of the standard normal distribution is

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \mathrm{e}^{-\frac{t^2}{2}} \, \mathrm{d}t,$$

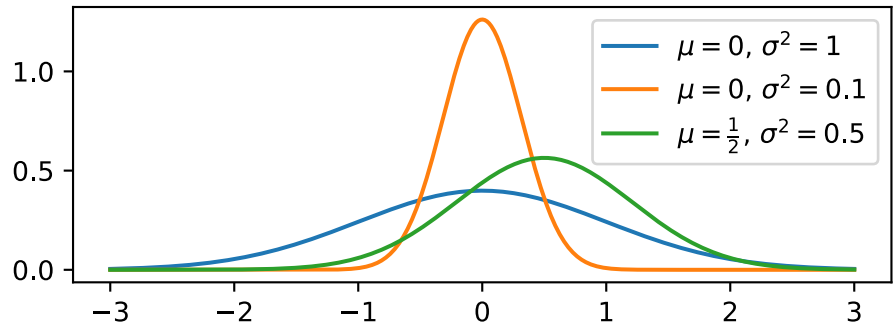
and the generic form is

$$F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right).$$

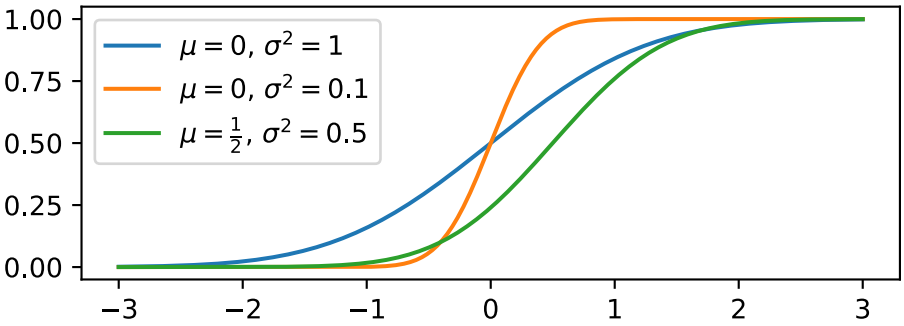
$\Phi(x)$ is expressed in terms of the error function

$$\Phi(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right), \quad \text{with} \quad \operatorname{erf}(x) = \frac{1}{\sqrt{2}}\int_0^x \mathrm{e}^{-t^2}\,\mathrm{d}t.$$

```
import numpy as np
import matplotlib.pyplot as plt
import math
%config InlineBackend.figure_formats = ["svg"]

f = lambda mu, sigma, x: 1 / np.sqrt(2 * np.pi * sigma**2) * \
    np.exp(- (x - mu)**2 / (2 * sigma**2))

phi = np.vectorize(lambda x: 1/2 * (1 + math.erf(x / np.sqrt(2))))
cfd = lambda mu, sigma, x: phi((x - mu) / sigma)
x = np.linspace(-3, 3, 1024)
plt.figure()
plt.plot(x, f(0, 1, x),
         label=r"$\mu=0$, $\sigma^2=1$")
plt.plot(x, f(0, np.sqrt(0.1), x),
         label=r"$\mu=0$, $\sigma^2=0.1$")
plt.plot(x, f(1/2, np.sqrt(0.5), x),
         label=r"$\mu=\frac{1}{2}$, $\sigma^2=0.5$")
plt.gca().set_aspect(6/(3 * 1.25))
plt.legend()
plt.figure()
plt.plot(x, cfd(0, 1, x),
         label=r"$\mu=0$, $\sigma^2=1$")
plt.plot(x, cfd(0, np.sqrt(0.1), x),
         label=r"$\mu=0$, $\sigma^2=0.1$")
plt.plot(x, cfd(1/2, np.sqrt(0.5), x),
         label=r"$\mu=\frac{1}{2}$, $\sigma^2=0.5$")
plt.gca().set_aspect(6/(3 * 1))
plt.legend()
plt.show()
```



(a) Probability density function.



(b) Cumulative distribution function.

Figure 15.5: Normal distribution - basic functions.

Example 15.7 (Inverse Γ distribution)**.** The main use of the inverse Γ distribution is in Bayesian statistics.
The probability density function is given by

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} \frac{1}{x^{\alpha+1}} \mathrm{e}^{-\frac{\beta}{x}},$$

