Co-funded by the
Erasmus+ Programme
of the European Union

Mathematical Methods using Python

Course Notes

Tom Trigano

BIOART
Engineering education
for human welfare

Contents

General Ideas about Python

1. Basic commands in Python

This introductory chapter aims to be a general, introduction to Python. It is not destined to a computer science audience, but rather to people whose major is not computer programming (e.g., people coming from the Electrical, Chemical, Mechanical Engineering world).

1.1 Let's get it on

1.1.1 What is Python?

Python is a general-purpose programming language, built on top of C. Its main advantage (besides its cost!) lies in its versatility. Indeed, many tasks (Machine Learning, data analysis, web, robotics, etc.) can be easily performed due to a huge community of Python developers, which provide free toolboxes (packages, libraries, wrappers) for all purposes.
Python is a scripting language (meaning it is slower than plain C), but due to numerous optimizations it can run fast enough for prototyping. Even so, its advantage when compared to C is that it is very easy to prototype.

1.1.2 Installing Python

Python is freely downloadable at the website `Python.org`, and exists for all platforms. It is extremely easy to install and maintain in Mac and Linux, using standard tools (wheel and pip), but the management under Windows is a bit more tricky. For Windows users, the use of the Anaconda (freely available) is recommended, since it allows to create different working environments.
In order to program, we will also need an editor. For standard scripting and execution via the console, Visual Studio Code is recommended. For live execution, Jupyter will be used all along this book (though alternatives exist, e.g. Spyder)

1.1.3 Hello world

The following line can be save into a file (say,*helloworld.py*), and be executed by calling the python interpreter (*python helloworld.py*). Alternatively, this line can be executed directly into Jupyter.

```python
print('Hello world!')
```

Listing 1.1: *Python "Hello World" code*

1.1.4 Python caveats

- Everything is object: meaning that each variable may have several methods included in it. The following lines, for example, replaces 'a' with 'e' everywhere.

```python
text = 'Python is tha coolast'
new_text=text.replace('a','e')
print(new_text)
```

Listing 1.2: *string basic manipulation*

- Everything is memory access: meaning that there is no copy of variables contrary to Matlab.

```python
a = ['P','y','t','h','o','n']
b = a[3::]
b[0] = 'A'
print(a)
```

Listing 1.3: *basic slicing*

- Everything is simple: meaning that no indexations are required, for example to iterate

```python
a = [0,1,3.14,'a','Python',[0,1]]
for value in a:
    print(value)
```

Listing 1.4: *basic slicing*

The overall philosophy of Python is humorously summarized inside the interpreter and denoted by "the Zen of Python"...

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900 64 bit (
    AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Listing 1.5: *Python philosophy*

1.2 Variable types

Similarly to C, Python is typed, meaning that each variable has an associated type. We review here the most commonly used types (warning: comparing variables with different types may results in an error)

1.2.1 Simple types

Name	Type	Possible values examples
`bool`	Boolean value	True, False
`int`	Integer of unlimited magnitude	42
`float`	Floating point number, system-defined precision	3.1415927
`str`	A character string	’hello’, """Spanning multiple lines"""

1.2.2 Complex types

Name	Type	Possible values examples
`dict`	dictionary of key and value pairs; can contain mixed types	`{'key1': 1.0, 3: False}`
`list`	list, can contain mixed types	`[4.0, 'string', True]`
`tuple`	unchangable list	`(4.0, 'string', True)`

1.3 Conditional programming

Several times, you wish to run some lines of codes if a specific condition holds, or a certain amount of times only, or until a certain condition is met but it is difficult to know when exactly. As most programming languages, Python has this covered.
Python supports the usual logical conditions from mathematics:

- Equals:

```
a == b
```

- Differs:

```
a != b
```

- Lesser (or equal) than:

```
a < b
a <= b
```

- Greater (or equal) than:

```
a > b
a >= b
```

Python relies on indentation, using whitespace, to define scope in the code. Other programming languages often use curly-brackets for this purpose, but this approach forces the programer to be organized and to produce readable codes. For example, the follow code will not work, because indentations (when entering inside the function code and loops) are missing:

```
def test_greater(a=14,b=3):
if b > a:
print('b is greater than a')
else:
print('b is smaller than a or equals to a')
```

Listing 1.6: *Code with a bug: indentation is missing*

The correct code should be as follows:

```
def test_greater(a=14,b=3):
    if b > a:
        print('b is greater than a')
    else:
        print('b is smaller than a or equals to a')
```

Listing 1.7: *Corrected code*

1.3.1 If . . . else . . . statements

The ”if. . . else. . . ” statement can be used to execute different parts of code whether a certain condition is verified or not. For example, the following code check whether two lists have a similar length or not:

```
a = [1, 2, 3, 4]
b = ['Tom','Kfir','Irit','Guy','Amir']
if len(a) == len(b):
    print('{} and {} have same length'.format('a','b'))
else:
    print('{} and {} have not the same length'.format('a','b'))
```

Listing 1.8: *Example of if statement*

1.3.2 For statements

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). It works as an iterator as in $C++$. With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc. Iterations can be controlled by the two following keywords:

- **break**: will stop the for loop and continue the program after the loop
- **continue**: will stop the current iteration inside the for loop and will jump to the next iteration.

Loops can be nested into other loops, similarly to other languages. However, when iterating over objects of similar sizes, you should consider the use of **zip** as an alternative.

```
market = ['apple','pear','banana','apple','pineapple','banana','coconut','
    banana','strawberry','banana']
basket = []
for fruit in market:
    if fruit=='banana':
        print('I dont like bananas!')
        continue
    if len(basket)==5:
        print('I dont want to eat to much!')
        break
    basket.append(fruit)

print('I have {} fruits in my basket, and here they are: {}'.format(len(basket)
    ,basket))
```

Listing 1.9: *Look at the following code: how many fruits do you have at the end of the loop in your basket?*

1.3.3 While statements

With the while loop we can execute a set of statements as long as a condition is true. Similarly to the for, we can control the loop with ”break” and ”continue”.

```
market = ['apple','pear','banana','apple','pineapple','banana','coconut','
    banana','strawberry','banana']
basket = []
n=0
```

```
while len(basket)!=3:
    fruit=market[n]
    n += 1
    if fruit in ['banana','apple']:
        print('I dont like bananas, or apples!')
        continue
    basket.append(fruit)

print('I have {} fruits in my basket, and here they are: {}'.format(len(basket)
    ,basket))
```

Listing 1.10: *Look at the following code: how many fruits do you have at the end of the loop in your basket?*

1.4 Functions in Python

Functions are an important part of modularity in computer programming. It allows to encapsulate parts of a program in a separate function, which will be called later. The generic syntax for a function is

```
def my_function(param_1, param_2, optional_param=1, optional_param_2 = 'yes'):
    '''
    This function does...
    '''
    # Code goes here

    return output # optional, since referencing is OK
```

Listing 1.11: *Function syntax*

In this syntax, parameters whose values are not specified are mandatory, whereas the other are optional (values will be passed on based on the definition).
Functions are very important to get a readable code. Very. When saving a set of functions in a separate file or module (say, *my_module.py*), the functions can be there loaded all or separately:

```
# will import all the functions at once
import my_module
# will import only one function from the module
from my_module import this_function
```

Listing 1.12: *Importing functions*

It is also possible to define small, anonymous functions, when we need this function locally or to define a series of parametric function:

```
# the lambda function
def my_parametric_function(a):
    return lambda x : x + a

# defining a series of functions
f1 = my_parametric_function(1)
f2 = my_parametric_function(2)

print(f1(3))
print(f2(3))
```

Listing 1.13: *Lambda functions*

1.4.1 A final word of advice

The author of this booklet (an avid user of Matlab) often hears in his classroom the following question:

"Why don't you teach Python instead of Matlab?"
To the author's point of view, this question makes little sense. Both are nice, easy to learn, scripting language. Python is extremely developed in several fields of computer science and machine learning, and lacks the nice model-based programming capabilities and stability of Matlab. Similarly, Python handles data in a much nicer way than Matlab, but involves usually a longer trial and error cycle in order to use freely available codes. On the other hand, we cannot ignore the economic aspects (most startups in Israel develop in Python due to the cost of the Matlab license, while big companies tend to use Matlab for its stability). Therefore, my very humble opinion in the matter is that a good engineer show know both, because he never know where he will eventually land.

1.5 Case study: playing Tic-Tac-Toe

Though it seems trivial, programming a tic-tac-toe is a great wrapping up example: it goes over basic programming, and forces to decompose the algorithm into small parts, making it easy to address. Though we could push the example further (e.g. by adding a GUI), let us focus only on a small text application.

1.5.1 Block diagram

A typical turn in the game goes like this:
1. the player whose turn it is chooses a box in the 3×3 grid.
2. if the chosen box does not exist or is occupied, we go back to step 1.
3. if the chosen box exists and is free, we put either an 'X' or 'O' (depending on the player).
4. if there are three 'X' or 'O' in a row, a column or in diagonals, the current player wins. The game ends here.
5. if no one wins but all the boxes are occupied, this is a draw; the game ends here.
6. otherwise, the next player goes over the same procedure.

1.5.2 The code

```python
def tic_tac_toe():

    game_board = [[' ', ' ', ' '],[' ', ' ', ' '],[' ', ' ', ' ']]
    end_game = False
    symbols=['X','O']
    player = 0
    display_board(game_board)

    for n in range(100):

        if end_game is True:
            break

        box = raw_input('Player {}, enter your move (A1,C2, etc...)'.format(
player+1))

        (is_free,bad_location) = check_if_free(game_board,box)

        if is_free and not bad_location:
            update_board(game_board,box,symbols[player])
            if current_player_wins(game_board):
                display_board(game_board)
                print('Player {} wins! Congrats!'.format(player+1))
                end_game = True
                continue
            player = (player + 1) % 2


        if not is_free:
            print('Place already taken! Again!')
            continue

        if bad_location:
            print('Not in the board! Again!')
            continue


        if check_draw(game_board):
            print('Draw game')
            end_game = True

        display_board(game_board)
```

Listing 1.14: *Main function*

```
def display_board(L):
    """Displays the board as it is now"""
    print('    A    B    C')
    for count,row in enumerate(L):
        print("{} {}".format(count,row))
```

Listing 1.15: *Displays the board in ASCII*

```
def check_if_free(L,box):

    return_value = False
    bad_location = False

    if box[0] not in ['A', 'B', 'C']:
        bad_location=True
        return (return_value,bad_location)

    if int(box[1]) not in [0,1,2]:
        bad_location = True
        return (return_value,bad_location)

    idx = ['A','B','C'].index(box[0])

    if L[int(box[1])][idx] == ' ':
        return_value = True
    return (return_value,bad_location)

def update_board(L,box,symbol):
    idx2 = ['A','B','C'].index(box[0])
    idx1 = int(box[1])
    L[idx1][idx2] = symbol
```

Listing 1.16: *Checks if a box is free and update the game board*

```
def current_player_wins(L):
    winning_combinations = ['XXX','OOO']
    states=[r[0]+r[1]+r[2] for r in L] + [L[0][k]+L[1][k]+L[2][k] for k in
    range(3)] + [L[0][0]+L[1][1]+L[2][2]] + [L[2][0]+L[1][1]+L[0][2]]

    win = False
    if 'XXX' in states:
        win = True
    if 'OOO' in states:
        win = True

    return win

current_player_wins(game_board)

def check_draw(L):
    table = L[0]+L[1]+L[2]
    result = 9
    is_draw = False
    for x in table:
        if x == ' ':
            result -= 1
    if result == 9:
        is_draw = True
    return is_draw
```

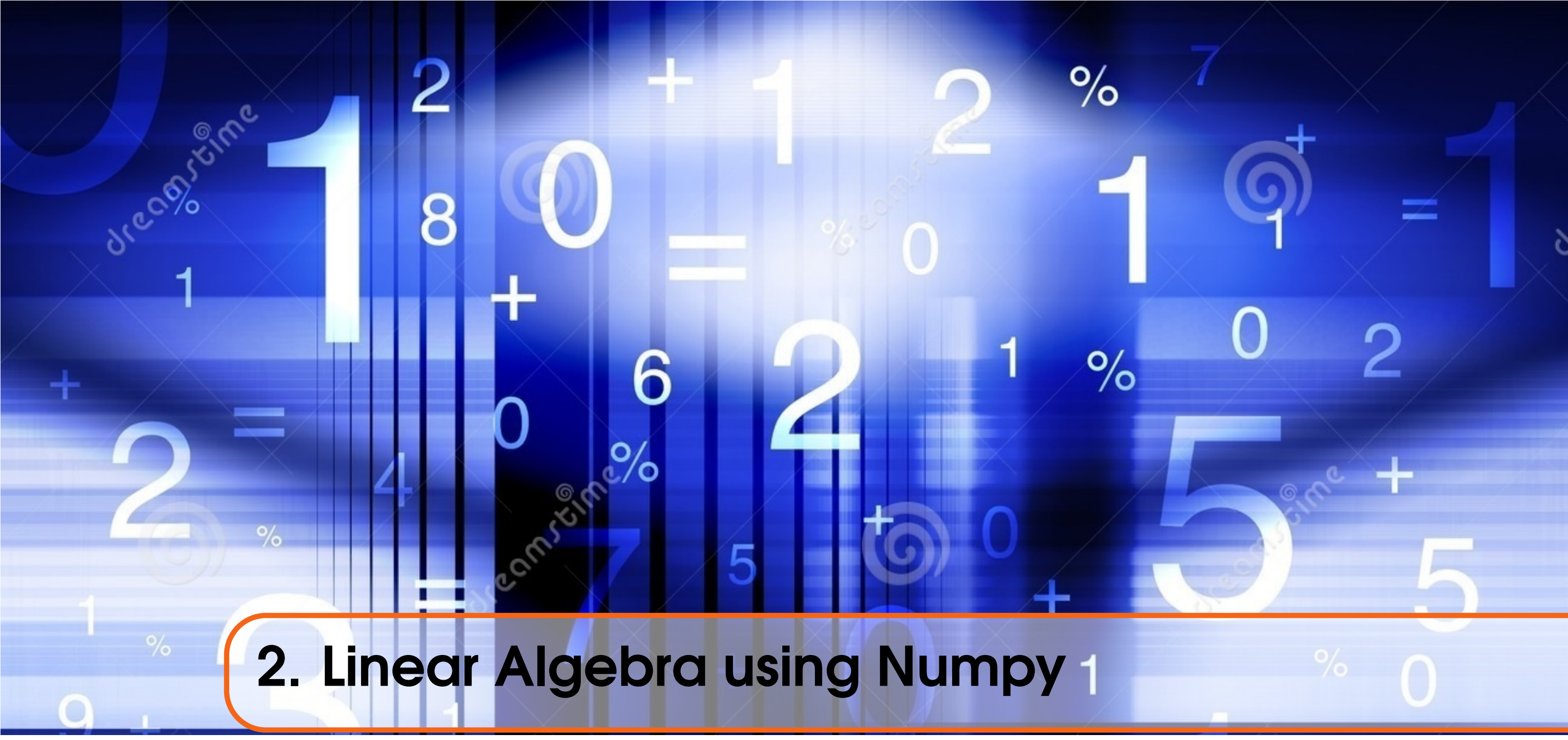
Listing 1.17: *Functions which checks if there is a win or a draw*

1.6 References

There are many references for Python programming

II

Linear Algebra



2. Linear Algebra using Numpy

2.1 Reach Out, I'll be There

Linear algebra is a field of mathematics which is encountered in all the fields of engineering. Among many uses, we can quote:

1. Modeling stress on materials in civil engineering,
2. Analyzing linear circuits using state-space models in electrical engineering,
3. modeling degrees of freedom for solids in mechanical engineering,
4. solving numerically differential equations
5. linear algebra is also an inherent part of most modern machine learning algorithms

The package `numpy` includes numerous functions to perform matrix and array manipulations, similarly to Matlab. This package is usually called as follows:

```python
import numpy as np
```

Listing 2.1: *Calling the numpy package*

Calling the package `np` is not mandatory; however, most developers use this convention, so to write code compliant to other libraries we strongly recommend to use the same convention as well.
For a more intuitive use of arrays similar to Microsoft Excel worksheets, the use of Pandas is also a nice alternative. It may be called as

```python
import pandas as pd
```

Listing 2.2: *Calling the pandas package*

2.2 Return on Linear Algebra

This chapter will present a short reminder on standard linear algebra operations. Besides the technical aspects of these operations, and their Python implementation, it is also important to understand the motivation for such operations.

Definition 2.2.1 A vector space is a set V of elements (the vectors) associated with a set of numbers K (usually $\mathbb{R}$ or $\mathbb{C}$) such that the following calculation rules are true:

- for all $\mathbf{u}, \mathbf{v} \in V$, $\mathbf{u} + \lambda \mathbf{v}$
- $\mathbf{u} + \mathbf{0} = \mathbf{u}$ (there exists a null element)
- $\lambda(\mathbf{x} + \mathbf{y}) = \lambda \mathbf{x} + \lambda \mathbf{y}$
- $(\lambda + \mu)\mathbf{x} = \lambda \mathbf{x} + \mu \mathbf{x}$
- $\lambda(\mu \mathbf{x}) = (\lambda \mu)\mathbf{x}$

As understood, the formal definition is very generic, and any space equipped with conveniently set calculation rules can be considered as a vector. Moreover, it is often of interest to equip this vector space of an inner product, this to quantify numerically how similar two vectors look like. A norm is usually induced from this inner product.[1]

2.2.1 Matrices and Linear Applications

Definition 2.2.2 — Linear mapping. A linear application is an application from one vector space (say, E) to another vector space (say, F), such that the following properties hold:

1. $f(\mathbf{u} + \mathbf{v}) = f(\mathbf{u}) + f(\mathbf{v})$
2. $f(\alpha \mathbf{u}) = \alpha f(\mathbf{u})$ for all $\mathbf{u} \in E, \alpha \in K$

This definition is general, and holds for any vector space. However, when both E and F have a finite dimension, the definition of a linear application can be summarized by means of an array of number, as seen below.
Loosely speaking, assuming that E has a finite dimension means that each vector $\mathbf{x} \in E$ can be written as a unique linear combination of a finite family of vectors $\mathbf{e}_i, i = 1 \dots n$, that we will call a basis:

$$\mathbf{x} = \sum_{k=1}^{n} \lambda_k \mathbf{e}_k.$$

For example, it is common to represent a point in our three dimensional world as three coordinates (and the uniqueness of these coordinates is the cornerstone of GPS positioning). There, we shall say that $E = \mathbf{R}^3$ is a vector space with dimension $\dim E = 3$. It also means there is a duality between a vector space with finite dimension and a defining basis[2]
Now, we will consider a linear application f from E to F, and that both E and F are vector spaces with finite dimension (not necessarily equal). We will also define $\mathbf{e}_i,\ i = 1 \dots n$ and $\mathbf{f}_j,\ j = 1 \dots m$ bases associated with E and F, respectively. Since bases define all the vectors in E and F, it is sufficient to known $f(\mathbf{e}_i),\ i = 1 \dots n$ to fully characterize the application f. Thus, we can characterize the linear application as

$$f(\mathbf{e}_j) = \sum_{i=1}^{m} a_{ij} \mathbf{f}_i,$$

and it is therefore convenient to summarize a linear application in an array of numbers, called *matrix*:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \tag{2.2.1}$$

[1]We do not aim in this chapter to make an exhaustive presentation of linear algebra: numerous - well-known - textbooks exist in the matter, and this incomplete presentation could appear plain stupid in comparison. However, not understanding the basic significance of the objects we manipulate at a daily basis is also plain stupid.

[2]Note that there is in fact an infinite number of bases which can characterize the same vector space. This is the underlying justification for dimensionality reduction in statistics.

where in (2.2.1) the first column includes the decomposition of $f(\mathbf{e}_1)$ in F, the second column includes the decomposition of $f(\mathbf{e}_2)$ and so forth. Consequently, the rules of matrix calculations are induced by their very construction:

- Matrices can be added only if they have the same dimension (since we can only add linear applications on the same vector spaces)
- Matrix multiplication follow the rules of applications' composition, since $\mathbf{ABx} = f_A(f_B(\mathbf{x}))$
- Matrix multiplication is not commutative
- and so forth. Matrix calculus rules are not "magic", they follow the rules of use of linear applications.

