



# Programming Basics and AI

*with Matlab and Python*

Lectures on YouTube:

<https://www.youtube.com/@mathtalent>

Seongjai Kim

Department of Mathematics and Statistics

Mississippi State University

Mississippi State, MS 39762 USA

Email: [skim@math.msstate.edu](mailto:skim@math.msstate.edu)

Updated: December 2, 2023

***Coding techniques may improve the computer program by tens of percents,  
while an effective algorithmic design can improve it by tens or hundreds of times***

***In computational literacy, the coding ability is to the tip of the iceberg  
as the ability of algorithmic design is to its remainder***

*Seongjai Kim*, Professor of Mathematics, Department of Mathematics and Statistics, Mississippi State University, Mississippi State, MS 39762 USA. Email: [skim@math.msstate.edu](mailto:skim@math.msstate.edu).

# Prologue

This lecture note provides an overview of scientific computing, i.e., of modern information engineering tasks to be tackled by powerful computer simulations. The emphasis throughout is on **the understanding of modern algorithmic designs** and their **efficient implementation**.

As well known as in the society of computational methods, **computer programming** is the process of constructing an executable computer program in order to accomplish a specific computational task. Programming in practice cannot be realized without incorporating computational languages. However, it is not a simple process of experiencing computational languages; it involves concerns such as

- mathematical analysis,
- generating computational algorithms,
- profiling algorithms' accuracy and cost, and
- the implementation of algorithms in selected programming languages (commonly referred to as **coding**).

The source code of a program can be written in one or more programming languages.

The manuscript is conceived as an introduction to the thriving field of **information engineering**, particularly for **early-year college students** who are interested in mathematics, engineering, and other sciences, without an already strong background in computational methods. It will also be suitable for **talented high school students**. All examples to be treated in this manuscript are implemented in Matlab and Python, and occasionally in Maple.

Currently, the lecture note is growing.

December 2, 2023

## Level of Lectures

- The target audience is **undergraduate students**.
- However, **talented high school students** would be able to follow the lectures.
- **Persons with no programming experience** will understand most of lectures.

## Goals of Lectures

Let the students understand

- Mathematical Basics: Calculus & Linear Algebra
- Programming, with Matlab and Python
- Artificial Intelligence (AI)
- Basics of Machine Learning (ML)
- An ML Software: **Scikit-Learn**

## Programming: the Programmers' Work

- Understand and analyze the problem
- Convert mathematical terms to computer programs
- Verify the code
- Get insights

**You will learn programming!**

# Contents

<b>Title</b>	<b>ii</b>
<b>Prologue</b>	<b>iv</b>
<b>Table of Contents</b>	<b>ix</b>
<b>1 Programming Basics</b>	<b>1</b>
1.1. What is Programming or Coding? . . . . .	2
1.1.1. Programming: Some Examples . . . . .	2
1.1.2. Functions: Generalization and Reusability . . . . .	5
1.1.3. Becoming a Good Programmer . . . . .	8
1.2. Matlab: A Powerful Computer Language . . . . .	12
1.2.1. Introduction to Matlab/Octave . . . . .	12
1.2.2. Repetition: Iteration Loops . . . . .	19
1.2.3. Anonymous Function . . . . .	25
1.2.4. Open Source Alternatives to Matlab . . . . .	25
Exercises for Chapter 1 . . . . .	26
<b>2 Programming Examples</b>	<b>29</b>
2.1. Area Estimation of the Region Defined by a Closed Curve . . . . .	30
2.2. Visualization of Complex-Valued Solutions . . . . .	35
2.3. Discrete Fourier Transform . . . . .	38
2.3.1. Discrete Fourier Transform . . . . .	39
2.3.2. Short-Time Fourier Transform . . . . .	44
2.4. Computational Algorithms and Their Convergence . . . . .	47
2.4.1. Computational Algorithms . . . . .	47
2.4.2. Big $\mathcal{O}$ and little $o$ notation . . . . .	50
2.5. Inverse Functions: Exponentials and Logarithms . . . . .	53
2.5.1. Inverse functions . . . . .	54
2.5.2. Logarithmic Functions . . . . .	59
Exercises for Chapter 2 . . . . .	62
<b>3 Programming with Calculus</b>	<b>65</b>
3.1. Differentiation . . . . .	66
3.1.1. The Slope of the Tangent Line . . . . .	66

3.1.2. Derivative and Differentiation Rules . . . . .	71
3.2. Basis Functions and Taylor Series . . . . .	76
3.2.1. Change of Variables & Basis Functions . . . . .	76
3.2.2. Power Series and the Ratio Test . . . . .	78
3.2.3. Taylor Series Expansion . . . . .	81
3.3. Polynomial Interpolation . . . . .	86
3.3.1. Lagrange Form of Interpolating Polynomials . . . . .	87
3.3.2. Polynomial Interpolation Error Theorem . . . . .	90
3.4. Numerical Differentiation: Finite Difference Formulas . . . . .	93
3.5. Newton's Method for the Solution of Nonlinear Equations . . . . .	99
3.6. Zeros of Polynomials . . . . .	104
3.6.1. Horner's Method . . . . .	105
Exercises for Chapter 3 . . . . .	109
<b>4 Linear Algebra Basics</b>	<b>113</b>
4.1. Solutions of Linear Systems . . . . .	114
4.1.1. Solving a linear system . . . . .	115
4.1.2. Matrix equation $Ax = b$ . . . . .	117
4.2. Row Reduction and the General Solution of Linear Systems . . . . .	119
4.2.1. Echelon Forms and the Row Reduction Algorithm . . . . .	120
4.2.2. The General Solution of Linear Systems . . . . .	123
4.3. Linear Independence and Span of Vectors . . . . .	126
4.4. Invertible Matrices . . . . .	129
Exercises for Chapter 4 . . . . .	133
<b>5 Programming with Linear Algebra</b>	<b>135</b>
5.1. Determinants . . . . .	136
5.2. Eigenvalues and Eigenvectors . . . . .	141
5.2.1. Characteristic Equation . . . . .	142
5.2.2. Matrix Similarity and The Diagonalization Theorem . . . . .	144
5.3. Dot Product, Length, and Orthogonality . . . . .	148
5.4. Vector Norms, Matrix Norms, and Condition Numbers . . . . .	151
5.5. Power Method and Inverse Power Method for Eigenvalues . . . . .	155
5.5.1. The Power Method . . . . .	155
5.5.2. The Inverse Power Method . . . . .	159
Exercises for Chapter 5 . . . . .	162
<b>6 Multivariable Calculus</b>	<b>163</b>
6.1. Multi-Variable Functions and Their Partial Derivatives . . . . .	164
6.1.1. Functions of Several Variables . . . . .	164
6.1.2. First-order Partial Derivatives . . . . .	165
6.2. Directional Derivatives and the Gradient Vector . . . . .	168

6.3. Optimization: Method of Lagrange Multipliers . . . . .	173
6.3.1. Optimization Problems with Equality Constraints . . . . .	174
6.3.2. Optimization Problems with Inequality Constraints . . . . .	177
6.4. The Gradient Descent Method . . . . .	181
6.4.1. Introduction to the Gradient Descent Method . . . . .	181
6.4.2. The Gradient Descent Method in Multi-Dimensions . . . . .	185
6.4.3. The Gradient Descent Method for Positive Definite Linear Systems . . . . .	188
Exercises for Chapter 6 . . . . .	192
<b>7 Least-Squares and Regression Analysis</b>	<b>193</b>
7.1. The Least-Squares Problem . . . . .	194
7.2. Regression Analysis . . . . .	198
7.2.1. Regression line . . . . .	198
7.2.2. Least-squares fitting of other curves . . . . .	201
7.2.3. Nonlinear regression: Linearization . . . . .	202
7.3. Scene Analysis with Noisy Data: Weighted Least-Squares and RANSAC . . . . .	204
7.3.1. Weighted Least-Squares . . . . .	204
7.3.2. RANDOM SAmple Consensus (RANSAC) . . . . .	206
Exercises for Chapter 7 . . . . .	209
<b>8 Python Basics</b>	<b>211</b>
8.1. Why Python? . . . . .	212
8.2. Python Essentials in 30 Minutes . . . . .	215
8.3. Zeros of a Polynomial in Python . . . . .	221
8.4. Python Classes . . . . .	225
Exercises for Chapter 8 . . . . .	232
<b>9 Vector Spaces and Orthogonality</b>	<b>233</b>
9.1. Subspaces of $\mathbb{R}^n$ . . . . .	234
9.2. Orthogonal Sets and Orthogonal Matrix . . . . .	238
9.3. Orthogonal Projections . . . . .	243
9.4. The Gram-Schmidt Process and QR Factorization . . . . .	248
9.5. QR Iteration for Finding Eigenvalues . . . . .	253
Exercises for Chapter 9 . . . . .	257
<b>10 Introduction to Machine Learning</b>	<b>259</b>
10.1. What is Machine Learning? . . . . .	260
10.2. Binary Classifiers . . . . .	265
10.2.1. The Perceptron Algorithm . . . . .	267
10.2.2. Adaline: ADAptive LInear NEuron . . . . .	271
10.3. Popular Machine Learning Classifiers . . . . .	275
10.3.1. Logistic Regression . . . . .	276
10.3.2. Support Vector Machine . . . . .	279

10.3.3. $k$ -Nearest Neighbors . . . . .	281
10.4. Neural Networks . . . . .	283
10.4.1. A Simple Network to Classify Hand-written Digits: MNIST Dataset . . . . .	284
10.4.2. Implementation for MNIST Digits Dataset [9] . . . . .	288
10.5. Scikit-Learn: A Python Machine Learning Library . . . . .	292
A Machine Learning Modelcode . . . . .	296
Exercises for Chapter 10 . . . . .	301
<b>11 Principal Component Analysis</b>	<b>303</b>
11.1. Principal Component Analysis . . . . .	304
11.1.1. The covariance matrix . . . . .	305
11.1.2. Computation of principal components . . . . .	309
11.1.3. Dimensionality reduction: Data compression . . . . .	311
11.2. Singular Value Decomposition . . . . .	315
11.2.1. Algebraic interpretation of the SVD . . . . .	317
11.2.2. Computation of the SVD . . . . .	319
11.3. Applications of the SVD to LS Problems . . . . .	321
Exercises for Chapter 11 . . . . .	329
<b>A Appendices</b>	<b>333</b>
A.1. Optimization: Primal and Dual Problems . . . . .	334
A.1.1. The Lagrangian . . . . .	334
A.1.2. Lagrange Dual Problem . . . . .	336
A.2. Weak Duality, Strong Duality, and Complementary Slackness . . . . .	338
A.2.1. Weak Duality . . . . .	339
A.2.2. Strong Duality . . . . .	340
A.2.3. Complementary Slackness . . . . .	341
A.3. Geometric Interpretation of Duality . . . . .	342
A.4. Rank-One Matrices and Structure Tensors . . . . .	349
A.5. Boundary-Effects in Convolution Functions in Matlab and Python SciPy . . . . .	353
A.6. From Python, Call C, C++, and Fortran . . . . .	357
<b>P Projects</b>	<b>365</b>
P.1. Project: Canny Edge Detection Algorithm for Color Images . . . . .	366
P.1.1. Noise Reduction: Image Blur . . . . .	370
P.1.2. Gradient Calculation: Sobel Gradient . . . . .	372
P.1.3. Edge Thinning: Non-maximum Suppression . . . . .	375
P.1.4. Double Threshold . . . . .	377
P.1.5. Edge Tracking by Hysteresis . . . . .	378
P.2. Project: Text Extraction from Images, PDF Files, and Speech Data . . . . .	380
<b>Bibliography</b>	<b>383</b>



*Contents*

ix

**Index**

**385**



# CHAPTER 1

## Programming Basics

In this chapter, you will learn

- what programming is
- what coding is
- what programming languages are
- how to convert mathematical terms to computer programs
- how to control repetitions

### Contents of Chapter 1

1.1. What is Programming or Coding? . . . . .	2
1.2. Matlab: A Powerful Computer Language . . . . .	12
Exercises for Chapter 1 . . . . .	26

## 1.1. What is Programming or Coding?

**Definition 1.1.** **Computer programming** is the process of building an executable computer program in order to accomplish a specific computational task.

- **Programming involves various concerns** such as
  - mathematical/physical analysis,
  - generating computational algorithms,
  - profiling algorithms' accuracy and cost, and
  - the implementation of algorithms in a chosen programming language (commonly referred to as **coding**).
- **The purpose of programming** is to find a sequence of instructions that will automate the performance of a task for solving a given problem.
- Thus, **the process of programming** often requires expertise in several different subjects, including
  - **knowledge of the application domain,**
  - **specialized algorithms,** and
  - **formal logic.**

### 1.1.1. Programming: Some Examples

**Example 1.2.** Assume that we need to find the sum of integers from 2 to 5:

$$2 + 3 + 4 + 5.$$

**Solution.** You may start with 2; add 3, add 4, and finally add 5; the answer is 14. This simple procedure is the result of programming in your brain.

**Programming is thinking.** □

**Example 1.3.** Let's try to get  $\sqrt{5}$ . Your calculator must have a function key  $\sqrt{\quad}$ . When you input  $\sqrt{5}$  and push  , your calculator displays the answer on the spot. How can the calculator get the answer?

**Solution.** Calculators or computers cannot keep a table to look the answer up. They compute the answer on the spot as follows.

Let  $Q = 5$ .

1. **initialization:**  $p$

2. **for**  $i = 1, 2, \dots, \text{itmax}$

$$p \leftarrow (p + Q/p)/2;$$

3. **end for**

squareroot\_Q.m

```

1 Q=5;
2
3 p = 1;
4 for i=1:8
5     p = (p+Q/p)/2;
6     fprintf("%3d  %.20f\n",i,p)
7 end

```

Output

```

1 1 3.00000000000000000000
2 2 2.333333333333333348136
3 3 2.23809523809523813753
4 4 2.23606889564336341891
5 5 2.23606797749997809888
6 6 2.23606797749978980505
7 7 2.23606797749978980505
8 8 2.23606797749978980505

```

The algorithm has converged to 20 decimal digit accuracy, just in 6 iterations.  $\square$

**Note:** The above example shows what really happens in your calculators and computers (sqrt). In general,  $\sqrt{Q}$  can be found in a few iterations of simple mathematical operations.

**Remark 1.4.** Note that

$$p \leftarrow (p + Q/p)/2 = p - \frac{p^2 - Q}{2p}, \quad (1.1)$$

which can be interpreted as follows.

1. Square the current iterate  $p$ ;
2. Measure the difference from  $Q$ ;
3. Scale the difference by twice the current iterate ( $2p$ )
4. Update  $p$  by subtracting the scaled difference (correction term)

**Question. How could we know:  
a good scaling factor in the correction term is  $2p$ ?**

- The answer comes from a **mathematical analysis**.
  - In general, programming consists of
    - (a) **mathematical analysis**,
    - (b) **algorithmic design**,
    - (c) **implementation to the computer (coding)**, and
    - (d) **verification for accuracy and efficiency**.
  - Once you have done a **mathematical analysis** and performed **algorithmic design**, the next step is to **implement** the algorithm for a code (**coding**).
  - In implementation, you can use one (or more) of computer languages such as Matlab, Python, C, C++, and Java.
- 
- Through the course, you will learn **programming techniques**, using **simple languages** such as Matlab and Python.
  - **Why simple languages?**

To focus on **mathematical logic** and **algorithmic design**

**Remark 1.5. (Coding vs. Programming)**

At this moment, you may ask questions like:

- *What is coding?*
- *How is it related to programming?*

Though the terms are often used interchangeably, coding and programming are two different things.

Particularly, in **Software Development Industries**,

- **Coding** refers to **writing codes** for applications, but **programming** is a **much broader term**.
  - **Coding** is basically the process of creating codes from one language to another,
  - while **programming** is to find solutions of problems and determine how they should be solved.
- **Programmers** generally deal with the big picture in applications.

**So you will learn programming!**

## 1.1.2. Functions: Generalization and Reusability

A good programmer must implement codes that are **effective, easy to modify, and reusable**. In order to understand **reusability**, let us consider the following simple programming.

**Example 1.6.** Find the sum of the square of consecutive integers from 1 to 10.

**Solution.**

- This example asks to evaluate the quantity:

$$1^2 + 2^2 + \cdots + 10^2 = \sum_{i=1}^{10} i^2. \quad (1.2)$$

- A Matlab code can be written as

```

1 sqsum = 0;
2 for i=1:10
3     sqsum = sqsum + i^2;
4 end

```

- When the code is executed, the variable `sqsum` saves 385.
- The code is a simple form of **repetition**, one of most common building blocks in programming.

### Remark 1.7. Reusability.

- The above Matlab program produces the square sum of integers from 1 to 10, which may not be useful for other occasions.
- In order to make programs **reusable** for various situations, operations or a group of operations must be
  - **implemented with variable inputs**, and
  - **saved as a form of function**.

**Example 1.8. (Generalization of Example 1.6).** Find the sum of the square of consecutive integers **from  $m$  to  $n$** .

**Solution.** As a generalization of the above Matlab code, it can be implemented and saved in `squaresum.m` as follows.

```

_____ squaresum.m _____
1 function sqsum = squaresum(m,n)
2 %function sqsum = squaresum(m,n)
3 % Evaluates the square sum of consecutive integers: m to n.
4 % input: m,n
5 % output: sqsum
6
7 sqsum = 0;
8 for i=m:n
9     sqsum = sqsum + i^2;
10 end

```

- In Matlab, each of saved function files is called an **M-file**, of which **the first line** specifies



- the function name (squaresum),
  - input variables (m,n),
  - outputs (sqsum).
- **Lines 2–5** of `squaresum.m`, beginning with the percent sign (%), are for a convenient user interface. A built-in function help can be utilized whenever we want to see what the programmer has commented for the function. For example,

```
help  
1 >> help squaresum  
2 function sqsum = squaresum(m,n)  
3     Evaluates the square sum of consecutive integers: m to n.  
4     input:  m,n  
5     output: sqsum
```

- **The last four lines** of `squaresum.m` include the required operations for the given task.

- 
- **On the command window**, the function is called for various  $m$  and  $n$ . For example,

```
1 >> squaresum(1,10)  
2 ans = 385
```

### 1.1.3. Becoming a Good Programmer

#### **In-Reality** 1.9. Aspects of Programming

As aforementioned, **computer programming** (or **programming**) is the process of building an executable computer program for accomplishing a specific computational task. **A task may consist of numerous sub-tasks each of which can be implemented as a function; some functions may be used more than once or repeatedly in a program.** The reader should consider following aspects of programming, before coding.

- **Task modularization:** The given computational task can be partitioned into **several small sub-tasks (modules)**, each of which is manageable conveniently and effectively in both mathematical analysis and computer implementation. The major goal of task modularization is to build a **backbone of programming**.
- **Development of algorithms:** For each module, computational algorithms must be **developed and saved in functions**.
- **Choice of computer languages:** One can choose one of computer languages in which all the sub-tasks are implemented. However, it is occasionally the case that sub-tasks are implemented in more than one computer language, **in order to maximize the performance of the resulting program and/or to minimize human efforts**.
- **Debugging:** Once all the modules are implemented and linked for the given computational task, the code must **be verified for correctness and effectiveness**. Such a process of finding and resolving defects or issues within a computer program is called **debugging**.

**Note:** It is occasionally the case that **verification and debugging** take much longer time than implementation itself.

**Remark 1.10. Tips for Programming:**

- **Add functions one-by-one:** Building a program is not a simple problem but a difficult project, particularly when the program should be constructed from scratch. A good strategy for an **effective programming** is:

(a) Add functions one-by-one.

(b) Check if the program is correct, **each time adding a function.**

That is, the programmer should keep the program in a working condition for the whole period of time of implementation.

- **Use/modification of functions:** One can build a new program pretty effectively by trying to modify and use old functions used for the same or similar projects. When it is the case,

*you may have to start the work **by copying old functions to newly-named functions** to modify, rather than adding/replacing lines of the original functions.*

Such a strategy will make the programmer debug much more easily and **keep the program in a working condition all the time.**

**Note:** For a successful programming, the programmer may consider the following, before he/she starts implementation (coding).

- **Understanding the problem:** inputs, operations, & outputs
- **Required algorithms:** reusable or new
- **Required mathematical methods/derivation**
- **Program structure:** how to place operations/functions
- **Verification:** How to verify the code to make sure correctness

**Example 1.11.** Let us write a program for sorting an array of numbers from smallest to largest.

**Solution.** We should consider the following before coding.

- **The goal:** A sorting algorithm.
- **Method:** Comparison of component pairs for the smaller to move up.
- **Verification:** How can I verify the program work correctly?  
Let's use e.g., a randomly-generated array of numbers.
- **Parameters:** Overall, what could be input/output parameters?

All being considered, a program is coded as follows.

```
----- mysort.m -----
1  function S = mysort(R)
2  %function S = mysort(R)
3  %   which sorts an array from smallest to largest
4
5  %% initial setting
6  S = R;
7
8  %% get the length
9  n = length(R);
10
11 %% begin sorting
12 for j=n:-1:2   %index for the largest among remained
13     for i=1:j-1
14         if S(i) > S(i+1)
15             tmp = S(i);
16             S(i) = S(i+1);
17             S(i+1) = tmp;
18         end
19     end
20 end
```

```
SortArray.m
1 % User parameter
2 n=10;
3
4 % An array of random numbers
5 % (1,n) vector of integer random values <= 100
6 R = randi(100,1,n)
7
8 % Call "mysort"
9 S = mysort(R)
```

```
Output
1 >> SortArray
2 R =
3     33     88     75     17     91     94     79     36     2     72
4 S =
5     2     17     33     36     72     75     79     88     91     94
```

**Note:** You may have to run “SortArray.m” a few times, to make sure that “mysort” works correctly.

### Summary 1.12. Programming vs. Coding

- **Programming** consists of *analysis, design, coding, & verification*. It requires **creative thinking and reasoning**, on top of coding.
- It would better begin with a **simple computer language**.

## 1.2. Matlab: A Powerful Computer Language

**Matlab** (matrix laboratory) is a multi-paradigm numerical computing environment and a proprietary programming language developed by MathWorks.

- **Flexibility:** Matlab allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran, and Python; it is particularly good at matrix manipulations.
- **Computer Algebra:** Although Matlab is intended primarily for numerical computing, an optional toolbox uses the MuPAD symbolic engine, allowing access to symbolic computing abilities.
- **Most Convenient Computer Language:** Overall, Matlab is *about the easiest computer language* to learn and to use as well.



**Remark 1.13.** For each of programming languages, there are **four essential components** to learn.

1. **Looping** – repetition
2. **Conditional statements** – dealing with cases
3. **Input/Output** – using data and saving/visualizing results
4. **Functions** – reusability and programming efficiency (§1.1.2)

### 1.2.1. Introduction to Matlab/Octave

#### **Vectors and Matrices**

The most basic thing you will need to do is to enter vectors and matrices. You would enter commands to Matlab at a prompt that looks like `>>`.

- Rows are separated by semicolons (;) or  .
- Entries in a row are separated by commas (,) or  .

```
                                Vectors and Matrices
1  >> v = [1; 2; 3]           % column vector
2  v =
3      1
4      2
5      3
6  >> w = [5, 6, 7, 8]      % row vector
7  w =
8      5      6      7      8
9  >> A = [2 1; 1 2]        % matrix
10 A =
11     2     1
12     1     2
13 >> B = [2, 1; 1, 2]
14 B =
15     2     1
16     1     2
```

- The symbols (,) and (;) can be used to combine more than one command in the same line.
- If we use semicolon (;), Matlab sets the variable but does not print the output.

```
1  >> p = [2; -3; 1], q = [2; 0; -3];
2  p =
3      2
4     -3
5      1
6  >> p+q
7  ans =
8      4
9     -3
10     -2
11 >> d = dot(p,q);
```

where dot computes the **dot product** of two vectors.