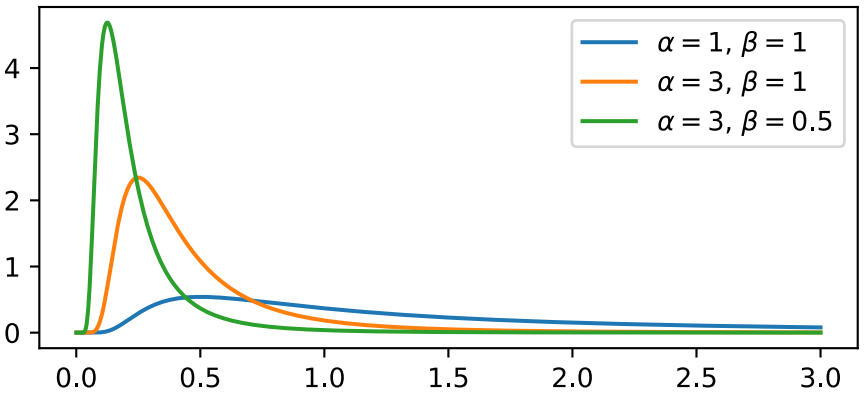
for the shape parameter α, the scale parameter β and the name giving Γ function.
The corresponding CDF is

$$F(x) = \frac{\Gamma\left(\alpha, \frac{\beta}{x}\right)}{\Gamma(\alpha)}$$

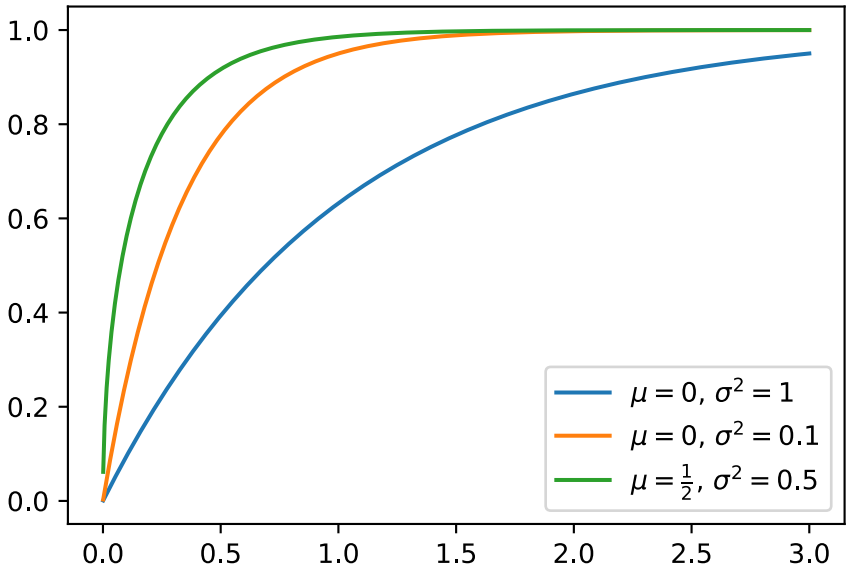
where the numerator is called the upper incomplete gamma function.

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

```
import math
import scipy
%config InlineBackend.figure_formats = ["svg"]

f = lambda alpha, beta, x: np.pow(beta, alpha) / math.gamma(alpha) * \
                           np.pow(1/x, 1 + alpha) * np.exp(-beta/x)

cfd = lambda alpha, beta, x: scipy.special.gdtr(alpha, beta, x)
x = np.linspace(1e-3, 3, 1024)
plt.figure()
plt.plot(x, f(1, 1, x), label=r"$\alpha=1$, $\beta=1$")
plt.plot(x, f(3, 1, x), label=r"$\alpha=3$, $\beta=1$")
plt.plot(x, f(3, 0.5, x), label=r"$\alpha=3$, $\beta=0.5$")
plt.gca().set_aspect(4/(3 * 5))
plt.legend()
plt.figure()
plt.plot(x, cfd(1, 1, x), label=r"$\mu=0$, $\sigma^2=1$")
plt.plot(x, cfd(3, 1, x), label=r"$\mu=0$, $\sigma^2=0.1$")
plt.plot(x, cfd(3, 0.5, x), label=r"$\mu=\frac{1}{2}$, $\sigma^2=0.5$")
plt.gca().set_aspect(6/(3 * 1))
plt.legend()
plt.show()
```



(a) Probability density function.



(b) Cumulative distribution function.

Figure 15.6: Inverse Gamma distribution - basic functions.

15.3 Bayes' theorem in action

What Bayes was after was the basic question of data science: find the parameters of a model given the outcome data together with the model. The theorem can be used as an update for the probability we have with additional information and therefore fitting our parameters.

Before we get to an example let us consider the terms of the theorem once more.

1. The prior expresses our initial believe of the outcome. If we know nothing about it we can use an non-informative prior like the uniform distribution.
2. The likelihood is similar to a probability distribution but the integral is not 1. As the name suggest it is the likelihood of a certain experiment result as a function of the parameters of the *model*. If we select the likelihood as a conjugate to the prior we know the shape of the posterior and this can speed up the calculation.
3. The denominator are there to normalize the posterior so that the posterior becomes a probability distribution with integral 1.
4. The posterior are our main result of the so called Bayers inference. If we have multiple samples we can use it as the prior for the next.

Let us look at some more elaborate examples to show how Bayes theorem can be used.

15.3.1 Election prediction as instructive example

Here we give a rudimentary introduction to election forecasts (this example follows the notes Mehrle (2024)).

> **Note**
>
> So far our probabilities where not continuous functions. For this example we need continuous functions and to reflect this in the notation we use small letters.