As an example, the following code performs the following operation:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```python

import numpy as np
import pprint # nice print of arrays

# one projection matrix on the plane z=0
A = np.zeros((2,3))
A[0,0] = 1
A[1,1] = 1
pprint.pprint(A)

# we project the vector x = [1 2 3]^T , and compute Ax
x = np.zeros((3,1))
x[0] = 1
x[1] = 2
x[2] = 3
y = A.dot(x)
pprint.pprint(y)
```

Listing 2.3: *matrix vector multiplication*

2.2.2 Standard Matrix Operations

The following operations can be performed on matrices (and it is always a good idea to remember the linear applications associated):

- Addition/subtraction: $\mathbf{A}+\mathbf{B}$ represent the addition of two linear applications $(f_A + f_B)$ applied on the same vector spaces, and we get:

$$\begin{aligned}
&\mathbf{A}+\mathbf{B} \\
&= \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \dots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix} \\
&= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \dots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \dots & a_{2n}+b_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \dots & a_{mn}+b_{mn} \end{bmatrix},
\end{aligned}$$

  This operation can be done in Python using the regular $+$ operations on Numpy arrays.

- Multiplication: **AB** represents the composition of two linear applications $f_A(f_B(\cdots))$:

$$
\begin{aligned}
&\mathbf{AB} \\
&= \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \dots & \vdots \\ a_{q1} & a_{q2} & \dots & a_{qm} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \dots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=1}^m a_{1i}b_{i1} & \sum_{i=1}^m a_{1i}b_{i2} & \dots & \sum_{i=1}^m a_{1i}b_{in} \\ \sum_{i=1}^m a_{2i}b_{i1} & \sum_{i=1}^m a_{2i}b_{i2} & \dots & \sum_{i=1}^m a_{2i}b_{in} \\ \vdots & \vdots & \dots & \vdots \\ \sum_{i=1}^m a_{qi}b_{i1} & \sum_{i=1}^m a_{qi}b_{i2} & \dots & \sum_{i=1}^m a_{qi}b_{in} \end{bmatrix},
\end{aligned}
$$

- Homogeneity: $\lambda\mathbf{A}$ represents the composition of two linear applications λf_A
- Transposition: $\mathbf{A}^T$ represents the adjoint operator of **A**

A Python summary of the previously described operation is presented below:

```python
import numpy as np
import pprint

A = np.zeros((2,3))
A[0,0] = 1
A[1,1] = 1

B = A.copy()
C=A+B
pprint.pprint(C)
```

Listing 2.4: *standard operations*

2.2.3 Advanced Common Operations in Engineering

Common linear algebra operations are the cornerstone of numerous engineering applications. In the field of electrical engineering, convolutions can be computed by multiplying a vector defining the input signal by a Toeplitz matrix characterizing the impulse response of the filter. Linear systems can be easily solved by multiplying (when possible) the solution vector by the inverse of the matrix defining the system's coefficients. The next chapter details one fundamental use of linear algebra: finding a good, compact way to represent complex data.

2.3 Numpy for matrix manipulations

2.3.1 Basic uses of numpy

numpy includes by its several useful submodules including functions for matrix creation, computations and handling. The most commonly known and used submodules are:

- **linalg**: a submodule of linear algebra routines to solve linear systems, invert matrices, compute eigenvalues and eigenvectors, perform matrix decompositions, and so forth.
- **random**: a submodule for generation of random variable samples based on probability density functions or probability mass functions.
- **fft**: a submodule for Fast Fourier Transform and related operations

Further documentation can be found on the numpy website, which works in a similar manner than Matlab's help online, and includes numerous examples. It is also worth mentioning the companion module **scipy**, which includes more advanced routines of linear algebra, optimization and advanced algorithms for scientific computing.

2.3.2 Plotting Results using Matplotlib

One of the best functionalities of Matlab is the easy way to plot results stored in arrays or matrices. Python offers similar tools, using the matplotlib package for plotting, and the seaborn package for statistical data representation (which will be detailed in a further chapter). Options in matplotlib are similar to Matlab, such that the learning curve from one language to the other is not steep.

The following example show how to display different types of curves using Matplotlib (note that there are many, many more)

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

def example_plots_matplotlib():
    t = np.linspace(0,4*np.pi,1000)
    f = np.sin(2*t) # one sine
    g = np.sin(t) * np.exp(-t/5) # another sine function, damped
    h = np.sin(t) + 0.1*np.random.randn(t.shape[0]) # a noisy sine

    fig = plt.figure(num=1,figsize=(20,10))
    plt.subplot(2,2,1)
    plt.plot(t,f)
    plt.xlabel('t')
    plt.ylabel('sin(2t)')
    plt.subplot(2,2,2)
    plt.plot(t,g,'r--',linewidth=3)
    plt.xlabel('t')
    plt.ylabel('Damped sine')
    plt.subplot(2,2,3)
    plt.plot(t,h,'.')
    plt.plot(t,np.sin(t),linewidth=2)
    plt.xlabel('t')
    plt.ylabel('Noisy data and ideal curve')
    plt.subplot(2,2,4)
    plt.hist(h,50,(-1.3,1.3))    plt.savefig('output.pdf',format='pdf')
plt.show()
```

Listing 2.5: *Code to plot some graphs and histograms*

Figures obtained after execution of the function are presented in figure 2.3.1.

2.3.3 Basic example: Filtering with a Toeplitz Matrix

Let us assume that we consider one finite-length signal $x[n] = [x_0\ x_1\ \ldots x_{N-1}]^T$ of size N, and one finite length impulse response of a digital filter $h[n] = [h_0\ h_1\ \ldots h_{M-1}]^T$ of size M. The output of the digital filter is given by the discrete convolution between $x[n]$ and $h[n]$:

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]x[n-k]$$

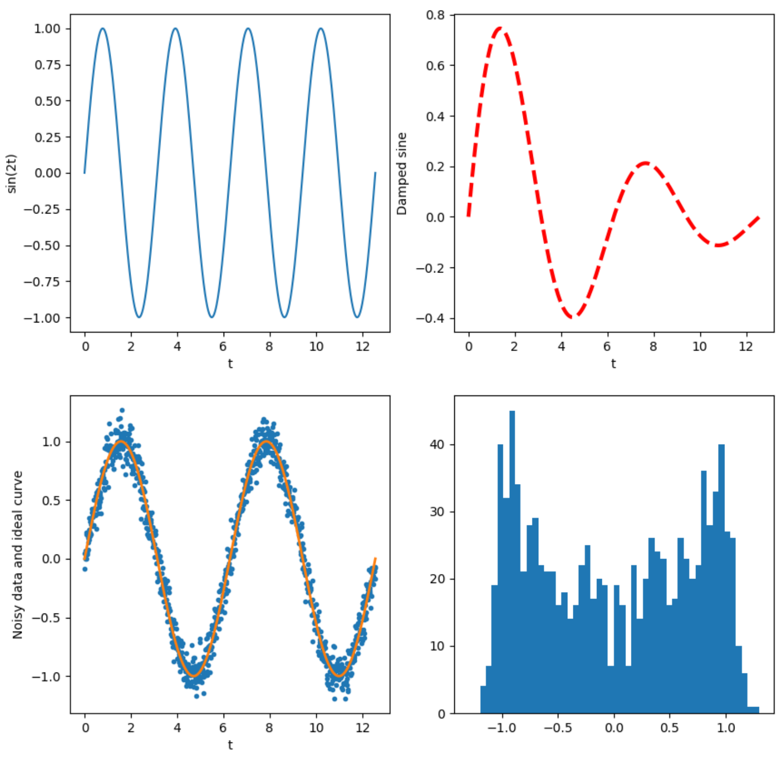


Figure 2.3.1: *Examples of figures obtained using matplotlib*

Since both h and x have finite length, the convolution can indeed be summarized by a matrix-vector multiplication:

$$y[n] = \begin{bmatrix} h_0 & 0 & 0 & \dots & 0 & 0 \\ h_1 & h_0 & 0 & \dots & 0 & 0 \\ h_2 & h_1 & h_0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{M-1} & h_{M-2} & h_{M-3} & \dots & h_1 & h_0 \\ 0 & h_{M-1} & h_{M-2} & h_{M-3} & \dots & h_1 \\ 0 & 0 & h_{M-1} & \dots & h_3 & h_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \dots & h_{M-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_{N-1} \end{bmatrix} \tag{2.3.1}$$

The following code is the implementation of Equation (2.3.1)

```python
import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt

h = np.arange(1, 6)
x = np.ones((5,1))

padding = np.zeros(h.shape[0] - 1, h.dtype)
first_col = np.r_[h, padding]
```

```
first_row = np.r_[h[0], padding]

H = linalg.toeplitz(first_col , first_row)

y = np.dot(H,x)
print y

fig = plt.figure(figsize=(20,10))
plt.subplot(1,3,1)
plt.stem(x)
plt.ylabel('x[n]')
plt.xlabel('n')
plt.subplot(1,3,2)
plt.stem(h)
plt.ylabel('h[n]')
plt.xlabel('n')
plt.subplot(1,3,3)
plt.stem(y)
plt.ylabel('y[n] = h * x [n]')
plt.xlabel('n')
plt.savefig('ch2_convolution_example.png')
plt.show()
```

Listing 2.6: *Code performing the convolution between x and h*

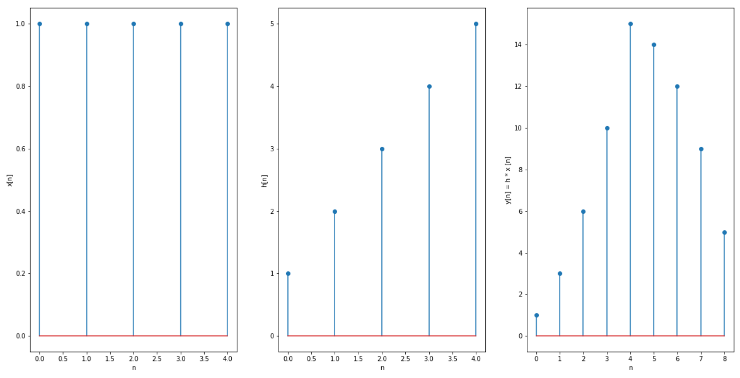


Figure 2.3.2: *Graphs obtained by running the convolution script*

2.4 Case study: an application of the Kalman filter

Kalman filtering was one of the most common linear filter used in the electrical engineering community. It was proved to be optimal when the investigated system is linear and the noise can be modeled as Gaussian.

2.4.1 State-space model representation of a dynamic system

When looking at a dynamic system, we can characterize it by different kinds of views: its inner state (meaning, how the system reacts when put in a known state and let to evolve freely), and its outer state (how it reacts when applying it a known input). All in all, a dynamic system's behavior is based on the evolution of its inner state and its outer state, jointly. A state-space model is a mathematical model used to describe the behavior of a dynamic system, based on an intern state and observations based on the current state and an input.

The state-space model investigated as part of the Kalman filter is described below:

$$\begin{aligned}\mathbf{x}_n &= \mathbf{F}_n\mathbf{x}_{n-1} + \mathbf{B}_n\mathbf{u}_n + \mathbf{w}_n \\ \mathbf{y}_n &= \mathbf{H}_n\mathbf{x}_n + \mathbf{v}_n\end{aligned}$$

when it is assumed that $\mathbf{F}_n, \mathbf{B}_n, \mathbf{u}_n, \mathbf{H}_n$, $\mathbf{w}_n \sim \mathcal{N}(0, \mathbf{Q}_n)$ and $\mathbf{v}_n \sim \mathcal{N}(0, \mathbf{R}_n)$.
The objective is to estimate $\mathbf{x}_n$ based on $\mathbf{y}_n$.

2.4.2 Kalman equations

We will define $\mathbf{x}_{n|n}$ as the estimate of $\mathbf{x}_n$, based on the knowledge of $\mathbf{y}_k, k = 1 \dots n$. The objective if to find the estimator which minimizes the mean squared error $E((\mathbf{x}_n - \mathbf{x}_{n|n})^2)$. The Kalman algorithm solves the computation of the estimate iteratively. One iteration is based on two steps: prediction and smoothing.
Prediction: The natural way to guess the state in the next time is to apply the formula to what was obtained in the previous stage:

$$\mathbf{x}_{n|n-1} = \mathbf{F}_n\mathbf{x}_{n-1|n-1} + \mathbf{B}_n\mathbf{u}_n$$

When doing so, we make an error whose covariance matrix is given by:

$$\begin{aligned}\mathbf{P}_{n|n-1} &= E((\mathbf{x}_n - \mathbf{x}_{n|n-1})(\mathbf{x}_n - \mathbf{x}_{n|n-1})^T) \\ &= E((\mathbf{F}_n\mathbf{x}_{n-1} + \mathbf{w}_n - \mathbf{F}_n\mathbf{x}_{n-1|n-1})(\mathbf{F}_n\mathbf{x}_{n-1} + \mathbf{w}_n - \mathbf{F}_n\mathbf{x}_{n-1|n-1})^T) \\ &= \mathbf{F}_n E(\mathbf{x}_{n-1} - \mathbf{x}_{n-1|n-1})(\mathbf{x}_{n-1} - \mathbf{x}_{n-1|n-1})^T)\mathbf{F}_n^T + \mathbf{Q}_n \\ &= \mathbf{F}_n\mathbf{P}_{n-1|n-1}\mathbf{F}_n^T + \mathbf{Q}_n\end{aligned}$$

Smoothing: we define the innovation as the difference between an estimated observation and its actual value:

$$\mathbf{I}_n = \mathbf{y}_n - \mathbf{H}_n\mathbf{x}_{n|n-1}$$

The covariance matgrix of the innovation is:

$$\mathbf{S}_n = \mathbf{H}_n\mathbf{P}_{n|n-1}\mathbf{H}_n^T + \mathbf{R}_n$$

The state estimate can be defined as a function of the innovation, as follows:

$$\mathbf{x}_{n|n} = \mathbf{x}_{n|n-1} + \mathbf{K}_n\mathbf{I}_n$$

The objective is now to find the value of $\mathbf{K}_n$ which minimizes the error

$$E[\|\mathbf{x}_{n|n} - \mathbf{x}\|_2^2] = E[(\mathbf{x}_{n|n} - \mathbf{x}_n)^T(\mathbf{x}_{n|n} - \mathbf{x}_n)] = E[\mathrm{Tr}((\mathbf{x}_{n|n} - \mathbf{x}_n)(\mathbf{x}_{n|n} - \mathbf{x}_n)^T)] \tag{2.4.1}$$

We now need to derive (2.4.1), and check when the derivative equals zero (as a function of $(\mathbf{x}_{n|n} - \mathbf{x}_n)(\mathbf{x}_{n|n} - \mathbf{x}_n)^T$):

$$\begin{aligned}(\mathbf{x}_{n|n} - \mathbf{x}_n)(\mathbf{x}_{n|n} - \mathbf{x}_n)^T &= (\mathbf{x}_{n|n-1} - \mathbf{K}_n(\mathbf{y}_n - \mathbf{H}_n\mathbf{x}_{n|n-1}) - \mathbf{x}_n)(\mathbf{x}_{n|n-1} - \mathbf{K}_n(\mathbf{y}_n - \mathbf{H}_n\mathbf{x}_{n|n-1}) - \mathbf{x}_n)^T \\ &= (\mathbf{I} - \mathbf{K}_n\mathbf{H}_n)(\mathbf{x}_n - \mathbf{x}_{n|n-1}) + \mathbf{K}_n\mathbf{v}_n)(\mathbf{I} - \mathbf{K}_n\mathbf{H}_n)(\mathbf{x}_n - \mathbf{x}_{n|n-1}) + \mathbf{K}_n\mathbf{v}_n)^T \\ &= (\mathbf{I} - \mathbf{K}_n\mathbf{H}_n)\mathbf{P}_{n|n-1}(\mathbf{I} - \mathbf{K}_n\mathbf{H}_n)^T + \mathbf{K}_n\mathbf{R}_n\mathbf{K}_n^T \\ &= \mathbf{K}_n\mathbf{S}_n\mathbf{K}_n^T + \mathbf{P}_{n|n-1} - \mathbf{K}_n\mathbf{H}_n\mathbf{P}_{n|n-1} - \mathbf{P}_{n|n-1}\mathbf{H}_n^T\mathbf{K}_n^T\end{aligned}$$

and we look for $\mathbf{K}_n$ such as

$$\frac{\partial \mathrm{Tr}((\mathbf{x}_{n|n}-\mathbf{x}_n)(\mathbf{x}_{n|n}-\mathbf{x}_n)^T)}{\partial \mathbf{K}_n} = \mathrm{Tr}\left(\frac{\partial (\mathbf{x}_{n|n}-\mathbf{x}_n)(\mathbf{x}_{n|n}-\mathbf{x}_n)^T}{\partial \mathbf{K}_n}\right) = 0$$

We define the functional $f:\ \mathbf{K}_n \mapsto \mathbf{K}_n\mathbf{S}_n\mathbf{K}_n^T + \mathbf{P}_{n|n-1} - \mathbf{K}_n\mathbf{H}_n\mathbf{P}_{n|n-1} - \mathbf{P}_{n|n-1}\mathbf{H}_n^T\mathbf{K}_n^T$ and will compute its derivative based on its variations:

$$\begin{aligned} f(\mathbf{K}_n+\delta\mathbf{K}_n)-f(\mathbf{K}_n) = {} & \mathbf{K}_n\mathbf{S}_n\delta\mathbf{K}_n^T + \delta\mathbf{K}_n\mathbf{S}_n\mathbf{K}_n^T + \delta\mathbf{K}_n\mathbf{S}_n\delta\mathbf{K}_n^T \\ & - \delta\mathbf{K}_n\mathbf{H}_n\mathbf{P}_{n|n-1} - \mathbf{P}_{n|n-1}\mathbf{H}_n^T\delta\mathbf{K}_n^T \end{aligned}$$

Thus, $f'(\mathbf{K}_n) = \mathbf{K}_n\mathbf{S}_n + \mathbf{S}_n\mathbf{K}_n^T - \mathbf{H}_n\mathbf{P}_{n|n-1} - \mathbf{P}_{n|n-1}\mathbf{H}_n^T$, and due to the identities $\mathrm{Tr}(A)=\mathrm{Tr}(A^T)$, $\mathbf{P}_{n|n-1} = \mathbf{P}_{n|n-1}^T$, $\mathbf{S}_n = \mathbf{S}_n^T$, we obtain

$$\mathrm{Tr}(f'(\mathbf{K}_n)) = 2\mathrm{Tr}(\mathbf{K}_n\mathbf{S}_n) - 2\mathrm{Tr}(\mathbf{H}_n\mathbf{P}_{n|n-1}) \tag{2.4.2}$$

Equation (2.4.2) equals zero when

$$\mathbf{K}_n = \mathbf{P}_{n|n-1}\mathbf{H}_n^T\mathbf{S}_n^{-1} \tag{2.4.3}$$

Once we found (2.4.3), it is easy to compute the covariance matrix of the smoothing

$$\mathbf{P}_{n|n} = (\mathbf{I}-\mathbf{K}_n\mathbf{H}_n)\mathbf{P}_{n|n-1}$$

and the error made on the observation

$$\mathbf{y}_{n|n} = \mathbf{y}_n - \mathbf{H}_n\mathbf{x}_{n|n}$$

2.4.3 Implementation

As an example of the Kalman filter, we shall use it to track a moving object automatically in a movie. In this framework, we will define the state as $\mathbf{x}_n = [x_n; y_n; \dot{x}_n; \dot{y}_n]$, the position of the center of mass of the object in the image n of the movie. If assuming that the acceleration is constant in the whole movie, we can define

$$\begin{bmatrix} x_n \\ y_n \\ \dot{x}_n \\ \dot{y}_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \\ \dot{x}_{n-1} \\ \dot{y}_{n-1} \end{bmatrix} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \frac{\Delta t^2}{2} \\ \Delta t \\ \Delta t \end{bmatrix} a + \mathbf{w}_n,\ \mathbf{Q}_n \propto \begin{bmatrix} \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} & 0 \\ 0 & \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & 0 & \Delta t^2 & 0 \\ 0 & \frac{\Delta t^3}{2} & 0 & \Delta t^2 \end{bmatrix}$$

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_n \\ y_n \\ \dot{x}_n \\ \dot{y}_n \end{bmatrix} + \mathbf{v}_n,\ \mathbf{R}_n = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}$$

Eventually, it turns out that both step of prediction and update are iterated matrix computations which can be efficiently implemented using Numpy capabilities, as follows:

```python
def run_kalman_filtering(folder_name ,x,y, start_frame ,end_frame):

    # options to play with
    dt = 1  # sampling rate
    S_frame = 10 # starting frame
    u = .05 # define acceleration magnitude
    Q = np.array([[x[0]], [y[0]], [0],[0]]) # initial state
    Q_estimate = Q.copy() # estimate of initial location estimation of where the hexbug
    HexAccel_noise_mag = 100 # process noise: (stdv of acceleration: me.s^-2)
    tkn_x = 1 # measurement noise in the horizontal direction (x axis).
    tkn_y = 1 # measurement noise in the horizontal direction (y axis).
    radius_circle = 20 # radius to play with

    Ez = np.array([[tkn_x,0],[0,tkn_y]])

    Ex = np.array([[dt**4/4, 0, dt**3/2, 0],[0, dt**4/4, 0, dt**3/2],[dt**3/2, 0, dt**2, 0],[0, dt**3/2, 0, dt**2]])

    P = Ex.copy()

    # state-space model matrices
    A = np.array([[1,0,dt,0],[0,1,0,dt],[0,0,1,0],[0,0,0,1]]) # state matrix
    B = np.array([[0.5*dt**2],[0.5*dt**2],[dt],[dt]])
    C = np.array([[1,0,0,0],[0,1,0,0]])

    if os.path.exists('./result') is False:
        os.mkdir('result')

    for n in range(start_frame,end_frame):
        Q_loc_obs = np.array([[x[n]],[y[n]]])
        # predict the next state
        Q_estimate = A.dot(Q_estimate) + HexAccel_noise_mag*np.random.randn()*u*B
        # update the covariance matrix
        P = np.dot(A,np.dot(P,A.T)) + Ex
        # Kalman Gain
        M = np.dot(C,np.dot(P,C.T)) + Ez
        M_inv = np.linalg.inv(M)
        K = np.dot(P,np.dot(C.T,M_inv))

        # update
        Q_estimate = Q_estimate + np.dot(K,Q_loc_obs - np.dot(C,Q_estimate) )
        # update cov
        P =  np.dot(np.eye(4) - np.dot(K,C),P)

        I = cv2.imread(os.path.join(folder_name,'frame'+str(n)+'.jpg'))
        cv2.circle(I,(int(x[n]),int(y[n])),100,(0,255,0),10)
        cv2.circle(I,(int(Q_estimate[0]),int(Q_estimate[1])),100,(0,0,255),10)
        savename = 'result/frame_filtered'+str(n)+'.jpg'
        cv2.imwrite(savename,I)
```