- Instead of entering a matrix at once, we can build it up from either its rows or its columns.

```
1 >> c1=[1; 2]; c2=[3; 4];
2 >> M=[c1,c2]
3 M =
4     1     3
5     2     4
6 >> c3=[5; 6];
7 >> M=[M,c3]
8 M =
9     1     3     5
10    2     4     6
11 >> c4=c1; r3=[2 -1 5 0];
12 >> N=[M, c4; r3]
13 N =
14     1     3     5     1
15     2     4     6     2
16     2    -1     5     0
```



## Operations with Vectors and Matrices

- Matlab uses the symbol (\*) for both scalar multiplication and **matrix-vector multiplication**.
- In Matlab, to retrieve the  $(i, j)$ -th entry of a matrix M, type  $M(i, j)$ .
- To retrieve more than one element at a time, give a list of columns and rows that you want.
  - For example,  $2:4$  is the same as  $[2\ 3\ 4]$ .
  - A **colon (:)** by itself means **all**. Thus,  $M(i, :)$  extracts the  $i$ -th row of M. Similarly,  $M(:, j)$  extracts the  $j$ -th column of M.

```
1 >> M=[1 2 3 4; 5 6 7 8; 9 10 11 12], v=[1;-2;2;1];
2 M =
3     1     2     3     4
4     5     6     7     8
5     9    10    11    12
6 >> M(2,3)
7 ans =
8     7
9 >> M(3,[2 4])
10 ans =
11     10    12
12 >> M(:,2)
13 ans =
14     2
15     6
16    10
17 >> 3*v
18 ans =
19     3
20    -6
21     6
22     3
23 >> M*v
24 ans =
25     7
26    15
27    23
```

- To multiply two matrices in Matlab, use the symbol (\*).
- The  $n \times n$  identity matrix is formed with the command `eye(n)`.
- You can ask Matlab for its reasoning using the command `why`. Unfortunately, Matlab usually takes attitude and gives a random response.

```
1 >> A=[1 2; 3 4], B=[4 5; 6 7],
2 A =
3     1     2
4     3     4
5 B =
6     4     5
7     6     7
8 >> A*B
9 ans =
10    16    19
11    36    43
12 >> I=eye(3)
13 I =
14     1     0     0
15     0     1     0
16     0     0     1
17 >> C=[2 4 6; 1 3 5; 0 1 1];
18 >> C_inv = inv(C)
19 C_inv =
20    1.0000   -1.0000   -1.0000
21    0.5000   -1.0000    2.0000
22   -0.5000    1.0000   -1.0000
23 >> C_inv2=C\I
24 C_inv2 =
25    1.0000   -1.0000   -1.0000
26    0.5000   -1.0000    2.0000
27   -0.5000    1.0000   -1.0000
28 >> C_inv*C
29 ans =
30     1     0     0
31     0     1     0
32     0     0     1
```

## Graphics with Matlab

In Matlab, the most popular graphic command is `plot`, which creates a 2D line plot of the data in  $Y$  versus the corresponding values in  $X$ . A general syntax for the command is

```
plot(X1,Y1,LineStyle1,...,Xn,Yn,LineStylen)
```

```
fig_plot.m  
1 close all  
2  
3 %% a curve  
4 X1=linspace(0,2*pi,11); % n=11  
5 Y1=cos(X1);  
6  
7 %% another curve  
8 X2=linspace(0,2*pi,51);  
9 Y2=sin(X2);  
10  
11 %% plot together  
12 plot(X1,Y1,'-or','linewidth',2, X2,Y2,'-b','linewidth',2)  
13 legend({'y=cos(x)', 'y=sin(x)'})  
14 axis tight  
15 print -dpng 'fig_cos_sin.png'
```

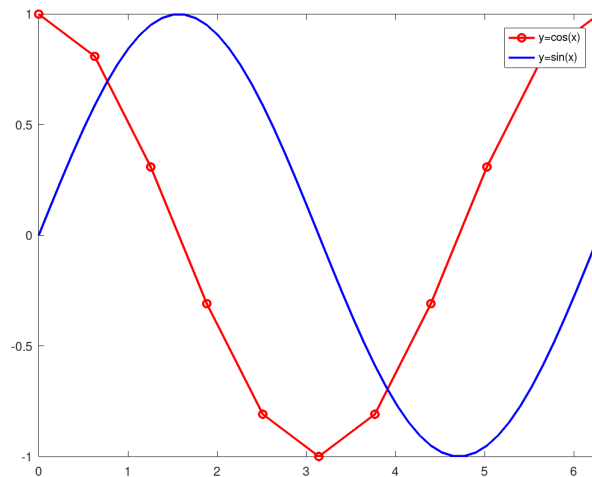


Figure 1.1: plot of  $y = \cos x$  and  $y = \sin x$ .

Above `fig_plot.m` is a typical M-file for figuring with `plot`.

- Line 1: It closes all figures currently open.
- Lines 3, 7, and 11 (comments): When the percent sign (%) appears, the rest of the line will be ignored by Matlab.
- Lines 4 and 8: The command `linspace(x1,x2,n)` returns a row vector of  $n$  evenly spaced points between  $x_1$  and  $x_2$ .
- Line 12: Its result is a figure shown in Figure 1.1.
- Line 15: it saves the figure into a png format, named `fig_cos_sin.png`. The first function ( $y = \cos x$ ) is plotted with 11 points so that its curve shows the local linearity, while the graph of  $y = \sin x$  looks smooth with 51 points.

- For contour plots, you may use `contour`.
- For figuring 3D objects, you may try `surf` and `mesh`.
- For function plots, you can use `fplot`, `fsurf`, and `fmesh`.

**Remark 1.14. (help and doc).**

Matlab is powerful and well-documented as well. To see what a built-in function do or how you can use it, type

```
help <name>   or   doc <name>
```

The command `doc` opens the Help browser. If the Help browser is already open, but not visible, then `doc` brings it to the foreground and opens a new tab. Try `doc surf`, followed by `doc contour`.

## 1.2.2. Repetition: Iteration Loops

**Recall: (Remark 1.13 on p.12)** For each of programming languages, there are **four essential components** to learn.

1. **Looping** – repetition
2. **Conditional statements** – dealing with cases
3. **Input/Output** – using data and saving/visualizing results
4. **Functions** – reusability and programming efficiency (§1.1.2)

### Note: Repetition

- In scientific computing, one of most frequently occurring events is **repetition**.
- Each repetition of the process is also called an **iteration**.
- It is the act of repeating a process, to generate a (possibly unbounded) sequence of outcomes, with the aim of approaching a desired goal, target or result. Thus,
  - (a) Iteration must start with an **initialization** (starting point), and
  - (b) Perform a step-by-step marching in which the results of one iteration are used as the starting point for the next iteration.

In the context of mathematics or computer science, iteration (along with the related technique of recursion) is a **very basic building block** in programming.

As in other computer languages, Matlab provides a few types of loops to handle looping requirements including: **while loops, for loops, and nested loops**.

## While loop

The **while loop** repeatedly executes statements while a specified condition is true. The syntax of a while loop in Matlab is as follows.

```
while <expression>
    <statements>
end
```

An expression is true when the result is nonempty and contains all nonzero elements, logical or real numeric; otherwise the expression is false.

**Example 1.15.** Here is an example for the while loop.

```
%% while loop
a=10; b=15;
fprintf('while loop execution: a=%d, b=%d\n',a,b);

while a<=b
    fprintf('    The value of a=%d\n',a);
    a = a+1;
end
```

When the code above is executed, the result will be:

```
while loop execution: a=10, b=15
    The value of a=10
    The value of a=11
    The value of a=12
    The value of a=13
    The value of a=14
    The value of a=15
```

**For loop**

A **for loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. The syntax of a for loop in Matlab is as following:

```
for index = values
    <program statements>
end
```

Here **values** can be any list of numbers. For example:

- `initval:endval` – increments the index variable from `initval` to `endval` by 1, and repeats execution of program statements while index is not greater than `endval`.
- `initval:step:endval` – increments index by the value `step` on each iteration, or decrements when `step` is negative.

**Example 1.16.** The code in Example 1.15 can be rewritten as a for loop.

```
%% for loop
a=10; b=15;
fprintf('for loop execution: a=%d, b=%d\n',a,b);

for i=a:b
    fprintf('    The value of i=%d\n',i);
end
```

When the code above is executed, the result will be:

```
for loop execution: a=10, b=15
    The value of i=10
    The value of i=11
    The value of i=12
    The value of i=13
    The value of i=14
    The value of i=15
```

## Nested loops

Matlab also allows to use one loop inside another loop. The syntax for a **nested loop** in Matlab is as follows:

```
for n = n0:n1
    for m = m0:m1
        <statements>;
    end
end
```

The syntax for a nested while loop statement in Matlab is as follows:

```
while <expression1>
    while <expression2>
        <statements>;
    end
end
```

For a nested loop, you can combine

- for loop and while loop
- more than two



## Loop Control Statements

### Break Statement

The break statement terminates execution of for or while loops.

- Statements in the loop that appear after the break statement are not executed.
- In nested loops, break exits only from the loop in which it occurs.
- Control passes to the statement following the end of that loop.

**Example 1.17.** Let's modify the code in Example 1.15 to involve a break statement.

```
%% "break" statement with while loop
a=10; b=15; c=12.5;
fprintf('while loop execution: a=%d, b=%d, c=%g\n',a,b,c);

while a<=b
    fprintf('    The value of a=%d\n',a);
    if a>c, break; end
    a = a+1;
end
```

When the code above is executed, the result is:

```
while loop execution: a=10, b=15, c=12.5
    The value of a=10
    The value of a=11
    The value of a=12
    The value of a=13
```

When the condition  $a > c$  is satisfied, break is invoked; which breaks the while loop to stop.

## Continue Statement

**continue** passes control to the next iteration of a for or while loop.

- It skips any remaining statements in the body of the loop for the current iteration; the program continues execution from the next iteration.
- `continue` applies only to the body of the loop where it is called.
- In nested loops, `continue` skips remaining statements only in the body of the loop in which it occurs.

**Example 1.18.** Consider a modification of the code in Example 1.16.

```
%% for loop with "continue"
a=10; b=15;
fprintf('for loop execution: a=%d, b=%d\n',a,b);

for i=a:b
    if mod(i,2), continue; end % even integers, only
    disp(['    The value of i=' num2str(i)]);
end
```

When the code above got executed, the result is:

```
for loop execution: a=10, b=15
    The value of i=10
    The value of i=12
    The value of i=14
```

**Note:** In the above, `mod(i,2)` returns the remainder when `i` is divided by 2 (so that the result is either 0 or 1). In general,

- `mod(a,m)` returns the remainder after division of `a` by `m`, where `a` is the dividend and `m` is the divisor.
- This `mod` function is often called the modulo operation.

### 1.2.3. Anonymous Function

**Matlab-code 1.19.** In Matlab, one can define an **anonymous function**, which is a function that is not stored in a program file.

```
anonymous_function.m
1  %% Define an anonymous function
2  f = @(x) x.^3-x-2;
3
4  %% Evaluate the function
5  f1 = f(1)
6  X = 1:6;
7  fX = feval(f,X)
8
9  %% Calculus
10 q = integral(f,1,3)
```

```
Output
1  >> anonymous_function
2  f1 =
3     -2
4  fX =
5     -2     4    22    58   118   208
6  q =
7     12
```

### 1.2.4. Open Source Alternatives to Matlab

- **Octave** is the best-known alternative to Matlab. Octave strives for exact compatibility, so many of your projects developed for Matlab may run in Octave with no modification necessary.
- **NumPy** is the main package for scientific computing with Python. It can process  $n$ -dimensional arrays, complex matrix transforms, linear algebra, Fourier transforms, and can act as a gateway for C and C++ integration. It is the fundamental data-array structure for the SciPy Stack, and an ecosystem of Python-based math, science, and engineering software. Python basics will be considered in Chapter 8, p. 211.

## Exercises for Chapter 1

1.1. On Matlab command window, perform the following

- 1:20
- 1:1:20
- 1:2:20
- 1:3:20;
- isprime(12)
- isprime(13)
- for i=3:3:30, fprintf(' [i,i^2]=[%d, %d]\n',i,i^2), end

The above is the same as  $\left[ \begin{array}{l} \text{for } i=3:3:30 \\ \quad \text{fprintf(' [i,i^2]=[%d, %d]\n',i,i^2)} \\ \text{end} \end{array} \right.$

- for i=1:10,if isprime(i),fprintf('prime=%d\n',i);end,end
- Rewrite it with linebreaks, rather than using comma (,).

1.2. Compose a code and write as a function for the sum of prime numbers not larger than a positive integer  $n$ .

1.3. Modify the function you made in Exercise 2 to count the number of prime numbers and return the result along with the sum. For **multiple output**, the function may start with

```
function [sum, numver] = <function_name>(inputs)
```

1.4. Let, for  $k, n$  positive integers,

$$S_k = 1 + 2 + \cdots + k = \sum_{i=1}^k i$$

and

$$T_n = \sum_{k=1}^n S_k.$$

Write a code to find and print out  $S_n$  and  $T_n$  for  $n = 1 : 10$ .

1.5. The **golden ratio** is the number  $\phi = \frac{1 + \sqrt{5}}{2}$ .

- (a) Verify that the golden ratio is the positive solution of  $x^2 - x - 1 = 0$ .
- (b) Evaluate the golden ratio in 12-digit decimal accuracy.

1.6. The **Fibonacci sequence** is a series of numbers, defined by

$$f_0 = 0, \quad f_1 = 1; \quad f_n = f_{n-1} + f_{n-2}, \quad n = 2, 3, \dots \quad (1.3)$$

The Fibonacci sequence has interesting properties; two of them are

- (i) The ratio  $r_n = f_n/f_{n-1}$  approaches the golden ratio, as  $n$  increases.  
 (ii) Let  $x_1$  and  $x_2$  be two solutions of  $x^2 - x - 1 = 0$ :

$$x_1 = \frac{1 - \sqrt{5}}{2} \quad \text{and} \quad x_2 = \frac{1 + \sqrt{5}}{2}.$$

Then

$$t_n := \frac{(x_2)^n - (x_1)^n}{\sqrt{5}} = f_n, \quad \text{for all } n \geq 0. \quad (1.4)$$

(a) Compose a code to print out the following in a table format.

$$n \quad f_n \quad r_n \quad t_n$$

for  $n \leq K = 20$ .

You may start with

```

_____ Fibonacci_sequence.m _____
1  K = 20;
2  F = zeros(K);
3  F(1)=1; F(2)=F(1);
4
5  for n=3:K
6      F(n) = F(n-1)+F(n-2);
7      rn = F(n)/F(n-1);
8      fprintf("n =%3d;  F = %7d;  rn = %.12f\n",n,F(n),rn);
9  end
  
```

(b) Find  $n$  such that  $r_n$  has 12-digit decimal accuracy to the golden ratio  $\phi$ .

*Ans:* (b)  $n = 32$



## CHAPTER 2

# Programming Examples

### Contents of Chapter 2

2.1. Area Estimation of the Region Defined by a Closed Curve . . . . .	30
2.2. Visualization of Complex-Valued Solutions . . . . .	35
2.3. Discrete Fourier Transform . . . . .	38
2.4. Computational Algorithms and Their Convergence . . . . .	47
2.5. Inverse Functions: Exponentials and Logarithms . . . . .	53
Exercises for Chapter 2 . . . . .	62

## 2.1. Area Estimation of the Region Defined by a Closed Curve

**Problem 2.1.** It is common in reality that a **region** is saved by a *sequence of points*: For some  $n > 0$ ,

$$(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n), \quad (x_n, y_n) = (x_0, y_0). \quad (2.1)$$

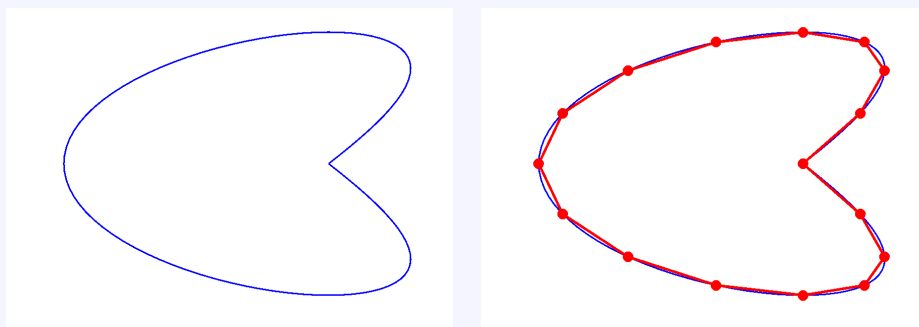


Figure 2.1: A region and its approximation.

**Here the question is:**

*If a sequence of points (2.1) represents a region, how can we compute its area accurately?*

### Derivation of Computational Formulas

**Example 2.2.** Let's begin with a very simple example.

(a) **Find the area of a rectangle**  $[a, b] \times [c, d]$ .

**Solution.** We know the area =  $(b - a) \cdot (d - c)$ .  
It can be rewritten as

$$b \cdot (d - c) - a \cdot (d - c) = b \cdot (d - c) + a \cdot (c - d)$$

from which we may guess that

$$\text{Area} = \sum_i x_i^* \cdot \Delta y_i, \quad (2.2)$$





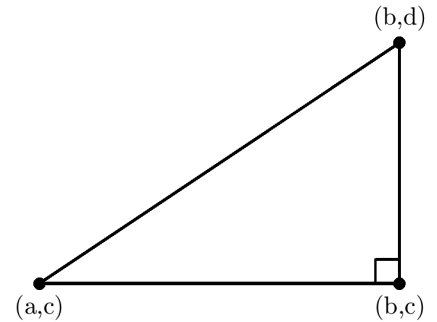
where the sum is carried out over line segments  $L_i$  and  $x_i^*$  denotes the mid value of  $x$  on  $L_i$ .

(b) **Find the area of a triangle.**

**Solution.** We know the area =  $\frac{1}{2}(b - a) \cdot (d - c)$ .  
 Now, let's try to find the area using the formula (2.2):

$$\text{Area} = \sum_i x_i^* \cdot \Delta y_i.$$

Let  $L_1, L_2, L_3$  be the bottom side, vertical side, and the hypotenuse, respectively.



Then

$$\begin{aligned} \text{Area} &= \frac{a+b}{2} \cdot (c - c) + \frac{b+b}{2} \cdot (d - c) + \frac{b+a}{2} \cdot (c - d) \\ &= 0 + b \cdot (d - c) + \frac{b+a}{2} \cdot (c - d) \\ &= \left(b - \frac{b+a}{2}\right) \cdot (d - c) = \frac{1}{2}(b - a) \cdot (d - c). \end{aligned}$$

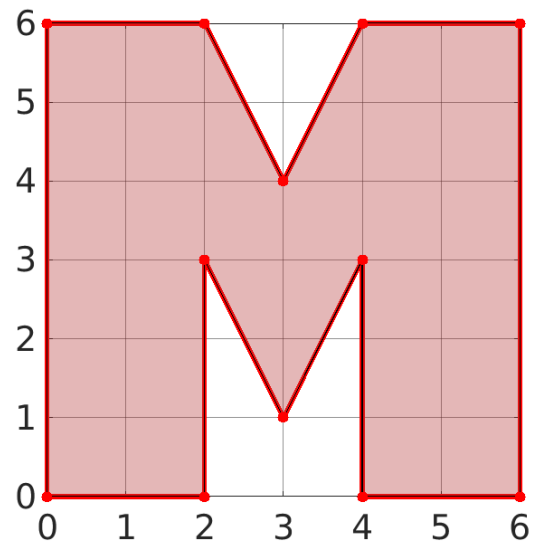
Okay. The formula is correct!

**Note:** Horizontal line segments makes **no** contribution to the area.

(c) **Let's verify the formula once more.**

The area of the M-shaped is 30.  
 Let's collect only nonzero values:

$$\begin{aligned} &2 \cdot 3 - 2.5 \cdot 2 + 3.5 \cdot 2 - 4 \cdot 3 \\ &\quad + 6 \cdot 6 \\ &\quad - 3.5 \cdot 2 + 2.5 \cdot 2 \\ &= 6 - 5 + 7 - 12 \\ &\quad + 36 \\ &\quad - 7 + 5 \\ &= 30 \end{aligned}$$



Again, the formula is correct!!

**Summary 2.3.** The above work can be summarized as follows.

- Let a region  $R$  be represented as a sequence of points

$$(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n), \quad (x_n, y_n) = (x_0, y_0). \quad (2.3)$$

- Let  $L_i$  be the  $i$ -th line segment connecting  $(x_{i-1}, y_{i-1})$  and  $(x_i, y_i)$ ,  $n = 1, 2, \dots, n$ . Then the area of  $R$  can be computed using the formula

$$\text{Area}(R) = \sum_{i=1}^n x_i^* \cdot \Delta y_i, \quad (2.4)$$

where

$$x_i^* = \frac{x_{i-1} + x_i}{2}, \quad \Delta y_i = y_i - y_{i-1}.$$

**Note:** The formula (2.4) is a result of **Green's Theorem** for the line integral and **numerical approximation**.

**Example 2.4.** We will generate a dataset, save it to a file, and read it to plot and measure the area.

- (a) Generate a dataset that represents the circle of radius centered the origin. For example, for  $i = 0, 1, 2, \dots, n$ ,

$$(x_i, y_i) = (\cos \theta_i, \sin \theta_i), \quad \theta_i = i \cdot \frac{2\pi}{n}. \quad (2.5)$$

Note that  $(x_n, y_n) = (x_0, y_0)$ .

- (b) Analyze accuracy improvement of the area as  $n$  grows. The larger  $n$  you choose, the more accurately the data would represent the region.

**Solution.**

```

_____ circle.m _____
1  n = 10;
2  %%---- Data generation -----
3  theta = linspace(0,2*pi,n+1)';    % a column vector
4  data = [cos(theta),sin(theta)];
5

```

```
6 %%---- Plot it & Save the image -----
7 figure,
8 plot(data(:,1),data(:,2),'r-','linewidth',2);
9 daspect([1 1 1]); axis tight;
10 xlim([-1 1]), ylim([-1 1]);
11 title(['Circle: n=' int2str(n)])
12 image_name = 'circle.png';
13 saveas(gcf,image_name);
14
15 %%---- Save the data -----
16 filename = 'circle-data.txt';
17 csvwrite(filename,data)
18 %writematrix(data,filename,'Delimiter',' ');
19
20 %%=====
21 %%---- Read the data -----
22 %%=====
23 DATA = load(filename);
24 X = DATA(:,1);
25 Y = DATA(:,2);
26
27 figure,
28 plot(X,Y,'b--','linewidth',2);
29 daspect([1 1 1]); axis tight
30 xlim([-1 1]), ylim([-1 1]);
31 title(['Circle: n=' int2str(n)])
32 yticks(-1:0.5:1)
33 saveas(gcf,'circle-dashed.png');
34
35 %%---- Area computation -----
36 area = area_closed_curve(DATA); %See an Exercise problem
37 fprintf('n = %3d; area = %.12f, misfit = %.12f\n', ...
38         size(DATA,1)-1,area, abs(pi-area));
```

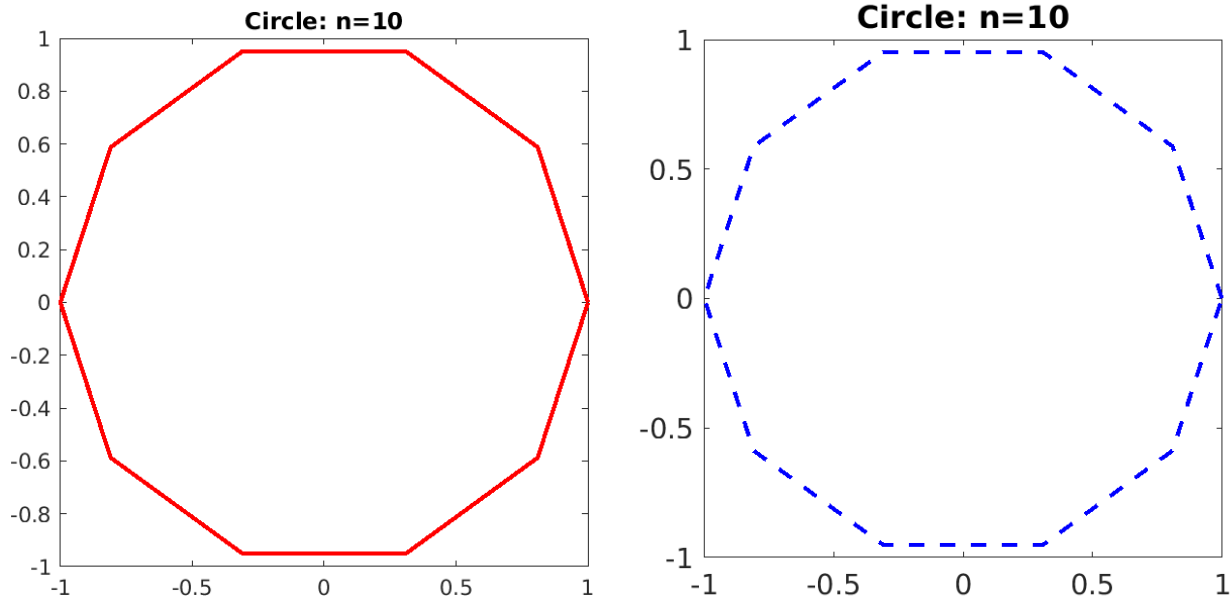


Figure 2.2: An approximation of the unit circle, with  $n = 10$ .