First we need to get a relative vote for party x which might come from the last elections or a prior poll. We assume $\mu = 20\%$ and express the uncertainty in our estimate with a normal distribution with standard deviation of $\sigma = 5\%$.

The resulting probability distribution becomes

$$p(x) = \mathcal{N}[\mu, \sigma^2](x) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

and in our case we get $p(x) = \mathcal{N}[0.2, 0.05^2](x)$.

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

normaldist = lambda mu, sigma, x: 1 / np.sqrt(2 * np.pi * sigma**2) * \
    np.exp(- (x - mu)**2 / (2 * sigma**2))

p = lambda x: normaldist(0.2, 0.05, x)
```

```
x = np.linspace(0, 1, 1024)
plt.figure()
plt.plot(x, p(x))
plt.grid("major")
plt.gca().set_aspect(1/(3 * 8))
plt.xlabel(r"$x$")
plt.ylabel(r"$p(x)$")
plt.show()
```

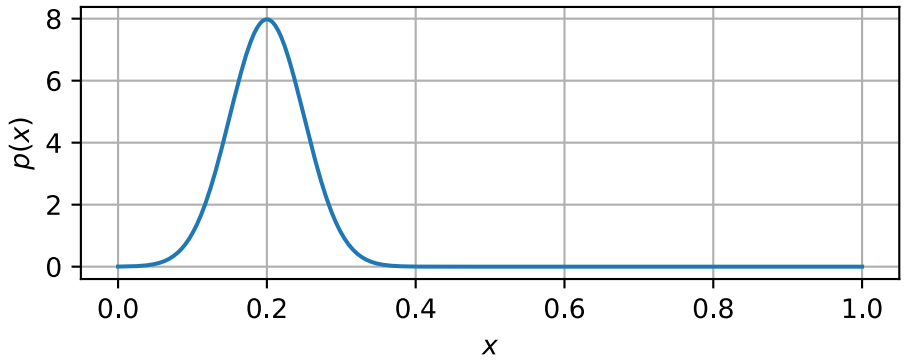


Figure 15.7: Probability density for our initial guess of the vote for party x.

Now we improvement our estimate with the help of Bayes' theorem by asking *random* people what they will vote.

> **Note**
>
> Of course this needs to be done in the proper way to make sure it is not biased. We assume to not be biased.

The probability that exactly k people of a random sample of size n vote for party x is a best described with a binomial distribution

$$p\left(\frac{k}{n}|x\right) = \left(\begin{array}{c} n \\ k \end{array} \right) x^k (1-x)^{n-k}.$$

```
import scipy

bino = lambda k, n, x: scipy.special.comb(n, k) * np.pow(x, k) * \
                       np.pow(1-x, n - k)
p2 = lambda x: bino(2, 10, x)
```

```
x = np.linspace(0, 1, 1024)
plt.figure()
plt.plot(x, p2(x))
plt.grid("major")
plt.gca().set_aspect(1)
plt.xlabel(r"$x$")
plt.ylabel(r"$p(\frac{5}{15}|x)$")
plt.show()
```

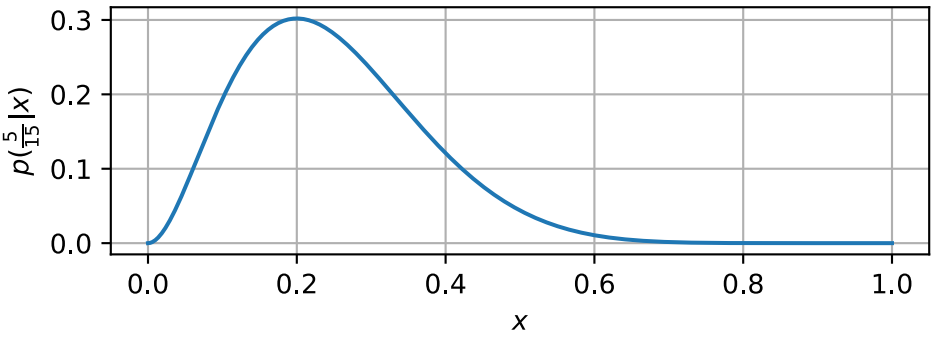


Figure 15.8: Probability density of k=5 positives out of n=15 samples.

In order to compute $p(\frac{k}{n})$ we need to integrate our two results

$$p\left(\frac{k}{n}\right) = \int_0^1 p\left(\frac{k}{n}|x\right) p(x)\,\mathrm{d}x$$

So for a given sample size n and positive samples k we get a new distribution.