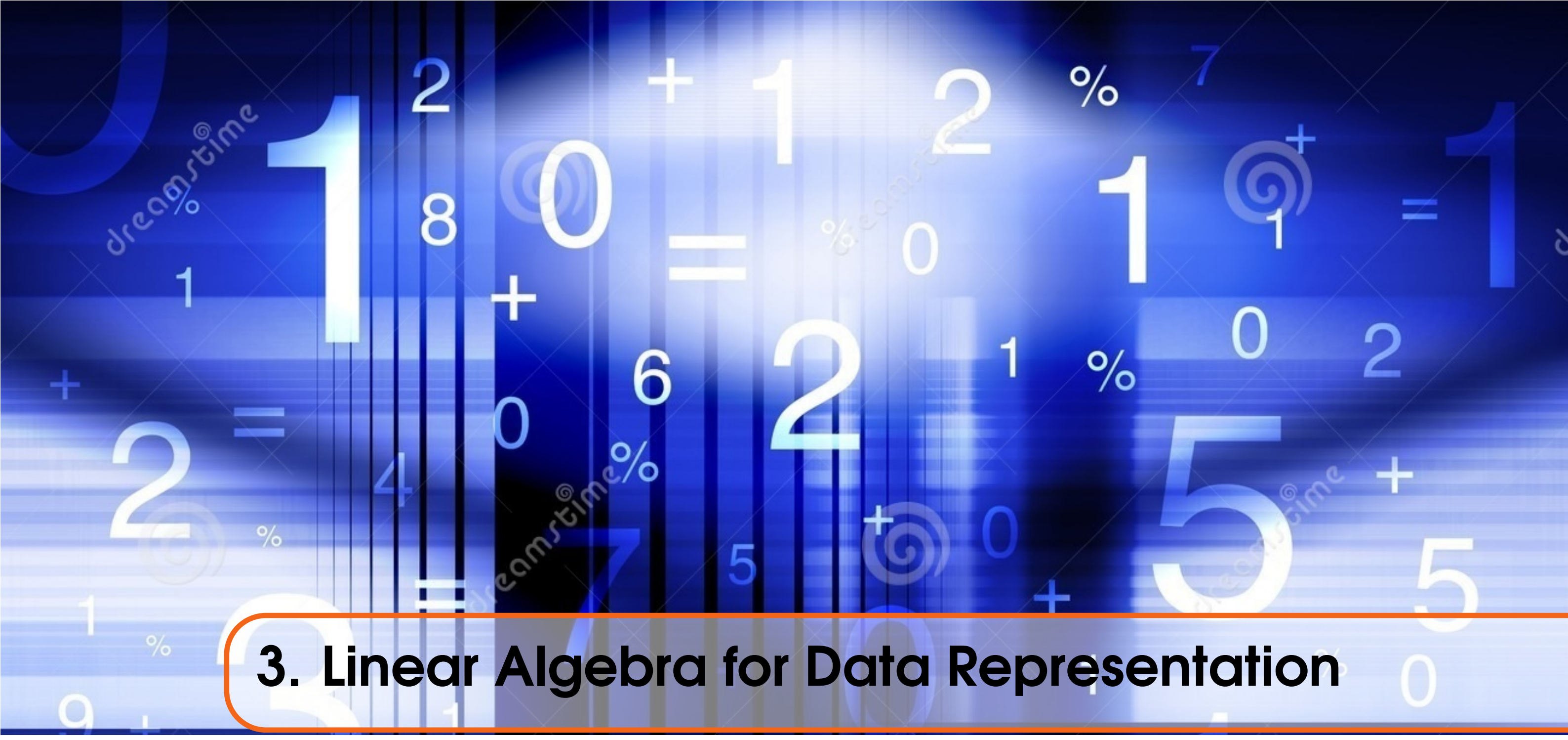
Listing 2.7: *Implementation of Kalman filter iterations*

2.5 Further Readings

link to
tlib help

references to
lgebra books

Further references on Matplotlib can be accessed online HERE .
Further references on linear algebra HERE



3. Linear Algebra for Data Representation

3.1 Ain't nothing like the real thing

Considering a vector space with finite dimensions, it is straight forward to understand that an infinite number of possible basis exist. The question thus arising is: is there a vector basis "better" than the other?
The answer, of course, is driven by the data at hand and the problem we wish to solve.

3.2 Matrix Decompositions

3.2.1 Why Decompositions are Useful

Matrix decompositions aim to express a matrix as a product of matrix. In a sense we aim to decompose an application as a sequence of “simpler” linear applications. For example, a geometric similarity can be decomposed as a sequence of a rotation, a scaling and possibly a translation, so that both objects are similar. These decompositions help in either denoising data, either characterizing datasets in a more useful basis. Most of the presented decompositions can be computed in practice by using the functions existing in the `numpy.linalg` and `scipy.linalg` package.

3.2.2 LU decomposition

LU decomposition is strongly related to the Gauss elimination procedure when solving a linear systems.

Definition 3.2.1 — LU decomposition. Given a square matrix $\mathbf{A}$, it can be decomposed into the product of one lower triangular matrix $\mathbf{L}$ with one upper triangular matrix $\mathbf{U}$:

$$\mathbf{A} = \mathbf{L}\mathbf{U} \tag{3.2.1}$$

Equation (3.2.1) summarizes the process of Gaussian elimination in matrix form, possibly with the addition of a permutation matrix. The following example illustrates the LU decomposition of a given matrix:

```python
import numpy as np
import scipy.linalg
import pprint

A = np.array([[1,2,3],[4,5,6],[7,8,9]])
L,U=scipy.linalg.lu(A, permute_l = True)
pprint.pprint(A)
pprint.pprint(L)
pprint.pprint(U)
pprint.pprint(np.dot(L,U))
```

Listing 3.1: *LU decomposition*

The LU decomposition is mainly used for solving linear systems, since in the case of triangular systems the number of operations for matrix inversion is approximately halved. However, since it involves numerical inversions in the process, it is not a numerically stable decomposition. The next decomposition, in that matter, is way more attractive.

3.2.3 QR decomposition

Definition 3.2.2 — QR decomposition. Given a square matrix $\mathbf{A}$, it is possible to decompose it as a product

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q}$ is an orthogonal matrix (meaning that $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$) and $\mathbf{R}$ is a upper triangular matrix.

This decomposition is inherently connected to Gram-Schmidt orthogonalization of a basis: the matrix $\mathbf{R}$ includes all the remainders of the projected vectors of the original basis, while the columns of $\mathbf{Q}$ denote the new orthonormal basis obtained.
The following script performs the QR decomposition of given matrix.

```
import numpy as np
import pprint

A = np.array([[1,2,3],[4,5,6],[7,8,9]])
Q,R=np.linalg.qr(A)
pprint.pprint(A)
pprint.pprint(R)
pprint.pprint(Q)
pprint.pprint(np.dot(Q,R))
```

Listing 3.2: *QR decomposition*

When solving numerical systems, the QR decomposition is more appealing, since one of the inversion appearing in the LU decomposition is replaced by a direct matrix multiplication (due to the orthonormality property, inverting $\mathbf{Q}$ is equivalent to multiplying by its transpose). However, a direct implementation is also numerically unstable, since the normalization of the orthogonal vectors in $\mathbf{Q}$ can yield large or very small values. In practice, QR decomposition is most often achieved using more complex algorithms, (Householder reflections in the Numpy module).

3.2.4 Cholesky Decomposition

Cholesky decomposition operates on symmetric positive-definite matrices; that is, matrices $\mathbf{A}$ such that $\mathbf{A} = \mathbf{A}^T$ and $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all vector $\mathbf{x}$. Such matrices can be associated to quadratic forms, are non-singular and have strictly positive eigenvalues.

Definition 3.2.3 — Cholesky decomposition. For such a matrix, there exists an upper triangular matrix $\mathbf{R}$ such that

$$\mathbf{A} = \mathbf{R}^T\mathbf{R} \tag{3.2.2}$$

Equation (3.2.2) is the equivalent of the squared root for real numbers. This decomposition is particularly important, since in many practical applications (gradient for regression, covariance matrix decomposition), we investigate matrices of the form $\mathbf{A}^T\mathbf{A}$, which are symmetric and semidefinite.

■ **Example 3.1** As an example, we will compute the Cholesky decomposition of the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 29 & -14 \\ 3 & -14 & 26 \end{bmatrix}.$$

This computation will also provide an insight on the algorithm used to compute this decomposition. The objective matrix is

$$\mathbf{R} = \begin{bmatrix} r_1 & r_2 & r_3 \\ 0 & r_4 & r_5 \\ 0 & 0 & r_6 \end{bmatrix},$$

and the following equality must hold:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 29 & -14 \\ 3 & -14 & 26 \end{bmatrix} = \begin{bmatrix} r_1 & 0 & 0 \\ r_2 & r_4 & 0 \\ r_3 & r_5 & r_6 \end{bmatrix} \cdot \begin{bmatrix} r_1 & r_2 & r_3 \\ 0 & r_4 & r_5 \\ 0 & 0 & r_6 \end{bmatrix}$$

Multiplying the first row by the columns gives

$$r_1^2 = 1, r_1r_2 = 2, r_1r_3 = 3,$$

and due to the known positivity of the diagonal

$$r_1 = 1, r_2 = 2, r_3 = 3.$$

Multipling the second row by the second and third lines gives

$$r_2^2 + r_4^2 = 29,\ r_2r_3 + r_4r_5 = -14,$$

thus

$$r_4 = 5, r_5 = -4,$$

and eventually multiplying the third row by the third column gives

$$r_3^2 + r_5^2 + r_6^2 = 26,$$

thus

$$r_6 = 1,\ \mathbf{R} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & -4 \\ 0 & 0 & 1 \end{bmatrix}$$

■

Note that the same algorithm can be used to check whether a matrix is symmetric definite. Cholesky decomposition is performed in Python using `numpy.linalg.cholesky`.

3.3 Eigenvectors simplify things

Given a linear application $\mathbf{A}$, we say that $\mathbf{x}$ is an eigenvector if there exists λ such that

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{3.3.1}$$

In (3.3.1), the parameter λ is called an eigenvalue.

3.3.1 The Idea Within

As aforementioned, a linear applications transforms a vector basis in another family of possibly dependent vectors. However, if the associated matrix is diagonal, the application can be easily understood as scaling / flipping vectors of the existing basis. Knowing which vectors remain invariant (up to a scaling) after applying a linear transformation is therefore important, since it allows to simplify the understanding of the linear application. When the matrix is related to data (e.g. a covariance matrix, which represents the linear dependence between couples of random variables), computing the eigenvectors provides the main directions around which the data are organized[1]. Finding eigenvectors/eigenvalues is also critical when solving differential systems, since it allows to deal with each differential equation independently.

■ **Example 3.2** Given the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 29 & -14 \\ 3 & -14 & 26 \end{bmatrix},$$

we will find its eigenvalues and eigenvectors. This can be easily computed with Python using the following code:

```python
import numpy as np
import pprint

A = np.array([[1,2,3],[2,29,-14],[3,-14,26]])

values, vectors=np.linalg.eig(A)

pprint.pprint(values)
pprint.pprint(vectors)
```

Listing 3.3: *Eigenvalues and eigenvectors with Numpy*

■

3.4 The SVD

Singular Value Decomposition (SVD) is one of the most important decompositions in numerous fields, e.g. statistical analysis and signal processing. It can be seen as an extension of diagonalization (finding all the eigenvalues and eigenvectors) of a square matrix, for matrices of general sizes and ranks. It is particularly useful for data dimensionality reduction.

3.4.1 The Idea and the Result

Definition 3.4.1 — SVD decomposition. Assume that we have a matrix $\mathbf{A}$ of size $n \times d$, where n and d are not necessarily equal. This matrix can represent d vectors of data organized in a

[1]Incidentally, eigenvectors are an inherent part of the PageRank algorithm of Google...

matrix way, or any other linear application. Denote by k its rank

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T, \tag{3.4.1}$$

where in (3.4.1) $\mathbf{U}$ denotes an orthonormal $n \times n$ matrix, $\mathbf{V}$ is a $d \times d$ orthonormal matrix and $\mathbf{D}$ is a $n \times d$ whose only diagonal terms can be different from 0.

The non-zero terms of $\mathbf{D}$ are called the *singular values* of $\mathbf{A}$, and are the generalization of the concept of eigenvalues for squared matrices. The intuition within this decomposition is easy to grasp: any linear application can be understood as a rotation, followed by a scaling along each new dimension, followed eventually by a rotation on the image subspace. Note that the decomposition is not unique, but we can make it unique by sorting the singular values from the largest to the smallest. SVD is often used in dimensionality reduction of problems, since the singular values of $\mathbf{A}$ are the squared roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$. The largest a singular value is, the more variance there is along this direction, therefore the more relevant is this eigenvector for data representation.

3.4.2 Case Study: PCA for faces (Eigenfaces)

Principal Component Analysis is a way to reduce the dimensionality of a dataset. Given a matrix dataset $\mathbf{A}$, it aims to represent the k most relevant statistical information, in the variance sense. More precisely, we represent the data by means of the eigenvectors of the covariance matrix $\mathbf{A}^T\mathbf{A}$. The steps to perform a PCA are:

1. Perform an SVD on $\mathbf{A}$
2. Use the columns of $\mathbf{V}$ associated with the k-largest singular values as the eigenvectors of $\mathbf{A}^T\mathbf{A}$.

On a given vector $\mathbf{x}$, we can thus apply the PCA transformation $\mathbf{D}\mathbf{V}^T\mathbf{x}$ to get the most relevant information in terms of maximal variances. When applying this decomposition to pictures of faces (reshaped in a vector form), it turns out that only a few eigenvectors (called eigenfaces) are enough to represent general faces with a fair accuracy. The following Python codes takes a dataset from well-known faces, performs a PCA to reduce the dimensionality of the problem to 100, and train a coarse classifier for face recognition. Besides the much faster execution times, we also get better recognition results, since we force the classifier to focus only on the most relevant features from the very beginning.

```python
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_lfw_people
from sklearn.metrics import classification_report
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

# Load dataset of presidents faces - it contains 1140 pictures of sizes 62x47
lfw_dataset = fetch_lfw_people(min_faces_per_person=100)

# saves the height and width of the images, as well as the data and labels
_, h, w = lfw_dataset.images.shape
# data is already preprocessed as vectors of size 62x47
X = lfw_dataset.data
y = lfw_dataset.target
target_names = lfw_dataset.target_names

# split into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

```

```
# Compute a PCA
n_components = 100
pca = PCA(n_components=n_components, whiten=True).fit(X_train)

# apply PCA transformation
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

# train a neural network , with PCA and without
print("Fitting the classifier to the training set")
clf = MLPClassifier(hidden_layer_sizes=(1024,512,100,), batch_size=256, verbose
    =True, early_stopping=True).fit(X_train_pca, y_train)
clf2 = MLPClassifier(hidden_layer_sizes=(4000,2000,100,), batch_size=256,
    verbose=True, early_stopping=True).fit(X_train, y_train)

y_pred = clf.predict(X_test_pca)
print(classification_report(y_test, y_pred, target_names=target_names))
y_pred2 = clf2.predict(X_test)
print(classification_report(y_test, y_pred2, target_names=target_names))
```

Listing 3.4: *Dimensionality reduction and eigenfaces*

3.5 Further Readings

III

Optimization

4. Basics of Unconstrained Optimization

4.1 Ain't no Mountain High Enough

Optimization deals with the selection of the best value with respect to a known criterion from a known set. From an engineering point of view, it is one the most important / commonly used field, since many engineering solutions can be seen as the solution of an optimization problem (maximizing the power, minimizing the cost, etc...). As such, it is important to understand notations which appear frequently in numerous fields, and to have a basic knowledge of optimization algorithms which are frequently used.

The most primitive approach we might think of is to investigate all the possible values that the function takes, and retain either its minimal / maximal value. This approach (also known as *brute force* approach), clearly, is not retained in practice: besides the fact that the function can take an infinite number of values, the number of verifications we would do here will grow exponentially with the dimension of the function. The intuition in the matter is that an efficient extremum search depends on the local properties of the function at hand. For example, very intuitively speaking, if we aim to find a minimum, and know that at a given point the function is non-increasing, it might be a good idea to investigate in priority this direction.

However, this kind of local idea can raise other issues. If we consider the function displayed in Figure 4.1.1, we can understand that this task can be very challenging: though this function on $[0,1]$ has one single global minimum, if we limit ourselves to $[0,0.2]$, it seems that the minimum is around -1. Obviously this is not the case, but this illustrates one of the main difficulties encountered in the field: in most algorithms, *initialization* (that is, where do we start our minimization finding) is critical. This also shows that the solution strongly depends on the domain we investigate. When we force the solution to belong to a given domain (or to satisfy a certain condition), we shall say that we address a *constrained optimization problem*. Otherwise, the optimization problem is said to be *unconstrained*.

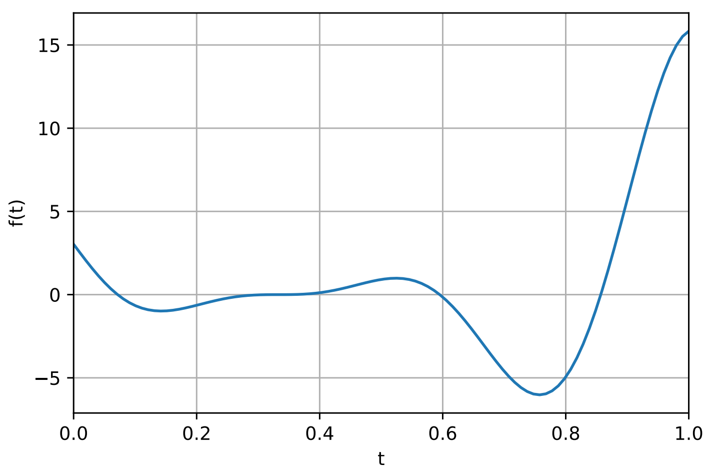


Figure 4.1.1: *The function $f(t) = (6t-2)^2 sin(12t-4)$ on $[0,1]$ - depending on where we start our search for a minimum, results may strongly differ.*

4.2 Mathematical Notations

The following notations are commonly used in optimization theory, and are important to know.

Definition 4.2.1 Given a function f defined on a domain D, the minimal value m that f attains on D is denoted by $\min_{x \in D} f(x)$, that is for all $x \in D$, $f(x) > m$; the element (not necessarily unique) $x_0 \in D$ such that $f(x_0) = \min_{x \in D} f(x)$ is denoted by $\arg\min_{x \in D} f(x)$.
Similarly, the maximal value of f and the element for which the maximum is attained are denoted respectively by $\max_{x \in D} f(x)$ and $\arg\max_{x \in D} f(x)$.

■ **Example 4.1** Consider the function $f : x \mapsto x^2 + 2$. It is straightforward to check that $\min_{\mathbb{R}} f = 2$, $\arg\min_{\mathbb{R}} f = 0$. ■

4.3 Convex Functions and Convex Optimization

One critical subproblem in the field of optimization is the problem of convex optimization, that is the particular case when the function we wish to optimize is convex. The following definition introduces the concept of convex functions.

Definition 4.3.1 — Convex function. A function f is convex when for all $0 \leq \lambda \leq 1$ and for all x_1, x_2, we have:

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2).$$

In case we have $f(\lambda x_1 + (1-\lambda)x_2) = \lambda f(x_1) + (1-\lambda) f(x_2)$ if and only if $\lambda = 0$ and $\lambda = 1$, we shall say that the function is strictly convex.

Graphically, given two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$, the graph reprensenting f in the domain $[x_1, x_2]$ will always be below the line generated by the two points, as seen in Figure 4.3.1.
For example, the functions e^{-x}, x^{2n} are convex. We will further on say that $f(x)$ is concave if and only if $-f(x)$ convex. Observe that most functions are neither convex nor concave, and that the linear functions are the only ones being both convex and concave.
The following result provides a sufficient condition for convextiy, which often holds on practice.

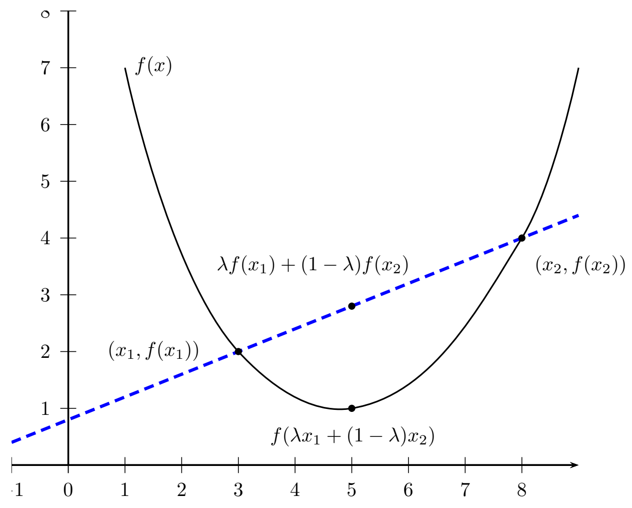


Figure 4.3.1: *Graphical example of a convex function*

Proposition 4.3.1 If $f''(x) \geq 0$, then $f(x)$ is convex.

Proof. Due to Taylor-Young equality:

$$f(b) = f(a) + f'(a)(b-a) + f''(x)\frac{(b-a)^2}{2}, \quad a \leq x \leq b.$$

On the one hand, for $a = \lambda x_1 + (1-\lambda)x_2, b = x_1$, we get that

$$f(x_1) \geq f(\lambda x_1 + (1-\lambda)x_2) + f'(\lambda x_1 + (1-\lambda)x_2)(1-\lambda)(x_1 - x_2),$$

and on the other hand for $a = \lambda x_1 + (1-\lambda)x_2,\ b = x_2$ we have

$$f(x_2) \geq f(\lambda x_1 + (1-\lambda)x_2) + f'(\lambda x_1 + (1-\lambda)x_2)\lambda(x_2 - x_1).$$

Thus

$$\lambda f(x_1) + (1-\lambda) f(x_2) \geq f(\lambda x_1 + (1-\lambda)x_2)$$

■

Furthermore, the following property allows to build easily convex functions based on a set of known convex functions.

Proposition 4.3.2 Every linear combination of convex functions is also a convex function.