#### Accuracy Improvement

```

1  n = 10; area = 2.938926261462, misfit = 0.202666392127
2  n = 20; area = 3.090169943749, misfit = 0.051422709840
3  n = 40; area = 3.128689300805, misfit = 0.012903352785
4  n = 80; area = 3.138363829114, misfit = 0.003228824476
5  n = 160; area = 3.140785260725, misfit = 0.000807392864

```

**Note:** The misfit becomes **a quarter** as the number of points is doubled.

**Remark 2.5.** From Example 2.4, you learn how to

- Generate datasets
- Save data into a file
- Read a data file
- Set figure environments
- Call functions

## 2.2. Visualization of Complex-Valued Solutions

**Problem 2.6.** Seeking the solutions of

$$f(x) = x^2 - x + 1 = 0, \quad (2.6)$$

we can easily find that the equation has **no real solutions**. However, by using the **quadratic formula**, the complex-valued solutions are

$$x = \frac{1 \pm \sqrt{3}i}{2}.$$

**Here we have questions:**

*What do the complex-valued solutions mean?*

*Can we visualize them?*

**Remark 2.7. Complex Number System:**

The most complete number system is the system of complex numbers:

$$\mathbb{C} = \{x + yi \mid x, y \in \mathbb{R}\}$$

where  $i = \sqrt{-1}$ , called the **imaginary unit**.

- Seeking a **real-valued solution** of  $f(x) = 0$  is the same as finding a solution of  $f(z) = 0$ ,  $z = x + yi$ , restricting on the  $x$ -axis ( $y = 0$ ).
- If

$$f(z) = A(x, y) + B(x, y)i, \quad (2.7)$$

then the complex-valued solutions are the points  $x + yi$  such that  $A(x, y) = B(x, y) = 0$ .

**Example 2.8.** For  $f(x) = x^2 - x + 1$ , express  $f(x + yi)$  in the form of (2.7).  
**Solution.**

$$\text{Ans: } f(z) = (x^2 - x - y^2 + 1) + ((2x - 1)y)i$$

**Example 2.9.** Implement a code to visualize complex-valued solutions of  $f(x) = x^2 - x + 1 = 0$ .

**Solution.** From Example 2.8,

$$f(z) = A(x, y) + B(x, y)i, \quad A(x, y) = x^2 - x - y^2 + 1, \quad B(x, y) = (2x - 1)y.$$

```

----- visualize_complex_solution.m -----
1  close all
2  if exist('OCTAVE_VERSION','builtin'), pkg load symbolic; end
3
4  syms x y real
5
6  %% z^2 -z +1 = 0
7  A = @(x,y) x.^2-x-y.^2+1;
8  B = @(x,y) (2*x-1).*y;
9  T = 'z^2-z+1=0';
10
11  figure, % Matlab 'fmesh' is not yet in Octave
12  np=41; X=linspace(-5,5,np); Y=linspace(-5,5,np);
13  mesh(X,Y,A(X,Y),'EdgeColor','r'), hold on
14  mesh(X,Y,B(X,Y),'EdgeColor','b'),
15  mesh(X,Y,zeros(np,np),'EdgeColor','k'),
16  legend("A","B","0"),
17  xlabel('x'), ylabel('y'), title(['A and B for ' T])
18  hold off
19  print -dpng 'complex-solutions-A-B-fmesh.png'
20
21  %%--- Solve A=0 and B=0 -----
22  [xs,ys] = solve(A(x,y)==0,B(x,y)==0,x,y);
23
24  figure,
25  np=101; X=linspace(-5,5,np); Y=linspace(-5,5,np);
26  contour(X,Y,A(X,Y'), [0 0],'r','linewidth',2), hold on
27  contour(X,Y,B(X,Y'), [0 0],'b--','linewidth',2)
28  plot(double(xs),double(ys),'r.','MarkerSize',30) % the solutions
29  grid on
30  %ax=gca; ax.GridAlpha=0.5; ax.FontSize=13;
31  legend("A=0","B=0")
32  xlabel('x'), ylabel('yi'), title(['Compex solutions of ' T])
33  hold off
34  print -dpng 'complex-solutions-A-B=0.png'

```

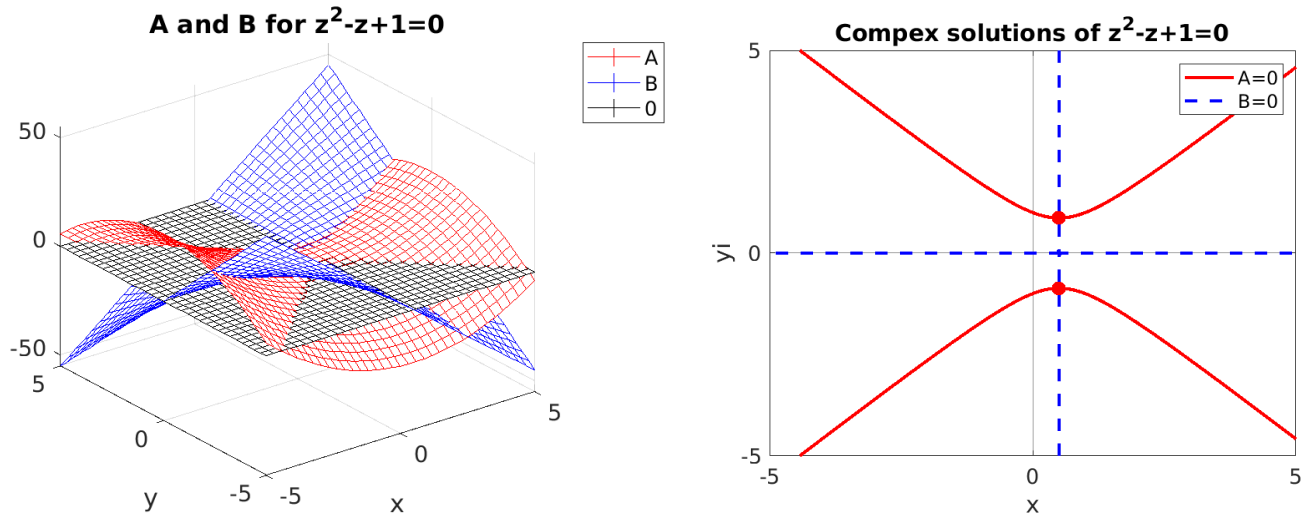


Figure 2.3: Two solutions are  $\frac{1}{2} + \frac{\sqrt{3}}{2}i$  and  $\frac{1}{2} - \frac{\sqrt{3}}{2}i$ .

**Remark 2.10.** You can easily find the **real part** and the **imaginary part** of polynomials of  $z = x + iy$  as follows.

```

_____ Real and Imaginary Parts _____
1  syms x y real
2  z = x + 1i*y;
3
4  g = z^2 - z + 1;
5  simplify(real(g))
6  simplify(imag(g))

```

Here “1i” (number 1 and letter i), appeared in Line 2, means the imaginary unit  $i = \sqrt{-1}$ .

```

_____ Output _____
1  ans = x^2 - x - y^2 + 1
2  ans = y*(2*x - 1)

```

**Summary 2.11. Visualization of Complex-Valued Solutions**

Seeking a **real-valued solution** of  $f(x) = 0$  is the same as finding a solution of  $f(z) = 0$ ,  $z = x + yi$ , restricting on the  $x$ -axis ( $y = 0$ ).

## 2.3. Discrete Fourier Transform

**Note:** In spectral analysis of audio data, our goal is to determine **the frequency content of a signal**.

- For **analog signals**: Use **Fourier transform**
- For **digital signals**: Use **discrete Fourier transform**

### **Definition** 2.12. **Fourier Transform**

- For a function  $x(t)$ , a continuous signal, the **Fourier transform** is defined as

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t} dt, \quad (2.8)$$

where  $\omega = 2\pi f$  is the angular frequency and  $f$  is the frequency.

- The **inverse Fourier transform** is defined as

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega)e^{i\omega t} d\omega. \quad (2.9)$$

The identity

$$e^{i\theta} = \cos \theta + i \sin \theta, \quad \theta \in \mathbb{R}, \quad (2.10)$$

is called the **Euler's identity**; see Exercise 3.3 on p.109.



### 2.3.1. Discrete Fourier Transform

#### **Definition 2.13. Discrete Fourier Transform (DFT)**

For a discrete signal  $\{x(n\Delta t)\}$ , the **discrete Fourier transform** is defined as

$$X(k\Delta f) = \sum_{n=0}^{N-1} x(n\Delta t)e^{-i(2\pi k\Delta f)(n\Delta t)}, \quad k = 0, 1, 2, \dots, N-1, \quad (2.11)$$

where

- $N$  = the total number of discrete data points taken
- $T$  = the total sampling time (second)
- $\Delta t$  = time between data points,  $\Delta t = T/N$
- $\Delta f$  = the **frequency increment (frequency resolution)**  
 $\Delta f = 1/T$  (Hz)
- $f_s$  = the sampling frequency (per second),  $f_s = 1/\Delta t = N/T$ .

**Note:**  $(k\Delta f)(n\Delta t) = kn/N$

#### **Definition 2.14. Inverse Discrete Fourier Transform (IDFT)**

The **inverse discrete Fourier transform** of  $X$  is defined as

$$x(n\Delta t) = \frac{1}{N} \sum_{k=0}^{N-1} X(k\Delta f)e^{i(2\pi k\Delta f)(n\Delta t)}, \quad n = 0, 1, 2, \dots, N-1, \quad (2.12)$$

#### **Remark 2.15. Fast Fourier Transform (FFT)**

The **fast Fourier transform** is *simply* a DFT that is fast.

- All the rules and details about DFTs apply to FFTs as well.
- **Power-of-2 Restriction**
  - Many FFTs (e.g., in Microsoft **Excel**) restrict  $N$  to a power of 2, such as 64, 128, 256, and so on.
  - The FFT in Matlab has no such restriction.

**Example 2.16.** Generate a synthetic signal for  $T = 4$  seconds, at a sampling rate of  $f_s = 100$  Hz. Then, compute its DFT and restore the original signal by applying the IDFT.

**Solution.** Let's begin with the DFT and the IDFT.

```

----- discrete_Fourier.m -----
1  function X = discrete_Fourier(x)
2  % function X = discrete_Fourier(x)
3  % Calculate the full DFT
4
5  N = length(x); % Length of input sequence
6  X = zeros(1,N); % Initialize output sequence
7
8  % (k*Df)*n*Dt = (k/T)*n*(T/N) = k*n/N
9  for k = 0:N-1      % Loop over all frequency components
10     for n = 0:N-1  % Loop over all time-domain samples
11         X(k+1) = X(k+1) + x(n+1)*exp(-1i*2*pi*k*n/N);
12     end
13 end

```

```

----- discrete_Fourier_inverse.m -----
1  function X = discrete_Fourier_inverse(x)
2  % function X = discrete_Fourier_inverse(x)
3  % Calculate the inverse DFT
4
5  N = length(x); % Length of input sequence
6  X = zeros(1,N); % Initialize output sequence
7
8  % (k*Df)*n*Dt = (k/T)*n*(T/N) = k*n/N
9  for n = 0:N-1      % Loop over all time-domain samples
10     for k = 0:N-1  % Loop over all frequency components
11         X(n+1) = X(n+1) + x(k+1)*exp(1i*2*pi*k*n/N);
12     end
13 end
14
15 X = X/N;

```

For a synthetic signal, we combine two sinusoidals, of which frequencies  $f = 10, 20$  and magnitudes are 1, 2.

```

----- signal_DFT.m -----
1  close all
2
3  T = 4; fs = 100;
4  t = 0:1/fs:T-1/fs;           % Time vector
5  x = sin(2*pi*10*t) + 2*sin(2*pi*20*t); % Signal
6
7  figure, plot(t,x,'-k')
8  print -dpng 'dft-data-signal.png'
9
10 %-----
11 X = discrete_Fourier(x);
12
13 % Compute magnitude and phase
14 N = length(X);
15 mag_X = zeros(1,N); % Initialize magnitude
16 phi_X = zeros(1,N); % Initialize phase
17 for k = 1:N
18     mag_X(k) = sqrt(real(X(k))^2 + imag(X(k))^2);
19     phi_X(k) = atan2(imag(X(k)), real(X(k)));
20 end
21
22 %-----
23 x_restored = discrete_Fourier_inverse(X);
24 misfit = max(abs(x-x_restored))
25
26 % Plots for Spectra
27 %-----
28 f = (0:N-1)*fs/N; % Frequency vector
29 figure, subplot(2,1,1);
30 plot(f, mag_X,'-r','linewidth',1.5);
31 xlabel('Frequency (Hz)'); ylabel('Magnitude'); title('Magnitude Spectrum');
32 subplot(2,1,2);
33 plot(f, phi_X,'-b');
34 xlabel('Frequency (Hz)'); ylabel('Phase (rad)'); title('Phase Spectrum');
35 print -dpng 'dft-magnitude-phase.png'

```

```

----- Output -----
1  misfit = 2.6083e-13

```

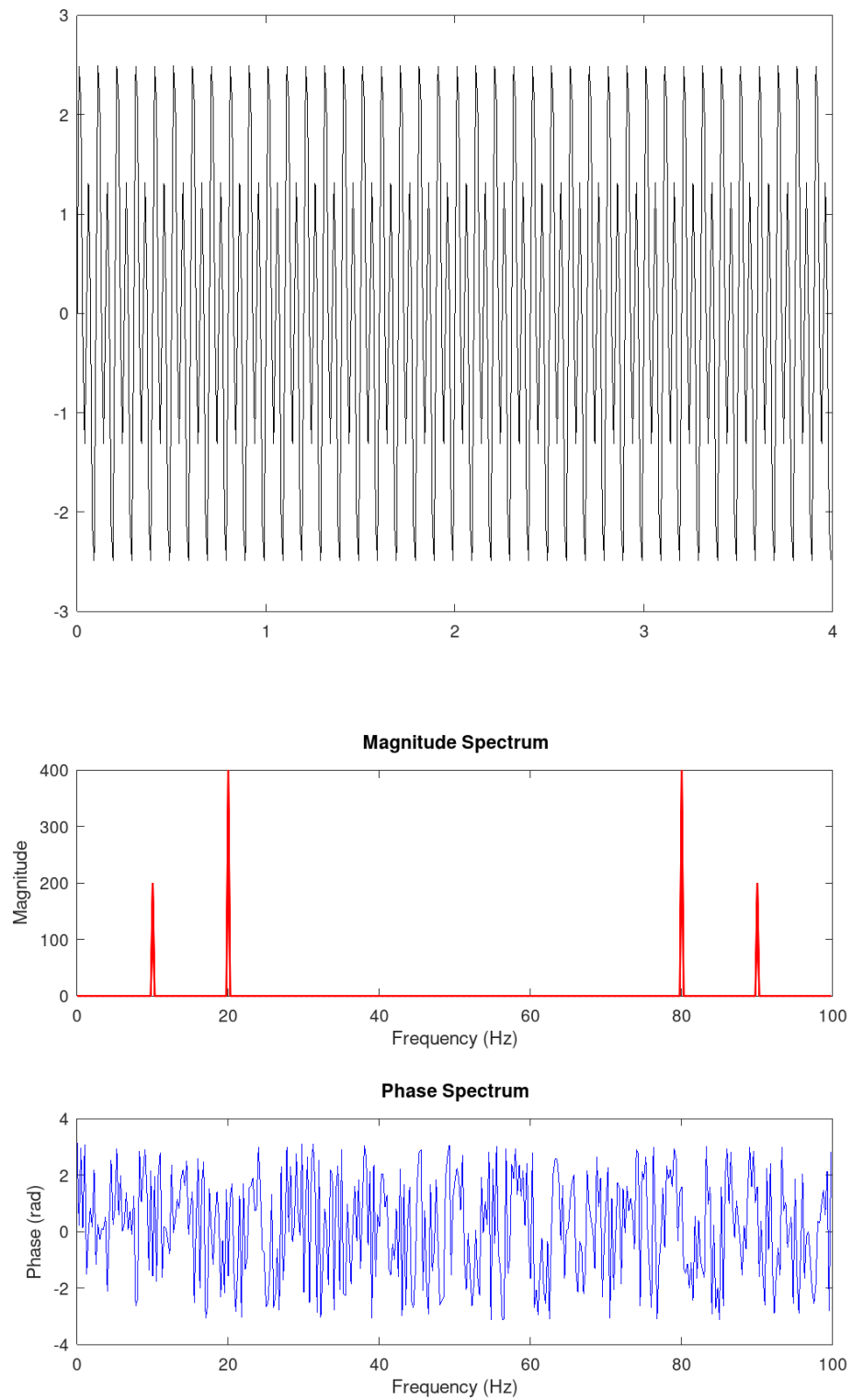


Figure 2.4: A synthetic signal and its spectra.

**Remark 2.17. Nyquist Criterion**

- The **Nyquist criterion** is important in DFT analysis.
  - **When sampling at frequency  $f_s$ , we can obtain reliable frequency information only for frequencies less than  $f_s/2$ .** (Here, reliable means without aliasing problems.)
- We can calculate at what value of  $k$  the frequency  $k\Delta f = f_s/2$ .

$$k = \frac{f_s/2}{\Delta f} = \frac{(N/T)/2}{1/T} = \frac{N}{2}. \quad (2.13)$$

- Therefore, we conclude that the **maximum useful frequency**  $f_{\max}$  from a DFT output (also called the **folding frequency**  $f_{\text{folding}}$ ) is

$$f_{\max} = f_{\text{folding}} = \frac{f_s}{2} = \frac{N}{2}\Delta f. \quad (2.14)$$

In other words, only half of the  $N$  available DFT output values are useful – for  $k = 0:N/2-1$ .

**Example 2.18.** Suppose we sample a signal for  $T = 4$  seconds, at a sampling rate of  $f_s = 100$  Hz.

- (a) How many data points are taken?

$$N = Tf_s = 4 \cdot 100 = 400$$

- (b) How many useful DFT output values are obtained?

$$N/2 = 200$$

- (c) What is  $\Delta f$ ?

$$\Delta f = 1/T = 1/4 = 0.25 \text{ Hz}$$

- (d) What is the maximum frequency for which the DFT output is useful and reliable?

$$f_{\max} = (N/2)\Delta f = (400/2) \cdot 0.25 = 50 \text{ Hz}$$

**Note:** *The other half of the output values ( $f > f_{\text{folding}}$ ) are thrown out or ignored.*

### 2.3.2. Short-Time Fourier Transform

The **short-time Fourier transform (STFT)** is used to analyze **how the frequency content of a signal changes over time**.

- The procedure for computing the STFT:
  - (a) **Divide the signal** into short segments of equal length.
  - (b) **Compute the DFT separately** on each short segment.
- The magnitude squared of the STFT is known as the **spectrogram**, a time-frequency representation of the signal.

#### Remark 2.19. Short-Time Fourier Transform.

- It requires to set
  - **An analysis window  $g(n)$  of length  $M$**
  - **$R$ : The window hops over the original signal by  $R$  samples.**
  - ⇒ The overlap  $L = M - R$
- Most window functions taper off at the edges to avoid spectral ringing.
- The DFT of each windowed segment is stored into a complex-valued matrix of  $\text{int}((N_s - L)/(M - L))$  columns, where  $N_s$  is the length of the original signal.

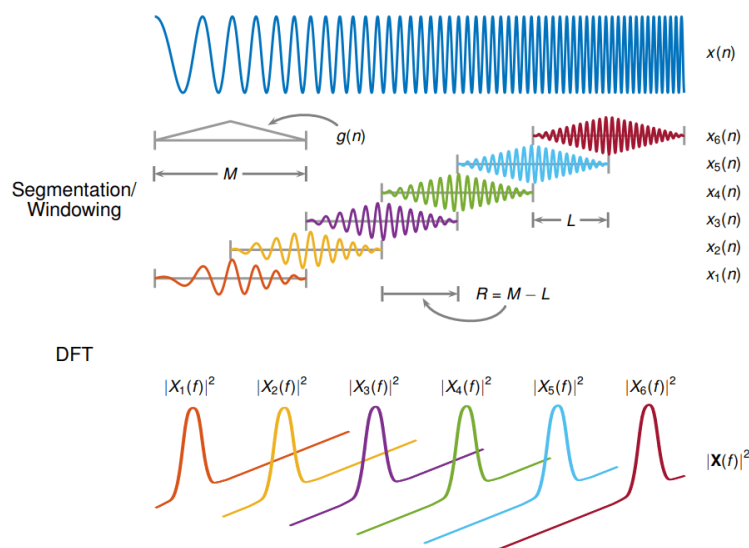


Figure 2.5: iscola\_stft.png from Matlab.

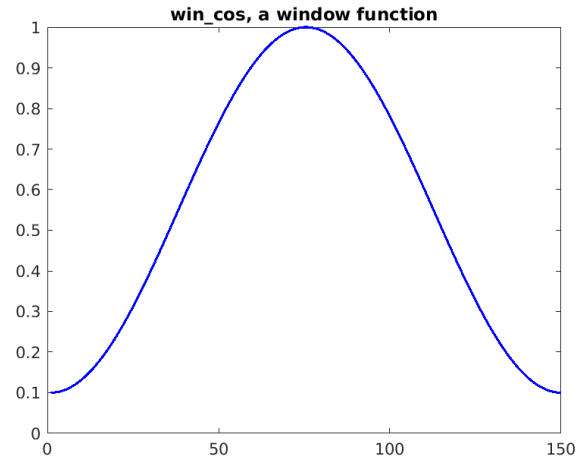
**Example 2.20.** Implement a code for the STFT.  
**Solution.**

We first select an analysis window.

```

----- win_cos.m -----
1 function g = win_cos(M)
2 % function g = win_cos(M)
3
4 t = linspace(0,2*pi,M);
5 g = 0.55-0.45*cos(t);

```



```

----- short_time_DFT.m -----
1 close all; clear all
2
3 fs=10000; %sampling frequency in Hz
4 T0=1;
5 t0=0:1/fs:T0-1/fs; % Time vector
6 x0=sin((100+3900*t0/2).*(2*pi*t0)); % a chirp signal
7
8 %-----
9 x = [x0,x0,x0]; %t = [t0,t0+T0,t0+2*T0];
10
11 %-----
12 M = 150; % window length
13 R = 60; % sliding length
14
15 g = win_cos(M);
16 F = stft2(x,g,R);
17 S = abs(F).^2; % spectrogram
18
19 %-- Plots -----
20 figure,plot(t0(1:1000), x0(1:1000),'-k','linewidth',1.5)
21 title('First 1000 Samples of the Chirp Signal')
22 print -dpng 'stft-chirp-signal.png'
23
24 figure,plot(1:M,g,'-b','linewidth',1.5);
25 ylim([0,1]); title('win\_cos, a window function')
26 print -dpng 'stft-window-g.png'
27

```

```

28 Df = fs/M; Dt = R/fs;
29 figure, imagesc((0:size(S,2)-1)*Dt,(0:M-1)*Df,S)
30 xlabel('Time (second)'); ylabel('Frequency (Hz)'); title('Spectrogram');
31 colormap('pink'); colorbar; set(gca,'YDir','normal')
32 print -dpng 'stft-spectrogram.png'

```

```

----- stft2.m -----
1 function F = stft2(x,g,R)
2 % function F = stft2(x,g,R)
3 %   x: the signal
4 %   g: the window function of length M
5 %   R: sliding length
6 % Output: F = the short-time DFT
7
8 Ns = length(x); M = length(g);
9 L = M-R; % overlap
10
11 Col = floor((Ns-L)/(M-L));
12 F = zeros(M,Col); c = 1;
13
14 while 1
15     if c==1, n0=1; else, n0=n0+R; end
16     n1=n0+M-1;
17     if n1>Ns, break; end
18     signal = x(n0:n1).*g;
19     F(:,c) = discrete_Fourier(signal)'; c=c+1;
20 end

```

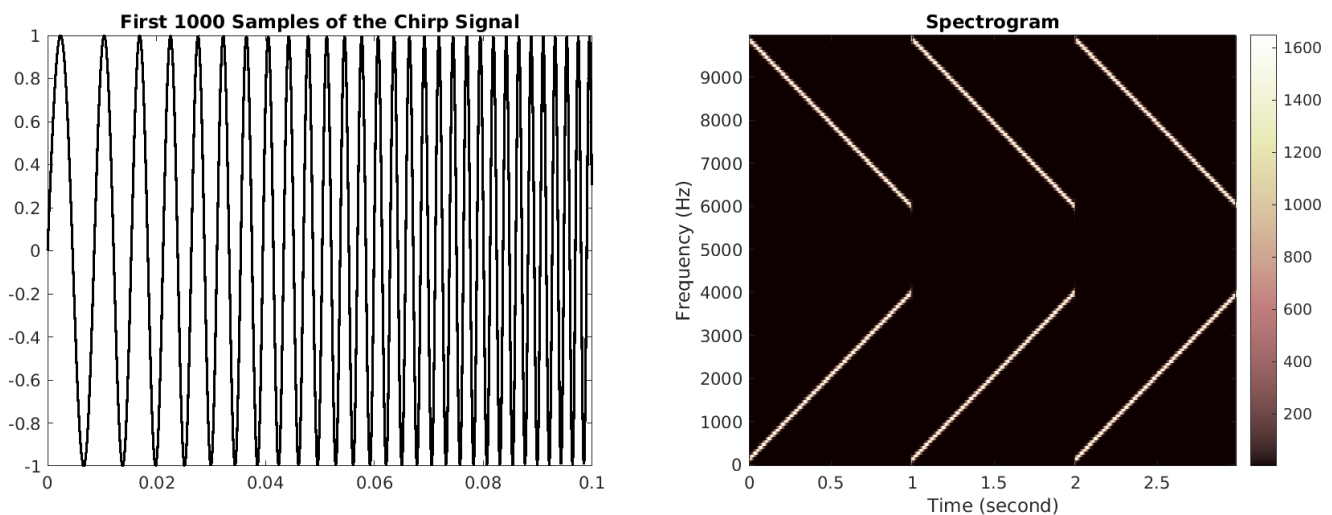


Figure 2.6: The first 1000 samples of the chirp signal and the spectrogram from the STFT.