```
p3 = lambda k, n: scipy.integrate.quad(lambda x: bino(k, n, x) * p(x), 0, 1)[0]

p4 = lambda k, n ,x: p(x) * bino(k, n, x) / p3(k, n)

x = np.linspace(0, 1, 1024)
plt.figure()
plt.plot(x, p4(5, 15, x), label=r"$n=5, k=15$")
plt.plot(x, p4(50, 150, x), ":", label=r"$n=50, k=150$")
plt.plot(x, p4(0, 15, x), "-.", label=r"$n=0, k=15$")
plt.legend()
plt.grid("major")
plt.gca().set_aspect(1 / (3 * 14))
plt.xlabel(r"$x$")
```

```
plt.ylabel(r"$p(x|\frac{k}{n})$")
plt.show()
```

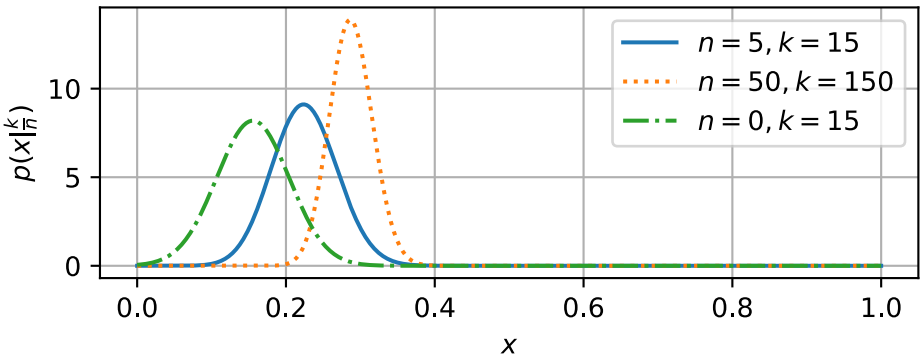


Figure 15.9: Parameter likelihood after a survey with k=5 positives out of n=15 samples (solid - blue) and k=15 out of n=150 samples (dotted).

In Figure 15.9 we can see that the peak moves from 0.2 closer to $\frac{1}{3}$ (our pooling results) and the bigger our sample size gets the more likely our change is (we get closer).

> **Warning**
>
> A point that is often criticised regarding the *Bayesian inference* we applied here is that it is highly dependent on the initial guess (Figure 15.7).

15.3.2 Material properties

To identify material properties such as strength we can also use Bayes' theorem.

> **Important**
>
> This example follows the notes Mehrle (2024) but as we do not have the data the plots are not generated interactively.

Again we start with an initial guess, in this case for the yield stress with 90% of the samples endure in our experiment.

For steel *S235JR* this is $R_{aH,0} = 235\text{MPa}$. We use a Gaussian normal distribution with $\mu_0 = 270\text{MPa}$ and $\sigma_0 = 27.3\text{MPa}$.

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ["svg"]

normaldist = lambda mu, sigma, x: 1 / np.sqrt(2 * np.pi * sigma**2) * \
    np.exp(- (x - mu)**2 / (2 * sigma**2))
Finv = lambda mu, sigma, p: mu + sigma * np.sqrt(2) * \
                            scipy.special.erfinv(2 * p - 1)

mu_0 = 270
sigma_0 = 27.31
RaH_0 = Finv(mu_0, sigma_0, 0.1)

p = lambda x: normaldist(mu_0, sigma_0, x)

x = np.linspace(0, 400, 1024)
plt.figure()
plt.plot(x, p(x))
plt.fill_between(x, p(x), where=(x > RaH_0))
my_max = np.round(np.max(p(x)), 3)
plt.plot([RaH_0, RaH_0], [0, my_max])
plt.grid("major")
plt.gca().set_aspect(400/(3 * my_max))
plt.xlabel(r"$x$")
plt.ylabel(r"$p(R_{aH})$")
plt.show()
```

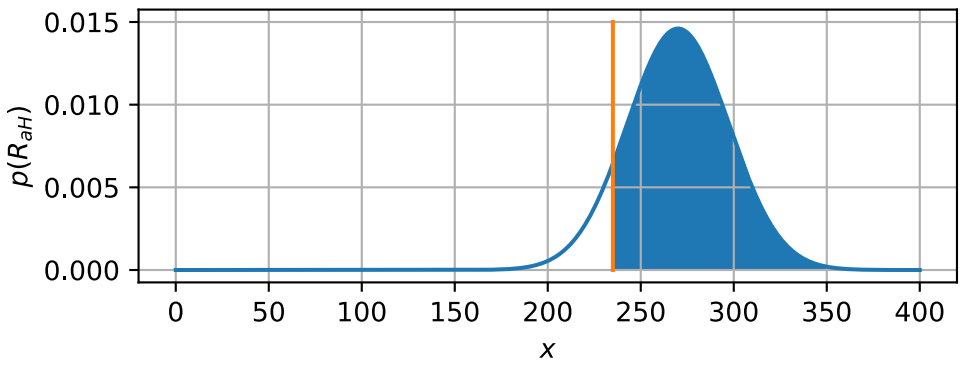


Figure 15.10: Initial assumption for the yield stress of the material, the coloured part represents the 90% of samples that have a yield stress larger than 235.