4.3.1 Why Convexity is Cool? Which Other Properties?

Many costs functions appearing in fields such as regression, statistics, estimation theory and so forth are naturally convex. One of the most important properties that convex function hold is that a local minimum is also global, provided the convex function is regular enough (differentiable). It means that in practice, when a numerical optimization technique converges, we can truly assess from the returned value the minimum in the convex case. Furthermore, if the function is strictly convex, the minimizer is also unique. All these results illustrate the importance of convex optimization.
As mentioned before, convexity is one important property for a function to have for any optimization problem. This is not the only one though, since any iterative method to find a minimum and its associated minimizer can also benefit for additional local knowledge. For example, if the function we wish to minimize is differentiable, we also have access to the function's local behavior. In that

case, it would be legitimate to look for the minimizer based on the steepest decrease, based on the derivatives' values. More generally, the more derivatives we know, the more we can use this knowledge to accelerate the convergence of our optimizations algorithms.
In the examples we give further, we shall assume that the function we wish to minimize is differentiable.

4.3.2 Main Ideas for Programming an Optimization Procedure - a Fable

Let's consider the toy example of four people who are standing in the middle of a mountain landscape, and wish to attain a valley at the lowest altitude.

- The first man goes along all the possible paths, and tries them all in an exhaustive manner. He will eventually attain the lowest altitude valley, but most chances are he will die of old age before he attains it. This approach illustrates the brute-force approach detailed before; it is interesting only for very simple functions and very low dimensions, but usually fails to provide satisfactory results in a reasonable computation time.
- The second guy looks where the steepest descent can be obtained by judging from where he stands, and goes several meters in that direction. He then stops, and investigates from his new position in which direction is the steepest descent, and then replicates his behavior. He will eventually reach a valley, but has no guarantee that he attained the lowest one. This idea illustrates gradient-based methods, which are commonly used in the field: they are generic, can be accelerated in a number of ways, and provide sub-optimal solutions easily. However, they can be slow on pathological examples or high dimensional functions, and only local optimizations can be attained.
- The third man measures the altitude of three points in the mountain that he can see, and goes along the following idea: the highest altitude point is retained, and we try to find a better candidate on the line built with this point and the center of mass of the two other points. If there is an alternative outside the defined triangle, we go along that path and extend our new triangle that way. Otherwise, we contract it. If we arrive at some valley, we restart the procedure. Eventually, this man will arrive to the lowest valley, at a time that is way more reasonable that the brute force approach. It symbolizes the function-only based exploratory approaches based on meta-heuristics, where no differentiation is required (either because we do not know them, or because we cannot compute them analytically). They are general methods, which also converge to local minimum (unless we do some warm start-over search), and are a good alternative when either the function is complicated, or not completely known. Note, however, that convergence in that case is more complicated to diagnose.
- The fourth man starts like the second one, but introduces a little randomness: he does not necessarily follow the steepest descent, but modifies it slightly and randomly. Furthermore, when finding a direction which makes him go up a little, he does not necessarily discard it, but rather tosses a coin: if he gets a "heads", he follows it, otherwise he stays where he stands. Eventually he will also get to the lowest valley, though there is no way to predict when. This man symbolizes the stochastic optimization approaches. Such methods are powerful in the sense that they allow to go out of local minima if any. The price to pay is a much slower rate of convergence.

4.4 Some Techniques of Unconstrained Optimization

For all the presented algorithms, we will present one example in the 1D case, and another one in the 2D case, for the sake of visualization. The 1D-function used is the Forrester function $f(x) = (6x-2)^2 \sin(12x-4)$ for x in $[0,1]$, presented in Figure 4.1.1. The 2D example used is the Rosenbrock function $f(x,y) = 100(y-x^2)^2 + (1-x)^2$ (banana-shaped function), whose minimum value is 0 and is attained at $(1,1)$; this function is presented in Figure 4.4.1.

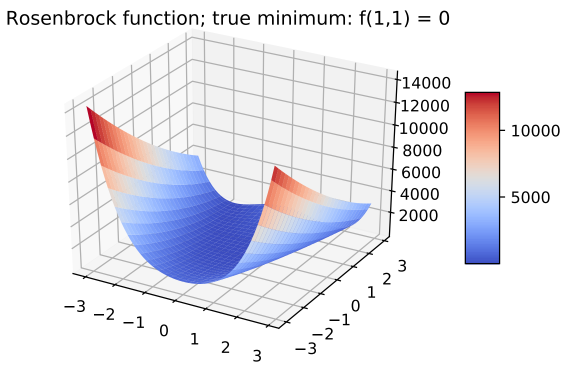


Figure 4.4.1: *Rosenbrock function. Minimizer is* $f(1,1)=0$*.*

4.4.1 Using only the Functions

When no information on the gradient is available, two strategies may be used: either compute the gradient information numerically (with the involved possible numerical instability), or try to optimize with using this information.
The Nelder-Mead algorithm can be seen as an extension of the dichotomy procedure, where instead of considering a line we use the points of a simplex to deal with multiple dimensions. This method

4.4.2 Using the Functions and their Gradients: The Gradient Method

The gradient descent is an iterative method whose objective is to provide a local minimum and its minimizer. Conceptually, we want to implement

$$x^{(k+1)} = x^{(k)} + t_k \Delta x^{(k)},$$

where $x^{(k)}$ denotes the value of the minimizer at iteration k, $\Delta x^{(k)}$ is the search direction of the minimum we wish to investigate (it is not necessarily a unit vector), and $t_k > 0$ is a weight (how faithful we are in that given direction).
For the sake of the argument, let us assume that the function f we wish to minimize is convex and differentiable. Then, we know due to a convexity argument that if $\nabla f(x^{(k)})^T(y - x^{(k)}) < 0$, then $f(y) > f(x^{(k)})$. Consequently, the search direction must satisfy $\nabla f(x^{(k)})^T \Delta(x^{(k)}) < 0$ to guarantee that we decrease the value of f at each iteration. Geometrically, it means that the search direction must make an acute angle with the opposite of the gradient.
Therefore, the gradient method for finding the minimum of a function can be summarized as follows:

1. Take a starting point $x^{(0)}$ (This point is often critical to guarantee the convergence - it is recommended to perform several runs)
2. Compute $\Delta x = -\nabla f(x^{(k)})$
3. Update $x^{(k+1)} = x^{(k)} + t_k \Delta x$ until convergence.

Convergence is usually evaluated based on $\dfrac{|x^{(k+1)} - x^{(k)}|}{x^{(k)}} < \varepsilon$, where ε is a user-defined threshold.
This algorithm is conceptually very simple, but convergence is usually very slow.
The following code presents the gradient descent for the Rosenbrock function:

```
def rosenbrock(x):
    return 100*(x[1]-x[0]*x[0])**2 + (1-x[0])**2

def rosenbrock_gradient(x):
    grad = np.zeros((2,))
    grad[0] = 400*x[0]*x[0]*x[0]-400*x[0]*x[1]+2*x[0]-2
    grad[1] = 200*x[1]*(x[1]-x[0]*x[0])
    return grad

def gradient_descent(x0,step,tol=1e-3,max_iter=10000,verbose = True):
    error = np.Inf
    stopping_criterion = False
    x = x0
    x_previous_iteration = x0
    current_iteration = 0
    values_x = []
    while stopping_criterion is False:
        current_iteration += 1
        gradient_value = rosenbrock_gradient(x)
        x = x - step*gradient_value
        error = np.linalg.norm(x-x_previous_iteration) / np.linalg.norm(
    x_previous_iteration)
        if error < tol or current_iteration==max_iter:
            stopping_criterion = True
        x_previous_iteration = x
        values_x.append(x)
        if verbose is True:
            print('Iteration {}, x = {}, error = {}'.format(current_iteration,x
    ,error))

    return values_x

plt.figure()
X = np.arange(-3, 3, 0.1)
Y = np.arange(-6, 3, 0.1)
X, Y = np.meshgrid(X, Y)
Z = 100*(Y-X*X)**2 + (1-X)**2

for x in np.arange(-3,4):
    for y in np.arange(-2,2):
        res = gradient_descent([x,y],0.00001,max_iter=100,verbose=False)
        x_coord = [a[0] for a in res]
        y_coord = [a[1] for a in res]
        plt.plot(x_coord,y_coord,'k')

plt.contour(X,Y,np.log(1+Z),20)
plt.show()
```

Listing 4.1: *Gradient descent for the Rosenbrock function*

The code shows the iterations of the gradient descent from several starting points. As it can be seen from Figure 4.4.2, convergence occurs if we have a good initialization, but the algorithm may diverge. This is due to the fact that in the case of complicated (ill-conditioned) functions, the gradient may not point to the actual direction of the steepest descent.

In the case of more regular functions, however, convergence to a local minimum occurs after few iterations.

In a more general case (multi-dimensional functions), the gradient descent is often slow, and is not used in practice due to the fact that the gradient not always point in the right direction. In practice, we rather use conjugate-gradient descent, which intuitively accelerates the gradient descent by

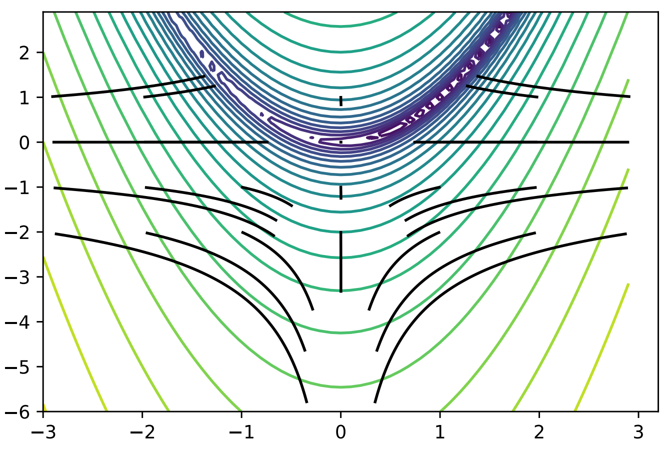


Figure 4.4.2: *Gradient descent on the Rosenbrock functions with different iterations*

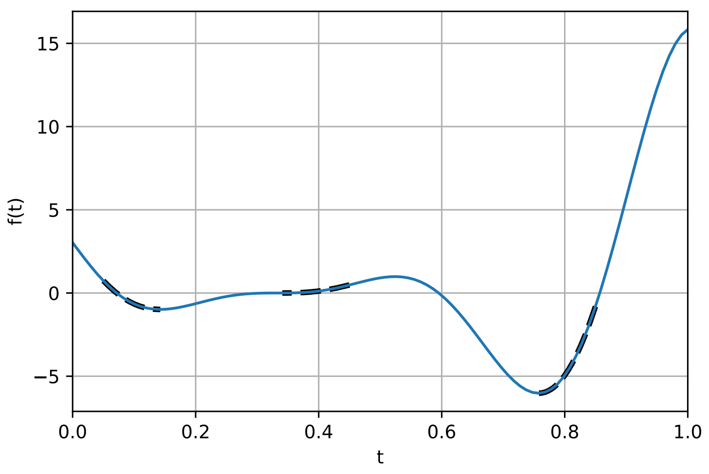


Figure 4.4.3: *Gradient descent on a 1D-function = gradient descent converges to local minima*

adding a momentum; numerically speaking, it means that the update rule is of the form

$$x^{(k+1)} = x^{(k)} - t_k \nabla f(x^{(k)}) + b_k(x^{(k)} - x^{(k-1)}),$$

where the term b_k depends on the Jacobian of f. A direct implementation is presented in the code below:

```
import scipy.optimize as opt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np

X = np.arange(-3, 3, 0.1)
Y = np.arange(-3, 3, 0.1)
X, Y = np.meshgrid(X, Y)

```

```python
def quadratic(x):
    return (x[0]-2)**2 /16 + (x[1]-1)**2/4

def quadratic_gradient(x):
    grad = np.zeros((2,))
    grad[0] = 2*(x[0]-2)/16
    grad[1] = 2*(x[1]-1)/4
    return grad

def save_step(k):
    global steps
    steps.append(k)


steps = []
steps.append([5,-5])
res = opt.minimize(quadratic,[5,-5],method='CG',jac=quadratic_gradient,hess=
    hessian,callback=save_step)
print(steps)
x_coord = [a[0] for a in steps]
y_coord = [a[1] for a in steps]
plt.plot(x_coord,y_coord,'b-o',Linewidth=1)
print(len(x_coord))
plt.contour(X,Y,Z,20)
plt.show()
```

Listing 4.2: *Conjugate gradient method on a quadratic function*

The obtained graph is displayed in Figure 4.4.4

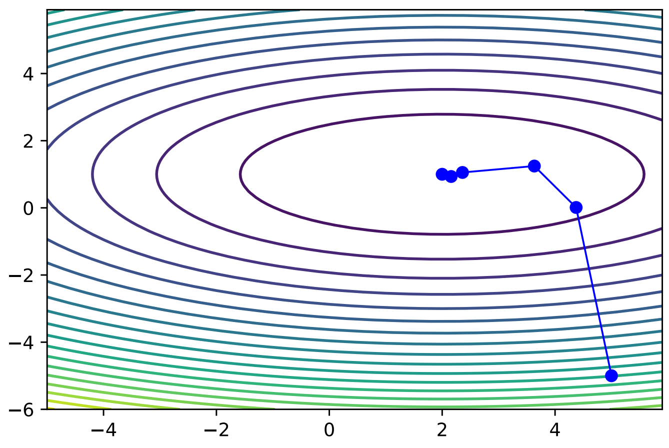


Figure 4.4.4: *Conjugate gradient applied on the function* $f(x,y) = \frac{1}{16}(x-2)^2 + \frac{1}{4}(y-1)^2$

4.4.3 An Important Variation: the Stochastic Gradient Descent

With the recent development of deep learning, the cost functions we wish to optimize are more and more defined on a very high dimension space (the dimension is here defined as the number of free variables, and in that framework is above several hundreds of millions!). Optimizing such functions is an extremely challenging problems, even on modern computers. However, the cost functions involved are usually of the form

$$f(x) = \sum_{k=1}^{N} f_k(x). \tag{4.4.1}$$

For example, the standard squared error term associated with data $x_1, x_2, \ldots, x_n$ is of the form $C(w) = \sum_{k=1}^{n} (w_i - x_i)^2$. When dealing with a large dimension problem, it is common to compute the gradient only on a sample of the f_k's randomly drawn (this is in the literature called a *mini-batch*), and iterate until all the subfunctions f_k have been used (this is called in the literature an *epoch*). When the learning rate t_k decreases with an appropriate rate, with relatively mild assumptions, it is shown in the field of stochastic optimization that stochastic gradient descent converges almost surely to a global minimum when the objective function is convex, and otherwise converges almost surely to a local minimum.
Stochastic gradient descent is implemented in the **sklearn** module, and is one of the basic optimizers which can be in the Deep Learning dedicated modules such as Pytorch, Tensorflow or Keras.

4.5 Going Further: Proximal Algorithms

In the case where functions are not differentiable, gradient methods cannot be applied out of the box. However, if the function to minimize has a similar form as in equation (4.4.1), the gradient method can be extended using projection operators. This approach is called proximal gradient descent, and is extremely efficient to minimize terms related to penalized regression.
For the sake of the discussion, assume we wish to solve

$$\arg\min_{\beta} \Big\{ f(\beta) + \lambda \mathrm{Pen}(\beta) \Big\}, \tag{4.5.1}$$

where $f(\beta) = \frac{1}{2N} \|\mathbf{A}\beta - \mathbf{y}\|_2^2$ denotes the standard convex objective cost function, differentiable with Lipschitz continuous gradient ∇f and Lipschitz constant L; and Pen the penalty, which is a continuous and convex function, not necessarily differentiable everywhere. For example, in the sparse regression framework, $\mathrm{Pen}(\beta)$ is the penalty introduced to enforce sparsity (e.g., $\mathrm{Pen}(\beta) = \|\beta\|_1$ for LASSO, and $\mathrm{Pen}(\beta) = \sum_{g \in \mathscr{G}} \|\beta_g\|_2$ for grouped LASSO – in the latter $\mathscr{G}$ represents the partition of the indexes of the β's into non-overlapping groups). In the compressive sensing literature, the minimization (4.5.1) is usually solved by means of one of the two following algorithms: proximal gradient or ADMM. Both rely on the iterative computation of one or two proximal operators, whose definition is now briefly recalled.
Given any convex, possibly non-smooth, function g, with domain of definition D_g, to be minimized, we denote the proximal operator by

$$\mathbf{prox}_g(\mathbf{v}) = \arg\min_{\mathbf{x}} \Big\{ g(\mathbf{x}) + \frac{1}{2} \|\mathbf{x} - \mathbf{v}\|_2^2 \Big\}. \tag{4.5.2}$$

Roughly speaking, (4.5.2) can be understood either as the generalization of a projection operator or as a compromise between minimizing g while staying close to $\mathbf{v}$. The cornerstone of proximal algorithms for optimization is that $\mathbf{x}^*$ is a minimizer of g if and only if it is a fixed point of $\mathbf{prox}_g$, that is $\mathbf{prox}_g(\mathbf{x}^*) = \mathbf{x}^*$. Therefore, a possible, preliminary algorithm for minimizing g would be to iterate until convergence the proximal operator:

$$\mathbf{x}^{(n+1)} = \mathbf{prox}_{\mu g}(\mathbf{x}^{(n)}), \tag{4.5.3}$$

where μ denotes a step size controlling on "how close" to $\mathbf{x}^{(n)}$ we wish to remain at each iteration. Though $\mathbf{prox}_{\mu f}$ is not a contracting, but only non-expansive, operator, it turns out that (4.5.3) can be adapted to a converging algorithm. We refer to [5, Chapter 3] for an in-depth explanation on this issue and how to solve it. For further explanation, we also define the subdifferential set of any function g as

$$\partial g(\mathbf{x}) = \left\{ \mathbf{y} \; ; \; g(\mathbf{z}) \geq g(\mathbf{x}) + \mathbf{y}^T(\mathbf{z}-\mathbf{x}), \; \mathbf{z} \in D_g \right\}. \tag{4.5.4}$$

Recall that, if g is differentiable on $\mathbf{x}$, then $\partial g(\mathbf{x})$ in (4.5.4) reduces to the singleton $\{\nabla g(\mathbf{x})\}$, and that $\mathbf{x}$ is the minimizer of g if and only if $0 \in \partial g(\mathbf{x})$. Thus, the subdifferential set can be understood as an extension of the differential operator, and it is useful to find the minimizer of convex functions which are non necessarily differentiable, as it happens in our case.
Proximal optimization methods applied on sparse regression problems rely on two important properties of the proximal operator:

1. $\mathbf{prox}_{\mu g} = (\mathbf{I} + \mu \partial g)^{-1}$
2. the separability property, namely if g is fully separable ($g(\mathbf{x}) = \sum_{i=1}^n g_i(x_i)$), then $(\mathbf{prox}_g(\mathbf{v}))_i = (\mathbf{prox}_{g_i}(v_i))$.

As detailed in [5], $(\mathbf{I} + \mu \partial g)^{-1}$ is a relation which is single valued, and therefore can be equaled with a function even if ∂g is not. The second property allows us to get a closed-form term for the proximal operator of Pen in many cases, for example for LASSO, Grouped LASSO and other related penalties. This stems from the form of the penalties introduced, which rely on ℓ_1 or ℓ_2 norms. Let us denote by $\hat{\beta}$ a minimizer of (4.5.1), whose existence is guaranteed by convexity. Then we have for all $\mu > 0$:

$$\begin{aligned} \mathbf{0} \in \partial(\mu f + \lambda\mu \mathrm{Pen})(\hat{\beta}) &\Leftrightarrow \mathbf{0} \in \mu \nabla f(\hat{\beta}) + \lambda\mu \partial \mathrm{Pen}(\hat{\beta}) \\ &\Leftrightarrow \hat{\beta} - \mu \nabla f(\hat{\beta}) \in (\mathbf{I} + \lambda\mu\partial \mathrm{Pen})(\hat{\beta}). \end{aligned} \tag{4.5.5}$$

Using the first property with (4.5.5), we can observe that $\hat{\beta} = \mathbf{prox}_{\mu\lambda\mathrm{Pen}}(\hat{\beta} - \mu\nabla f(\hat{\beta}))$. Thus, a proximal gradient algorithm to obtain the solution of (4.5.1) is based on iterated computations of the proximal operator with $g = \lambda\mathrm{Pen}$, that is:

$$\begin{cases} \text{Initialize} & \beta^{(0)} \\ \text{Repeat} & \beta^{(n+1)} = \mathbf{prox}_{\mu\lambda\mathrm{Pen}}(\beta^{(n)} - \mu\nabla f(\beta^{(n)})), \end{cases} \tag{4.5.6}$$

where μ in (4.5.6) denotes a positive non-increasing step size, possibly constant. As shown in [1], this iterative scheme is guaranteed to converge to $\hat{\beta}$, provided that ∇f is L-Lipschitz, that is $\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\|_2 \leq L\|\mathbf{x}-\mathbf{y}\|_2$, for all $\mathbf{x}, \mathbf{y}$, and provided that $\mu \leq \frac{1}{L}$. For the sparse reconstruction framework, $f(\beta) = \frac{1}{2N}\|\mathbf{A}\beta - \mathbf{y}\|_2^2$ and $\nabla f(\beta) = \frac{1}{N}\mathbf{A}^T(\mathbf{A}\beta - \mathbf{y})$, so we can easily see that L is the highest eigenvalue of $\frac{1}{N}\mathbf{A}^T\mathbf{A}$. Consequently, in that case the optimal value of μ is constant and known.
Obviously, the usefulness of the presented approach depends strongly on whether $\mathbf{prox}_{\lambda\mu\mathrm{Pen}}$ is easily computable or not. Fortunately, this is the case for a wide variety of sparsity-driven regression problems. Due to the separability property, it can be shown that, for the LASSO penalty ($\mathrm{Pen}(\beta) = \|\beta\|_1$) we have

$$\mathbf{prox}_{\lambda\mu\mathrm{Pen}}(\mathbf{v}) = \begin{cases} v_i - \lambda\mu, & \text{if } v_i > \lambda\mu; \\ v_i + \lambda\mu, & \text{if } v_i < -\lambda\mu; \\ 0, & \text{if } |v_i| \leq \lambda\mu; \end{cases}$$

whereas the explicit computation for the grouped LASSO penalty ($\mathrm{Pen}(\beta) = \sum_{g\in\mathscr{G}} \|\beta_g\|_2$) yields

$$(\mathbf{prox}_{\lambda\mu\mathrm{Pen}}(\mathbf{v}))_g = \max\left\{0, 1 - \frac{\lambda\mu}{\|\mathbf{v}_g\|_2}\right\}\mathbf{v}_g.$$

Moreover, in practice the convergence of (4.5.6) can be accelerated when the same idea of adding a momentum in the conjugate gradient is applied with $\beta^{(n)}$ and $\beta^{(n-1)}$. This leads to the FISTA approach [1], whose rate of convergence is better than the direct application of (4.5.6) by an order of magnitude:

```
Data: $\lambda > 0, 0 < \mu < 1/L$
Result: $\hat{\beta}$, a closed-form solution of (4.5.1)
Initialization: set $\mathbf{y}^{(1)} = \beta^{(0)}, t_1 = 1$;
while Convergence is not attained do
    Compute;
    $\beta^{(n+1)} = \mathbf{prox}_{\mu\lambda \mathrm{Pen}}(\mathbf{y}^{(n)} - \mu\nabla f(\mathbf{y}^{(n)}))$;
    $t_{n+1} = \frac{1+\sqrt{1+4t_n^2}}{2}$;
    $\mathbf{y}^{(n+1)} = \beta^{(n)} + \left(\frac{t_n - 1}{t_{n+1}}\right)(\beta^{(n+1)} - \beta^{(n)})$;
end
```

Algorithm 1: The FISTA optimization method [1].