## 2.4. Computational Algorithms and Their Convergence

**Definition 2.21.** Suppose that  $p^*$  is an approximation to  $p$ . Then

- The **absolute error** is  $|p - p^*|$ , and
- the **relative error** is  $\frac{|p - p^*|}{|p|}$ , provided that  $p \neq 0$ .

### 2.4.1. Computational Algorithms

**Definition 2.22.** An **algorithm** is a procedure that describes, in an unambiguous manner, a finite sequence of steps to be carried out in a specific order.

Algorithms consist of various steps for inputs, outputs, and functional operations, which can be described effectively by a so-called **pseudocode**.

**Definition 2.23.** An algorithm is called **stable**, if small changes in the initial data produce correspondingly small changes in the final results. Otherwise, it is called **unstable**. Some algorithms are stable only for certain choices of data/parameters, and are called **conditionally stable**.

**Notation 2.24. (Growth rates of the error):** Suppose that  $E_0 > 0$  denotes an error introduced at *some* stage in the computation and  $E_n$  represents the magnitude of the error after  $n$  subsequent operations.

- If  $E_n = C \times n E_0$ , where  $C$  is a constant independent of  $n$ , then the growth of error is said to be **linear**, for which the algorithm is stable.
- If  $E_n = C^n E_0$ , for some  $C > 1$ , then the growth of error is **exponential**, which turns out unstable.

## Rates (Orders) of Convergence

**Definition 2.25.** Let  $\{x_n\}$  be a sequence of real numbers tending to a limit  $x^*$ .

- The rate of convergence is at least **linear** if there are a constant  $c_1 < 1$  and an integer  $N$  such that

$$|x_{n+1} - x^*| \leq c_1 |x_n - x^*|, \quad \forall n \geq N. \quad (2.15)$$

- We say that the rate of convergence is at least **superlinear** if there exist a sequence  $\varepsilon_n$  tending to 0 and an integer  $N$  such that

$$|x_{n+1} - x^*| \leq \varepsilon_n |x_n - x^*|, \quad \forall n \geq N. \quad (2.16)$$

- The rate of convergence is at least **quadratic** if there exist a constant  $C$  (not necessarily less than 1) and an integer  $N$  such that

$$|x_{n+1} - x^*| \leq C |x_n - x^*|^2, \quad \forall n \geq N. \quad (2.17)$$

- In general, we say that the rate of convergence is **of  $\alpha$  at least** if there exist a constant  $C$  (not necessarily less than 1 for  $\alpha > 1$ ) and an integer  $N$  such that

$$|x_{n+1} - x^*| \leq C |x_n - x^*|^\alpha, \quad \forall n \geq N. \quad (2.18)$$

**Example 2.26.** Consider a sequence defined recursively as

$$x_1 = 2, \quad x_{n+1} = \frac{x_n}{2} + \frac{1}{x_n}. \quad (2.19)$$

(a) Find the limit of the sequence; (b) show that the convergence is quadratic.

**Hint:** You may first check the behavior of the sequence. Then prove its convergence, by verifying  $x_n > \sqrt{2}$  for all  $n \geq 1$  ( $\odot x_{n+1}^2 - 2 > 0$ ) and  $x_{n+1} < x_n$  ( $\odot x_n - x_{n+1} = x_n(\frac{1}{2} - \frac{1}{x_n^2}) > 0$ ).

**Solution.**

```

sequence_sqrt2.m
1  x = 2;
2  for n=1:5
3      x = x/2 + 1/x;
4      fprintf('n=%d: xn = %.10f\n',n,x)
5  end

```

```

Output
1  n=1: xn = 1.5000000000
2  n=2: xn = 1.4166666667
3  n=3: xn = 1.4142156863
4  n=4: xn = 1.4142135624
5  n=5: xn = 1.4142135624

```

**It looks monotonically decreasing and bounded below  $\Rightarrow$  Converge!**

(It converges to  $\sqrt{2} \approx 1.41421356237310$ .)

## 2.4.2. Big $\mathcal{O}$ and little $o$ notation

### **Definition 2.27.**

- A sequence  $\{\alpha_n\}_{n=1}^{\infty}$  is said to be **in  $\mathcal{O}$  (big Oh)** of  $\{\beta_n\}_{n=1}^{\infty}$  if a positive number  $K$  exists for which

$$|\alpha_n| \leq K|\beta_n|, \text{ for large } n \left( \text{or equivalently, } \frac{|\alpha_n|}{|\beta_n|} \leq K \right). \quad (2.20)$$

In this case, we say “ $\alpha_n$  is in  $\mathcal{O}(\beta_n)$ ” and denote  $\alpha_n \in \mathcal{O}(\beta_n)$  or  $\alpha_n = \mathcal{O}(\beta_n)$ .

- A sequence  $\{\alpha_n\}$  is said to be **in  $o$  (little oh)** of  $\{\beta_n\}$  if there exists a sequence  $\varepsilon_n$  tending to 0 such that

$$|\alpha_n| \leq \varepsilon_n|\beta_n|, \text{ for large } n \left( \text{or equivalently, } \lim_{n \rightarrow \infty} \frac{|\alpha_n|}{|\beta_n|} = 0 \right). \quad (2.21)$$

In this case, we say “ $\alpha_n$  is in  $o(\beta_n)$ ” and denote  $\alpha_n \in o(\beta_n)$  or  $\alpha_n = o(\beta_n)$ .

**Example 2.28.** Show that  $\alpha_n = \frac{n+1}{n^2} = \mathcal{O}\left(\frac{1}{n}\right)$  and

$$f(n) = \frac{n+3}{n^3+20n^2} \in \mathcal{O}(n^{-2}) \cap o(n^{-1}).$$

**Solution.**

**Definition 2.29.** Suppose  $\lim_{h \rightarrow 0} G(h) = 0$ . A quantity  $F(h)$  is said to be in  $\mathcal{O}$  (**big Oh**) of  $G(h)$  if a positive number  $K$  exists for which

$$\frac{|F(h)|}{|G(h)|} \leq K, \text{ for } h \text{ sufficiently small.} \quad (2.22)$$

In this case, we say  $F(h)$  is in  $\mathcal{O}(G(h))$ , and denote  $F(h) \in \mathcal{O}(G(h))$ . **Little oh of  $G(h)$**  can be defined the same way as for sequences.

**Example 2.30.** Taylor's series expansion for  $\cos(x)$  is given as

$$\begin{aligned} \cos(x) &= 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \dots \\ &= 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 - \frac{1}{720}x^6 + \dots \end{aligned}$$

If you use a computer algebra software (e.g. Maple), you will obtain

$$\text{taylor}(\cos(x), x = 0, 4) = 1 - \frac{1}{2!}x^2 + \mathcal{O}(x^4)$$

which implies that

$$\underbrace{\frac{1}{24}x^4 - \frac{1}{720}x^6 + \dots}_{=: F(x)} = \mathcal{O}(x^4). \quad (2.23)$$

Indeed,

$$\frac{|F(x)|}{|x^4|} = \left| \frac{1}{24} - \frac{1}{720}x^2 + \dots \right| \leq \frac{1}{24}, \text{ for sufficiently small } x. \quad (2.24)$$

Thus  $F(x) \in \mathcal{O}(x^4)$ .  $\square$

**Example 2.31.** Choose the correct assertions (in each,  $n \rightarrow \infty$ )

- $(n^2 + 1)/n^3 \in o(1/n)$
- $(n + 1)/\sqrt{n} \in o(1)$
- $1/\ln n \in \mathcal{O}(1/n)$
- $1/(n \ln n) \in o(1/n)$
- $e^n/n^5 \in \mathcal{O}(1/n)$

**Example 2.32.** Let  $f(h) = \frac{1}{h}(1 + h - e^h)$ . What are the limit and the rate of convergence of  $f(h)$  as  $h \rightarrow 0$ ?

**Solution.**

**Self-study 2.33.** Show that these assertions are not true.

- a.  $e^x - 1 = \mathcal{O}(x^2)$ , as  $x \rightarrow 0$
- b.  $x = \mathcal{O}(\tan^{-1} x)$ , as  $x \rightarrow 0$
- c.  $\sin x \cos x = o(x)$ , as  $x \rightarrow 0$

**Solution.**

## 2.5. Inverse Functions: Exponentials and Logarithms

**In-Reality 2.34.** A function  $f$  is a rule that assigns an **output**  $y$  to each **input**  $x$ :  $f(x) = y$ . Thus a function is a set of actions that determines the **system**. However, in reality, it is often the case that the equation must be **solved for either the input or the function**.

1. Given  $(f, x)$ , **getting  $y$**  is the simplest and most common task.
2. Given  $(f, y)$ , **solving for  $x$**  is to find the **inverse function** of  $f$ .
3. Given  $(x, y)$ , **solving for  $f$**  is not a simple task in practice.
  - Using many data points  $\{(x_i, y_i)\}$ , finding an approximation of  $f$  is the core subject of **polynomial interpolation** (§ 3.3), **regression analysis** (Ch. 7), and **machine learning** (Ch. 10).

### **Key Idea 2.35. What is the Inverse of a Function?**

Let  $f : X \rightarrow Y$  be a function. For simplicity, consider

$$y = f(x) = 2x + 1. \quad (2.25)$$

- Then,  $f$  is a rule that performs two actions:  $\times 2$  and followed by  $+1$ .
- The **reverse** of  $f$  must be:  $-1$  followed by  $\div 2$ .

– Let  $y \in Y$ . Then the reverse of  $f$  can be written as

$$x = (y - 1)/2 =: g(y) \quad (2.26)$$

The function  $g$  is the **inverse function** of  $f$ .

– However, it is conventional to choose  $x$  for the independent variable. Thus it can be formulated as

$$y = g(x) = (x - 1)/2. \quad (2.27)$$

- Let's summarize the above:

(a) **Solve  $y = f(x)$  for  $x$ :**  $x = (y - 1)/2 =: g(y)$ .

(b) **Exchange  $x$  and  $y$ :**  $y = g(x) = (x - 1)/2$ .

## 2.5.1. Inverse functions

**Note:** The first step for finding the inverse function of  $f$  is to solve  $y = f(x)$  for  $x$ , to get  $x = g(y)$ . Here the required is **for  $g$  to be a function.**

**Definition 2.36.** A function  $f$  is called a **one-to-one function** if it never takes on the same value twice; that is,

$$f(x_1) \neq f(x_2) \quad \text{whenever } x_1 \neq x_2. \quad (2.28)$$

**Claim 2.37. Horizontal Line Test.**

A function is **one-to-one** if and only if no horizontal line intersects its graph more than once.

**Example 2.38.** Check if the function is one-to-one.

a.  $f(x) = x^2$

b.  $g(x) = x^2, x \geq 0$

c.  $h(x) = x^3$

**Solution.**

**Definition 2.39.** Let  $f$  be a **one-to-one function** with domain  $X$  and range  $Y$ . Then its **inverse function**  $f^{-1}$  has domain  $Y$  and range  $X$  and is defined by

$$f^{-1}(y) = x \Leftrightarrow f(x) = y, \quad (2.29)$$

for any  $y \in Y$ .



**Solution.** Write  $y = x^3 + 2$ .

**Step 1:** Solve it for  $x$ :

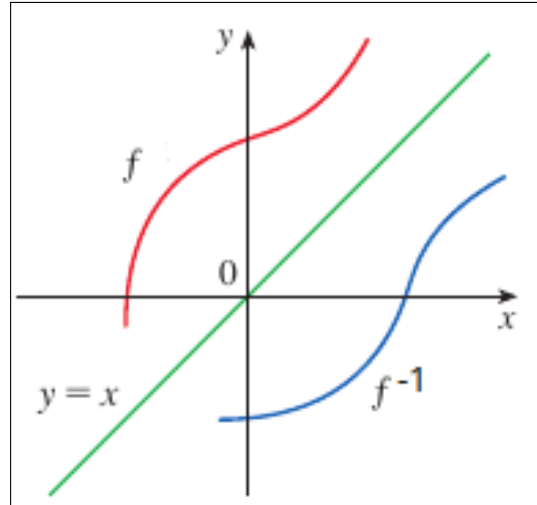
$$x^3 = y - 2 \Rightarrow x = \sqrt[3]{y - 2}.$$

**Step 2:** Exchange  $x$  and  $y$ :

$$y = \sqrt[3]{x - 2}.$$

Therefore the inverse function is

$$f^{-1}(x) = \sqrt[3]{x - 2}.$$



## Exponential Functions

**Definition 2.40.** A function of the form

$$f(x) = a^x, \quad \text{where } a > 0 \text{ and } a \neq 1, \quad (2.30)$$

is called an **exponential function** (with base  $a$ ).

- All exponential functions have domain  $(-\infty, \infty)$  and range  $(0, \infty)$ , so an exponential function never assumes the value 0.
- All exponential functions are either increasing ( $a > 1$ ) or decreasing ( $0 < a < 1$ ) over the whole domain.

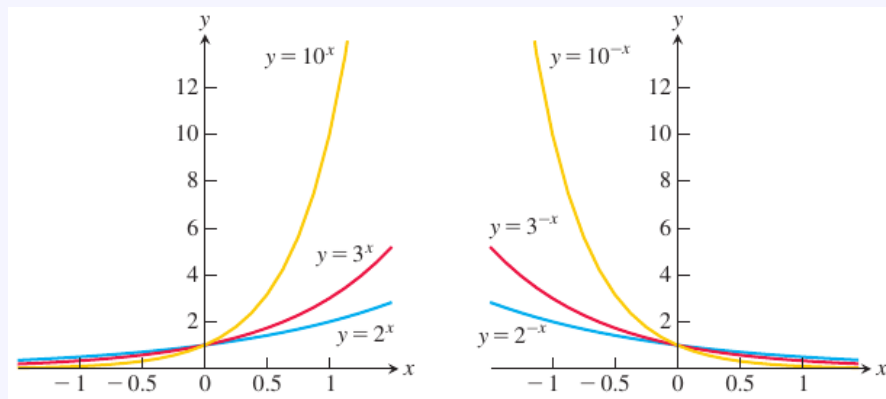


Figure 2.7: Exponential functions.

**Example 2.41.** Table 2.1 shows data for the **population of the world** in the 20th century. Figure 2.8 shows the corresponding **scatter plot**.

- The pattern of the data points suggests an exponential growth.
- Use an **exponential regression** algorithm to find a model of the form

$$P(t) = a \cdot b^t, \quad (2.31)$$

where  $t = 0$  corresponds to 1900.

Table 2.1

$t$ (years since 1900)	Population $P$ (millions)
0	1650
10	1750
20	1860
30	2070
40	2300
50	2560
60	3040
70	3710
80	4450
90	5280
100	6080
110	6870

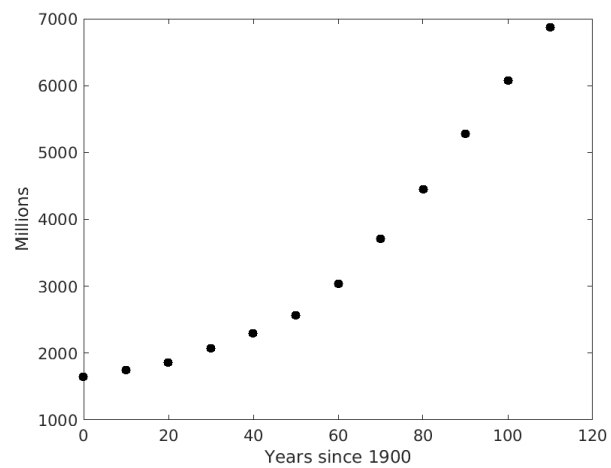


Figure 2.8: Scatter plot for world population growth.

**Remark 2.42.** The exponential regression (2.31) can be rewritten as

$$\ln P = \ln(a \cdot b^t) = \ln a + t \ln b = \alpha + t\beta. \quad (2.32)$$

One can find the parameters  $(\alpha, \beta)$  which fit best the following:

$$\left. \begin{array}{l} \alpha + 0\beta = \ln 1650 \\ \alpha + 10\beta = \ln 1750 \\ \alpha + 20\beta = \ln 1860 \\ \vdots \\ \alpha + 110\beta = \ln 6870 \end{array} \right\} \Rightarrow A \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \mathbf{r} \quad (2.33)$$

Then recover  $(a, b)$ :  $a = e^\alpha$ ,  $b = e^\beta$ .

**Solution.** We will see details of the exponential regression later.

```

population.m
1 Data =[0 1650; 10 1750; 20 1860; 30 2070;
2       40 2300; 50 2560; 60 3040; 70 3710;
3       80 4450; 90 5280; 100 6080; 110 6870];
4 m = size(Data,1);
5
6 % exponential model, through linearization
7 A = ones(m,2);
8 A(:,2) = Data(:,1);
9 r = log(Data(:,2));
10 p = (A'*A)\(A'*r);
11 a = exp(p(1)), b = exp(p(2)),
12
13 plot(Data(:,1),Data(:,2),'k.','MarkerSize',20)
14     xlabel('Years since 1900');
15     ylabel('Millions'); hold on
16     print -dpng 'population-data.png'
17 t = Data(:,1);
18 plot(t,a*b.^t,'r-','LineWidth',2)
19     print -dpng 'population-regression.png'
20     hold off

```

The program results in

$$a = 1.4365 \times 10^3, \quad b = 1.0140.$$

Thus the exponential model reads

$$P(t) = (1.4365 \times 10^9) \cdot (1.0140)^t. \quad (2.34)$$

Figure 2.9 shows the graph of this exponential function together with the original data points. We see that the exponential curve fits the data reasonably well.  $\square$

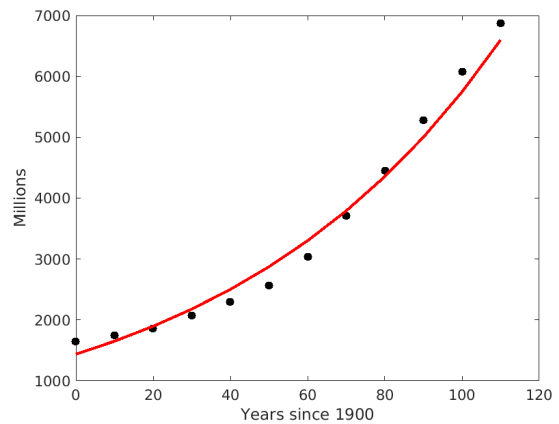


Figure 2.9: Exponential model for world population growth.

## The Number $e$

Of all possible bases for an exponential function, there is one that is most convenient for the purposes of calculus. The choice of a base  $a$  is influenced by the way the graph of  $y = a^x$  crosses the  $y$ -axis.

- Some of the formulas of calculus will be greatly simplified, if we choose the base  $a$  so that **the slope of the tangent line to  $y = a^x$  at  $x = 0$  is exactly 1**.
- In fact, there is such a number and it is denoted by the letter  $e$ . (This notation was chosen by the Swiss mathematician **Leonhard Euler** in 1727, probably standing for exponential.)
- It turns out that the number  $e$  lies between 2 and 3:

$$e \approx 2.718282 \quad (2.35)$$

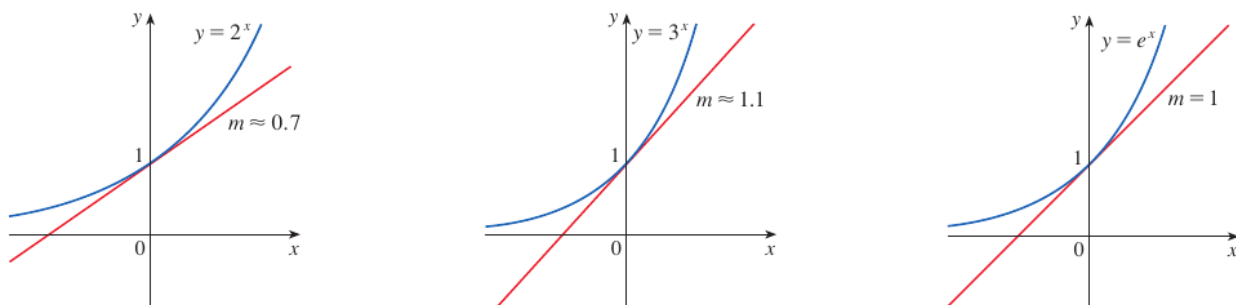


Figure 2.10: The number  $e$ .

### Remark 2.43. The Euler's number $e$ as a Limit

It can be calculated as the limit

$$e = \lim_{x \rightarrow 0} (1 + x)^{1/x}. \quad (2.36)$$

```

1  % An increasing sequence
2
3  for n=1:8
4      x=1/10^n;
5      en = (1+x)^(1/x);
6      fprintf('e_%d = %.10f\n',n,en)
7  end

```

```

1  e_1 = 2.5937424601
2  e_2 = 2.7048138294
3  e_3 = 2.7169239322
4  e_4 = 2.7181459268
5  e_5 = 2.7182682372
6  e_6 = 2.7182804691
7  e_7 = 2.7182816941
8  e_8 = 2.7182817983

```

## 2.5.2. Logarithmic Functions

**Recall:** The exponential function  $f(x) = a^x$  is either increasing ( $a > 1$ ) or decreasing ( $0 < a < 1$ ).

- It is one-to-one by the Horizontal Line Test.
- Therefore it has its inverse.

**Definition 2.44.** The **logarithmic function with base  $a$** , written  $y = \log_a x$ , is the inverse of  $y = a^x$  ( $a > 0$ ,  $a \neq 1$ ). That is,

$$y = \log_a x \Leftrightarrow a^y = x. \quad (2.37)$$

**Example 2.45.** Find the inverse of  $y = 2^x$ .

**Solution.**

1. **Solve  $y = 2^x$  for  $x$ :**

$$x = \log_2 y$$

2. **Exchange  $x$  and  $y$ :**

$$y = \log_2 x$$

Thus the graph of  $y = \log_2 x$  must be the reflection of the graph of  $y = 2^x$  about  $y = x$ .

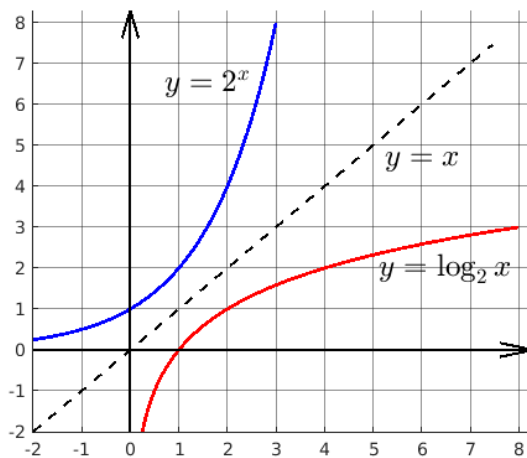


Figure 2.11: Graphs of  $y = 2^x$  and  $y = \log_2 x$ .

**Note:**

- Equation (2.37) represents the action of “**solving for  $x$** ”
- The domain of  $y = \log_a x$  must be the range of  $y = a^x$ , which is  $(0, \infty)$ .

## The Natural Logarithm and the Common Logarithm

Of all possible bases  $a$  for logarithms, we will see later that the most convenient choice of a base is the number  $e$ .

### Definition 2.46.

- The logarithm with base  $e$  is called the **natural logarithm** and has a special notation:

$$\log_e x = \ln x \quad (2.38)$$

- The logarithm with base 10 is called the **common logarithm** and has a special notation:

$$\log_{10} x = \log x \quad (2.39)$$

### Remark 2.47.

- From your calculator, you can see buttons of **LN** and **LOG**, which represent  $\ln = \log_e$  and  $\log = \log_{10}$ , respectively.
- When you implement **a code on computers**, the functions  $\ln$  and  $\log$  can be called by “log” and “log10”, respectively.

## Properties of Logarithms

- Algebraic Properties:** for ( $a > 0, a \neq 1$ )

---


$$\text{Product Rule: } \log_a xy = \log_a x + \log_a y$$

$$\text{Quotient Rule: } \log_a \frac{x}{y} = \log_a x - \log_a y$$

$$\text{Power Rule: } \log_a x^\alpha = \alpha \log_a x$$

$$\text{Reciprocal Rule: } \log_a \frac{1}{x} = -\log_a x$$


---

(2.40)

- Inverse Properties**

---


$$a^{\log_a x} = x, \quad x > 0; \quad \log_a a^x = x, \quad x \in \mathbb{R}$$

$$e^{\ln x} = x, \quad x > 0; \quad \ln e^x = x, \quad x \in \mathbb{R}$$


---

(2.41)

**Example 2.48.** Solve for  $x$ .

(a)  $e^{5-3x} = 3$ .