This time our parameters μ and σ are uncertain and need to be modified, furthermore, they do not follow a binomial distribution but our assumption is that they are better described by a normal distribution

$$p(\mu) = \mathcal{N}(\mu_0, \sigma_{\mu_0}), \quad \text{with} \quad \sigma_{\mu_0} = \frac{\mu_0}{10},$$

and

$$p(\sigma) = \mathcal{N}(\sigma_0, \sigma_{\sigma_0}), \quad \text{with} \quad \sigma_{\sigma_0} = \frac{\sigma_0}{10}.$$

The likelihood of drawing a sample with yield strength $R_{aH}^{(i)}$ we calculate the dependent probabilities separate by fixing the *current* values of $\mu = \check{\mu}$ and $\sigma = \check{\sigma}$, respectively.

$$p\left(R_{aH}^{(i)}|\mu\right) = \mathcal{N}[\mu, \check{\sigma}](R_{aH}^{(i)})$$

and

$$p\left(R_{aH}^{(i)}|\sigma\right) = \mathcal{N}[\check{\mu}, \sigma](R_{aH}^{(i)})$$

and with Bayes' theorem we can compute

$$p\left(\mu|R_{aH}^{(i)}\right) = \frac{p\left(R_{aH}^{(i)}|\mu\right) p(\mu)}{\int p\left(R_{aH}^{(i)}|\mu\right) p(\mu)\,\mathrm{d}\mu},$$

and

$$p\left(\sigma|R_{aH}^{(i)}\right) = \frac{p\left(R_{aH}^{(i)}|\sigma\right) p(\sigma)}{\int p\left(R_{aH}^{(i)}|\sigma\right) p(\sigma)\,\mathrm{d}\sigma},$$

Now by computing the inverse from our above probabilities we can update our μ_0 and σ_0 accordingly. This we can continue for several samples.

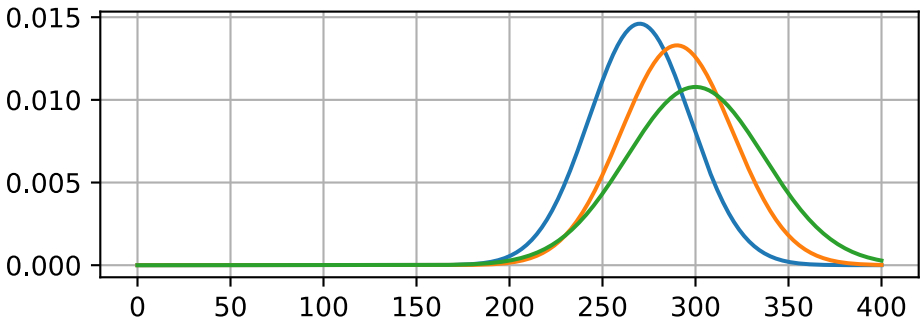


Figure 15.11: Update after 10 and 100 iterations

The mean converges relatively fast and stays but the deviation still changes after 100 sample drawn. This is due to the fact that the two variables are not independent of each other and the assumption that of a normal distribution is not a good estimate for σ.

15.4 Conjugate priors

The above updates are relatively expensive in terms of computing and are not handy. The idea is to find the updates analytically.

In general this is not *likely to happen* but for certain combinations of density functions the convert neatly into each other under Bayers' theorem.

For the normal distribution this is the case and the update becomes

$$\frac{1}{\sigma_{k+1}^2} = \frac{1}{\sigma_k^2} + \frac{1}{\sigma^2},$$

and

$$\mu_{k+1} = \left(\frac{\mu_k}{\sigma_k^2} + \frac{\mu}{\sigma^2} \right) \sigma_{k+1}^2,$$

16 Kalman Filter

The Kalman Filter is one of the most important contributions to engineering sciences in the 20th century and it helped to develop the space program.

The filter is an iterative process to estimate the parameters of that describe some system. In engineering terms it is used to estimate the underlying states of a state space model via a two-step predictor-corrector approach. For both, the state and the uncertainty P in this state, normal distributions are assumed and the updates are computed analytically.

> **!** Important
>
> We follow Faragher (2012) for this introduction.

The algorithm assumes that we have a state at time k that evolved from a prior state at time $k-1$ from the following equation

$$x_k = F_k x_{k-1} + B_k u_k + w_k, \tag{16.1}$$

with x_k being the state vector, u_k contains the control inputs like position, velocity, etc., F_k is called the state transition matrix, B_k the control input matrix, and last w_k contains the noise of the process. The process noise w_k is assumed to be drawn from a zero mean multivariate normal distribution with covariance matrix Q_k. Furthermore, we measure the system according to the model

$$z_k = H_k x_k + \nu_k$$

for z_k being the measurements, H_k the transformation matrix and ν_k contains the measurement noise.