4.6 Case study: A Tikhonov regularization

4.6.1 The Equations

Given a matrix $\mathbf{A}$, a vector of observations $\mathbf{y}$ and a real number $r > 0$, Tikhonov regularization is defined as the following optimization problem:

$$\mathbf{x}_{Tikh} = \arg\min_{\mathbf{x}}\{\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2 + r\|\mathbf{x}\|_2^2\} \tag{4.6.1}$$

In equation (4.6.1), the parameter r is a trade-off parameter between the precision of the regressor (we wish to fit the curve $\mathbf{y}$ with a linear combination of the columns of $\mathbf{A}$) and a penalty term which shrinks the values of the entries of $\mathbf{x}$. This problem is more complex than linear regression, however it remains a convex problem where the objective function is differentiable. Namely:

$$\begin{aligned} f(\mathbf{x}) &= \|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2 + r\|\mathbf{x}\|_2^2 \\ \nabla f(\mathbf{x}) &= -2\mathbf{A}^T(\mathbf{y}-\mathbf{A}\mathbf{x}) + 2r\mathbf{x} \end{aligned}$$

Thus, the minimization can be easily solved using one of the aforementioned methods. The code can be found in the companion Jupyter notebook.

4.7 Further Readings

It is impossible to learn optimization without at first look at [3]: this book provides a full coverage of the field of convex optimization, and is freely accessible online. Furthermore, it is exhaustive, and provides clear insights on how to solve in practice optimization problems.
The rather nice example of the one-dimensional function exhibiting all the problematic points at once in Figure 4.1.1 was taken from [4].

5. Constrained Optimization Techniques

5.1 Seasons in the Abyss

In the previous chapter, we investigated methods used in unconstrained optimizations. Namely, the task at hand was either to find a minimal/maximal value, or to find where this minimal/maximal value is attained. However, there are in most practical problems applications, the practicioner has to take into account additional constraints about the solution (positivity, sparsity, definition in a known domain, etc.). Though we can still verify at each iteration of the previous method whether we still stand in the domain defined by the constrained, there is no guarantee that we attain an optimal value by doing so. Thus, we need additional insights to address constrained optimization problems

5.2 Lagrange Multipliers for Equality Constraints

The first kind of constraints we wish to investigate are the equality constraints, that is the problem at hand is

$$\begin{array}{cc} \text{Find} & \arg\min f(\mathbf{x}) \\ \text{such that} & g(\mathbf{x}) = 0 \end{array} \tag{5.2.1}$$

Obviously, a non-zero constraint can be included in the very definition of g. This problem is usually solved by the Lagrange multipliers, which allows to address (5.2.1) as an unconstrained optimization problem. To further grasp the general description of the method, we first focus on a simple example.

5.2.1 Introductory example

Assume we wish to solve

$$\begin{array}{cc} \text{Find} & \arg\min x^2 + y^2 \\ \text{such that} & y = 3 \end{array} \tag{5.2.2}$$

Here $f(x,y) = x^2 + y^2$, and $g(x,y) = y$. The function is displayed in Figure 5.2.1

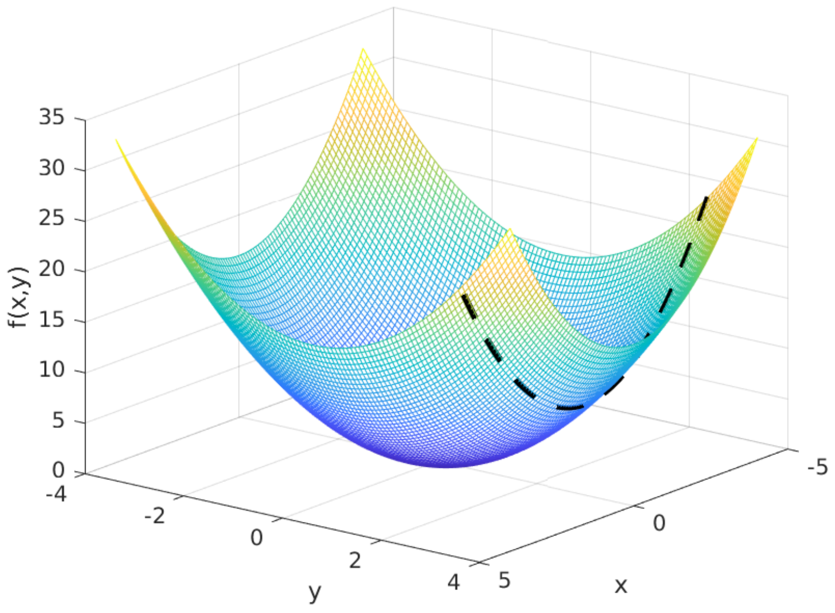


Figure 5.2.1: *The function we wish to minimize. The admissible set is displayed in black.*

Since we have an additional constraint, we want to find the solution on the admissible set displayed in black. In that case, the solution is clearly the point $(0,3)$, and in that point the admissible set $\{x^2+y^2\ ;\ y=3\}$ is tangent to the level line $f(x,y)=9$. Since both curves are tangent, it means that at this point the gradients of f and g are orthogonal to the same vector $\mathbf{x_0}=(1,0,0)$. We have

$$\nabla f(0,3)=\begin{bmatrix}0\\9\\0\end{bmatrix},\ \nabla g(0,3)=\begin{bmatrix}0\\1\\0\end{bmatrix}$$

At the solution point, the gradients are collinear as well, thus we have

$$\nabla f(0,3)=9\nabla g(0,3)$$

5.2.2 Method of Lagrange Multipliers

The idea underneath Lagrange multipliers is that we must find the admissible points at which the gradients of the constraint and the function are orthogonal to the same vector space. To do so, we define

$$L(\mathbf{x},\lambda)=f(\mathbf{x})-\lambda g(\mathbf{x}). \tag{5.2.3}$$

[Note that in (5.2.3), the minus sign is not mandatory and can be replaced by an addition, since we are interested in collinearity.]
We assume here that both functions f and g are differentiable. First of all, we can remark that if $\mathbf{x}^*$ is an admissible solution of (5.2.1), then the solution of the problem

$$\text{Find}\quad \arg\min L(\mathbf{x},\lambda) \tag{5.2.4}$$

is of the form $(\mathbf{x}^*,\lambda^*)$; indeed, as discussed above, at the solution point $\mathbf{x}^*$ both gradients of f and g are collinear, thus there exists a real number λ^* such that $\nabla L(\mathbf{x}^*,\lambda^*)=0$. Thus, the interest of introducing the function L is double. First, the solution of (5.2.1) is one of the solution candidates of $\arg\min L$. Second, we can solve (5.2.1) by solving the unconstrained optimization problem (5.2.4), on which the methods of the previous chapters can be applied. Thus, if both functions f and g are differentiable, the steps of the Lagrange multipliers method are:

1. Define the Lagrangian (5.2.3)
2. Compute the gradient $\nabla L(\mathbf{x},\lambda)$
3. Solve $\nabla L(\mathbf{x},\lambda) = 0$
4. Find among the solutions of $\nabla L(\mathbf{x},\lambda) = 0$ the solution of (5.2.1).

Note that if one of the functions f or g is not differentiable everywhere, steps 2 and 3 of the method can be replaced by a proximal algorithm to find the minimizer of L, as seen in the previous chapter. For example, a code solving (5.2.2) can be as follows:

```
def f(x):
    return x[0]**2 + x[1]**2

def lagrangian_f(x):
    return x[0]**2 + x[1]**2 - x[2]*(x[1]-3)

def lagrangian_gradient(x):
    grad = np.zeros((3,))
    grad[0] = 2*x[0]
    grad[1] = 2*x[1]-x[2]
    grad[2] = -(x[1]-3)
    return grad

res = opt.root(lagrangian_gradient,[1,1,1])
print(res)
```

Listing 5.1: *Finding the root of the Lagrangian's gradient*

The code returns the following:

```
   fjac: array([[-1.00000000e+00, -8.89843754e-14, -1.77968751e-13],
       [-2.81366973e-17, -8.94427191e-01,  4.47213596e-01],
       [ 1.99007475e-13, -4.47213596e-01, -8.94427191e-01]])
     fun: array([-8.07793567e-28,  0.00000000e+00, -0.00000000e+00])
 message: 'The solution converged.'
    nfev: 6
     qtf: array([4.36228831e-12, 1.98602730e-16, 4.87728590e-12])
       r: array([-2.00000000e+00,  8.90325845e-13,  5.34128297e-13, -2.23606798e+00,
        8.94427191e-01,  4.47213595e-01])
  status: 1
 success: True
       x: array([-4.03896783e-28,  3.00000000e+00,  6.00000000e+00])
```

and we can see that the only candidate point is $(0,3,6)$, thus the solution should be $(0,3)$, and it is indeed the case.
Alternatively, we can program the optimization in a straightforward manner:

```
def function(x):
    return x[0]**2+x[1]**2

def constraint(x):
    return x[1]-3

cons = {'type': 'eq', 'fun': constraint}

```

```
res = opt.minimize(function ,[0 ,0] , constraints=cons)

print(res)
```

Listing 5.2: *Equality Constraint Optimization*

and the solution displayed is

```
     fun: 9.0
     jac: array([0., 6.])
 message: 'Optimization terminated successfully.'
    nfev: 13
     nit: 3
    njev: 3
  status: 0
 success: True
       x: array([-2.59052039e-16,  3.00000000e+00])
```

Obviously, this method can be extended to a set of constraints: assume now that we wish to solve

$$\begin{array}{rc} \text{Find} & \arg\min f(\mathbf{x}) \\ \text{such that} & g_1(\mathbf{x}) = c_1 \\ & g_2(\mathbf{x}) = c_2 \\ & \vdots \\ & g_N(\mathbf{x}) = c_N \end{array}$$

The same idea can be generalized to the multidimensional case, if we define $\lambda = [\lambda_1 \ \dots \ \lambda_N]^T$ and $\mathbf{g}(\mathbf{x}) = [g_1(\mathbf{x}) - c_1 \ g_2(\mathbf{x}) - c_2 \ \dots \ g_N(\mathbf{x}) - c_N]^T$, the Lagrangian functional can be defined as

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda_1(g_1(\mathbf{x}) - c_1) - \dots - \lambda_N(g_N(\mathbf{x}) - c_N) = f(\mathbf{x}) - \lambda^T \mathbf{g}(\mathbf{x}),$$

and the same procedure can be applied.

5.3 Karush Kuhn Tucker Conditions for General Constraints

Karush Kuhn Tucker (KKT) conditions are a generalization of the Lagrange multiplier idea. They provide necessary conditions for a point to be the solution of an optimization problem which includes a set of equality constraints and a set of inequality constraints. The system of equations and inequalities corresponding to the KKT conditions is usually not solved directly, except in the few special cases where a closed-form solution can be derived analytically. In general, many optimization algorithms can be interpreted as methods for numerically solving the KKT system of equations and inequalities.

5.3.1 Overview of KKT Conditions

Assume that we wish to optimize

$$\begin{array}{rc} \text{Find} & \arg\min f(\mathbf{x}) \\ \text{such that} & g_i(\mathbf{x}) \leq 0, \ i = 1 \dots N \\ \text{and} & h_j(\mathbf{x}) = 0, \ j = 1 \dots M \end{array} \tag{5.3.1}$$

Based on (5.3.1), we form the vectors $\mathbf{g}(\mathbf{x}) = [g_1(\mathbf{x}) \ \dots \ g_N(\mathbf{x})]^T$ and $\mathbf{h}(\mathbf{x}) = [h_1(\mathbf{x}) \ \dots \ h_M(\mathbf{x})]^T$. The Lagrangian function is defined as

$$L(\mathbf{x}, \lambda, \mu) = f(\mathbf{x}) - \lambda^T \mathbf{g}(\mathbf{x}) - \mu^T \mathbf{h}(\mathbf{x})$$

Then the solution candidates must be looked in the set of the solutions of $\nabla L = 0$. If $\mathbf{x}^*$ is a local minimum, under regularity conditions (independance of the gradients' contraints, for example), then there exists a λ^* and a μ^* such that

$$\nabla f(\mathbf{x}^*) = (\lambda^*)^T \mathbf{g}(\mathbf{x}) + (\mu^*)^T \mathbf{h}(\mathbf{x}) \tag{5.3.2}$$

Equation (5.3.2) provides a system to solve numerically, which returns a set of possible solutions.

5.3.2 Numerical Example

We will solve numerically the following optimization problem:

$$\begin{cases} \arg\min x(x-1)(x+1) \\ \text{s.t. } x \geq 0, \\ \sin(2\pi x) = -0.5 \end{cases}$$

Figure 5.3.1 represents the cost function and the equality constraint $\sin(2\pi t) - 0.5 = 0$. A quick glance at the graph shows that the solution should lie between 0.5 and 1, as the solution we wish is one of the root of the raised sine. Note that we took into account the inequality constraint by only plotting the relevant part of the graph. In order to solve this numerically, we use the

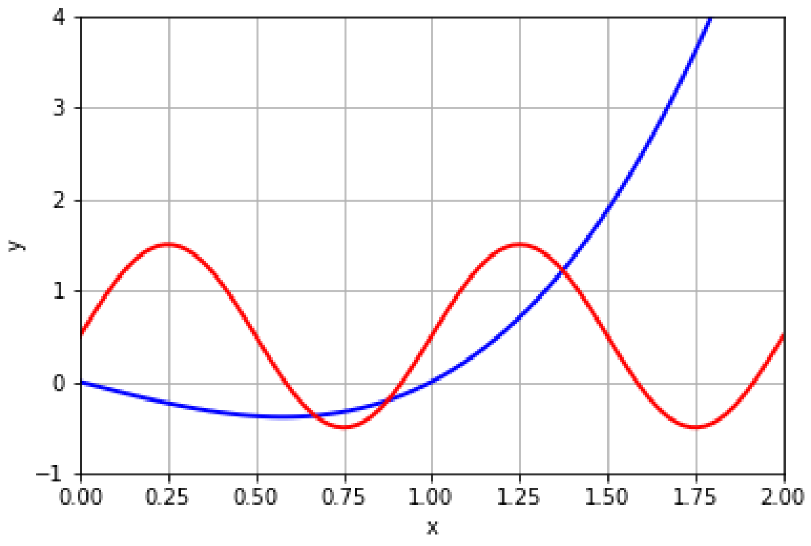


Figure 5.3.1: *Graph of the cost function f (blue) and the constraint equality (red).*

function *scipy.optimize.minimize*, which allows to take into account boundaries, linear and non linear constraints, inequalities and equalities altogether. We define the function, the constraint and the bound as follows:

```python
my_bounds = [[0,np.inf]]

def my_function(x):
    return x[0]*(x[0]-1)*(x[0]+1)

def my_jac(x):
    return 2*x[0]*x[0]-1

def my_constraint(x):
    return np.sin(2*np.pi*x[0])+0.5

cons = {'type': 'eq', 'fun': my_constraint}
```

Listing 5.3: *Definition of the constraints*

Then we run the minimizer:

```
res = opt.minimize(my_function,[0.7],bounds=my_bounds,constraints=cons,jac=
    my_jac)
print(res)
```

The output of the previous lines is

```
     fun: -0.3848379629816655
     jac: array([-0.31949638])
 message: 'Optimization terminated successfully.'
    nfev: 6
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([0.58333333])
```

The returned solution is close to the true optimum. The reader will find in the official documentation of scipy a more detailed examples.

5.4 The ADMM for Linear/Quadratic Constraints

The alternating direction method of multipliers (ADMM) is an algorithm that solves convex optimization problems by breaking them into smaller pieces, each of which are then easier to handle. It has recently found wide application in a number of areas, from optics to signal processing. ADMM is used in a large number of papers at this point, so it is impossible to be complete in only one paragraph.

5.4.1 Mathematical Description

The algorithm solves problems of the form

$$\begin{array}{ll} \arg\min & f(\mathbf{x})+g(\mathbf{z}) \\ \text{such that} & \mathbf{Ax}+\mathbf{Bz}=\mathbf{c} \end{array}$$

where $\mathbf{x}\in\mathbf{R}^n$, $\mathbf{z}\in\mathbf{R}^m$, $\mathbf{c}\in\mathbf{R}^p$, , and $\mathbf{A}$, $\mathbf{B}$ are matrices with suitable dimensions.
To solve this problem, we introduce the "augmented Lagrangian", which takes both from Lagrange multipliers and proximal methods:

$$L_\mu(\mathbf{x},\mathbf{z},\lambda)=f(\mathbf{x})+g(\mathbf{z})+\lambda^T(\mathbf{Ax}+\mathbf{Bz}-\mathbf{c})+\frac{\mu}{2}\|\mathbf{Ax}+\mathbf{Bz}-\mathbf{c}\|_2^2 \tag{5.4.1}$$

ADMM can be understand as iterating the optimization one variable after the other in (5.4.1), until the algorithm convergences to a local minimizer:

$$\begin{cases} \mathbf{x}^{(k+1)} & =\arg\min_{\mathbf{x}} L_\mu(\mathbf{x},\mathbf{z}^{(\mathbf{k})},\lambda^{(k)}) \\ \mathbf{z}^{(k+1)} & =\arg\min_{\mathbf{z}} L_\mu(\mathbf{x}^{(k+1)},\mathbf{z},\lambda^{(k)}) \\ \lambda^{(k+1)} & =\lambda^{(k)}+\mu(\mathbf{Ax}^{(k+1)}+\mathbf{Bz}^{(k+1)}-\mathbf{c}) \end{cases} \tag{5.4.2}$$

It is common to scale the problem in practice, in order to get shorter formulae. if we define the residual $\mathbf{r} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} - \mathbf{c}$, we get $\lambda^T \mathbf{r} + \frac{\mu}{2}\|\mathbf{r}\|_2^2 = \frac{\mu}{2}\|\mathbf{r}+\mathbf{u}\|_2^2 - \frac{\mu}{2}\|\mathbf{u}\|_2^2$, where we define $\mathbf{u} = \frac{1}{\mu}\lambda$. Therefore, (5.4.2) becomes:

$$\begin{cases} \mathbf{x}^{(k+1)} & = \arg\min_{\mathbf{x}} \left\{ f(\mathbf{x}) + \frac{\mu}{2}\|\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z}^{(k)} - \mathbf{c} + \mathbf{u}^{(k)}\|_2^2 \right\} \\ \mathbf{z}^{(k+1)} & = \arg\min_{\mathbf{z}} \left\{ g(\mathbf{z}) + + \frac{\mu}{2}\|\mathbf{A}\mathbf{x}^{(k+1)} + \mathbf{B}\mathbf{z} - \mathbf{c} + \mathbf{u}^{(k)}\|_2^2 \right\} \\ \mathbf{u}^{(k+1)} & = \mathbf{u}^{(k)} + \mu(\mathbf{A}\mathbf{x}^{(k+1)} + \mathbf{B}\mathbf{z}^{(k+1)} - \mathbf{c}) \end{cases} \tag{5.4.3}$$

We refer to [2] for an description of the conditions which guarantee the convergence of the algorithm to a local minimum. In general, ADMM is used for cases when modest accuracy is sufficient for the problem at hand. As for most iterative algorithms, a stopping criterion must be provided for (5.4.3)by the user for practical use. Define the following quantities:

$$\mathbf{r}^{(k)} := \mathbf{A}\mathbf{x}^{(k)} + \mathbf{B}\mathbf{z}^{(k)} - \mathbf{c}$$

$$\mathbf{s}^{(k)} := \mu\mathbf{A}^T\mathbf{B}(\mathbf{z}^{(k)} - \mathbf{z}^{(k-1)})$$

$$\varepsilon_p := \sqrt{p}\varepsilon_{abs} + \varepsilon_{rel}\max\{\|\mathbf{A}\mathbf{x}\|_2, \|\mathbf{B}\mathbf{z}\|_2, \|\mathbf{c}\|_2\} \tag{5.4.4}$$

$$\varepsilon_d := \sqrt{n}\varepsilon_{abs} + \varepsilon_{rel}\|\mathbf{A}^T\lambda\|_2, \tag{5.4.5}$$

where in (5.4.4) and (5.4.5), ε_{abs} and ε_{rel} are absolute precision and relative precision are specified by the user, n is the dimension of $\mathbf{x}$, p is the number of rows in $\mathbf{A}$. [2] recommends the following choices as stopping criteria:

$$\|\mathbf{r}^{(k)}\|_2 \le \varepsilon_p, \quad \|\mathbf{s}^{(k)}\|_2 \le \varepsilon_d$$

Obviously, (5.4.3) is of interest only if the first two steps are numerically solvable. In several cases of interest, this will be the case. Recall that the proximal operator of any function was defined in (4.5.2) as

$$\mathbf{prox}_g(\mathbf{v}) = \arg\min_{\mathbf{x}} \left\{ g(\mathbf{x}) + \frac{1}{2}\|\mathbf{x} - \mathbf{v}\|_2^2 \right\}.$$

So, in practice, when matrices $\mathbf{A}$ and $\mathbf{B}$ reduce to identity, the first two steps of one ADMM iteration can be seen as the computation of two proximal operators, which can be in many applications solvable.