(b)  $\log_3 x + \log_3(x - 2) = 1$

(c)  $\ln(\ln x) = 0$

**Solution.**

*Ans:* (a)  $x = \frac{1}{3}(5 - \ln 3)$ . (b)  $x = 3$ . (**Caution:**  $x = -1$  cannot be a solution.)

**Claim 2.49.**

(a) Every exponential function is a power of the natural exponential function.

$$a^x = e^{x \ln a}. \quad (2.42)$$

(b) Every logarithmic function is a constant multiple of the natural logarithm.

$$\log_a x = \frac{\ln x}{\ln a}, \quad (a > 0, a \neq 1) \quad (2.43)$$

which is called the **Change-of-Base Formula**.

**Proof.** (a).  $a^x = e^{\ln(a^x)} = e^{x \ln a}$ .

(b).  $\ln x = \ln(a^{\log_a x}) = (\log_a x)(\ln a)$ , from which one can get (2.43).  $\square$

**Remark 2.50.** Based on Claim 2.49, all exponential and logarithmic functions can be evaluated by the natural exponential function and the natural logarithmic function; which are named “exp()” and “log()”, in Matlab.

**Note:** You will work on a project, *Canny Edge Detection Algorithm For Color Images*, while you are studying the next chapter.

## Exercises for Chapter 2

2.1. Download a dataset saved in `heart-data.txt`:

<https://skim.math.msstate.edu/LectureNotes/data/heart-data.txt>

- (a) Draw a figure for it.
- (b) Use the formula (2.4) to find the area.

**Hint:** You may use the following. You should finish the function `area_closed_curve`. Note that the index in Matlab arrays begins with 1, not 0.

```

                                heart.m
1  DATA = load('heart-data.txt');
2
3  X = DATA(:,1); Y = DATA(:,2);
4  figure, plot(X,Y,'r-','linewidth',2);
5
6  [m,n] = size(DATA);
7  area = area_closed_curve(DATA);
8
9  fprintf('# of points = %d; area = %g\n',m,area);

```

```

                                area_closed_curve.m
1  function area = area_closed_curve(data)
2  % compute the area of a region of closed curve
3
4  [m,n] = size(data);
5  area = 0;
6
7  for i=2:m
8      %FILL HERE APPROPRIATELY
9  end

```

Ans: (b) 9.41652.

2.2. Function  $f(x) = x^3 - 2x^2 + x - 2$  has two complex-values zeros and a real zero. Implement a code to visualize all the solutions in the complex coordinates.

**Hint:** Find the real and imaginary parts of  $f(z)$  as in Remark 2.10.

2.3. Either produce or download a sound file and then compute its spectrogram. For a wav file, you may try <https://www2.cs.uic.edu/~i101/SoundFiles/> or <https://voiceage.com/Audio-Samples-AMR-WB.html>.

**Hint:** Assume you have got `StarWars3.wav`. Then you may start with the following.

```

                                real_STFT.m
1  filename = 'StarWars3.wav';
2  [x,fs] = audioread(filename);
3

```



```

4 | gong = audioplayer(x,fs);
5 | play(gong)

```

The above works on both Matlab and Octave.

2.4. For the pair  $(x_n, \alpha_n)$ , is it true that  $x_n = \mathcal{O}(\alpha_n)$  as  $n \rightarrow \infty$ ?

- (a)  $x_n = \sqrt{n^2 - 10}$ ;  $\alpha_n = \sqrt{n}$
- (b)  $x_n = 3n - n^4 + 1$ ;  $\alpha_n = n^3$
- (c)  $x_n = n - \frac{1}{\sqrt{n}} + 1$ ;  $\alpha_n = \sqrt{n}$
- (d)  $x_n = n^2 + n$ ;  $\alpha_n = n^3$

2.5. The population of **Starkville**, Mississippi, was 2,689 in the year 1900 and 25,495 in 2020. Assume that the population in Starkville grows exponentially with the model

$$P_n = P_0 \cdot (1 + r)^n, \quad (2.44)$$

where  $n$  is the elapsed year and  $r$  denotes the growth rate per year.

- (a) Find the growth rate  $r$ .
- (b) Estimate the population in 1950 and 2000.
- (c) Approximately when is the population going to reach 50,000?

**Hint:** Applying the natural log to (2.44) reads  $\log(P_n/P_0) = n \log(1 + r)$ . Dividing it by  $n$  and applying the natural exponential function gives  $1 + r = \exp(\log(P_n/P_0)/n)$ , where  $P_n = 25495$ ,  $P_0 = 2689$ , and  $n = 120$ .

*Ans:* (a)  $r = 0.018921 (= 1.8921\%)$ . (c) 2056.



## CHAPTER 3

# Programming with Calculus

In modern scientific computing (particularly, for AI), **calculus** and **linear algebra** play crucial roles.

- In this chapter, you will learn **certain selected topics in calculus** which are essential for advanced computing tasks.
- We will consider basic concepts and their applications as well.

### Contents of Chapter 3

3.1. Differentiation . . . . .	66
3.2. Basis Functions and Taylor Series . . . . .	76
3.3. Polynomial Interpolation . . . . .	86
3.4. Numerical Differentiation: Finite Difference Formulas . . . . .	93
3.5. Newton's Method for the Solution of Nonlinear Equations . . . . .	99
3.6. Zeros of Polynomials . . . . .	104
Exercises for Chapter 3 . . . . .	109

## 3.1. Differentiation

### 3.1.1. The Slope of the Tangent Line

**Problem 3.1.** A function  $y = f(x)$  can be graphed as a curve.

- In many applications, the **tangent line** plays a crucial role for the computation of approximate solutions.
- Here are questions:
  - What is the tangent line?
  - How can we find it?

#### Average Speed and Instantaneous Speed

When  $f(t)$  measures the distance traveled at time  $t$ ,

- **Average Speed.**

$$\text{Average speed over } [t_0, t_0 + h] = \frac{\text{distance traveled}}{\text{elapsed time}} = \frac{f(t_0 + h) - f(t_0)}{(t_0 + h) - t_0} \quad (3.1)$$

- **Instantaneous Speed.** For  $h$  very small,

$$\text{Instantaneous speed at } t_0 \approx \frac{f(t_0 + h) - f(t_0)}{h} \quad (3.2)$$

**Example 3.2.** If  $y$  denotes the distance fallen in feet after  $t$  seconds, then the **Galileo's law of free-fall** is

$$y = 16t^2 \text{ ft.} \quad (3.3)$$

Let  $t_0 = 1$ .

- (a) Find average speed, the **difference quotient**,

$$\frac{f(t_0 + h) - f(t_0)}{h}$$

for various  $h$ , positive and negative.

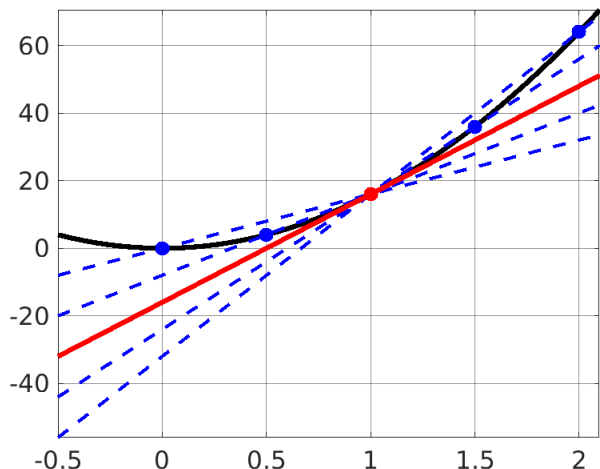
- (b) Estimate the instantaneous speed at  $t = t_0$ .

**Solution.**

```

                                free_fall.m
1  syms f(t) Q(h)    %also, views t and h as symbols
2
3  f(t) = 16*t.^2; t0=1;
4  Int = [t0-1.5,t0+1.1];
5  fplot(f(t),Int, 'k-', 'LineWidth',3)
6  hold on
7
8  %%---- Difference Quotient (DQ) -----
9  Q(h) = (f(t0+h)-f(t0))/h;
10 S(t,h) = Q(h)*(t-t0)+f(t0); % Secant line
11
12 %%---- Secant Lines, with Various h -----
13 for h0 = [-1 -0.5 0.5 1]
14     fplot(S(t,h0),Int, 'b--', 'LineWidth',2)
15     plot([t0+h0], [f(t0+h0)], 'b.', 'markersize',30)
16 end
17
18 %%---- Limit of the DQ -----
19 tan_slope = limit(Q(h),h,0);
20 T(t) = tan_slope*(t-t0)+f(t0);
21 fplot(T(t),Int, 'r-', 'LineWidth',3)
22 plot([t0], [f(t0)], 'r.', 'markersize',30)
23
24 axis tight, grid on; hold off
25 ax=gca; ax.FontSize=15; ax.GridAlpha=0.5;
26 print -dpng 'free-fall-tangent.png'
27
28 %%---- Measure Q(h) wih h=+-10^-i -----
29 for i = 1:5
30     h=-10^(-i); fprintf(" h= %.5f; Q(h) = %.8f\n",h,Q(h))
31 end
32 for i = 1:5
33     h=10^(-i); fprintf(" h= %.5f; Q(h) = %.8f\n",h,Q(h))
34 end

```



Difference Quotient at $t_0 = 1$	
1	$h = -0.10000$ ; $Q(h) = 30.40000000$
2	$h = -0.01000$ ; $Q(h) = 31.84000000$
3	$h = -0.00100$ ; $Q(h) = 31.98400000$
4	$h = -0.00010$ ; $Q(h) = 31.99840000$
5	$h = -0.00001$ ; $Q(h) = 31.99984000$
6	$h = 0.10000$ ; $Q(h) = 33.60000000$
7	$h = 0.01000$ ; $Q(h) = 32.16000000$
8	$h = 0.00100$ ; $Q(h) = 32.01600000$
9	$h = 0.00010$ ; $Q(h) = 32.00160000$
10	$h = 0.00001$ ; $Q(h) = 32.00016000$

**Let's confirm this algebraically.**

- When  $f(t) = 16t^2$  and  $t_0 = 1$ , the difference quotient reads

$$\begin{aligned}
 \frac{\Delta y}{\Delta t}(t_0 = 1) &= \frac{f(1+h) - f(1)}{h} = \frac{16(1+h)^2 - 16(1)^2}{h} \\
 &= \frac{16(1 + 2h + h^2) - 16(1)^2}{h} = \frac{32h + 16h^2}{h} & (3.4) \\
 &= 32 + 16h
 \end{aligned}$$

- As  $h$  gets closer and closer to 0, the average speed has the limiting value 32 ft/sec when  $t_0 = 1$  sec.  $\square$
- **Thus, the slope of the tangent line is 32.**

**Example 3.3.** Find an equation of the tangent line to the graph of  $y = x^2$  at  $x_0 = 2$ .

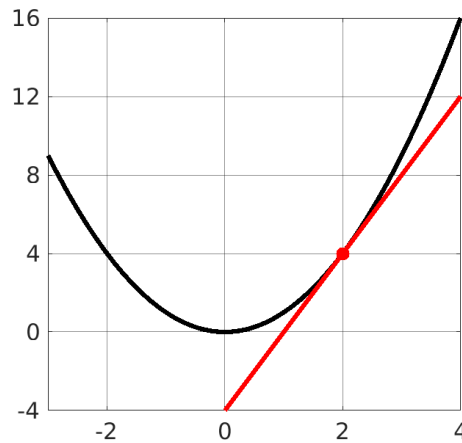


Figure 3.1: Graph of  $y = x^2$  and the tangent line at  $x_0 = 2$ .

**Solution.** Let's first try to find the slope, as the limit of the difference quotient.

**Definition 3.4.**

The **slope of the curve**  $y = f(x)$  at the point  $P(x_0, f(x_0))$  is the number

$$\lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h} \quad (=: \mathbf{f}'(\mathbf{x}_0), \text{ provided the limit exists}). \quad (3.5)$$

The **tangent line** to the curve at  $P$  is the line through  $P$  with this slope:

$$y - f(x_0) = \mathbf{f}'(\mathbf{x}_0)(x - x_0). \quad (3.6)$$

**Example 3.5.** Can you find the tangent line to  $y = |x^2 - 1|$  at  $x_0 = 1$ ?

**Solution.** As one can see from Figure 3.2, the **left-hand limit** and the **right-hand slope** of the difference quotient are not the same. Or, you may say the left-hand and the right-hand secant lines converge differently. Thus no tangent line can be defined.

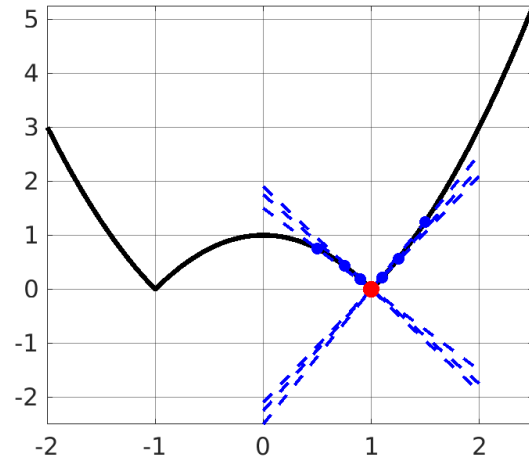


Figure 3.2: Graph of  $y = |x^2 - 1|$  and secant lines at  $x_0 = 1$ .

```

secant_lines_abs_x2_minus_1.m
1  syms f(x) Q(h) %also, views t and h as symbols
2
3  f(x)=abs(x.^2-1); x0=1;
4  figure, fplot(f(x),[x0-3,x0+1.5], 'k-', 'LineWidth', 3)
5  hold on
6
7  Q(h) = (f(x0+h)-f(x0))/h;
8  S(x,h) = Q(h)*(x-x0)+f(x0); % Secant line
9  %%---- Secant Lines, with Various h -----
10 for h0 = [-0.5 -0.25 -0.1 0.1 0.25 0.5]
11     fplot(S(x,h0),[x0-1,x0+1], 'b--', 'LineWidth', 2)
12     plot([x0+h0], [f(x0+h0)], 'b.', 'markersize', 25)
13 end
14 plot([x0], [f(x0)], 'r.', 'markersize', 35)
15 daspect([1 2 1])
16 axis tight, grid on
17 ax=gca; ax.FontSize=15; ax.GridAlpha=0.5;
18 hold off
19 print -dpng 'secant-y=abs-x2-1.png'

```



### 3.1.2. Derivative and Differentiation Rules

You have calculated **average slopes**, for various interval lengths  $h$ , and estimated the **instantaneous slope** by letting  $h$  approach zero.

**Definition 3.6.** The **derivative** of a function  $f(x)$ , denoted  $f'(x)$  or  $\frac{df(x)}{dx}$ , is

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}, \quad (3.7)$$

provided that the limit exists.

**Example 3.7.** Use the definition to find derivatives of the functions:

Find the difference quotient  $\frac{f(x+h)-f(x)}{h}$ , simplify it, and then apply  $\lim_{h \rightarrow 0}$ .

(a)  $f(x) = x$

(b)  $f(x) = x^2$

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

**Solution.**

**Formula 3.8.** From the last example,

$$\begin{aligned} f(x) = x &\Rightarrow f'(x) = 1 \\ f(x) = x^2 &\Rightarrow f'(x) = 2x \\ f(x) = x^3 &\Rightarrow f'(x) = 3x^2 \\ &\vdots \\ f(x) = x^n &\Rightarrow f'(x) = nx^{n-1} \end{aligned}$$

**Example 3.9.** Differentiate the following powers of  $x$ .

(a)  $x^6$

(b)  $x^{3/2}$

(c)  $x^{1/2}$

**Solution.**

**Formula 3.10. Differentiation Rules:**

- Let  $f(x) = au(x) + bv(x)$ , for some constants  $a$  and  $b$ . Then

$$\begin{aligned} \frac{f(x+h) - f(x)}{h} &= \frac{[au(x+h) + bv(x+h)] - [au(x) + bv(x)]}{h} \\ &= a \frac{u(x+h) - u(x)}{h} + b \frac{v(x+h) - v(x)}{h} \\ &\rightarrow au'(x) + bv'(x) \end{aligned} \quad (3.8)$$

- Let  $f(x) = u(x)v(x)$ . Then

$$\begin{aligned} \frac{f(x+h) - f(x)}{h} &= \frac{u(x+h)v(x+h) - u(x)v(x)}{h} \\ &= \frac{u(x+h)v(x+h) - u(x)v(x+h) + u(x)v(x+h) - u(x)v(x)}{h} \\ &= v(x+h) \frac{u(x+h) - u(x)}{h} + u(x) \frac{v(x+h) - v(x)}{h} \\ &\rightarrow u'(x)v(x) + u(x)v'(x) \end{aligned} \quad (3.9)$$

**Example 3.11.** Use the product rule (3.9) to find the derivative of the function

$$f(x) = x^6 = x^2 \cdot x^4$$

**Solution.**

**Example 3.12.** Does the curve  $y = x^4 - 2x^2 + 2$  have any horizontal tangent line? Use the information you just found, to sketch the graph.

**Solution.**

**Example 3.13.** Consider a computer program.

```

----- derivative_rules.m -----
1  syms n a b real
2  syms u(x) v(x)
3  syms f(x) Q(h)
4
5  %%-- Define f(x) -----
6  f(x) = x^n;
7
8  %%-- Define Q(h) and take limit -----
9  Q(h) = (f(x+h)-f(x))/h;
10 fp = limit(Q(h),h,0); simplify(fp)

```

Use the program to verify various rules of differentiation.

**Solution.**

Table 3.1: Rules of Derivative

$f(x)$	Results	Mathematical formula	
$x^n$	$n \cdot x^{(n-1)}$	$(x^n)' = nx^{n-1}$	<b>(power rule)</b>
$a \cdot u(x) + b \cdot v(x)$	$a \cdot D(u)(x) + b \cdot D(v)(x)$	$(au + bv)' = au' + bv'$	<b>(linearity rule)</b>
$u(x) \cdot v(x)$	$D(u)(x) \cdot v(x) + u(x) \cdot D(v)(x)$	$(uv)' = u'v + uv'$	<b>(product rule)</b>
$u(x)/v(x)$	$(D(u)(x) \cdot v(x) - D(v)(x) \cdot u(x)) / v(x)^2$	$(u/v)' = (u'v - uv') / v^2$	<b>(quotient rule)</b>
$u(v(x))$	$D(v)(x) \cdot D(u)(v(x))$	$(u(v(x)))' = u'(v(x)) \cdot v'(x)$	<b>(chain rule)</b>

**Example 3.14.** Find the derivative of  $g(x) = (2x + 1)^{10}$ .

**Solution.** Let  $u(x) = x^{10}$  and  $v(x) = 2x + 1$ . Then  $g(x) = u(v(x))$ .

- $u'(x) = 10x^9$  and  $v'(x) = 2$ .
- Thus

$$g'(x) = u'(v(x)) \cdot v'(x) = 10(v(x))^9 \cdot 2 = 20(2x + 1)^9.$$

**Remark 3.15. Differentiation Rules:**

- The **power rule** holds for **real number**  $n$ .
- The **quotient rule** can be explained using the **product rule**.

$$\begin{aligned} \left(\frac{u}{v}\right)' &= (u \cdot v^{-1})' = u' \cdot v^{-1} + u \cdot (v^{-1})' \\ &= u' \cdot v^{-1} + u \cdot (-v^{-2}v') = \frac{u'}{v} - \frac{u \cdot v'}{v^2} \\ &= \frac{u'v - uv'}{v^2} \end{aligned} \quad (3.10)$$

- The **chain rule** can be verified algebraically.

$$\begin{aligned} \frac{du(v(x))}{dx} &= \lim_{h \rightarrow 0} \frac{u(v(x+h)) - u(v(x))}{h} \\ &= \lim_{h \rightarrow 0} \frac{u(v(x+h)) - u(v(x))}{v(x+h) - v(x)} \cdot \frac{v(x+h) - v(x)}{h} \\ &= u'(v(x)) \cdot v'(x). \end{aligned} \quad (3.11)$$

Here  $u'(v(x))$  means the rate of change of  $u$  **with respect to**  $v(x)$ .

- We may rewrite (3.11):

$$\frac{du(v(x))}{dx} = \lim_{\Delta x \rightarrow 0} \frac{\Delta u}{\Delta x} = \lim_{\Delta x \rightarrow 0} \frac{\Delta u}{\Delta v} \cdot \frac{\Delta v}{\Delta x} = u'(v(x)) \cdot v'(x). \quad (3.12)$$

**Self-study 3.16.** Find the derivative of the functions.

(a)  $f(x) = (3x - 1)^7 x^5$

(b)  $g(x) = \frac{(3x - 1)^7}{x^5}$

**Solution.**

*Ans:* (b)  $((3x - 1)^6 (6x + 5))/x^6$

## 3.2. Basis Functions and Taylor Series

### 3.2.1. Change of Variables & Basis Functions

#### **In-Reality** 3.17. Two Major Mathematical Techniques.

In the history of the engineering **research and development (R&D)**, there have been two major mathematical techniques: the **change of variables** and **representation by basis functions**.

- In a nut shell, the **change of variables** is a basic technique used **to simplify problems**.
- A function can be **either represented or approximated** by a linear combination of **basis functions**.

**Example** 3.18. Find solutions of the system of equations

$$\begin{cases} xy + 2x + 2y = 20 \\ x^2y + xy^2 = 48 \end{cases} \quad (3.13)$$

where  $x$  and  $y$  are positive real numbers with  $x < y$ .

**Solution.** Let  $s = xy$  and  $t = x + y$  (a change of variables).

*Ans:*  $(x, y) = (2, 4)$

**Note:** Most subjects in Calculus, particularly Vector Calculus, are deeply related to the change of variables, to handle differentiation and integration over general 2D and 3D domains more effectively.

**Definition 3.19.** A **basis** for a vector space  $V$  is a **set of vectors** (functions) that

1. is *linearly independent*, and
2. *spans*  $V$ .

**Note:** *Linear Independence* and *Span* are defined in § 4.3.

**Example 3.20. Basis Functions**

- The **monomial basis** for the polynomial space  $\mathbb{P}_n$  is given by

$$\{1, x, x^2, \dots, x^n\}. \quad (3.14)$$

Each polynomial  $p \in \mathbb{P}_n$  is expressed as a linear combination of the monomial basis functions:

$$p = c_0 + c_1x + c_2x^2 + \dots + c_nx^n. \quad (3.15)$$

- The **standard unit vectors** in  $\mathbb{R}^n$  are

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \dots, \quad \mathbf{e}_n = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \quad (3.16)$$

which form the **standard basis for**  $\mathbb{R}^n$ ; any  $\mathbf{x} \in \mathbb{R}^n$  can be written as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \dots + x_n \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2 + \dots + x_n\mathbf{e}_n.$$

- Various other bases can be formulated.

### 3.2.2. Power Series and the Ratio Test

The **monomial basis** for analytic functions is given by

$$\{1, x, x^2, \dots\}. \quad (3.17)$$

This basis is used in power series and Taylor series.

**Definition 3.21.** A **power series about**  $x = 0$  is a series of the form

$$\sum_{n=0}^{\infty} c_n x^n = c_0 + c_1 x + c_2 x^2 + \dots + c_n x^n + \dots. \quad (3.18)$$

A **power series about**  $x = a$  is a series of the form

$$\sum_{n=0}^{\infty} c_n (x - a)^n = c_0 + c_1 (x - a) + c_2 (x - a)^2 + \dots + c_n (x - a)^n + \dots, \quad (3.19)$$

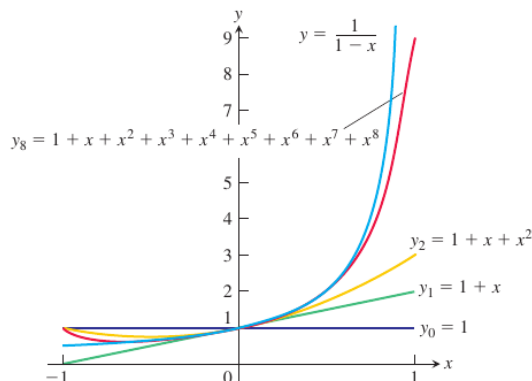
in which the **center**  $a$  and the **coefficients**  $c_0, c_1, c_2, \dots, c_n, \dots$  are constants.

**Example 3.22.** Taking all the coefficients to be 1 in (3.18) gives the geometric series

$$\sum_{n=0}^{\infty} x^n = 1 + x + x^2 + \dots + x^n + \dots,$$

which converges to  $1/(1 - x)$  **only if**  $|x| < 1$ . That is,

$$\frac{1}{1 - x} = 1 + x + x^2 + \dots + x^n + \dots, \quad |x| < 1. \quad (3.20)$$





**Remark 3.23.** It follows from Example 3.20 that

1. A function can be approximated by a power series.
2. A power series may converge only on a certain interval, of radius  $R$ .