For this introduction we us a simple example with a train going along a track. Therefore,

$$x_k = \begin{bmatrix} \mathrm{x}_k \\ \dot{\mathrm{x}}_k \end{bmatrix}.$$

As the train can be accelerated or the driver can use the break this will be our input parameters. We can model them as a function f_k and the train mass is constant with m. As a result u_t has the form

$$u_k = \frac{f_k}{m}.$$

We assume that to move from state $k-1$ to k the time Δt is needed. Hence,

$$\begin{aligned}\mathrm{x}_k &= \mathrm{x}_{k-1} + (\dot{\mathrm{x}}_{k-1}\Delta t) + \frac{f_k(\Delta t)^2}{2m} \\ \dot{\mathrm{x}}_k &= \dot{\mathrm{x}}_{k-1}\frac{f_k\Delta t}{m}.\end{aligned}$$

or in matrix form

$$x_k = \begin{bmatrix} \mathrm{x}_k \\ \dot{\mathrm{x}}_k \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{x}_{k-1} \\ \dot{\mathrm{x}}_{k-1} \end{bmatrix} + \frac{f_k}{m} \begin{bmatrix} \frac{(\Delta t)^2}{2} \\ \Delta t \end{bmatrix}$$

and we can see the matrices of our initial problem statement. As it is not possible to observe the true state of the system directly we use the Kalman filter as an algorithm to determine an estimate of x_k. As the system includes uncertainties the state is given in terms of probability density functions rather than discrete values.

So how can we derive an estimate for the current step from known steps and the uncertainty we have in the system. This uncertainty is model as normal distribution and the parameters are stored in the covariance matrix P_k (zero mean is assumed). As mentioned before we need to do a prediction and the standard equation for the Kalman filter is

$$\hat{x}_{k|k-1} = F_k\hat{x}_{k-1|k-1} + B_k u_k, \tag{16.2}$$

and

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^{\mathsf{T}} + Q_k.$$

Here $\star_{k|k-1}$ denotes the state at time t_k derived from the state at time t_{k-1}.

To get the second equation we need to know that the variance associated with the prediction $\hat{x}_{k|k-1}$ of the unknown true value x_k is given by

$$P_{k|k-1} = E\left[(x_k - x_{k|k-1})(x_k - x_{k|k-1})^{\mathsf{T}}\right].$$

If we now compute the difference of Equation 16.1 and Equation 16.2 we note that the state estimation errors as well as the process noise are uncorrelated and therefore their correlation is zero.

The measurement update equations are given by

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H_k\hat{x}_{k|k-1})$$

and

$$P_{k|k} = P_{k|k-1} + K_k H_k P_{k|k-1}$$

for

$$K_k = P_{k|k}H_k^{\mathsf{T}}(H_k P_{k|-1}H_k^{\mathsf{T}} + R_k)^{-1}.$$

Here, R_k is the covariance of the measurement noise.

To make (more) sense of the above formula we should consider the train example in more detail.

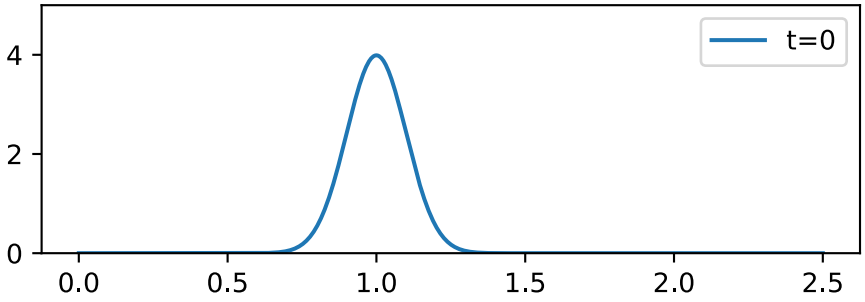
At time $t = 0$ (and $k = 0$) our tain is at a specific position but due to measurement uncertainty this position is expressed as a probability distribution. As mentioned before we use a normal Gaussian distribution with mean μ_0 and variance δ_0^2.

> **Note**
>
> Going back to our model description before, we have $\hat{x}_{1|0} = \mu_0$ and $P_{1|0} = \sigma_0^2$.

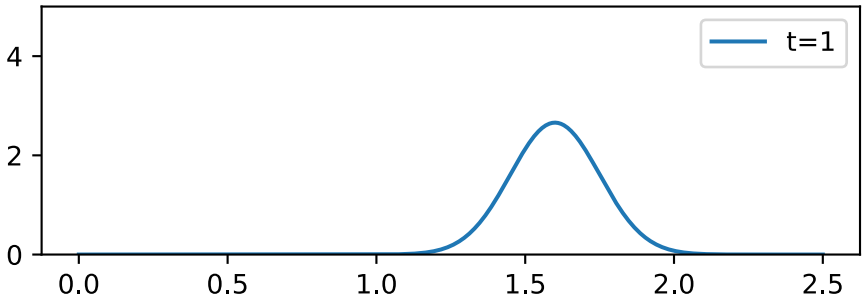
```
import numpy as np
import matplotlib.pyplot as plt
import math
%config InlineBackend.figure_formats = ["svg"]

f = lambda mu, sigma, x: 1 / np.sqrt(2 * np.pi * sigma**2) * \
                         np.exp(- (x - mu)**2 / (2 * sigma**2))

x = np.linspace(0, 2.5, 1024)
plt.figure()
plt.plot(x, f(1, 0.1, x), label=r"t=0")
plt.gca().set_aspect(2.5 / (3 * 5))
plt.ylim([0, 5])
plt.legend()
plt.figure()
plt.plot(x, f(1.6, 0.15, x), label=r"t=1")
plt.gca().set_aspect(2.5 / (3 * 5))
plt.legend()
plt.ylim([0, 5])
plt.show()
```