5.4.2 Object Oriented Programming: my ADMM in a Box

We will implement the ADMM inside a class, to grasp some concepts of object oriented programming. For simplicity, we will program it when the objective function is quadratic, in order to use closed-form terms when possible for steps 1 and 2 of (5.4.3). Roughly speaking, a class is a way to encapsulate variables and functions inside a "box", which has the advantage of modularity, reuseability and makes the code easier to maintain. The solver will need inputs set by the user, namely: $\mathbf{A}$, $\mathbf{B}$, $\mathbf{c}$, f, g, μ, ε_{abs} and ε_{rel}. The class will also have inner variables $\mathbf{x}$, $\mathbf{z}$, λ. The methods (functions included in the class) will do the following:

1. compute the first step, either in closed form or using a scipy solver.
2. compute the second step, either in closed form or using a scipy solver.
3. Update the auxiliary variable
4. Check whether the stopping criterion has been attained; if so, stop the solver, if not, iterate

The general architecture of the class we wish to program is as follows:

```python
class ADMM_quadratic_solver:
    import numpy as np
    import scipy.optimize as opt
    def __init__(self,A,B,c,P,mu,type_penalty=['lasso',0.1],tol_abs = 0.1,
    tol_rel = 0.01, max_iter = 1000):
        self.A = A
        self.B = B
        self.c = c
        self.P = P
        self.type_penalty = type_penalty
        self.mu = mu
        self.tol_abs = tol_abs
        self.tol_rel = tol_rel
        self.max_iter = max_iter
        # Initializes the variables to zero
        self.x = np.zeros((A.shape[1],1))
        self.z = np.zeros((B.shape[1],1))
        self.u = np.zeros((A.shape[0],1))
        self.eps_primal = np.Inf
        self.eps_dual = np.Inf

    def get_solution(self):
        return [self.x,self.z,self.u]

    def compute_step1_ADMM(self):
        #
        # INCLUDE THE CODE HERE
        #

    def compute_step2_ADMM(self):
        #
        # INCLUDE THE CODE HERE
        #


    def compute_step3_ADMM(self):
        #
        # INCLUDE THE CODE HERE
        #


    def check_stopping_criterion(self,residual_primal,residual_dual):
        #
        # INCLUDE THE CODE HERE
        #

    def solve(self):
        #
        # INCLUDE THE CODE HERE
        #
```

Listing 5.4: *General architecture of the ADMM class*

The reader can fill the functions based on the following case study, or during the working session on the dedicated Jupyter notebook.

5.5 Case Study: Variations on the Least-Square Estimate

Least square estimate is often used for regression purposes in the field of machine learning and statistical inference. We aim to solve the following problem:

$$\arg\min\left\{\frac{1}{2}\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2+r\mathrm{Pen}(\mathbf{x})\right\} \tag{5.5.1}$$

where in (5.5.1) $\mathbf{y}$ is a signal of interest, $\mathbf{A}$ is a dictionary of basic shapes, and Pen is a penalty put on the regressor $\mathbf{x}$. The signal we want to investigate is a sample of an ECG recording, displayed in Figure 5.5.1. Our objective is to locate the peaks in this signal. In order to do so, we will use a ℓ_1

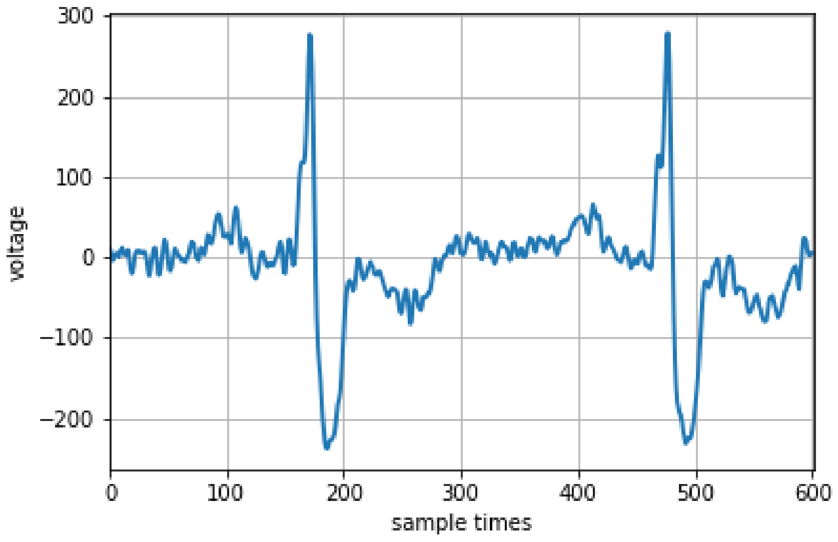


Figure 5.5.1: *Part of the ECG recording of interest*

driven penalty, namely

$$\mathrm{Pen}(\mathbf{x})=\|\mathbf{F}\mathbf{x}\|_1.$$

We will investigate LASSO ($\mathbf{F}=\mathbf{I}$). The optimization problem will be solved using ADMM. Indeed, if the objective function f is a quadratic function (as it is the case in regression problems)

$$f(\mathbf{x})=\frac{1}{2}\mathbf{x}^T\mathbf{P}^T\mathbf{P}\mathbf{x}+\mathbf{q}^T\mathbf{x}+\mathbf{r},$$

then it can be shown that

$$\arg\min_{\mathbf{x}}\left\{f(\mathbf{x})+\frac{\mu}{2}\|\mathbf{x}-\mathbf{v}\|_2^2\right\}=(\mathbf{I}+\mu\mathbf{P}^T\mathbf{P})^{-1}(\mu\mathbf{P}^T\mathbf{v}-\mathbf{q})$$

In our case

$$f(\mathbf{x})=\frac{1}{2}\mathbf{y}^T\mathbf{y}-\frac{1}{2}\mathbf{y}^T\mathbf{A}\mathbf{x}-\frac{1}{2}\mathbf{x}^T\mathbf{A}^T\mathbf{y}+\frac{1}{2}\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}=\frac{1}{2}\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}-\mathbf{y}^T\mathbf{A}\mathbf{x}+\frac{1}{2}\mathbf{y}^T\mathbf{y},$$

so the first step of the ADMM can be easily computed with

$$\mathbf{P}=\mathbf{A},\ \mathbf{q}=-\mathbf{A}^T\mathbf{y},\ \mathbf{r}=\frac{1}{2}\mathbf{y}^T\mathbf{y}$$

For LASSO, the second step of ADMM can also be computed explicitely as

$$\max(\mathbf{x}-r/\rho,0)-\max(-\mathbf{x}-r/\rho,0)$$

The code for ADMM for LASSO is displayed below.

```python
def ADMM_for_generalized_LASSO(dictionary, F, signal, sparsity_parameter,
    admm_parameter, tol_x = 0.1, tol_y = 0.1, max_iter=1000):

    # x,z,u are initialized randomly
    x = np.random.rand(dictionary.shape[1],1)
    z = np.random.rand(dictionary.shape[1]-1,1)
    u = np.random.rand(dictionary.shape[1]-1,1)

    # Big computations must be done outside the loop - THIS IS HUGE !
    M = np.dot(dictionary.T,dictionary) + admm_parameter * np.dot(F.T,F)
    inverse_AF = np.linalg.inv(M)
    ATy = np.dot(dictionary.T,signal)

    stopping_criterion2 = np.linalg.norm(signal-np.dot(dictionary,x))
    print('x_err = 0, y_err = {}'.format(stopping_criterion2))

    for current_iter in np.arange(max_iter):
        x_old = np.copy(x)

        vec = ATy + admm_parameter * np.dot(F.T,z-u)
        x = np.dot(inverse_AF,vec)

        vec = np.dot(F,x) + u
        thresh = sparsity_parameter / admm_parameter
        z = ((np.abs(vec) - thresh)>0)*(np.abs(vec)-thresh) * np.sign(vec)

        u = np.dot(F,x) + u - z

        print('iteration {} / {}'.format(current_iter,max_iter))
        stopping_criterion = np.linalg.norm(x-x_old)
        residual = np.linalg.norm(signal-np.dot(dictionary,x))
        stopping_criterion_2 = residual
        print('x_err = {}, y_err = {}'.format(stopping_criterion,
    stopping_criterion_2))

        if stopping_criterion < tol_x or stopping_criterion_2 < tol_y:
            break

        if current_iter==max_iter:
            print('The maximum number of iterations allowed has been reached.
    Result may be imprecise')

    return [x,z,u]
```

Listing 5.5: *ADMM for LASSO penalized regression*

To obtain the figures displayed in this chapter, we made use of data saved in a Matlab file, as follows:

```python
import numpy as np
import scipy.io as sio
import matplotlib.pyplot as plt
matlab_data = sio.loadmat('ECG_data.mat')
dictionary = matlab_data['B']
signal = matlab_data['sample_ECG']
signal = signal.T

F = np.eye(601,602)
res = ADMM_for_generalized_LASSO(dictionary, F, signal, 0, 10, tol_x = 0.01,
    tol_y = 900)
res2 = ADMM_for_generalized_LASSO(dictionary, F, signal, 100, 10, tol_x = 0.01,
    tol_y = 900)
```

Figure 5.5.2 illustrates the results obtained with Ordinary Least Squares (OLS) estimate. The approximation of the signal can be very good, however we can make very little use of the regressor to locate the peaks, particularly if the signal is noisy.

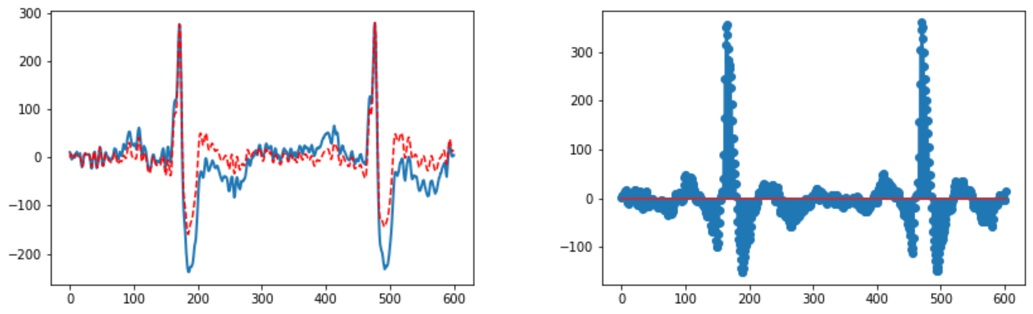


Figure 5.5.2: *Left: original signal and OLS approximation. Right: OLS regressor*

On the other hand, we can see that the LASSO provides a regressor which can be expoited to find the pulses efficiently, since it enforces the sparsity of the regressor. We encourage the reader to use the Jupyter notebook associated with this chapter to experiment with the different parameters of the provided function.

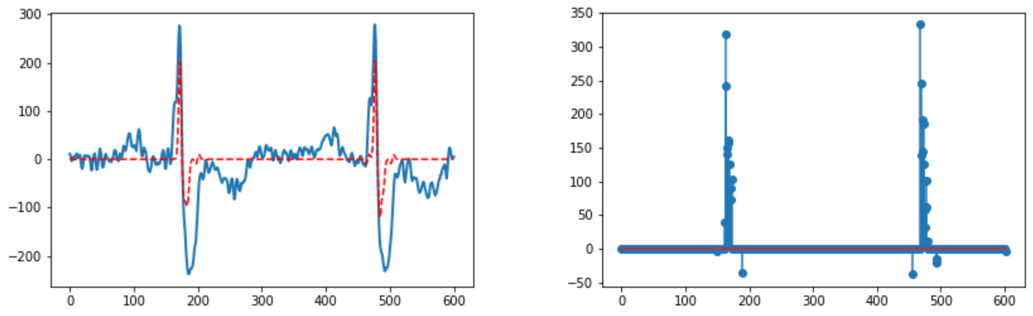
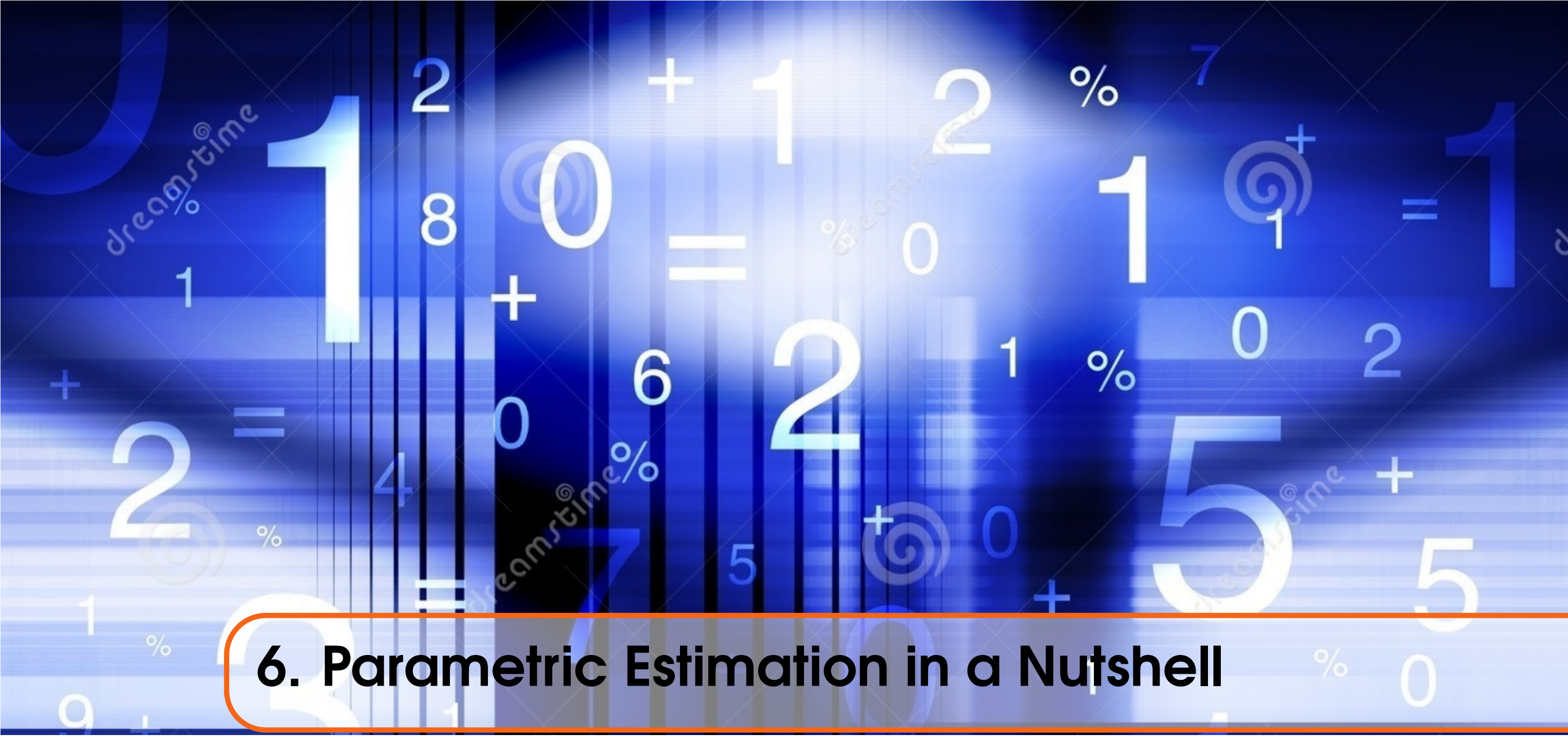


Figure 5.5.3: *Left: original signal and LASSO approximation. Right: LASSO regressor*

5.6 Further Readings

As aforementioned, the book [3] is a goldmine to further investigate the mathematical aspects of convex optimization, constrained of not. It further presents useful algorithms in the third part of the book. For further, modern, insights on proximal methods and ADMM, we refer to [5] and [2] for a friendly, engineering-driven, presentation of these methods.

IV A bit of Statistics and Machine Learning



6. Parametric Estimation in a Nutshell

6.1 Fear of the Dark

In numerous experiments, it is common to model the observations by means of a mathematical model. For example, we may model the ECG pulse repartition in a signal by a point process, the electrical signal recorded from a muscle as a function of the sensor resistivity, and so forth.
The common assumption in the framework of frequentist inference is the existence of a "true" model, under which the data are generated [1]. It is therefore necessary to get several observations from this model, in order to perform on it inference, that is finding a relevant model among many. Statistics is the field of mathematics which provides the tools required for inference (also called estimation), and it is obviously an important topic for all engineers in any field.

6.2 Estimation Theory, Parametric, Non-parametric, Semi-parametric

Statistical estimation can be performed in three ways: under a parametric setting, in which we assume that the model depends on a finite dimensional parameter $\theta \in \mathbb{R}^n$; In that case, we aim to estimate θ.

■ **Example 6.1 — Parametric estimation.** We assume that the grades in a class are ditributed according to a Gaussian distribution. Under this assumption, we wish to estimate the average μ and the standard deviation σ in order to fully characterize the model. Therefore $\theta = (\mu, \sigma)$. ■

In a non-parametric setting, we assume that the data distribution has some regularity functional properties, but we don't assume that it depends on a parameter. Semiparametric approaches perform inference on a partly parametric model, while reducing the influence of a nuisance contribution, possibly non parametric. We will focus in this chapter on the parametric approach.

Definition 6.2.1 — sample. A statistical sample will be defined as a sequence of n independent and identically distributed (i.i.d.) random variables $X_1,\ X_2,\ \dots,\ X_n$, with common distribution $f(\cdot;\theta)$ depending on a parameter $\theta \in \mathbb{R}^n$.

[1] This approach differs from the Bayesian paradigm, in which data rather influence the degree of belief we put on a given prior model.

6.3 Basics of Parametric Estimation

Since samples are assumed to be i.i.d., the joint distribution of the vector $(X_1, X_2, \ldots, X_n)$ is known and is equal to

$$f(x_1, x_2, \ldots, x_n; \theta) = \prod_{k=1}^{n} f(x_k; \theta) \ .$$

We provide here a series of definition which shall be used along the whole chapter.

Definition 6.3.1 — Estimator. An estimator of θ is a random variable defined as a functional of the sample $(X_1, X_2, \ldots, X_n)$, and is denoted by $\hat{\theta}(X_1, X_2, \ldots X_n)$, or more shortly $\hat{\theta}$.

Remember the an estimator is a random variable, and therefore its numeral value is strongly dependent on the sample itself.

■ **Example 6.2** Assume that we dispose of a sample $X_a, X_2, \ldots X_n$ i.i.d with common distribution $U([0, \theta])$. We can suggest as an example two estimators of θ:

1. $\hat{\theta}_1 = \frac{2}{n} \sum_{k=1}^{n} X_k$
2. $\hat{\theta}_2 = \max_{1 \leq k \leq n} X_k$

Both approaches are legitimate, but the second estimate should be preferred in practice (as an exercise: why?) ■

Definition 6.3.2 — Unbiased estimator. The bias of an estimator $\hat{\theta}$ is defined as

$$\text{bias}(\hat{\theta}) = E(\hat{\theta} - \theta).$$

When $E(\hat{\theta}) = \theta$, we shall that the estimator $\hat{\theta}$ is unbiased.

Unbiasedness only states that the average value of an estimate (should we repeat the experiments) is close to the true value of the estimator. It does not say that the estimate is close to the true value of θ (here, the sample size is fixed, and the number of experiments is large).

Definition 6.3.3 — Consistent estimator. The estimator $\hat{\theta}$ is consistent is it converges in probability to θ as the sample size n tends to infinity:

$$\forall \varepsilon > 0, \ P(|\hat{\theta} - \theta| > \varepsilon) \longrightarrow 0 \text{ as } n \to \infty.$$

Consistency is an important property, in the sense that it defines a "good" estimator of the parameter of interest. It is a guarantee that the numerical value of the estimator is close to the true value, provided the sample size is large enough (here, the sample size tends to infinity, while the number of experiments equals one).

Definition 6.3.4 — Asymptotically normal estimator. A consistent estimator $\hat{\theta}$ is said to be asymptotically normal when $\sqrt{n}(\hat{\theta} - \theta)$ converges in distribution to a normal distribution.

Asymptotic normality states that the distribution of the estimator's values around the true value is normally distributed, with a variance decreasing to 0 with a $\sqrt{n}$ rate.