**Theorem 3.24. The Ratio Test:**

Let  $\sum a_n$  be any series and suppose that

$$\lim_{n \rightarrow \infty} \left| \frac{a_{n+1}}{a_n} \right| = \rho. \quad (3.21)$$

- (a) If  $\rho < 1$ , then the series **converges absolutely**. ( $\sum |a_n|$  converges)
- (b) If  $\rho > 1$ , then the series **diverges**.
- (c) If  $\rho = 1$ , then the test is **inconclusive**.

**Example 3.25.** For what values of  $x$  do the following power series converge?

$$(a) \sum_{n=1}^{\infty} (-1)^{n-1} \frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots \quad (b) \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

**Solution.**

*Ans: (a)  $x \in (-1, 1]$ .*

**Theorem 3.26. Term-by-Term Differentiation.**

If  $\sum_{n=0}^{\infty} c_n(x-a)^n$  has **radius of convergence**  $R > 0$ , it defines a function

$$f(x) = \sum_{n=0}^{\infty} c_n(x-a)^n \quad \text{on} \quad |x-a| < R. \quad (3.22)$$

This function  $f$  has derivatives of all orders inside the interval, and we obtain the derivatives by differentiating the original series term by term:

$$\begin{aligned} f'(x) &= \sum_{n=1}^{\infty} n c_n (x-a)^{n-1}, \\ f''(x) &= \sum_{n=2}^{\infty} n(n-1) c_n (x-a)^{n-2}, \end{aligned} \quad (3.23)$$

and so on. Each of these derived series converges at every point of the interval  $a - R < x < a + R$ .

This theorem similarly holds for **Term-by-Term Integration**.

**Example 3.27.** A power series is given as in Example 3.25 (b):

$$\sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Find its derivative.

**Solution.**

### 3.2.3. Taylor Series Expansion

**Remark 3.28.** We have seen how a **converging power series** defines a function. In order to make infinite series more useful:

- Here we will try to express a **given function** as an infinite series called the **Taylor series**.
- In many cases, the Taylor series provides useful **polynomial approximation** of the original function.
- Because approximation by polynomials is extremely useful to both mathematicians and scientists, Taylor series are at the core of the theory of infinite series.

#### Series Representations

**Key Idea 3.29. Taylor Series.**

- Let's assume that a **given function**  $f$  is expressed as a power series about  $x = a$ :

$$\begin{aligned} f(x) &= \sum_{n=0}^{\infty} c_n (x-a)^n \\ &= c_0 + c_1(x-a) + c_2(x-a)^2 + \cdots + c_n(x-a)^n + \cdots, \end{aligned} \quad (3.24)$$

with a positive radius of convergence  $R > 0$ . ⇒ What are  $c_n$ ?

- Term-by-term derivatives read

$$\begin{aligned} f'(x) &= c_1 + 2c_2(x-a) + 3c_3(x-a)^2 + \cdots + nc_n(x-a)^{n-1} + \cdots \\ f''(x) &= 2c_2 + 3 \cdot 2c_3(x-a) + \cdots + n(n-1)c_n(x-a)^{n-2} + \cdots \\ f'''(x) &= 3 \cdot 2c_3 + 4 \cdot 3 \cdot 2c_4(x-a) + \cdots + n(n-1)(n-2)c_n(x-a)^{n-3} + \cdots \end{aligned} \quad (3.25)$$

with the  $n$ th derivative being

$$f^{(n)}(x) = n!c_n + (n+1)!c_{n+1}(x-a) + \cdots \quad (3.26)$$

- Thus, when  $x = a$ ,

$$f^{(n)}(a) = n!c_n \Rightarrow c_n = \frac{f^{(n)}(a)}{n!}. \quad (3.27)$$

**Definition 3.30.** Let  $f$  be a function with derivatives of all orders throughout some interval containing  $a$  as an interior point. Then the **Taylor series** generated by  $f$  at  $x = a$  is

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!} (x-a)^2 + \dots \quad (3.28)$$

The **Maclaurin series** of  $f$  is the Taylor series generated by  $f$  at  $x = 0$ :

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n = f(0) + f'(0)x + \frac{f''(0)}{2!} x^2 + \dots \quad (3.29)$$

**Example 3.31.** Find the Taylor series and Taylor polynomials generated by  $f(x) = \cos x$  at  $x = 0$ .

**Solution.** The cosine and its derivatives are

$$\begin{aligned} f(x) &= \cos x & f'(x) &= -\sin x \\ f''(x) &= -\cos x & f^{(3)}(x) &= \sin x \\ &\vdots & &\vdots \\ f^{(2n)}(x) &= (-1)^n \cos x & f^{(2n+1)}(x) &= (-1)^{n+1} \sin x. \end{aligned}$$

At  $x = 0$ , the cosines are 1 and the sines are 0, so

$$f^{(2n)}(0) = (-1)^n, \quad f^{(2n+1)}(0) = 0. \quad (3.30)$$

The Taylor series generated by  $\cos x$  at  $x = 0$  is

$$1 + 0 \cdot x - \frac{1}{2!} x^2 + \frac{0}{3!} x^3 + \frac{1}{4!} x^4 + \dots = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \quad (3.31)$$

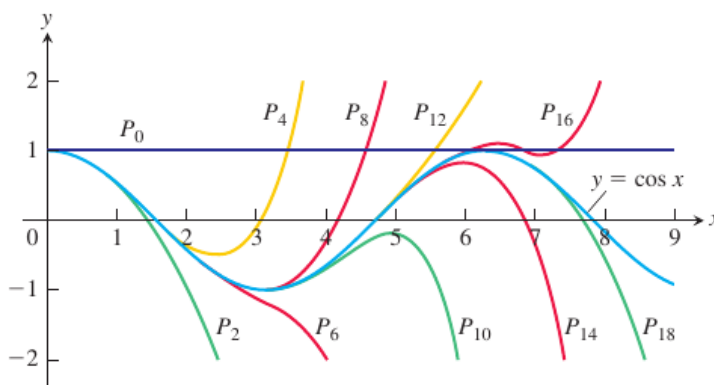


Figure 3.3:  $y = \cos x$  and its Taylor polynomials near  $x = 0$ .

### Commonly Used Taylor Series

Function	Series	Convergence
$\frac{1}{1-x}$	$= 1 + x + x^2 + x^3 + \dots = \sum_{n=0}^{\infty} x^n$	$x \in (-1, 1)$
$e^x$	$= 1 + x + \frac{x^2}{2!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$	$x \in \mathbb{R}$
$\cos x$	$= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$	$x \in \mathbb{R}$
$\sin x$	$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$	$x \in \mathbb{R}$
$\ln(1+x)$	$= x - \frac{x^2}{2} + \frac{x^3}{3} - \dots = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n}$	$x \in (-1, 1]$
$\tan^{-1} x$	$= x - \frac{x^3}{3} + \frac{x^5}{5} - \dots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{2n+1}$	$x \in [-1, 1]$

(3.32)

**Note:** The **interval of convergence** can be verified using e.g. the **ratio test**, presented in Theorem 3.24, p. 79.

**Self-study 3.32.** Plot the **sinc function**  $f(x) = \frac{\sin x}{x}$  and its Taylor polynomials of order 4, 6, and 8, about  $x = 0$ .

**Solution.** *Hint:* Use e.g., `syms x; T4 = taylor(sin(x)/x,x,0,'Order',5)`. Here “Order” means the leading order of truncated terms.

### Taylor Polynomials

**Definition 3.33.** Let  $f$  be a function with derivatives of order  $k = 1, 2, \dots, N$  in some interval containing  $a$  as an interior point. Then for any integer  $n$  from 0 through  $N$ , the **Taylor polynomial of order  $n$**  generated by  $f$  at  $x = a$  is the polynomial

$$P_n(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \dots + \frac{f^{(n)}(a)}{n!}(x-a)^n. \quad (3.33)$$

**Theorem 3.34. Taylor's Theorem with Lagrange Remainder**

Suppose  $f \in C^n[a, b]$ ,  $f^{(n+1)}$  exists on  $(a, b)$ , and  $x_0 \in [a, b]$ . Then, for every  $x \in [a, b]$ ,

$$f(x) = \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k + \mathcal{R}_n(x), \quad (3.34)$$

where, for some  $\xi$  between  $x$  and  $x_0$ ,

$$\mathcal{R}_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1}.$$

**Note:** The above theorem is useful in various engineering applications; **the error of the polynomial approximation** can be verified by measuring the Lagrange Remainder.

**Example 3.35.** Let  $f(x) = \cos(x)$  and  $x_0 = 0$ . Determine the second and third Taylor polynomials for  $f$  about  $x_0$ .

Maple-code

```

1  f := x -> cos(x):
2  fp := x -> -sin(x):
3  fpp := x -> -cos(x):
4  fp3 := x -> sin(x):
5  fp4 := x -> cos(x):
6
7  p2 := x -> f(0) + fp(0)*x/1! + fpp(0)*x^2/2!:
8  p2(x);
9      = 1 - 1/2 x^2
10 R2 := fp3(xi)*x^3/3!;
11     = 1/6 sin(xi) x^3
12 p3 := x -> f(0) + fp(0)*x/1! + fpp(0)*x^2/2! + fp3(0)*x^3/3!:
13 p3(x);
14     = 1 - 1/2 x^2
15 R3 := fp4(xi)*x^4/4!;
16     = 1/24 cos(xi) x^4
17
18 # On the other hand, you can find the Taylor polynomials easily
19 # using built-in functions in Maple:
20 s3 := taylor(f(x), x = 0, 4);
21     = 1 - 1/2 x^2 + 0(x^4)
22 convert(s3, polynom);
23     = 1 - 1/2 x^2

```

```

1 plot([f(x), p3(x)], x = -2 .. 2, thickness = [2, 2],
2     linestyle = [solid, dash], color = black,
3     legend = ["f(x)", "p3(x)"],
4     legendstyle = [font = ["HELVETICA", 10], location = right])

```

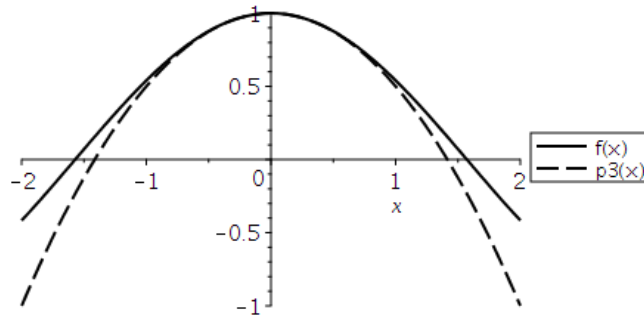


Figure 3.4:  $f(x) = \cos x$  and its third Taylor polynomial  $P_3(x)$ .

**Note:** When  $n = 0$ ,  $x = b$ , and  $x_0 = a$ , the Taylor's Theorem reads

$$f(b) = f(a) + \mathcal{R}_0(b) = f(a) + f'(c) \cdot (b - a), \quad (3.35)$$

equivalently,

$$f'(c) = \frac{f(b) - f(a)}{b - a}, \quad \text{for some } c \in (a, b), \quad (3.36)$$

which is the **Mean Value Theorem**.

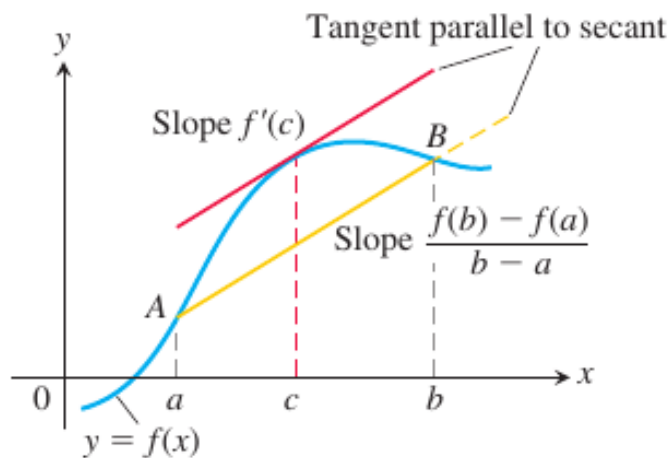


Figure 3.5: The Mean Value Theorem

### 3.3. Polynomial Interpolation

#### Polynomial Approximation and Interpolation

**Theorem 3.36. (Weierstrass Approximation Theorem)**

Suppose  $f \in C[a, b]$ . Then, for each  $\varepsilon > 0$ , there exists a polynomial  $P(x)$  such that

$$|f(x) - P(x)| < \varepsilon, \text{ for all } x \in [a, b]. \quad (3.37)$$

Every continuous function can be approximated by a polynomial, **arbitrarily close**.

**Theorem 3.37. (Polynomial Interpolation Theorem)**

If  $x_0, x_1, x_2, \dots, x_n$  are  $(n + 1)$  distinct real numbers, then for arbitrary values  $y_0, y_1, y_2, \dots, y_n$ , **there is a unique polynomial  $p_n$**  of degree at most  $n$  such that

$$p_n(x_i) = y_i \quad (0 \leq i \leq n). \quad (3.38)$$

The graph of  $y = p_n(x)$  passes all points  $\{(x_i, y_i)\}$ .

For values at  $(n + 1)$  distinct points, the interpolating polynomial  $p_n \in \mathbb{P}_n$  is **unique**.

**Example 3.38.** Find the interpolating polynomial  $p_2$  passing  $(-2, 3)$ ,  $(0, -1)$ , and  $(1, 0)$ .

**Solution.**



### 3.3.1. Lagrange Form of Interpolating Polynomials

Let data points  $(x_k, y_k)$ ,  $0 \leq k \leq n$  be given, where  $n + 1$  *abscissas*  $x_i$  are distinct. The interpolating polynomial will be sought in the form

$$p_n(x) = y_0L_{n,0}(x) + y_1L_{n,1}(x) + \cdots + y_nL_{n,n}(x) = \sum_{k=0}^n y_kL_{n,k}(x), \quad (3.39)$$

where  $L_{n,k}(x)$  are **basis polynomials** that depend on the nodes  $x_0, x_1, \dots, x_n$ , but not on the *ordinates*  $y_0, y_1, \dots, y_n$ .

See **Definition 3.19**, p.77, for basis.

For example, for  $\{(x_0, y_0), (x_1, y_1), (x_2, y_2)\}$ , the Lagrange form of interpolating polynomial reads

$$p_2(x) = y_0L_{2,0}(x) + y_1L_{2,1}(x) + y_2L_{2,2}(x). \quad (3.40)$$

#### How to Determine the Basis $\{L_{n,k}(x)\}$

**Observation 3.39.** Let all the ordinates be 0 except for a 1 occupying  $i$ -th position, **i.e.,  $y_i = 1$  and other ordinates are all zero.**

- Then,

$$p_n(x) = \sum_{k=0}^n y_kL_{n,k}(x) = L_{n,i}(x) \Rightarrow \mathbf{p}_n(\mathbf{x}_j) = \mathbf{L}_{n,i}(\mathbf{x}_j). \quad (3.41)$$

- On the other hand, the polynomial  $p_n$  interpolating the data must satisfy  $\mathbf{p}_n(\mathbf{x}_j) = \delta_{ij}$ , where  $\delta_{ij}$  is the **Kronecker delta**

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

- Thus all the basis polynomials must satisfy

$$L_{n,i}(x_j) = \delta_{ij}, \quad \text{for all } 0 \leq i, j \leq n. \quad (3.42)$$

Polynomials satisfying such a property are known as the **cardinal functions**.

**Example 3.40. Construction of  $L_{n,0}(x)$ :** It is to be an  $n$ th-degree polynomial that takes the value 0 at  $x_1, x_2, \dots, x_n$  and the value 1 at  $x_0$ . Clearly, it must be of the form

$$L_{n,0}(x) = c(x - x_1)(x - x_2) \cdots (x - x_n) = c \prod_{j=1}^n (x - x_j), \quad (3.43)$$

where  $c$  is determined for which  $L_{n,0}(x_0) = 1$ . That is,

$$1 = L_{n,0}(x_0) = c(x_0 - x_1)(x_0 - x_2) \cdots (x_0 - x_n) \quad (3.44)$$

and therefore

$$c = \frac{1}{(x_0 - x_1)(x_0 - x_2) \cdots (x_0 - x_n)}. \quad (3.45)$$

Hence, we have

$$L_{n,0}(x) = \frac{(x - x_1)(x - x_2) \cdots (x - x_n)}{(x_0 - x_1)(x_0 - x_2) \cdots (x_0 - x_n)} = \prod_{j=1}^n \frac{(x - x_j)}{(x_0 - x_j)}. \quad (3.46)$$

**Summary 3.41.** Each cardinal function is obtained by similar reasoning; the general formula is then

$$L_{n,i}(x) = \prod_{j=0, j \neq i}^n \frac{(x - x_j)}{(x_i - x_j)}, \quad i = 0, 1, \dots, n. \quad (3.47)$$

**Example 3.42.** Determine the Lagrange interpolating polynomial that passes through  $(2, 4)$  and  $(5, 1)$ .

**Solution.**

**Example 3.43.** Let  $x_0 = 2$ ,  $x_1 = 4$ ,  $x_2 = 5$ . Use the points to find the second Lagrange interpolating polynomial  $p_2$  for  $f(x) = 1/x$ .

**Solution.**

$$\text{Ans: } p_2 = \frac{1}{12}(x-4)(x-5) - \frac{1}{8}(x-2)(x-5) + \frac{1}{15}(x-2)(x-4)$$

```

1  import sympy
2
3  def Lagrange(Lx,Ly):
4      X=sympy.symbols('X')
5      if len(Lx)!= len(Ly):
6          print("ERROR"); return 1
7      p=0
8      for k in range(len(Lx)):
9          t=1
10         for j in range(len(Lx)):
11             if j != k:
12                 t *= ( (X-Lx[j])/(Lx[k]-Lx[j]) )
13         p += t*Ly[k]
14     return p
15
16 if __name__ == "__main__":
17     Lx=[2,4,5]; Ly=[1/2,1/4,1/5]
18     p2 = Lagrange(Lx,Ly)
19     print(p2); print(sympy.simplify(p2))

```

Output

```

1  [Tue Aug.29] python Lagrange_interpol.py
2  0.5*(5/3 - X/3)*(2 - X/2) + 0.25*(5 - X)*(X/2 - 1) + 0.2*(X/3 - 2/3)*(X - 4)
3  0.025*X**2 - 0.275*X + 0.95

```

### 3.3.2. Polynomial Interpolation Error Theorem

**Q:** When an interpolating polynomial  $P_n \approx f$ , what is the error  $|f - P_n|$ ?

**Theorem 3.44. (Polynomial Interpolation Error Theorem).** Let  $f \in C^{n+1}[a, b]$ , and let  $P_n$  be the polynomial of degree  $\leq n$  that interpolates  $f$  at  $n + 1$  distinct points  $x_0, x_1, \dots, x_n$  in the interval  $[a, b]$ . Then, for each  $x \in (a, b)$ , there exists a number  $\xi_x$  between  $x_0, x_1, \dots, x_n$ , hence in the interval  $[a, b]$ , such that

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!} \prod_{i=0}^n (x - x_i) =: R_n(x). \quad (3.48)$$

**Example 3.45.** If the function  $f(x) = \sin(x)$  is approximated by a polynomial of degree 5 that interpolates  $f$  at six equally distributed points in  $[-1, 1]$  including end points, how large is the error on this interval?

**Solution.** The nodes  $x_i$  are  $-1, -0.6, -0.2, 0.2, 0.6,$  and  $1$ . It is easy to see that

$$|f^{(6)}(\xi)| = |-\sin(\xi)| \leq \sin(1).$$

```
g := x -> (x+1)*(x+0.6)*(x+0.2)*(x-0.2)*(x-0.6)*(x-1):
gmax := maximize(abs(g(x)), x = -1..1)
0.06922606316
```

Thus,

$$\begin{aligned} |\sin(x) - P_5(x)| &= \left| \frac{f^{(6)}(\xi)}{6!} \prod_{i=0}^5 (x - x_i) \right| \leq \frac{\sin(1)}{6!} \text{gmax} \\ &= 0.00008090517158 \end{aligned} \quad (3.49)$$

**Theorem 3.46. (Polynomial Interpolation Error Theorem for Equally Spaced Nodes):** Let  $f \in C^{n+1}[a, b]$ , and let  $P_n$  be the polynomial of degree  $\leq n$  that interpolates  $f$  at

$$x_i = a + ih, \quad h = \frac{b - a}{n}, \quad i = 0, 1, \dots, n.$$

Then, for each  $x \in (a, b)$ ,

$$|f(x) - P_n(x)| \leq \frac{h^{n+1}}{4(n+1)} M, \quad (3.50)$$

where

$$M = \max_{\xi \in [a, b]} |f^{(n+1)}(\xi)|.$$

**Proof.** Recall the interpolation error  $R_n(x)$  given in (3.48). We consider bounding

$$\max_{x \in [a, b]} \prod_{j=1}^n |x - x_j|.$$

Start by picking an  $x$ . We can assume that  $x$  is not one of the nodes, because otherwise the product in question is zero. Let  $x \in (x_j, x_{j+1})$ , for some  $j$ . Then we have

$$|x - x_j| \cdot |x - x_{j+1}| \leq \frac{h^2}{4}. \quad (3.51)$$

Now note that

$$|x - x_i| \leq \begin{cases} (j+1-i)h & \text{for } i < j \\ (i-j)h & \text{for } j+1 < i. \end{cases} \quad (3.52)$$

Thus

$$\prod_{j=1}^n |x - x_j| \leq \frac{h^2}{4} [(j+1)! h^j] [(n-j)! h^{n-j-1}]. \quad (3.53)$$

Since  $(j+1)!(n-j)! \leq n!$ , we can reach the following bound

$$\prod_{j=1}^n |x - x_j| \leq \frac{1}{4} h^{n+1} n!. \quad (3.54)$$

The result of the theorem follows from the above bound.  $\square$

**Example 3.47. (Revisit to Example 3.45)** The function  $f(x) = \sin(x)$  is approximated by a polynomial of degree 5 that interpolates  $f$  at six equally distributed points in  $[-1, 1]$  including end points. Use (3.50) to estimate **the upper bound of the interpolation error**.

**Solution.**

interpol\_error.py

```

1  from sympy import *
2  x = symbols('x')
3
4  a,b=-1,1; n=5
5
6  f = sin(x)
7  print( diff(f,x,n+1) )
8
9  h = (b-a)/n
10 M = abs(-sin(1.));
11
12 err_bound = h**(n+1)/(4*(n+1)) *M
13 print(err_bound)

```

Output

```

1  -sin(x)
2  0.000143611048073881

```

Compare the above error with the one in (3.49), p.90.

## 3.4. Numerical Differentiation: Finite Difference Formulas

**Note:** The derivative of  $f$  at  $x_0$  is defined as

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}. \quad (3.55)$$

This formula gives an obvious way to generate an approximation of  $f'(x_0)$ :

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}. \quad (3.56)$$

**Formula 3.48. (Two-Point Difference Formulas):** Let  $x_1 = x_0 + h$  and  $P_{0,1}$  be the first **Lagrange polynomial** interpolating  $f$  on  $[x_0, x_1]$ . Then

$$\begin{aligned} f(x) &= \mathbf{P}_{0,1}(\mathbf{x}) + \frac{(x - x_0)(x - x_1)}{2!} f''(\xi) \\ &= \frac{\mathbf{x} - \mathbf{x}_1}{-\mathbf{h}} \mathbf{f}(\mathbf{x}_0) + \frac{\mathbf{x} - \mathbf{x}_0}{\mathbf{h}} \mathbf{f}(\mathbf{x}_1) + \frac{(x - x_0)(x - x_1)}{2!} f''(\xi). \end{aligned} \quad (3.57)$$

Differentiating it, we obtain

$$f'(x) = \frac{\mathbf{f}(\mathbf{x}_1) - \mathbf{f}(\mathbf{x}_0)}{\mathbf{h}} + \frac{2x - x_0 - x_1}{2} f''(\xi) + \frac{(x - x_0)(x - x_1)}{2!} \frac{d}{dx} f''(\xi). \quad (3.58)$$

Thus

$$\begin{aligned} f'(x_0) &= \frac{f(x_1) - f(x_0)}{h} - \frac{h}{2} f''(\xi(x_0)) \\ f'(x_1) &= \frac{f(x_1) - f(x_0)}{h} + \frac{h}{2} f''(\xi(x_1)) \end{aligned} \quad (3.59)$$

**Definition 3.49.** For  $h > 0$ ,

$$\begin{aligned} f'(x_i) &\approx D_x^+ f(x_i) = \frac{f(x_i + h) - f(x_i)}{h}, \quad \text{(forward-difference)} \\ f'(x_i) &\approx D_x^- f(x_i) = \frac{f(x_i) - f(x_i - h)}{h}. \quad \text{(backward-difference)} \end{aligned} \quad (3.60)$$

**Example 3.50.** Use the forward-difference formula to approximate  $f(x) = x^3$  at  $x_0 = 1$  using  $h = 0.1, 0.05, 0.025$ .