Now we can predict the position of the train at $t = 1$ ($k = 1$) with the data we know at time $t = 0$ like its maximum speed, possible acceleration and deceleration, etc.. This is can be done via Equation 16.1. We can see a possible outcome in Figure 16.1b where the train has moved in positive direction but the variance has increased indicating the increased uncertainty in the position du to the fact that we do not have noise from the acceleration or deceleration happening in the span Δt.

Now at time $t = 1$ we can measure the position of the train. We do so via a radio signal and of course this process is also having some noise added to it resulting in a Gaussian probability to the position.



(a) Position at time t=0.



(b) Position at time t=1.

Figure 16.1: Position of the train

! Important

A key feature of the normal (Gaussian) distribution is that fact that the *product of two Gaussian distributions* is again a Gaussian distribution.
This process can be repeated over and over again and we always end up with a Gaussian distribution. This is a main feature in the Kalman filter.

$$\begin{aligned}\mathcal{N}[\mu_1,\sigma_1](x)\cdot\mathcal{N}[\mu_2,\sigma_2](x) &= \frac{1}{\sqrt{2\pi\sigma_1^2}}\mathrm{e}^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}}\cdot\frac{1}{\sqrt{2\pi\sigma_2^2}}\mathrm{e}^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}}\\ &= \frac{1}{2\pi\sqrt{\sigma_1^2\sigma_2^2}}\mathrm{e}^{-\left(\frac{(x-\mu_1)^2}{2\sigma_1^2}+\frac{(x-\mu_2)^2}{2\sigma_2^2}\right)}\\ &= \frac{1}{\sqrt{2\pi\sigma^2}}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}\\ &= \mathcal{N}[\mu,\sigma](x)\end{aligned}$$

with

$$\mu = \frac{\mu_1\sigma_2^2+\mu_2\sigma_1^2}{\sigma_1^2+\sigma_2^2} = \mu_1+\frac{\sigma_1^2(\mu_2-\mu_1)}{\sigma_1^2+\sigma_2^2} \qquad \sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2+\sigma_2^2} = \sigma_1^2-\frac{\sigma_1^4}{\sigma_1^2+\sigma_2^2} \tag{16.3}$$

Let us use this to get an estimate of our train after the measurement by multiplying the two Gaussian distributions, as seen in Figure 16.2.

```
import numpy as np
import matplotlib.pyplot as plt
import math
%config InlineBackend.figure_formats = ["svg"]

f = lambda mu, sigma, x: 1 / np.sqrt(2 * np.pi * sigma**2) * \
    np.exp(- (x - mu)**2 / (2 * sigma**2))

update = lambda mu, sigma: ((mu[0] * np.pow(sigma[1], 2) + mu[1] *
                             np.pow(sigma[0], 2)) / \
                            (np.sum(np.pow(sigma, 2))),
                            np.prod(sigma) / np.sqrt(np.sum(np.pow(sigma, 2))))

x = np.linspace(0, 2.5, 1024)
plt.figure()
plt.plot(x, f(1.6, 0.2, x), label=r"predictor")
plt.plot(x, f(2, 0.1, x), label=r"corrector")
plt.plot(x, f(*update([1.6, 2.0], [0.2, 0.1]), x), label=r"updated position")
plt.gca().set_aspect(2.5 / (3 * 5))
plt.legend()
```

```
plt.ylim([0, 5])
plt.show()
```

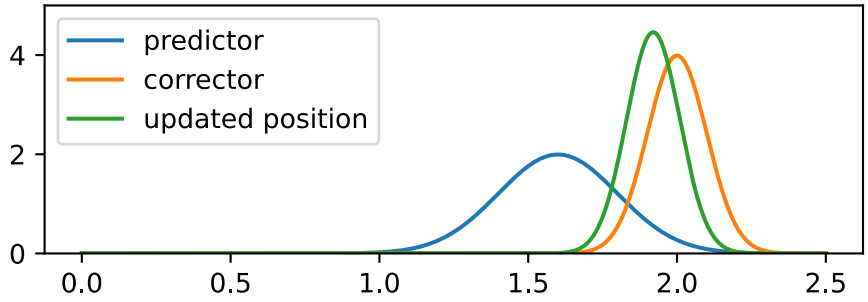


Figure 16.2: Position of the train at t=1 with measurement and resulting distribution.

The new distribution considers the information of our guess and the update provided by the measurement and therefore our best estimation of the current position of the train. Equation 16.3 represents the update step of the Kalman filter Equation 16.2.

To bring in all the components of the Kalman filter we assume that the measurement and the prediction is not in the same domain and therefore we need the matrix H_k.