■ **Example 6.3** We would like to check numerically if the estimators defined in the previous example verify the aforementioned properties.

```
import numpy as np
import numpy as np
import scipy.stats
import matplotlib.pyplot as plt
import pandas as pd
```

```python
import seaborn as sns

# the numerical true value of theta is defined
true_theta = np.pi
nb_exp = 100
sample_size = 10000
data = np.random.uniform(low=0,high=true_theta, size=(nb_exp,sample_size))

sample_sizes = np.matlib.repmat(np.arange(1,sample_size+1),nb_exp,1)
# This estimator is the sample mean times 2
estimator_1 = 2*np.cumsum(data, axis=1) / sample_sizes

# This computes the max estimator
estimator_2 = np.zeros(data.shape)
estimator_2[:,0]=data[:,0]
for k in np.arange(1,sample_size):
    estimator_2[:,k] = np.amax(data[:,0:k],axis=1)

plt.plot(estimator_1.T)
plt.plot(np.arange(10000),true_theta * np.ones((sample_size,)),'k--',linewidth
    =3)
plt.ylim((2.5,3.5))
plt.xlim((0,sample_size))
plt.xlabel('Sample size')
plt.ylabel('Estimator value')
plt.grid()
plt.savefig('ch6_estimator1_graph.pdf')
plt.show()
plt.plot(estimator_2.T)
plt.plot(np.arange(10000),true_theta * np.ones((sample_size,)),'k--',linewidth
    =3)
plt.ylim((3,3.2))
plt.xlim((0,sample_size))
plt.xlabel('Sample size')
plt.ylabel('Estimator value')
plt.grid()
plt.savefig('ch6_estimator2_graph.pdf')
plt.show()

# import as a pandas dataframe the last values of the estimate
data_est1 = pd.DataFrame(np.sqrt(sample_size) * (estimator_1[:,-1] - true_theta
    ))
sns_plot = sns.distplot(data_est1,bins=20)
fig = sns_plot.get_figure()
fig.savefig('ch6_est1_an.pdf')

# import as a pandas dataframe the last values of the estimate
data_est2 = pd.DataFrame(np.sqrt(sample_size) * (estimator_2[:,-1] - true_theta
    ))
sns_plot=sns.distplot(data_est2,bins=20)
fig = sns_plot.get_figure()
fig.savefig('ch6_est2_an.pdf')
```

Listing 6.1: *Numerical checks on previous estimates*

■

The graphs obtained by executing the previous script are displayed in Figure 6.3.1 for the first estimator, based on the sample mean, and in Figure 6.3.2 for the maximal value estimator.
From the previous investigation, we can conclude that there exists a strong belief that both estimators are consistent, that the sample mean based estimator is asymptotically normal, while the maximum

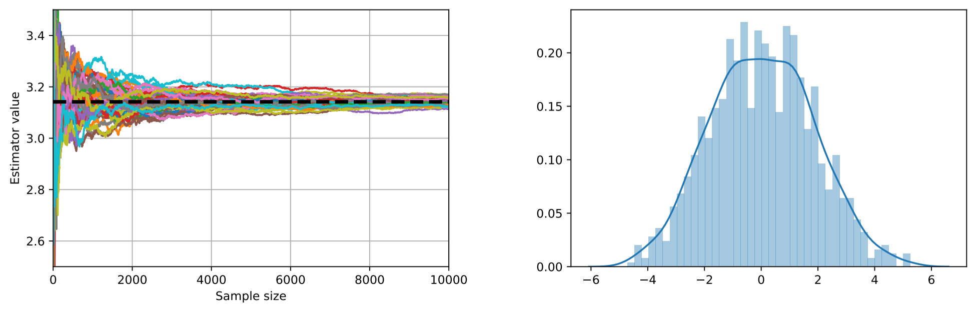


Figure 6.3.1: *Numerical investigation of consistency (left) and asymptotic normality (right) for the sample mean based estimator.*

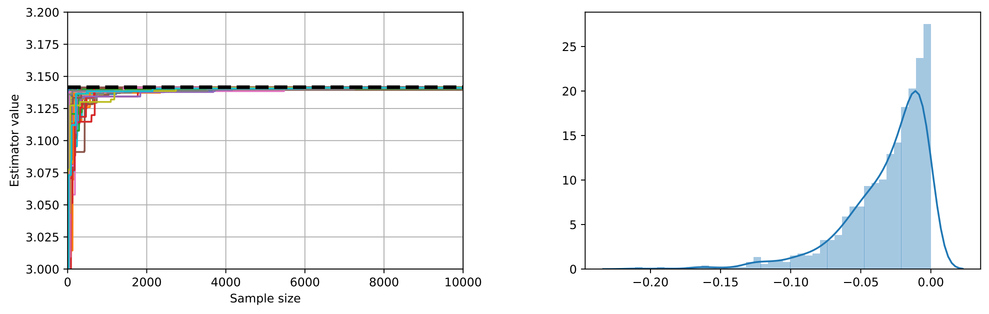


Figure 6.3.2: *Numerical investigation of consistency (left) and asymptotic normality (right) for the maximum based estimator.*

based estimator is not (as an exercise: why what this last point trivial to guess?). We can also have a strong belief that the first estimator is unbiased, while the second is not, while being asymptotically unbiased.
Note that asymptotic normality, though it explains the behavior of the estimator with respect to the data sample size, is not a necessary condition to attain: in practice, the second estimator should be preferred due to its much smaller variance.

6.4 The Maximum Likelihood Estimator

6.4.1 Definition

Considering the sample of observations $(X_1, X_2, \ldots, X_n)$ at hand, we can consider the functional $\prod_{k=1}^{n} f(x_k;\theta)$ as a function of θ only, since the values of x_k are given by the observations.

Definition 6.4.1 — likelihood and log-likelihood function. We define the likelihood function as

$$\mathscr{L}(\theta) = \prod_{k=1}^{n} f(x_k;\theta),$$

and the log-likelihood function as

$$\ell(\theta) = \sum_{k=1}^{n} \log f(x_k;\theta),$$

In the likelihood function, we invert the role of θ and the X_i's: the observations are considered fixed (which is conform to their inherent significance of observations - they do not vary once the experiment is finished), whereas θ varies. In that matter ℓ provides a degree of confidence on a given value θ, given the recorded observed: the higher ℓ is, the most likely θ is close to the true value of the unknown parameter. Based on this considerations, the maximum likelihood estimator (MLE) is defined as

$$\hat{\theta}_{MLE} = \arg\max_{\theta} \ell(\theta)$$

■ **Example 6.4** Going back the i.i.d sample distributed according to $U(0,\theta)$, we can compute analytically the likelihood as

$$\mathscr{L}(\theta) = \begin{cases} 0 & \text{if } \theta < \max(X_1, X_2, \dots X_n) \\ \frac{1}{\theta^n} & \text{if } \theta \geq \max(X_1, X_2, \dots X_n) \end{cases}$$

and observe that this function is non-decreasing in the second case. Thus, the MLE is equal to $\hat{\theta}_{MLE} = \max(X_1, X_2, \dots X_n)$. ■

Note that in many cases (such that in a multivariate setting), the MLE is not easily computable. When confronted to such a case, we perform the optimization numerically, either as described in the previous chapter, or by using a stochastic optimization approach.

6.5 Example: Monte-Carlo Method for Numerical Integration

Monte-Carlo methods are powerful techniques to perform numerical optimization and integration, they are part of a large class of stochastic algorithms, largely used in inference. In this Python example, we shall use a Monte-Carlo method to estimate numerically an integral, and compare it to a regular numerical integration.

6.5.1 Problem Overview

Let us assume that we wish to integrate

$$I = \int_0^{\infty} f(x)\,dx, \quad f(x) = \frac{e^{-2x}}{1+(x-1)^4}$$

This integral clearly converges to a finite value, though there is no existing closed-form of the integral's solution. Thus, we want to estimate this value numerically.

6.5.2 Vanilla Numerical Integration

Since we have a close-form for the integrand, the fast way to approximate this integral would be by numerical integration, using for example a trapezoid approximation:

$$I \approx \sum_{k=0}^{\infty} \frac{(f(x_k)+f(x_{k+1})(x_{k+1}-x_k)}{2},$$

where x_n is a grid of points taken from the x-axis.
The code for numerical integration is displayed below:

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

def my_function(x):
    return np.exp(-0.5*x) / (1+(x-1)**4)
```

```

x = np.arange(0,100,1)
y = my_function(x)
I1 = np.trapz(y,x)
```

Listing 6.2: *Numerical Integration Using the Trapeze Method*

The obtained numerical value is 1.05. We can refine the grid by either taking more point, or taking a larger domain of integration (however, this function descreases fast to 0, so the last idea has little chance to work). The values obtained after refining the grid are close to 1.15927. All in all, deterministic numerical integration works; however:

- We need to define manually the inf value (and we might be wrong about it)
- We need to define manually the grid size (too small = bad approximation, too large = waste of computations)

6.5.3 Ordinary Monte-Carlo Integration

The idea underneath Monte Carlo integration is that any integral can be seen as the expectation of the functional of a random variable. Let's say that we approximate I by the integral from 0 to 10, then

$$\int_0^{10} \frac{e^{-0.5x}}{1+(x-1)^4}\,dx = 10E\left(\frac{e^{-0.5U}}{1+(U-1)^4}\right),$$

where $U \sim U([0,10])$ is a uniform random variable on $[0,10]$. The Law of Large Numbers guarantees the convergence of the empirical mean to the expectation in that case, so the algorithm is

1. draw a sample $U_1, U_2, \ldots$ of n uniform random variables in [0,10]
2. approximate

$$I \approx \frac{10}{n}\sum_{k=1}^{n} \frac{e^{-0.5U_k}}{1+(U_k-1)^4}$$

The Python code is as follows:

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

def my_function(x):
    return np.exp(-0.5*x) / (1+(x-1)**4)

def vanilla_MC_for_my_function(sample_size=100,upper_bound=10):
    # draw the samples
    sample = 10*np.random.random(sample_size)
    values = my_function(sample)
    return 10*np.mean(values)
```

Listing 6.3: *Standard Monte Carlo Integration*

Results for up to 10000 iterations of the Monte Carlo integration are displayed in Figure 6.5.1. In all cases, the estimator converges to the true value of the integral. Since it is a stochastic approach, convergence also depends on the samples drawn.

6.5.4 Refining Results With Importance Sampling

Note that in the previous method, we did not need to specify a grid for integration. However, there is a larger variance in the result, and it still requires lots of points and an upper limit.
Thus, we use the fundamental idea of importance sampling: we can observe that

$$I = \int_0^{\infty} \frac{e^{-0.5x}}{1+(x-1)^4}\,dx = 2E\left(\frac{1}{1+(X-1)^4}\right),$$

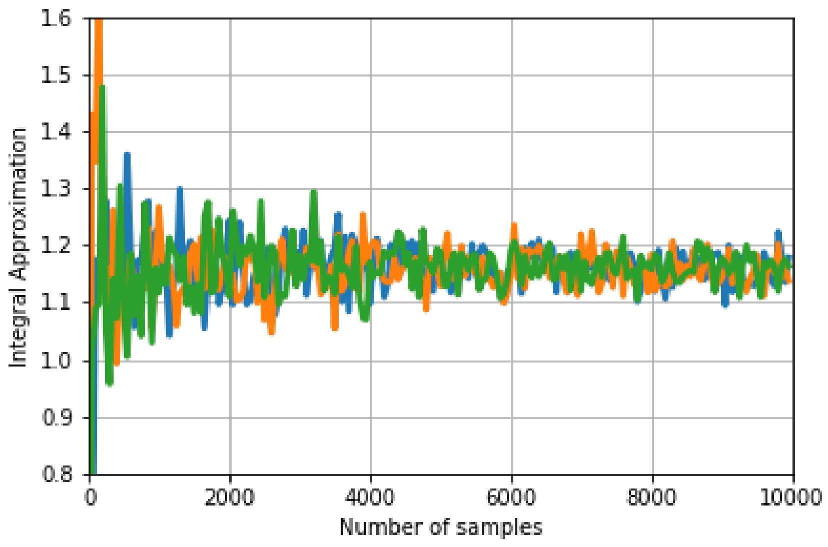


Figure 6.5.1: *Convergence of the Monte Carlo integral estimator for up to 10000 iterations and three runs.*

where $X \sim Exp(0.5)$ is distributed according to an exponential distribution. More generally, if we write

$$\int f(x)\,dx = \int \frac{f(x)}{g(x)}\,g(x)\,dx = E_g(f(X)/g(X)),$$

we get a flexible method which can be used to evaluate improper integral with less iterations. The code is as follows:

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

def my_function(x):
    return np.exp(-0.5*x) / (1+(x-1)**4)


def IS_MC_for_my_function(sample_size=100,upper_bound=10):
    # draw the samples
    sample = np.random.exponential(scale=2,size=sample_size)
    values = 1/(1+(sample-1)**4)
    return 2*np.mean(values)
```

Listing 6.4: *Standard Monte Carlo Integration*

Results for up to 10000 iterations of the Monte Carlo integration are displayed in Figure 6.5.2. Importance sampling allows to reduce significantly the variance of the integral estimator, thus we can rely on less samples and obtain a more precise result than by means of regular Monte Carlo method.

6.6 Further Readings

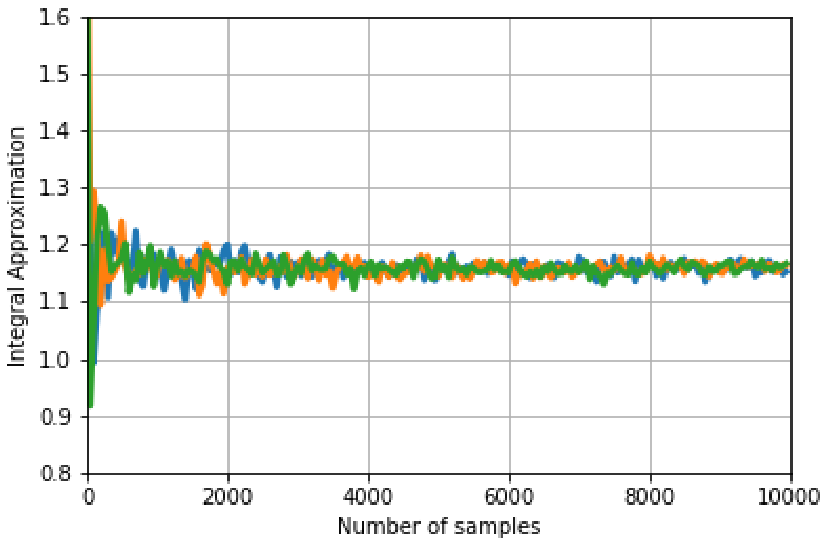
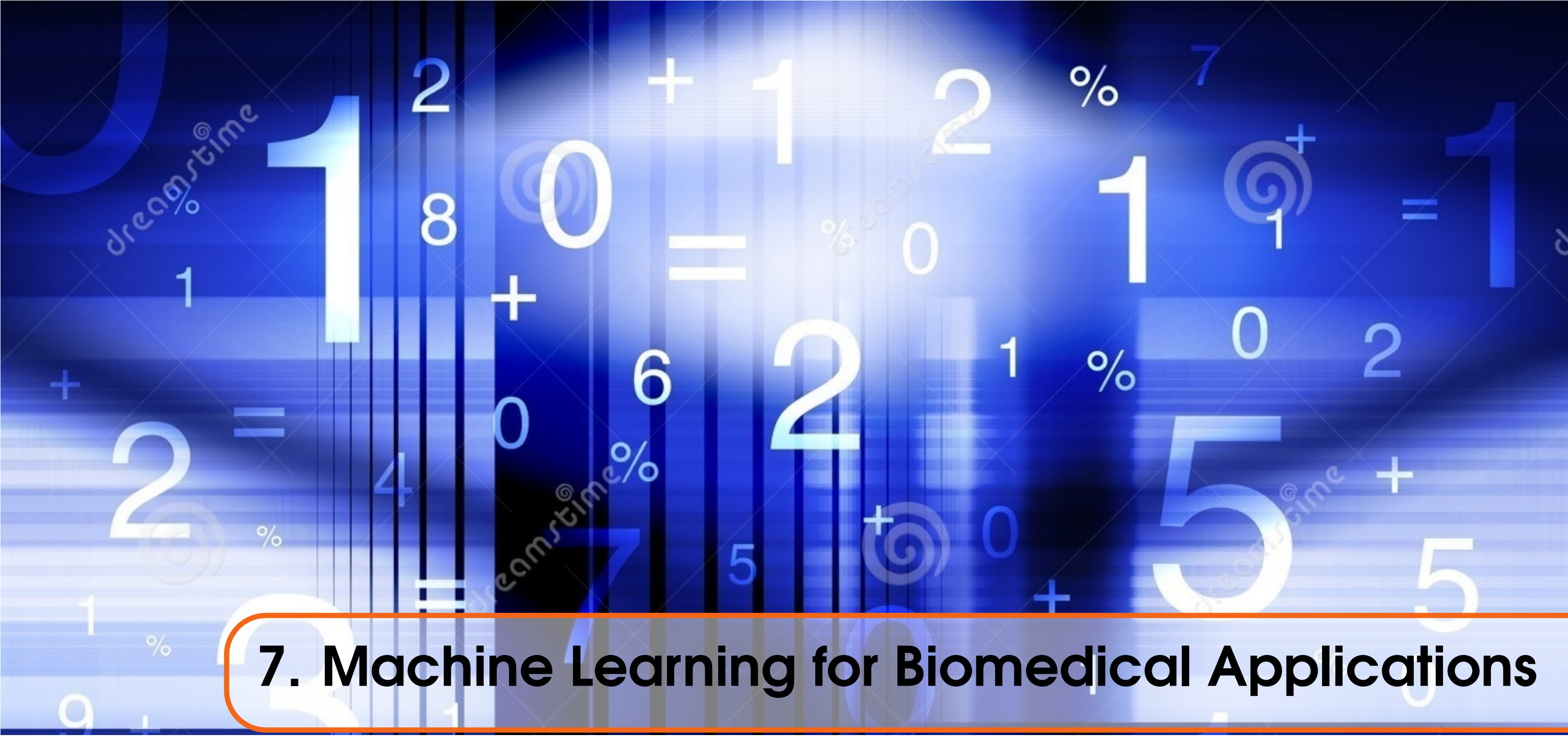


Figure 6.5.2: *Convergence of the Monte Carlo integral estimator for up to 10000 iterations and three runs.*



7. Machine Learning for Biomedical Applications

7.1 21st Century Digital Boy

This is no surprise: data has become an integral part of the engineering world. With the emergence of new tools for data processing, the field of machine learning has gained momentum in the last twenty years. This chapter will present basic concepts of machine learning and the use of the scikit-learn Python library to address machine learning problem.
This chapter has not the pretention to cover everything: a full book would not be enough to cover all the material exhaustively. The objective is rather to get a coarse understanding of the main ideas of machine learning, and to implement several ideas efficiently.

7.1.1 What is Machine Learning? (ML)

In its essence, Machine Learning (ML) is a change of paradigm in computer programming. A simple computer program (say, which sorts arrays of number from the smallest to the largest) obeys to a set of rules which are determined at the developement stage by the programer. We will refer to this as *hard-coding approach*. Hard-coding is fine as long as the problem is finite in its dimensions, and addresses tasks which may be easlity automated.
Let us consider now a more complex task, which is to identify an obstacle on a driveway in real time, or to recognize a target for a prosthetics. Obstacles may have different shapes, sizes, colors, and it is unlikely that looking at all the possible options manually and taking them into account in the programming stage will be efficient. On the other hand, a human being can solve such problems in a very easy way, based on a set of rules he learned during his lifetime.
In a sense, ML programming (or, at least, supervised ML programming), aims to mimic the human behavior by a change of paradigm, as shown in Figure 7.1.1.
More specifically, we provide to the computer a set of inputs and expected results, and we wish to obtain a given set of rules learned by means of an algorithm, and leave the decision to the computer. Since the learning itself is based on some randomness (appearing in optimization, learning stages, noise, etc.), we cannot expect a deterministic output as in the hard-coding appraoch; for example, the same algorithm trained on the same data twice can provide different answers to the same problem. Obviously, we aim to minimize this type of error, as well as to maximize the efficiency of

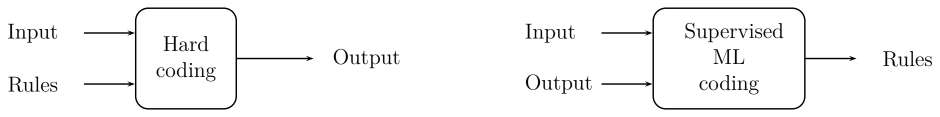


Figure 7.1.1: *Differences between hard-coding (left) and ML (right)*

the rules learned.

7.1.2 What is not ML?

Though machine learning techniques achieve impressive results in numerous tasks, it is important to understand each method's limitations and to put expectations too high: ML is still an active research topic. Consequently, we developing an ML pipeline, we should always keep in mind the following:

- ML find features, it does not interpret them: current supervised methods make wonders finding objects of interest in images, but cannot explain why they are here in the first place.
- ML algorithms infer, but cannot extrapolate: they learn what they have been taught to learn, no more, no less.
- ML has no ultimate tool: although deep learning has taken most of the scene in terms of results, they require lots of data and lots of computation power. Several concurrent approaches should be investigated when lacking data
- ML is not AI, only a subset
- Deep Learning is not ML, only a subset.

7.1.3 Examples of ML Tasks

A learning problem is usually based on a data set, and tries to predict properties of unknown additional data. This learning is based on inner properties of the data, which are either learned or extracted. These inner properties are called *features*. A supervised learning algorithm provides a rule of decision based on what he learned. An unsupervised learning algorithm provides ways to characterize the data, and has a more exploratory meaning. Reinforcement learning, a more recent branch of ML, aims to improve a learning algorithm performances based on the environment. By providing a reward function at the end of the prediction, the algorithm can learn a series of action to maximize a reward.

For supervised learning, we can distinguish several tasks:

- **Regression:** If the rule of decision is a continuous function, the problem is called a regression problem.
- **Classification:** If the rule of decision is a discrete valued function, the problem is called a classification problem.

For unsupervised learning, the objectives would be to find similarities inside unlabelled datasets (Clustering), or to determine the distribution of the data (Density Estimation), or either to extract features of interest in this dataset (Dimensionality Reduction / Compression).

7.1.4 Different Types of ML

7.2 Standard ML Pipeline

An usual pipeline in the development of ML algorithms is presented in Figure 7.2.1. We detail each stage below.

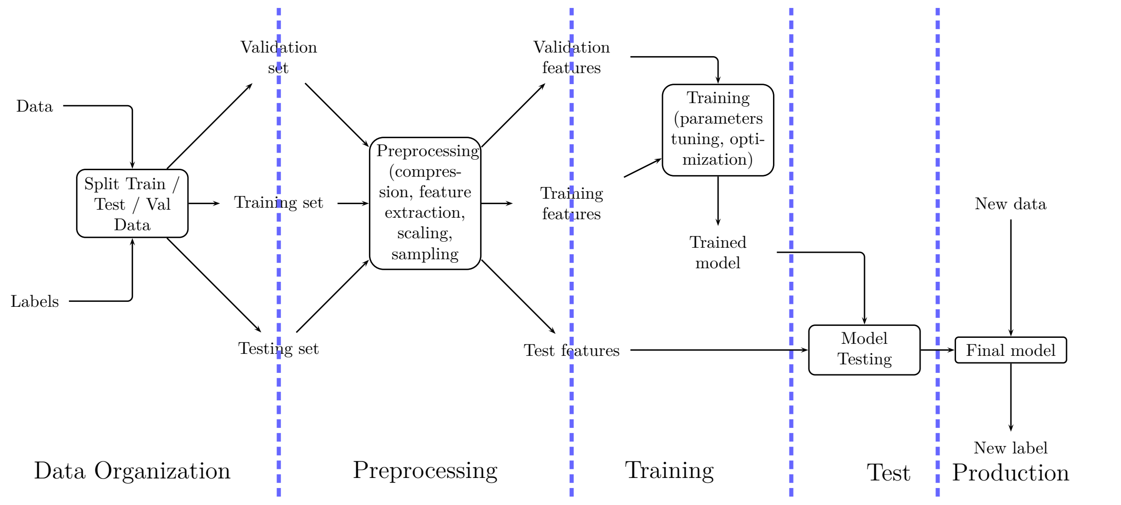


Figure 7.2.1: *A standard supervised machine learning pipeline - Deep learning merges preprocessing and training altogether.*

7.2.1 Data Organization

The preliminary steps of data organization is often neglected by praticioners who are already working on existing datasets. Yet, from all points of view, this step is crucial.

For the sake of the argument, let's say that we wish to classify CT scans provided by an hospital based on the fact that they show tumors or not. Assume that about 5% of the patients who made a CT scan actually have a tumor. In that framework, a supervised method may provide a very high precision rate; however, such classifier is useless: since most patients are healthy, a trained classifier will be inclined to classify everyone as healthy, since by doing so it does not go wrong often.

Therefore, before going any further, it is essential to investigate the dataset (e.g. using unsupervised techniques) in order to check that it is well suited for supervised learning. In particular, we should check:

- The classes imbalance: if one class is over-represented, the results obtained by the algorithm are not interpretable. Thus, all the classes should be well represented, with a sufficiently high number of labels for all classes.
- Extra-class variance: if we train an object recognition to classify images of dogs and cats, we are certain to do well, because the "objects" we wish to classify are different. Classifying between a sunscreen or an umbrella is a most challenging task. Thus, classes should be as separable as possible to maximize the precision.
- Inter-class variance: if we wish to classify between pop song and classical music pieces, but train the classical music on Bach only, more recent classical music will be wrongly defined as pop songs. Thus, a given class should exhibit some variability, in order to learn as many features as possible.

Under the assumption that we have a "good" dataset at hand, the first thing to do is to split it into three part: the *training set*, which will be used to train the model based on its labels, the *validation set*, which will be used to optimize the parameters the model depends on (optimizing on the training set might degrade the generalization capabilities of the model), and a *test set*, which will be used to validate the model on data he doesn't know to investigate if we trained a model generic enough. Usually, the split is done based on the following 70%-train, 10%-validation, 20%-test, though other choices are possible.

7.2.2 Preprocessing

The preprocessing steps aim to simplify the data at hand, or to extract from the data the most significant information (color, number of pixels,...). Moreover, most ML algorithms require the features to be on the same scale for optimal performances, so they have to be normalized. Dimensionality reduction techniques can also be applied when features are very correlated or redundant (this is often the case in images). This step yields faster algorithms running on less storage space.
The deep training paradigm includes the feature extraction inside the learning procedure: the early layers of the neural networks capture small features, whereas more complex features are captured in the later stages of the neural network.

7.2.3 Training

For machines and humans, the learning process is the same: it comes with a cost. More specifically, we need at first to establish a cost criterion under which we can evaluate the performances of the model's training. For example, for classification, *precision* as the proportion of correctly classified data is often used for classification tasks. Most of the time, the training process relies on minimizing the cost criterion chosen with the data, but this approach could also depend on several intern parameters (regularization, sparsity, learning rate, etc.). The validation set can be used to fine tune these parameters, so that the training set cannot have a bad influence on the parameter's choice.

7.2.4 Testing

As promising as it may be, a low cost value on training data is not very interesting per se. We would rather like the trained model to generalize, that is provide accurate results for data which are not included in the training set. At the end of the training, we perform an inference on the test set: if predictions on the test set are accurate, it means that the model can generalize well. Otherwise, other approaches and parameters in the training should be chosen.

7.2.5 Production

Once the model has been successfully tested, it is usually put in production for real-time use. At this point, the inner parameters of the model are frozen, and are not to be changed again. The work consists in looking at the prediction results the model provides, and if it fails to predicts several instances of the data, it has to be retrained with an updated dataset. This last stage obviously industrial, and is generally omitted while doing research in the field.

7.3 ML Using sklearn and Python

In this section, we will present several ML applications using the scikit-learn library (for approaches which are not related to neural networks), and Keras/Tensorflow for neural networks approaches.

7.3.1 The Philosophy of sklearn

Before looking at some examples of Machine Learning algorithms, it is important to understand how sklearn expects the data to be organized. In that matter, we will dig the Iris dataset:

```python
import pprint as pp
from sklearn import datasets
iris_dataset = datasets.load_iris()
pp.pprint(type(iris_dataset))
pp.pprint(iris_dataset)
```

The obtained result is

```
<class 'sklearn.utils.Bunch'>
{'DESCR': '.. _iris_dataset:\n'
          '\n'
          'Iris plants dataset\n'
          '--------------------\n'
          '\n'
          '**Data Set Characteristics:**\n'
          '\n'
          '    :Number of Instances: 150 (50 in each of three classes)\n'
          '    :Number of Attributes: 4 numeric, predictive attributes and the '
          'class\n'
          '    :Attribute Information
          .
          .
          .
          '     conceptual clustering system finds 3 classes in the data.\n'
          '   - Many, many more ...',
 'data': array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],

       .
       .
       .

       [6.2, 3.4, 5.4, 2.3],
       [5.9, 3. , 5.1, 1.8]]),
 'feature_names': ['sepal length (cm)',
                   'sepal width (cm)',
                   'petal length (cm)',
                   'petal width (cm)'],
 'filename': '/usr/local/lib/python3.6/dist-packages/sklearn/datasets/data/iris.csv',
 'target': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]),
 'target_names': array(['setosa', 'versicolor', 'virginica'], dtype='<U10')}
```

So, a dataset is stored in a Bunch class, which has the same shape than a dictionary. All classes IDs and data are stored in Numpy arrays, as well as the names of the features and classes. The field 'data' always has to be a Numpy array of size Number of samples $\times$ Number of features. It is clear that only very small datasets can be fully stored in memory. In practice, we cannot do so, thus the data needs to be prepared, for example using hdf5 files and XML files for information. This point is crucial for deep learning applications, where training must be performed on thousands of differents images, if not more.
Sklearn follows the procedure detailed in Figure 7.2.1. In this library, it is called a *pipeline*. A pipeline is the concatenation of several operations included in sklearn, that we can use either for

dimensionality reduction or training. A classifier or pipeline, once defined, is trained using the method *fit*. Test/Validation is done using the method *predict*.

7.3.2 Learning Example 1: K-Nearest Neighbors and PCA

K-Nearest Neighbors (KNN) is the easiest method for unsupervised classification - it can be summarized by the sentence "if in the k-nearest neighbors of an unknown input, there is a majority of a given class, then the unknown input should be labelled as belonging to this majority class". This paradigm, though simple, allows to perform unsupervised classification fast to get insights on the organization of the data.
On the first example, we'll work on the Iris dataset. First, we'll split the dataset between train and test, and use KNN out of the box to classify the test data.

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.model_selection import train_test_split
X, y = datasets.load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size
    =0.5, random_state=14)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
knn.score(X_test, y_test)
```

Listing 7.1: *KNN on the Iris dataset*

The classification score obtained is close to 0.98, which means that classification is very efficient. The reader can see the influence of the number of neighbors, which has to be carefully chosen, on the Jupyter notebook of the chapter.
One might wonder whether dimensionality improves the classification in that case:

```python
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA

pca = PCA()
combined_pca_knn = Pipeline(steps=[('pca',pca),('knn',knn_classifer)],verbose=
    True)
combined_pca_knn.fit(X_train,y_train)
combined_pca_knn.score(X_test,y_test)
```

Here, dimensionality reduction helps little to none; this is no surprise, since the dimensions of the features were very small from the very beginning.
For the Digits datasets (images of size 8×8), the same can be applied directly.

```python
X,y = datasets.load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3)
knn = KNeighborsClassifier(n_neighbors=21)
knn.fit(X_train,y_train)
knn.score(X_test,y_test)

pca = PCA()
combined_pca_knn = Pipeline(steps=[('pca',pca),('knn',knn)])
combined_pca_knn.fit(X_train,y_train)
combined_pca_knn.score(X_test,y_test)
```

In both cases, the score is around 0.97, and PCA little helps (indeed, images of 64 pixels is already a case of dimensionality reduced drastically...).

7.3.3 Learning Example 2: Classifying ECG pulses using a deep neural network

A neural network is a succession of linear and non-linear mappings, used to perform supervised learning. For a more detailed description of neural networks, we refer to the companion course on

deep learning from the BIOART consortium.
First, we load the data from the files

```python
data_normal = np.genfromtxt('ptbdb_normal.csv',delimiter=',')
print(data_normal.shape)
data_abnormal = np.genfromtxt('ptbdb_abnormal.csv',delimiter=',')
print(data_abnormal.shape)
```

The sizes are

```
(4046, 188)
(10506, 188)
```

Thus, we see that we dataset is not completely balanced. Though we should either perform data augmentation or split the dataset to keep approximately the same number of files for normal and abnormal data, we will let the data unchanged. We then create a matrix which will include the data, and a label ('1' if the ECG is normal, '0' if it is abnormal). The rows of the matrix are then shuffled, to guarantee that the split train/test works fine. The labels are changed to categorical data, so that the softmax decision will be applied.

```python
complete_data_set = np.zeros((data_normal.shape[0]+data_abnormal.shape[0],
    data_normal.shape[1]+1))
complete_data_set[0:4046,:-1] = data_normal
complete_data_set[0:4046,-1] = 1
complete_data_set[4046:,:-1] = data_abnormal
np.random.shuffle(complete_data_set)

X0, X_test, y0, y_test = train_test_split(complete_data_set[:,:-2],
    complete_data_set[:,-1],test_size=0.2)
X_train, X_val, y_train, y_val = train_test_split(X0,y0,test_size=0.1)
print(X_train.shape,y_train.shape,X_val.shape,y_val.shape,X_test.shape,y_test.
    shape)

# Change to categorical
y_train = keras.utils.to_categorical(y_train)
y_val = keras.utils.to_categorical(y_val)
y_test = keras.utils.to_categorical(y_test)
```

We will use the Keras library to build the neural network. Keras can be used either in a sequential manner (sequential API), either as a sequence of functions (functional API). Since the last option is in the long run more flexible, we shall program it using the functional API.

```python
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, Dropout, Flatten,
    Conv1D, MaxPooling1D
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(187,1))

def my_layer(x,pooling=False):
    # a layer instance is callable on a tensor, and returns a tensor
    conv = Conv1D(16,5,activation='relu',padding='same',input_shape=(187,1))(x)
    if pooling is True:
        pooled = MaxPooling1D(3)(conv)
        return pooled
    else:
        return conv

x = my_layer(inputs)
y = my_layer(x)
y2 = my_layer(y)
```

```
z = my_layer(y2,pooling=True)

vector = Flatten()(z)
final_vector = Dense(64,activation='relu')(vector)
predictions = Dense(2,activation='softmax')(final_vector)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.summary()
```

The method *summary* displays the architecture of the neural network, and is a good way to check whether the dimensions at each step are well respected.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 187, 1)            0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 187, 16)           96
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 187, 16)           1296
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 187, 16)           1296
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 187, 16)           1296
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 62, 16)            0
_________________________________________________________________
flatten_1 (Flatten)          (None, 992)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                63552
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 130
=================================================================
Total params: 67,666
Trainable params: 67,666
Non-trainable params: 0
_________________________________________________________________
```

The mappings depends on 67666 inner parameters, which are optimized to fit the training set. The hyperparameters (parameters which do not define the functions, such as the learning rate) are set using the validation dataset).

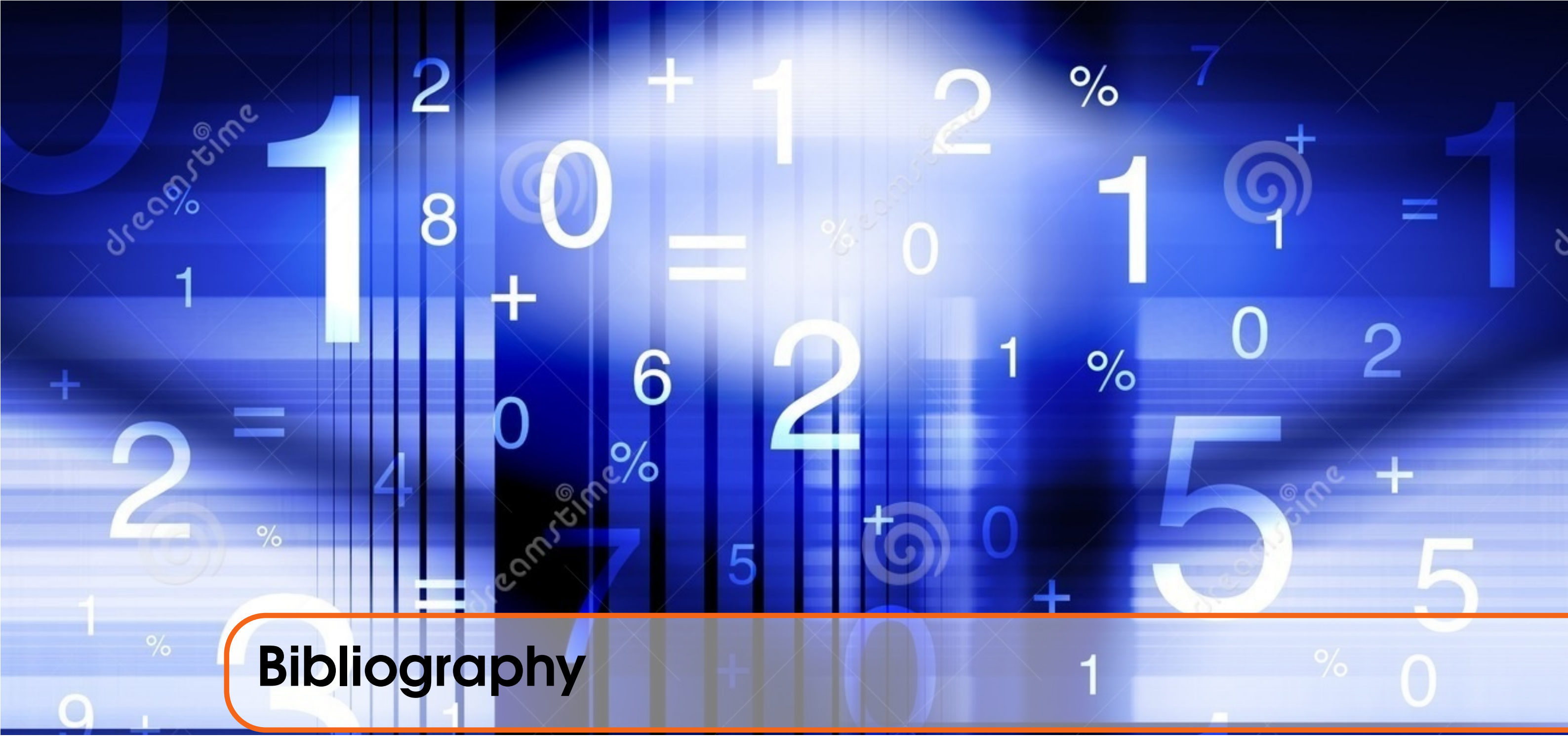
```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])


X_train = np.expand_dims(X_train,axis=2)
X_val = np.expand_dims(X_val,axis=2)
X_test = np.expand_dims(X_test,axis=2)

history=model.fit(X_train,y_train,batch_size=64,epochs=10,validation_data=(
    X_val,y_val))
```

After the training, Keras returns the accuracy and loss obtained both on the training set and the validation set.

```
Epoch 10/10
10476/10476 [==============================] - 3s 328us/step -
 loss: 1.2317e-04 - acc: 1.0000 -
val_loss: 0.0533 - val_acc: 0.9914
```

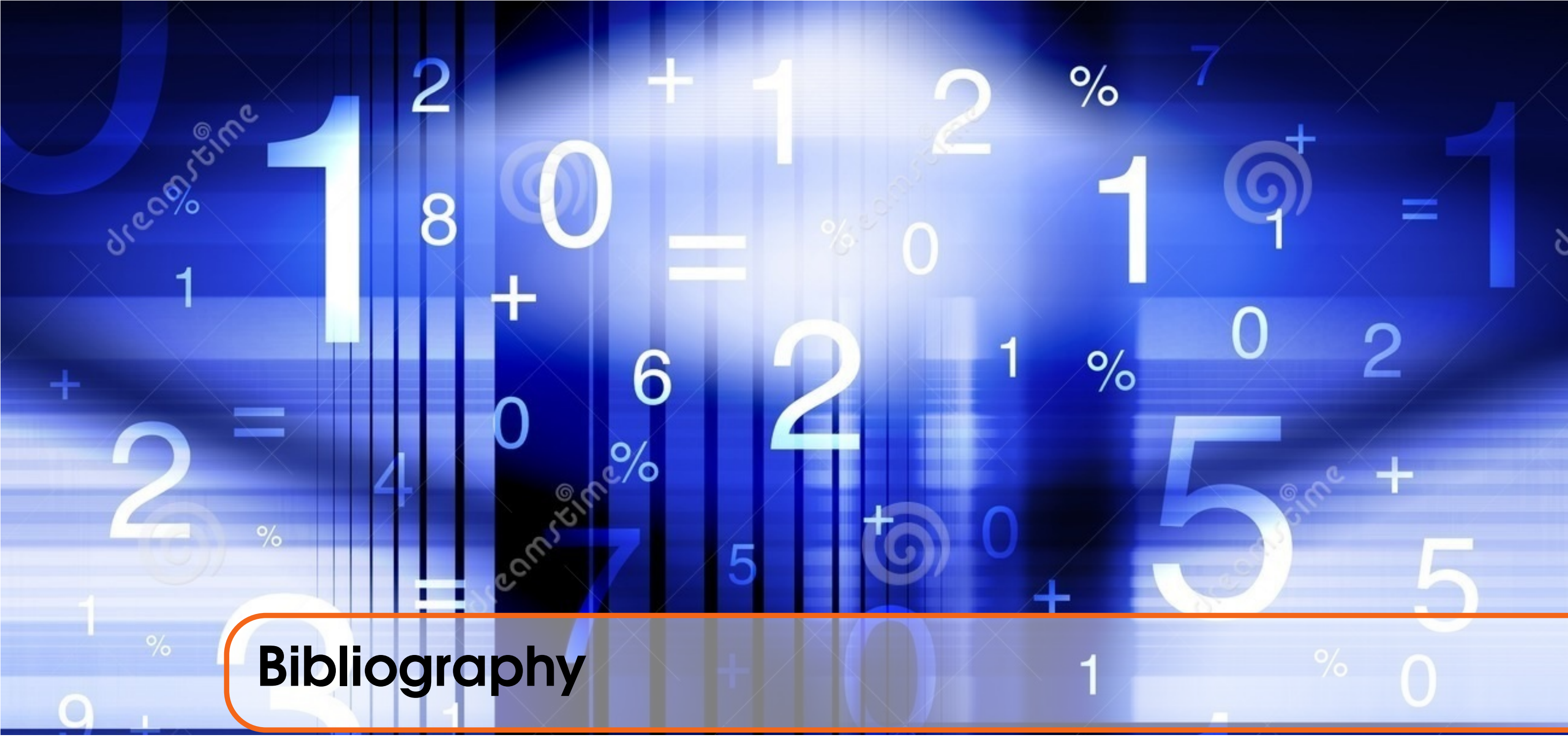
On the test set, the accuracy goes up to 98%, thus the model is well suited to the data.

```
_, accuracy = model.evaluate(X_test, y_test)
print(accuracy)
```



Bibliography

Books



Bibliography

[1] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2:183–202, 2009.

[2] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. *Distributed Optimization and Statistical Learning via the Alterniating Direction Method of Multipliers*. Foundations and Trends in Machine Learning, 2010.

[3] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[4] A. Forrester, A. Sobester, and A. Keane. *Engineering Design via Surrogate Modelling: a Practical Guide.* Wiley, 2008.

[5] N. Parikh and S. Boyd. Proximal Algorithms. *Foundations and Trends in Optimization*, 1(3):127–239, 2014.