**Solution.** Note that  $f'(1) = 3$ .

Maple-code

```

1  f := x -> x^3: x0 := 1:
2
3  h := 0.1:
4  (f(x0 + h) - f(x0))/h
5                                     3.310000000
6  h := 0.05:
7  (f(x0 + h) - f(x0))/h
8                                     3.152500000
9  h := 0.025:
10 (f(x0 + h) - f(x0))/h
11                                    3.075625000

```

The error becomes half, as  $h$  halves?

**Formula 3.51. (In general):** Let  $\{x_0, x_1, \dots, x_n\}$  be  $(n + 1)$  distinct points in some interval  $I$  and  $f \in C^{n+1}(I)$ . Then the Interpolation Error Theorem reads

$$f(x) = \sum_{k=0}^n f(x_k) L_{n,k}(x) + \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^n (x - x_k). \quad (3.61)$$

Its derivative gives

$$f'(x) = \sum_{k=0}^n f(x_k) L'_{n,k}(x) + \frac{d}{dx} \left( \frac{f^{(n+1)}(\xi)}{(n+1)!} \right) \prod_{k=0}^n (x - x_k) + \frac{f^{(n+1)}(\xi)}{(n+1)!} \frac{d}{dx} \left( \prod_{k=0}^n (x - x_k) \right). \quad (3.62)$$

Hence,

$$f'(x_i) = \sum_{k=0}^n f(x_k) L'_{n,k}(x_i) + \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0, k \neq i}^n (x_i - x_k), \quad (3.63)$$

which is the  $(n + 1)$ -point difference formula to approximate  $f'(x_i)$ .



**Formula 3.52. (Three-Point Difference Formulas ( $n = 2$ )):**

For convenience, let

$$x_0, \quad x_1 = x_0 + h, \quad x_2 = x_0 + 2h, \quad h > 0.$$

Recall the second-order cardinal basis functions

$$L_{2,0}(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}, \quad L_{2,1}(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)},$$

$$L_{2,2}(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

It follows from the **Polynomial Interpolation Error Theorem** that

$$f(x) = f(x_0)L_{2,0}(x) + f(x_1)L_{2,1}(x) + f(x_2)L_{2,2}(x) + \frac{f^{(3)}(\xi)}{3!} \prod_{k=0}^2 (x - x_k), \quad (3.64)$$

and its derivative reads

$$f'(x) = f(x_0)L'_{2,0}(x) + f(x_1)L'_{2,1}(x) + f(x_2)L'_{2,2}(x) + \frac{d}{dx} \left[ \frac{f^{(3)}(\xi)}{3!} \prod_{k=0}^2 (x - x_k) \right]. \quad (3.65)$$

Thus, the **three-point formulas** read

$$f'(x_0) = f(x_0)L'_{2,0}(x_0) + f(x_1)L'_{2,1}(x_0) + f(x_2)L'_{2,2}(x_0) + \frac{f^{(3)}(\xi)}{3!} \prod_{k=1}^2 (x_0 - x_k)$$

$$= \frac{-3f(x_0) + 4f(x_1) - f(x_2)}{2h} + \frac{h^2}{3} f^{(3)}(\xi_0),$$

$$f'(x_1) = \frac{f(x_2) - f(x_0)}{2h} - \frac{h^2}{6} f^{(3)}(\xi_1),$$

$$f'(x_2) = \frac{f(x_0) - 4f(x_1) + 3f(x_2)}{2h} + \frac{h^2}{3} f^{(3)}(\xi_2). \quad (3.66)$$

**Formula 3.53. (Five-Point Difference Formulas):**

Let  $f_i = f(x_0 + i h)$ ,  $h > 0$ ,  $-\infty < i < \infty$ .

$$\begin{aligned} f'(x_0) &= \frac{f_{-2} - 8f_{-1} + 8f_1 - f_2}{12h} + \frac{h^4}{30} f^{(5)}(\xi), \\ f'(x_0) &= \frac{-25f_0 + 48f_1 - 36f_2 + 16f_3 - 3f_4}{12h} + \frac{h^4}{5} f^{(5)}(\xi). \end{aligned} \quad (3.67)$$

**Summary 3.54. Numerical Differentiation:**

1.  $f(x) = P_n(x) + R_n(x)$ ,  $P_n(x) \in \mathbb{P}_n$ ,  $R_n(x) = \mathcal{O}(h^{n+1})$
2.  $f'(x) = P'_n(x) + \mathcal{O}(h^n)$ ,
3.  $f''(x) = P''_n(x) + \mathcal{O}(h^{n-1})$ , and so on.

**Note:** We can see from the above summary that **for  $f''$**

- The **three-point formula** ( $n = 2$ ): its accuracy is  $\mathcal{O}(h)$
- The **five-point formula** ( $n = 4$ ): its accuracy is  $\mathcal{O}(h^3)$

These hold for **every point in  $[x_0, x_n]$**  including all nodal points  $\{x_i\}$ .

**Midpoint Formula for  $f''$ : A higher-order Accuracy**

**Recall: (Theorem 3.34, p.84). Taylor's Theorem with Lagrange Remainder**

Suppose  $f \in C^n[a, b]$ ,  $f^{(n+1)}$  exists on  $(a, b)$ , and  $x_0 \in [a, b]$ . Then, for every  $x \in [a, b]$ ,

$$f(x) = \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k + \mathcal{R}_n(x), \quad (3.68)$$

where, for some  $\xi$  between  $x$  and  $x_0$ ,

$$\mathcal{R}_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1}.$$

Replace  $x$  by  $x_0 + h$ :

### Alternative Form of Taylor's Theorem

**Remark 3.55.** Replacing  $x$  by  $x_0 + h$  in the Taylor's Theorem, we have

$$f(x_0 + h) = \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!} h^k + \mathcal{R}_n(h), \quad \mathcal{R}_n(h) = \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1}, \quad (3.69)$$

for some  $\xi$  between  $x_0$  and  $x_0 + h$ . In detail,

$$\begin{aligned} f(x_0 + h) = & f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2!}h^2 + \frac{f'''(x_0)}{3!}h^3 + \dots \\ & + \frac{f^{(n)}(x_0)}{n!}h^n + \mathcal{R}_n(h). \end{aligned} \quad (3.70)$$

**Example 3.56.** Use the *Taylor series* to derive **the midpoint formula**

$$\begin{aligned} f''(x_0) = & \frac{f_{-1} - 2f_0 + f_1}{h^2} \\ & - \frac{h^2}{12}f^{(4)}(x_0) - \frac{h^4}{360}f^{(6)}(x_0) - \frac{h^6}{20160}f^{(8)}(x_0) - \dots \end{aligned} \quad (3.71)$$

**Solution.** It follows from the **Taylor's series formula (3.70)** that

$$\begin{aligned} f(x_0 + h) &= f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2!}h^2 + \frac{f'''(x_0)}{3!}h^3 + \frac{f^{(4)}(x_0)}{4!}h^4 + \dots \\ f(x_0 - h) &= f(x_0) - f'(x_0)h + \frac{f''(x_0)}{2!}h^2 - \frac{f'''(x_0)}{3!}h^3 + \frac{f^{(4)}(x_0)}{4!}h^4 - \dots \end{aligned} \quad (3.72)$$

Adding these two equations, we have

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + 2\frac{f''(x_0)}{2!}h^2 + 2\frac{f^{(4)}(x_0)}{4!}h^4 + \dots \quad (3.73)$$

Solve it for  $f''(x_0)$ .  $\square$

**Note:** The higher-order accuracy in (3.71) can be achieved at the **midpoint** only.

**Example 3.57.** Use the second-derivative midpoint formula to approximate  $f''(1)$  for  $f(x) = x^5 - 3x^2$ , using  $h = 0.2, 0.1, 0.05$ .

**Solution.**

Maple-code

```
1 f := x -> x^5 - 3*x^2:
2 x0 := 1:
3
4 eval(diff(f(x), x, x), x = x0)
5                                     14
6 h := 0.2:
7 (f(x0 - h) - 2*f(x0) + f(x0 + h))/h^2
8                                     14.40000000
9 h := 0.1:
10 (f(x0 - h) - 2*f(x0) + f(x0 + h))/h^2
11                                    14.10000000
12 h := 0.05:
13 (f(x0 - h) - 2*f(x0) + f(x0 + h))/h^2
14                                    14.02500000
```

## 3.5. Newton's Method for the Solution of Nonlinear Equations

The **Newton's method** is also called the **Newton-Raphson method**.

The objective is to find a zero  $p$  of  $f$ :

$$f(p) = 0. \quad (3.74)$$

**Strategy 3.58.** Let  $p_0$  be an approximation of  $p$ . We will try to find a **correction term**  $h$  such that  $(p_0 + h)$  is a better approximation of  $p$  than  $p_0$ ; ideally  $(p_0 + h) = p$ .

- If  $f''$  exists and is continuous, then by Taylor's Theorem

$$0 = f(p) = f(p_0 + h) = f(p_0) + hf'(p_0) + \frac{h^2}{2}f''(\xi), \quad (3.75)$$

where  $h = p - p_0$  and  $\xi$  lies between  $p$  and  $p_0$ .

- If  $|h|$  is small, it is reasonable to ignore the last term of (3.75) and solve for  $h = p - p_0$ :

$$h = p - p_0 \approx -\frac{f(p_0)}{f'(p_0)}. \quad (3.76)$$

- Define

$$p_1 = p_0 - \frac{f(p_0)}{f'(p_0)}; \quad (3.77)$$

then  $p_1$  may be a better approximation of  $p$  than  $p_0$ .

- The above can be repeated.

**Algorithm 3.59. Newton's method for solving  $f(x) = 0$**

For  $p_0$  chosen close to a root  $p$ , compute  $\{p_n\}$  repeatedly satisfying

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \quad n \geq 1. \quad (3.78)$$

### Graphical interpretation

- Let  $p_0$  be the initial approximation close to  $p$ . Then, the **tangent line** at  $(p_0, f(p_0))$  reads

$$L(x) = f'(p_0)(x - p_0) + f(p_0). \quad (3.79)$$

- To find the  **$x$ -intercept** of  $y = L(x)$ , let

$$0 = f'(p_0)(x - p_0) + f(p_0).$$

Solving the above equation for  $x$  becomes

$$x = p_0 - \frac{f(p_0)}{f'(p_0)}, \quad (3.80)$$

of which the right-side is the same as in (3.77).

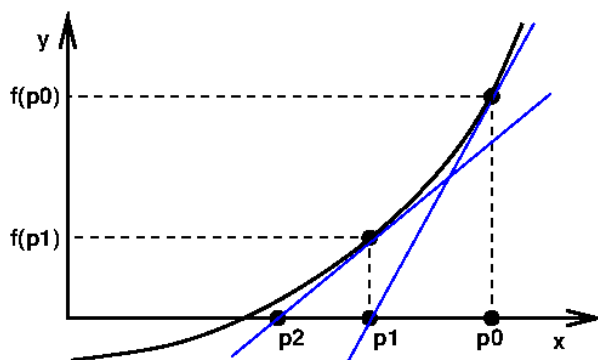


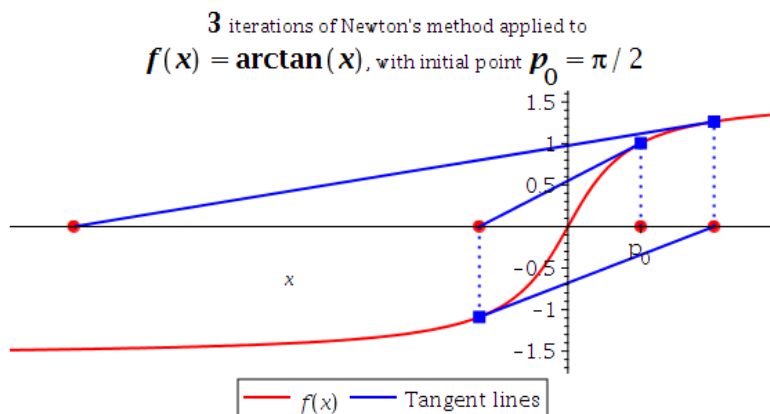
Figure 3.6: Graphical interpretation of the Newton's method.

#### An Example of Divergence

```

1 f := arctan(x);
2 Newton(f, x = Pi/2, output = plot, maxiterations = 3);

```



**Remark 3.60.**

- The Newton's method may diverge, unless the initialization is accurate.
- It cannot be continued if  $f'(p_{n-1}) = 0$  for some  $n$ . As a matter of fact, the Newton's method is most effective when  $f'(x)$  is bounded away from zero near  $p$ .

**Convergence analysis for the Newton's method**

Define the error in the  $n$ -th iteration:  $e_n = p_n - p$ . Then

$$e_n = p_n - p = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})} - p = \frac{e_{n-1}f'(p_{n-1}) - f(p_{n-1})}{f'(p_{n-1})}. \quad (3.81)$$

On the other hand, it follows from the Taylor's Theorem that

$$0 = f(p) = f(p_{n-1} - e_{n-1}) = f(p_{n-1}) - e_{n-1}f'(p_{n-1}) + \frac{1}{2}e_{n-1}^2f''(\xi_{n-1}), \quad (3.82)$$

for some  $\xi_{n-1}$ . Thus, from (3.81) and (3.82), we have

$$e_n = \frac{1}{2} \frac{f''(\xi_{n-1})}{f'(p_{n-1})} e_{n-1}^2. \quad (3.83)$$

**Theorem 3.61. (Convergence of Newton's method):** *Let  $f \in C^2[a, b]$  and  $p \in (a, b)$  is such that  $f(p) = 0$  and  $f'(p) \neq 0$ . Then, there is a neighborhood of  $p$  such that if the Newton's method is started  $p_0$  in that neighborhood, it generates a convergent sequence  $p_n$  satisfying*

$$|p_n - p| \leq C|p_{n-1} - p|^2, \quad (3.84)$$

for a positive constant  $C$ .

**Example 3.62.** Apply the Newton's method to solve  $f(x) = \arctan(x) = 0$ , with  $p_0 = \pi/5$ .

```
1 Newton(arctan(x), x = Pi/5, output = sequence, maxiterations = 5)
2 0.6283185308, -0.1541304479, 0.0024295539, -9.562*10^-9, 0., 0.
```

Since  $p = 0$ ,  $e_n = p_n$  and

$$|e_n| \leq 0.67|e_{n-1}|^3, \quad (3.85)$$

which is an occasional **super-convergence**.  $\square$

**Theorem 3.63. Newton's Method for a Convex Function**

Let  $f \in C^2(\mathbb{R})$  be increasing, convex, and of a zero  $p$ . Then, the zero  $p$  is unique and the Newton iteration converges to  $p$  from any starting point.

**Example 3.64.** Use the Newton's method to find the **square root** of a positive number  $Q$ .

**Solution.** Let  $x = \sqrt{Q}$ . Then  $x$  is a root of  $x^2 - Q = 0$ . Define  $f(x) = x^2 - Q$ ; set  $f'(x) = 2x$ . The Newton's method reads

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})} = p_{n-1} - \frac{p_{n-1}^2 - Q}{2p_{n-1}} = \frac{1}{2} \left( p_{n-1} + \frac{Q}{p_{n-1}} \right). \quad (3.86)$$

mysqrt.m

```
1 function x = mysqrt(q)
2 %function x = mysqrt(q)
3
4 x = (q+1)/2;
5 for n=1:10
6     x = (x+q/x)/2;
7     fprintf('x_%02d = %.16f\n',n,x);
8 end
```



## Results

```
1 >> mysqrt(16);
2 x_01 = 5.1911764705882355
3 x_02 = 4.1366647225462421
4 x_03 = 4.0022575247985221
5 x_04 = 4.0000006366929393
6 x_05 = 4.0000000000000506
7 x_06 = 4.0000000000000000
8 x_07 = 4.0000000000000000
9 x_08 = 4.0000000000000000
10 x_09 = 4.0000000000000000
11 x_10 = 4.0000000000000000
```

```
1 >> mysqrt(0.1);
2 x_01 = 0.3659090909090910
3 x_02 = 0.3196005081874647
4 x_03 = 0.3162455622803890
5 x_04 = 0.3162277665175675
6 x_05 = 0.3162277660168379
7 x_06 = 0.3162277660168379
8 x_07 = 0.3162277660168379
9 x_08 = 0.3162277660168379
10 x_09 = 0.3162277660168379
11 x_10 = 0.3162277660168379
```

**Note:** The function `sqrt` is implemented the same way as `mysqrt.m`.

## 3.6. Zeros of Polynomials

**Definition 3.65.** A polynomial of degree  $n$  has the form

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0, \quad (3.87)$$

where  $a_i$ 's are called the **coefficients** of  $P$  and  $a_n \neq 0$ .

The objective is to find zeros of  $P$ .

**Theorem 3.66. (Theorem on Polynomials).**

- **Fundamental Theorem of Algebra:**

*Every nonconstant polynomial has at least one root (possibly, in the complex field).*

- **Complex Roots of Polynomials:**

*A polynomial of degree  $n$  has **exactly  $n$  roots in the complex plane**, being agreed that each root shall be counted a number of times equal to its multiplicity. That is, there are unique (complex) constants  $x_1, x_2, \dots, x_k$  and unique integers  $m_1, m_2, \dots, m_k$  such that*

$$P(x) = a_n (x - x_1)^{m_1} (x - x_2)^{m_2} \cdots (x - x_k)^{m_k}, \quad \sum_{i=1}^k m_i = n. \quad (3.88)$$

- **Localization of Roots:**

*All roots of the polynomial  $P$  lie in the open disk centered at the origin and of radius of*

$$\rho = 1 + \frac{1}{|a_n|} \max_{0 \leq i < n} |a_i|. \quad (3.89)$$

- **Uniqueness of Polynomials:**

*Let  $P(x)$  and  $Q(x)$  be polynomials of degree  $n$ . If  $x_1, x_2, \dots, x_r$ , with  $r > n$ , are distinct numbers with  $P(x_i) = Q(x_i)$ , for  $i = 1, 2, \dots, r$ , then  $P(x) = Q(x)$  for all  $x$ .*

*– For example, two polynomials of degree  $n$  are the same if they agree at  $(n + 1)$  points.*

### 3.6.1. Horner's Method

**Note:** Known as **nested multiplication** and also as **synthetic division**, **Horner's method** can evaluate polynomials very efficiently. It requires  $n$  multiplications and  $n$  additions to evaluate an arbitrary  $n$ -th degree polynomial.

**Algorithm 3.67.** Let us try to evaluate  $P(x)$  at  $x = x_0$ .

- Utilizing the **Remainder Theorem**, we can rewrite the polynomial as

$$P(x) = (x - x_0)Q(x) + r = (x - x_0)Q(x) + P(x_0), \quad (3.90)$$

where  $Q(x)$  is a polynomial of degree  $n - 1$ , say

$$Q(x) = b_n x^{n-1} + \cdots + b_2 x + b_1. \quad (3.91)$$

- Substituting the above into (3.90), utilizing (3.87), and setting equal the coefficients of like powers of  $x$  on the two sides of the resulting equation, we have

$$\begin{array}{rcl} \hline b_n & = & a_n \\ b_{n-1} & = & a_{n-1} + x_0 b_n \\ & \vdots & \\ b_1 & = & a_1 + x_0 b_2 \\ \hline P(x_0) & = & a_0 + x_0 b_1 \\ \hline \end{array} \quad (3.92)$$

- Introducing  $b_0 = P(x_0)$ , the above can be rewritten as

$$b_{n+1} = 0; \quad b_k = a_k + x_0 b_{k+1}, \quad n \geq k \geq 0. \quad (3.93)$$

- If the calculation of Horner's algorithm is to be carried out with pencil and paper, the following arrangement is often used (known as **synthetic division**):

$x_0$	$a_n$	$a_{n-1}$	$a_{n-2}$	$\cdots$	$a_0$
	$x_0 b_n$	$x_0 b_{n-1}$	$\cdots$	$x_0 b_1$	
	$b_n$	$b_{n-1}$	$b_{n-2}$	$\cdots$	$P(x_0) = b_0$

**Example 3.68.** Use Horner's algorithm to evaluate  $P(3)$ , where

$$P(x) = x^4 - 4x^3 + 7x^2 - 5x - 2. \quad (3.94)$$

**Solution.** For  $x_0 = 3$ , we arrange the calculation as mentioned above:

3	1	-4	7	-5	-2
		3	-3	12	21
	1	-1	4	7	19 = P(3)

Note that the 4-th degree polynomial in (3.94) is written as

$$P(x) = (x - 3)(x^3 - x^2 + 4x + 7) + 19. \quad \square$$

**Remark 3.69.** When the Newton's method is applied for finding an approximate zero of  $P(x)$ , the iteration reads

$$x_n = x_{n-1} - \frac{P(x_{n-1})}{P'(x_{n-1})}. \quad (3.95)$$

Thus both  $P(x)$  and  $P'(x)$  must be evaluated in each iteration.

**Strategy 3.70. How to evaluate  $P'(x)$ :** **The derivative  $P'(x)$  can be evaluated by using the Horner's method with the same efficiency.** Indeed, differentiating (3.90)

$$P(x) = (x - x_0)Q(x) + P(x_0)$$

reads

$$P'(x) = Q(x) + (x - x_0)Q'(x). \quad (3.96)$$

Thus

$$P'(x_0) = Q(x_0). \quad (3.97)$$

That is, the evaluation of  $Q$  at  $x_0$  becomes the desired quantity  $P'(x_0)$ .  $\square$

**Example 3.71.** Evaluate  $P'(3)$  for  $P(x)$  considered in Example 3.68, the previous example.

**Solution.** As in the previous example, we arrange the calculation and carry out the synthetic division one more time:

3	1	-4	7	-5	-2	
		3	-3	12	21	
3	1	-1	4	7	19	= P(3)
		3	6	30		
	1	2	10			37 = Q(3) = P'(3)

**Example 3.72.** Implement the Horner's algorithm to evaluate  $P(3)$  and  $P'(3)$ , for the polynomial in (3.94):  $P(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$ .

**Solution.**

```

_____ horner.m _____
1 function [p,d] = horner(A,x0)
2 % input: A = [a_0,a_1,...,a_n]
3 % output: p=P(x0), d=P'(x0)
4
5 n = size(A(:,1));
6 p = A(n); d=0;
7
8 for i = n-1:-1:1
9     d = p + x0*d;
10    p = A(i) +x0*p;
11 end

```

```

_____ Call_horner.m _____
1 a = [-2 -5 7 -4 1];
2 x0=3;
3 [p,d] = horner(a,x0);
4 fprintf(" P(%g)=%g; P' (%g)=%g\n",x0,p,x0,d)
5     Result:  P(3)=19; P' (3)=37

```

**Example 3.73.** Let  $P(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$ , as in (3.94). Use the Newton's method and the Horner's method to implement a code and find an approximate zero of  $P$  near 3.

**Solution.**

```

newton_horner.m
1 function [x,it] = newton_horner(A,x0,tol,itmax)
2 %   input:  A = [a_0,a_1,...,a_n]; x0: initial for P(x)=0
3 %   outpue: x: P(x)=0
4
5 x = x0;
6 for it=1:itmax
7     [p,d] = horner(A,x);
8     h = -p/d;
9     x = x + h;
10    if(abs(h)<tol), break; end
11 end

```

```

Call_newton_horner.m
1 a = [-2 -5 7 -4 1];
2 x0=3;
3 tol = 10^-12; itmax=1000;
4 [x,it] = newton_horner(a,x0,tol,itmax);
5 fprintf("  newton_horner: x0=%g; x=%g, in %d iterations\n",x0,x,it)
6   Result:  newton_horner: x0=3; x=2, in 7 iterations

```

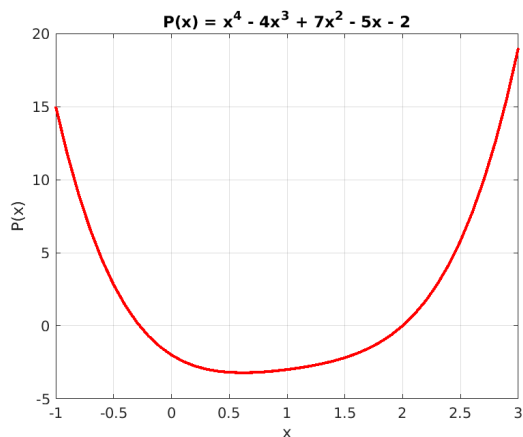


Figure 3.7: Polynomial  $P(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$ . Its two zeros are  $-0.275682$  and  $2$ .

### Exercises for Chapter 3

3.1. In Example 3.5, we considered the curve  $y = |x^2 - 1|$ . Find the left-hand limit and right-hand slope of the difference quotient at  $x_0 = 1$ .