To do so assume that the signal for the estimate is a radio signal that travels with the speed of light c and therefore we need to transform $\mathcal{N}[\mu_2, \sigma_2](x)$ as $\mathcal{N}[\frac{\mu_2}{c}, \frac{\sigma_2}{c}](x)$ and our update becomes

$$\mu = \mu_1 + K(\mu_2 - H\mu_1) \qquad \sigma^2 = \sigma_1^2 - KH\sigma_1^2,$$

with

$$H = \frac{1}{c} \qquad K = \frac{H\sigma_1^2}{H^2\sigma_1^2 + \sigma_2^2}.$$

> **Note**
>
> Going back to our model description before, we have $z_k = \mu_2$ and $R_k = \sigma_2^2$, $H = H_k$, $K = K_k$.

This the qualities given in the notes it is easy to see how the Kalman filter fits into our little illustration.

There are three main applications for the Kalman filter in engineering which differ mainly in the input provide to the model:

1. Measurement - since by definition the input is not known, it is neglected in the model ($u_k = 0$)
2. Control - the input is calculated by the controller and therefore known and considered ($u_k = -K_k x_k$)
3. Sensor fusion - the input is provided by a measurement from a complementary sensor ($u_k = z_k$)

Summary

This concludes our introduction to the basis of data science with a focus on engineering topics.

These notes introduced the main concepts with a clear focus of accurate mathematical representation and close illustration with programmatic examples.

If we need to sum up the main concept that is present in large sections of these notes it is that the correct representation is key in finding good concepts of computer based processing, To this extend the second key concept is to make sure the concepts can be handled via a computer and the student, so theory and programmatic application go hand in hand.

Both concepts stay true if we move to more evolved data science methods in further classes.

References

Brunton, Steven L., and J. Nathan Kutz. 2022. *Data-Driven Science and Engineering - Machine Learning, Dynamical Systems, and Control*. 2nd ed. Cambridge: Cambridge University Press.

Candès, Emmanuel J., Xiaodong Li, Yi Ma, and John Wright. 2011. "Robust Principal Component Analysis?" *J. ACM* 58 (3). https://doi.org/10.1145/1970392.1970395.

Cooley, James W., and John W. Tukey. 1965. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation* 19 (90): 297–301. https://doi.org/10.1090/s0025-5718-1965-0178586-1.

Downey, Austin, and Laura Micheli. 2024. "Vibration Mechanics: A Practical Introduction for Mechanical, Civil, and Aerospace Engineers." https://doi.org/10.5281/ZENODO.12539013.

Faragher, Ramsey. 2012. "Understanding the Basis of the Kalman Filter via a Simple and Intuitive Derivation." *IEEE Signal Processing Magazine* 29 (5): 128–32.

Gauß, Carl Friedrich. 1866. "Theoria Interpolationis Methodo Nova Tractata." Göttingen: Königliche Gesellschaft der Wissenschaften.

Golub, Gene H., and Charles F. Van Loan. 2013. *Matrix Computations*. 4th ed. Johns Hopkins Studies in the Mathematical Sciences.

Inden, Michael. 2023. *Python Lernen – Kurz & Gut* -. Sebastopol: O'Reilly.

Lambert, Ben. 2018. *A Student's Guide to Bayesian Statistics*. London, England: SAGE Publications.

Manohar, Krithika, Bingni W. Brunton, J. Nathan Kutz, and Steven L. Brunton. 2018. "Data-Driven Sparse Sensor Placement for Reconstruction: Demonstrating the Benefits of Exploiting Known Patterns." *IEEE Control Systems Magazine* 38 (3): 63–86. https://doi.org/10.1109/MCS.2018.2810460.

Matthes, Eric. 2023. *Python Crash Course - a Hands-on, Project-Based Introduction to Programming*. 3rd ed. No Starch Press. https://ehmatthes.github.io/pcc_3e/.

McKinney, Wes. 2022. *Python for Data Analysis 3e*. 3rd ed. Sebastopol, CA: O'Reilly Media. https://wesmckinney.com/book/.

Mehrle, Andreas. 2024. "Data Science." *Management Center Innsbruck, Reader for the Course*.

Meyberg, Kurt, and Peter Vachenauer. 1992. *Höhere Mathematik 2*. Springer-Lehrbuch. New York, NY: Springer.

Needell, D., and J. A. Tropp. 2009. "CoSaMP: Iterative Signal Recovery from Incomplete and Inaccurate Samples." *Applied and Computational Harmonic Analysis* 26 (3): 301–21. https://doi.org/https://doi.org/10.1016/j.acha.2008.07.002.

Ritchie, Hannah, Lucas Rodés-Guirao, Edouard Mathieu, Marcel Gerber, Esteban Ortiz-Ospina, Joe Hasell, and Max Roser. 2023. "Population Growth." *Our World in Data*.

Vasiliev, Yuli. 2022. *Python for Data Science - a Hands-on Introduction*. München: No Starch Press.