*Ans:*  $-2$  and  $2$ .

3.2. The number  $e$  is determined so that the slope of the graph of  $y = e^x$  at  $x = 0$  is exactly 1. Let  $h$  be a point near 0. Then

$$Q(h) := \frac{e^h - e^0}{h - 0} = \frac{e^h - 1}{h}$$

represents the **average slope** of the graph between the two points  $(0, 1)$  and  $(h, e^h)$ . Evaluate  $Q(h)$ , for  $h = 0.1, 0.01, 0.001, 0.0001$ . What can you say about the results?

*Ans:* For example,  $Q(0.01) = 1.0050$ .

3.3. Recall the Taylor series for  $e^x$ ,  $\cos x$  and  $\sin x$  in (3.32). Let  $x = i\theta$ , where  $i = \sqrt{-1}$ . Then

$$e^{i\theta} = 1 + i\theta + \frac{i^2\theta^2}{2!} + \frac{i^3\theta^3}{3!} + \frac{i^4\theta^4}{4!} + \frac{i^5\theta^5}{5!} + \frac{i^6\theta^6}{6!} + \cdots \quad (3.98)$$

(a) Prove that  $e^{i\theta} = \cos \theta + i \sin \theta$ , which is called the **Euler's identity**.

(b) Prove that  $e^{i\pi} + 1 = 0$ .

3.4. Implement a code to visualize complex-valued solutions of  $e^z = -1$ .

- Use `fimplicit`
- Visualize, with `ylim([-2*pi 4*pi]), yticks(-pi:pi:3*pi)`

**Hint:** Use the code in § 2.2, starting with

```

eulers_identity.m
1  syms x y real
2  z = x+1i*y;
3
4  %% ---- Euler's identity
5  g = exp(z)+1;
6  RE = simplify(real(g))
7  IM = simplify(imag(g))
8
9  A = @(x,y) <Copy RE appropriately>
10 B = @(x,y) <Copy IM appropriately>
11
12 %%--- Solve A=0 and B=0 -----
13

```

3.5. Derive the following **midpoint formula**

$$f''(x_0) = \frac{-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2}{12h^2} + \frac{h^4}{90}f^{(6)}(x_0) + \frac{h^6}{1008}f^{(8)}(x_0) + \dots \quad (3.99)$$

**Hint:** Use the technique in Example 3.56, with  $f(x_0 + ih)$ ,  $i = -2, -1, 0, 1, 2$ .

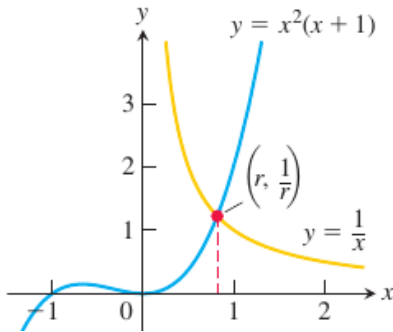
3.6. Use your calculator (or pencil-and-paper) to run two iterations of Newton's method to find  $x_2$  for given  $f$  and  $x_0$ .

(a)  $f(x) = x^4 - 2$ ,  $x_0 = 1$

(b)  $f(x) = xe^x - 1$ ,  $x_0 = 0.5$

Ans: (b)  $x_2 = 0.56715557$

3.7. The graphs of  $y = x^2(x + 1)$  and  $y = 1/x$  ( $x > 0$ ) intersect at one point  $x = r$ . Use Newton's method to estimate the value of  $r$  to eight decimal places.



3.8. Let  $f(x) = \cos x + \sin x$  be defined on the interval  $[-1, 1]$ .

(a) How many equally spaced nodes are required to interpolate  $f$  to within  $10^{-8}$  on the interval?

(b) Evaluate the interpolating polynomial **at the midpoint of a subinterval** and verify that the error is not larger than  $10^{-8}$ .

**Hint:** (a). Recall the formula:  $|f(x) - P_n(x)| \leq \frac{h^{n+1}}{4(n+1)}M$ . Then, for  $n$ , solve

$$\frac{(2/n)^{n+1}}{4(n+1)}\sqrt{2} \leq 10^{-8}.$$

3.9. Use the most accurate three-point formulas to determine the missing entries.

$x$	$f(x)$	$f'(x)$	$f''(x)$
1.0	2.0000		6.00
1.2	1.7536		
1.4	1.9616		
1.6	2.8736		
1.8	4.7776		
2.0	8.0000		
2.2	12.9056		52.08



**Hint:** The central scheme is more accurate than one-sided schemes.

3.10. Consider the polynomial

$$P(x) = 3x^5 - 7x^4 - 5x^3 + x^2 - 8x + 2.$$

- (a) Use the Horner's algorithm to find  $P(4)$ .
- (b) Use the Newton's method to find a real-valued root, starting with  $x_0 = 4$ . and applying the Horner's algorithm for the evaluation of  $P(x_k)$  and  $P'(x_k)$ .



## CHAPTER 4

# Linear Algebra Basics

**Real-world systems** can be approximated/represented as a **system of linear equations**

$$Ax = \mathbf{b}, \quad A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}, \quad (4.1)$$

where  $\mathbf{b}$  is the source and  $\mathbf{x}$  is the solution.

In this chapter, we will study topics in **linear algebra basics** including

- Elementary row operations
- Row reduction algorithm
- Linear independence
- Invertible matrices ( $m = n$ )

### Contents of Chapter 4

4.1. Solutions of Linear Systems . . . . .	114
4.2. Row Reduction and the General Solution of Linear Systems . . . . .	119
4.3. Linear Independence and Span of Vectors . . . . .	126
4.4. Invertible Matrices . . . . .	129
Exercises for Chapter 4 . . . . .	133

## 4.1. Solutions of Linear Systems

**Definition 4.1.** A **linear equation** in the variables  $x_1, x_2, \dots, x_n$  is an equation that can be written in the form

$$a_1 x_1 + a_2 x_2 + \dots + a_n x_n = b, \quad (4.2)$$

where  $b$  and the coefficients  $a_1, a_2, \dots, a_n$  are real or complex numbers.

A **system of linear equations** (or a **linear system**) is a collection of one or more linear equations involving the same variables – say,  $x_1, x_2, \dots, x_n$ .

**Example 4.2.**

$$(a) \begin{cases} 4x_1 - x_2 = 3 \\ 2x_1 + 3x_2 = 5 \end{cases} \qquad (b) \begin{cases} 2x + 3y - 4z = 2 \\ x - 2y + z = 1 \\ 3x + y - 2z = -1 \end{cases}$$

- **Solution:** A **solution** of the system is a list  $[s_1, s_2, \dots, s_n]$  of numbers that makes **each equation a true statement**, when

$$[x_1, x_2, \dots, x_n] \leftarrow [s_1, s_2, \dots, s_n].$$

- **Solution Set:** The set of all possible solutions is called the **solution set** of the linear system.
- **Equivalent System:** Two linear systems are called **equivalent** if they have the same solution set.

For example, Example 4.2 (a) is equivalent to

$$\begin{cases} 2x_1 - 4x_2 = -2 \\ 2x_1 + 3x_2 = 5 \end{cases} \quad \textcircled{R_1} \leftarrow \textcircled{R_1} - \textcircled{R_2}$$

**Remark 4.3.** Linear systems may have

no solution	:	<b>inconsistent system</b>
exactly one (unique) solution	}	: <b>consistent system</b>
infinitely many solutions		

**Example 4.4.** Consider the case of two equations in two unknowns.

$$(a) \begin{cases} -x + y = 1 \\ -x + y = 3 \end{cases} \quad (b) \begin{cases} x + y = 1 \\ x - y = 2 \end{cases} \quad (c) \begin{cases} -2x + y = 2 \\ -4x + 2y = 4 \end{cases}$$

### 4.1.1. Solving a linear system

Consider a simple system of 2 linear equations:

$$\begin{cases} -2x_1 + 3x_2 = -1 \\ x_1 + 2x_2 = 4 \end{cases} \quad (4.3)$$

Such a system of linear equations can be treated *much more conveniently and efficiently with matrix form*; (4.3) reads

$$\underbrace{\begin{bmatrix} -2 & 3 \\ 1 & 2 \end{bmatrix}}_{\text{coefficient matrix}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 4 \end{bmatrix}. \quad (4.4)$$

The **essential information** of the system can be recorded compactly in a rectangular array called an **augmented matrix**:

$$\begin{bmatrix} -2 & 3 & -1 \\ 1 & 2 & 4 \end{bmatrix} \quad \text{or} \quad \left[ \begin{array}{cc|c} -2 & 3 & -1 \\ 1 & 2 & 4 \end{array} \right] \quad (4.5)$$

**Solving (4.3):****System of linear equations**

$$\begin{cases} -2x_1 + 3x_2 = -1 & \textcircled{1} \\ \underline{x_1} + 2x_2 = 4 & \textcircled{2} \end{cases}$$

① ↔ ②: (interchange)

$$\begin{cases} x_1 + 2x_2 = 4 & \textcircled{1} \\ \underline{-2x_1} + 3x_2 = -1 & \textcircled{2} \end{cases}$$

② ← ② + 2 · ①: (replacement)

$$\begin{cases} x_1 + 2x_2 = 4 & \textcircled{1} \\ \underline{7x_2} = 7 & \textcircled{2} \end{cases}$$

② ← ②/7: (scaling)

$$\begin{cases} x_1 + \underline{2x_2} = 4 & \textcircled{1} \\ x_2 = 1 & \textcircled{2} \end{cases}$$

① ← ① - 2 · ②: (replacement)

$$\begin{cases} x_1 = 2 & \textcircled{1} \\ x_2 = 1 & \textcircled{2} \end{cases}$$

**Matrix form**

$$\begin{bmatrix} -2 & 3 & -1 \\ 1 & 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 4 \\ -2 & 3 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 4 \\ 0 & 7 & 7 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 4 \\ 0 & 1 & 1 \end{bmatrix}$$

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

**At the last step:**

$$\text{LHS: solution : } \begin{cases} x_1 = 2 \\ x_2 = 1 \end{cases}$$

$$\text{RHS : } \left[ \mathbf{I} \mid \begin{array}{c} 2 \\ 1 \end{array} \right]$$

**Tools 4.5. Three Elementary Row Operations (ERO):**

- **Replacement:** Replace one row by the sum of itself and a multiple of another row

$$R_i \leftarrow R_i + k \cdot R_j, \quad j \neq i$$

- **Interchange:** Interchange two rows

$$R_i \leftrightarrow R_j, \quad j \neq i$$

- **Scaling:** Multiply all entries in a row by a nonzero constant

$$R_i \leftarrow k \cdot R_i, \quad k \neq 0$$

**Definition 4.6.** Two matrices are **row equivalent** if there is a sequence of EROs that transforms one matrix to the other.

**4.1.2. Matrix equation  $Ax = b$** 

A fundamental idea in linear algebra is to view a **linear combination of vectors** as a **product of a matrix and a vector**.

**Definition 4.7.** Let  $A = [a_1 \ a_2 \ \cdots \ a_n]$  be an  $m \times n$  matrix and  $x \in \mathbb{R}^n$ , then the **product of  $A$  and  $x$** , denoted by  $Ax$ , is the linear combination of columns of  $A$  using the corresponding entries of  $x$  as weights, i.e.,

$$Ax = [a_1 \ a_2 \ \cdots \ a_n] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = x_1 a_1 + x_2 a_2 + \cdots + x_n a_n. \quad (4.6)$$

A **matrix equation** is of the form  $Ax = b$ , where  $b$  is a column vector of size  $m \times 1$ .

**Example 4.8.**  $x = [x_1, x_2]^T = [-3, 2]^T$  is the solution of the linear system.

Linear system

$$\begin{aligned} x_1 + 2x_2 &= 1 \\ 3x_1 + 4x_2 &= -1 \end{aligned}$$

Matrix equation

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Vector equation

$$x_1 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

**Theorem 4.9.** Let  $A = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n]$  be an  $m \times n$  matrix,  $\mathbf{x} \in \mathbb{R}^n$ , and  $\mathbf{b} \in \mathbb{R}^m$ . Then the matrix equation

$$A\mathbf{x} = \mathbf{b} \quad (4.7)$$

has the same solution set as the vector equation

$$x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \cdots + x_n \mathbf{a}_n = \mathbf{b}, \quad (4.8)$$

which, in turn, has the same solution set as the system with augmented matrix

$$[\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n : \mathbf{b}]. \quad (4.9)$$

### Two Fundamental Questions about a Linear System:

1. **(Existence):** Is the system consistent; that is, does at least one solution *exist*?
2. **(Uniqueness):** If a solution exists, is it the only one; that is, is the solution *unique*?

**Example 4.10.** Determine the values of  $h$  such that the given system is a consistent linear system

$$\begin{aligned} x + hy &= -5 \\ 2x - 8y &= 6 \end{aligned}$$

**Solution.**

*Ans:*  $h \neq -4$



## 4.2. Row Reduction and the General Solution of Linear Systems

**Example 4.11.** Solve the following system of linear equations, using the **three EROs**. Then, determine if the system is consistent.

$$\begin{aligned}x_2 - 2x_3 &= 0 \\x_1 - 2x_2 + 2x_3 &= 3 \\4x_1 - 8x_2 + 6x_3 &= 14\end{aligned}$$

**Solution.**

$$\text{Ans: } \mathbf{x} = [1, -2, -1]^T$$

**Note:** The system of linear equations can be solved by transforming the **augmented matrix** to the **reduced row echelon form (rref)**.

```

_____ linear_equations_rref.m _____
1  A = [0 1 -2; 1 -2 2; 4 -8 6];
2  b = [0; 3; 14];
3
4  Ab = [A b];
5  rref(Ab)

```

```

_____ Result _____
1  ans =
2      1      0      0      1
3      0      1      0     -2
4      0      0      1     -1

```

### 4.2.1. Echelon Forms and the Row Reduction Algorithm

**Definition 4.12. Echelon form:** A rectangular matrix is in an **echelon form** if it has following properties.

1. All **nonzero rows** are above any zero rows (rows of all zeros).
2. Each **leading entry** in a row is in a column to the right of leading entry of the row above it.
3. All entries **below a leading entry** in a column are zeros.

**Row reduced echelon form:** If a matrix in an echelon form satisfies 4 and 5 below, then it is in the **row reduced echelon form (RREF)**, or the **reduced echelon form (REF)**.

4. The **leading entry** in each nonzero row is **1**.
5. Each leading 1 is the **only nonzero entry** in its column.

**Example 4.13.** Verify whether the following matrices are in echelon form, row reduced echelon form.

$$(a) \begin{bmatrix} 1 & 0 & 2 & 0 & 1 \\ 0 & 1 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$(b) \begin{bmatrix} 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 9 \\ 0 & 1 & 0 & 6 \end{bmatrix}$$

$$(c) \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$(d) \begin{bmatrix} 1 & 1 & 2 & 2 & 3 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

$$(e) \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(f) \begin{bmatrix} 0 & 1 & 0 & 5 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

**Solution.**

### Terminologies

- 1) A **pivot position** is a location in  $A$  that corresponds to a leading 1 in the reduced echelon form of  $A$ .
- 2) A **pivot column** is a column of  $A$  that contains a pivot position.

**Example 4.14.** The matrix  $A$  is given with its reduced echelon form. Find the pivot positions and pivot columns of  $A$ .

$$A = \begin{bmatrix} 1 & 1 & 0 & 2 & 0 \\ 1 & 1 & 1 & 3 & 0 \\ 1 & 1 & 0 & 2 & 4 \end{bmatrix} \xrightarrow{R.E.F.} \begin{bmatrix} 1 & 1 & 0 & 2 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

**Solution.**

### Terminologies

- 3) **Basic variables:** In the system  $Ax = b$ , the variables that correspond to pivot columns (in  $[A : b]$ ) are **basic variables**.
- 4) **Free variables:** In the system  $Ax = b$ , the variables that correspond to non-pivotal columns are **free variables**.

**Example 4.15.** For the system of linear equations, identify its **basic variables** and **free variables**.

$$\begin{cases} -x_1 - 2x_2 & = & -3 \\ & 2x_3 & = & 4 \\ & 3x_3 & = & 6 \end{cases}$$

**Solution.** *Hint:* You may start with its augmented matrix, and apply row operations.

### Row Reduction Algorithm

**Example 4.16.** Row reduce the matrix into the **reduced echelon form**.

$$A = \begin{bmatrix} 0 & -3 & -6 & 4 & 9 \\ -2 & -3 & 0 & 3 & -1 \\ 1 & 4 & 5 & -9 & -7 \end{bmatrix}$$

**Solution.**

$$\textcircled{f} \quad A \xrightarrow{R_1 \leftrightarrow R_3} \begin{bmatrix} \boxed{1} & 4 & 5 & -9 & -7 \\ -2 & -3 & 0 & 3 & -1 \\ 0 & -3 & -6 & 4 & 9 \end{bmatrix} \xrightarrow{R_2 \leftarrow R_2 + 2R_1} \begin{bmatrix} \boxed{1} & 4 & 5 & -9 & -7 \\ 0 & \boxed{5} & 10 & -15 & -15 \\ 0 & -3 & -6 & 4 & 9 \end{bmatrix}$$

$$\textcircled{f} \quad \xrightarrow{R_2 \leftarrow R_2/5} \begin{bmatrix} \boxed{1} & 4 & 5 & -9 & -7 \\ 0 & \boxed{1} & 2 & -3 & -3 \\ 0 & -3 & -6 & 4 & 9 \end{bmatrix} \xrightarrow{R_3 \leftarrow R_3 + 3R_2} \begin{bmatrix} \boxed{1} & 4 & 5 & -9 & -7 \\ 0 & \boxed{1} & 2 & -3 & -3 \\ 0 & 0 & 0 & \boxed{-5} & 0 \end{bmatrix}$$

$$\textcircled{b} \quad \xrightarrow{R_3 \leftarrow R_3/-5} \begin{bmatrix} \boxed{1} & 4 & 5 & -9 & -7 \\ 0 & \boxed{1} & 2 & -3 & -3 \\ 0 & 0 & 0 & \boxed{1} & 0 \end{bmatrix} \xrightarrow{\substack{R_1 \leftarrow R_1 + 9R_3 \\ R_2 \leftarrow R_2 + 3R_3}} \begin{bmatrix} \boxed{1} & 4 & 5 & 0 & -7 \\ 0 & \boxed{1} & 2 & 0 & -3 \\ 0 & 0 & 0 & \boxed{1} & 0 \end{bmatrix}$$

$$\textcircled{b} \quad \xrightarrow{R_1 \leftarrow R_1 - 4R_2} \begin{bmatrix} \boxed{1} & 0 & -3 & 0 & 5 \\ 0 & \boxed{1} & 2 & 0 & -3 \\ 0 & 0 & 0 & \boxed{1} & 0 \end{bmatrix}$$

The combination of operations in Line  $\textcircled{f}$  is called the **forward phase** of the row reduction, while that of Line  $\textcircled{b}$  is called the **backward phase**.

### Remark 4.17. Pivot Positions

Once a matrix is **in an echelon form**, further row operations do not change the positions of leading entries. Thus, the **leading entries** become the **leading 1's** in the reduced echelon form.

### Uniqueness of the Reduced Echelon Form

**Theorem 4.18.** *Each matrix is row equivalent to one and only one reduced echelon form.*

### 4.2.2. The General Solution of Linear Systems

1) For example, for an augmented matrix, its R.E.F. is given as

$$\begin{bmatrix} 1 & 0 & -5 & 1 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (4.10)$$

2) Then, the associated system of equations reads

$$\begin{aligned} x_1 - 5x_3 &= 1 \\ x_2 + x_3 &= 4 \\ 0 &= 0 \end{aligned} \quad (4.11)$$

where  $\{x_1, x_2\}$  are basic variables ( $\because$  pivots).

3) Rewrite (4.11) as

$$\begin{cases} x_1 = 1 + 5x_3 \\ x_2 = 4 - x_3 \\ x_3 \text{ is free} \end{cases} \quad (4.12)$$

4) The system (4.12) can be expressed as

$$\begin{cases} x_1 = 1 + 5x_3 \\ x_2 = 4 - x_3 \\ x_3 = x_3 \end{cases} \quad (4.13)$$

5) Thus, the solution of (4.11) can be written as

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 5 \\ -1 \\ 1 \end{bmatrix}, \quad (4.14)$$

in which you are free to choose any value for  $x_3$ . (That is why it is called a “**free variable**”.)

- The description in (4.14) is called a **parametric description** of solution set; the free variable  $x_3$  acts as a parameter.
- The solution in (4.14) represents **all the solutions of the system (4.10)**, which is called the **general solution** of the system.

**Example 4.19.** Find the general solution of the system whose augmented matrix is

$$[A|\mathbf{b}] = \begin{bmatrix} 1 & 0 & -5 & 0 & -8 & 3 \\ 0 & 1 & 4 & -1 & 0 & 6 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

**Solution.** *Hint:* You should first row reduce it for the reduced echelon form.

**Example 4.20.** Choose  $h$  and  $k$  such that the system has

a) No solution

b) Unique solution

c) Many solutions

$$\begin{cases} x_1 - 3x_2 = 1 \\ 2x_1 + hx_2 = k \end{cases}$$

**Solution.**

*Ans:* (a)  $h = -6, k \neq 2$

**Example 4.21.** Find the **general solution** of the system of which

the augmented matrix is  $[A|\mathbf{b}] = \begin{bmatrix} 1 & 0 & 0 & 1 & 7 \\ 0 & 1 & 3 & 0 & -1 \\ 2 & -1 & -3 & 2 & 15 \\ 1 & 0 & -1 & 0 & 4 \end{bmatrix}$

**Solution.**

```

_____ linear_equations_rref.m _____
1  Ab = [1 0 0 1 7; 0 1 3 0 -1; 2 -1 -3 2 15; 1 0 -1 0 4];
2  rref(Ab)

```

```

_____ Result _____
1  ans =
2      1      0      0      1      7
3      0      1      0     -3     -10
4      0      0      1      1      3
5      0      0      0      0      0

```

**True-or-False 4.22.**

- The row reduction algorithm applies to only to augmented matrices for a linear system.
- If one row in an echelon form of an augmented matrix is  $[0 \ 0 \ 0 \ 0 \ 2 \ 0]$ , then the associated linear system is inconsistent.
- The pivot positions in a matrix depend on whether or not row interchanges are used in the row reduction process.
- Reducing a matrix to an echelon form is called the **forward phase** of the row reduction process.

**Solution.**

*Ans:* F,F,F,T

### 4.3. Linear Independence and Span of Vectors

**Definition 4.23.** A set of vectors  $S = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_p\}$  in  $\mathbb{R}^n$  is said to be **linearly independent**, if the vector equation

$$x_1 \mathbf{v}_1 + x_2 \mathbf{v}_2 + \dots + x_p \mathbf{v}_p = \mathbf{0} \quad (4.15)$$

has only the trivial solution (i.e.,  $x_1 = x_2 = \dots = x_p = 0$ ). The set of vectors  $S$  is said to be **linearly dependent**, if there exist weights  $c_1, c_2, \dots, c_p$ , not all zero, such that

$$c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \dots + c_p \mathbf{v}_p = \mathbf{0}. \quad (4.16)$$

**Note:** A vector in a linearly independent set  $S$  cannot be expressed as a linear combination of other vectors in  $S$ .

**Example 4.24.** Determine if the set  $\{\mathbf{v}_1, \mathbf{v}_2\}$  is linearly independent.

$$1) \mathbf{v}_1 = \begin{bmatrix} 3 \\ 0 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 0 \\ 5 \end{bmatrix} \qquad 2) \mathbf{v}_1 = \begin{bmatrix} 3 \\ 0 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

**Remark 4.25.** Let  $A = [\mathbf{v}_1 \ \mathbf{v}_2 \ \dots \ \mathbf{v}_p]$ . The matrix equation  $A\mathbf{x} = \mathbf{0}$  is equivalent to  $x_1 \mathbf{v}_1 + x_2 \mathbf{v}_2 + \dots + x_p \mathbf{v}_p = \mathbf{0}$ .

- Columns of  $A$  are **linearly independent** if and only if  $A\mathbf{x} = \mathbf{0}$  has *only* the trivial solution. ( $\Leftrightarrow Ax = 0$  has no free variable  $\Leftrightarrow$  Every column in  $A$  is a **pivot column**.)
- Columns of  $A$  are **linearly dependent** if and only if  $A\mathbf{x} = \mathbf{0}$  has nontrivial solution. ( $\Leftrightarrow Ax = 0$  has at least one free variable  $\Leftrightarrow A$  has at least one *non*-pivot column.)



**Example 4.26.** Determine if the vectors are linearly independent.

$$\begin{bmatrix} 0 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -8 \end{bmatrix}, \begin{bmatrix} -1 \\ 3 \\ 1 \end{bmatrix}$$

**Solution.**

**Example 4.27.** Determine if the vectors are linearly independent.

$$\begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}, \begin{bmatrix} -2 \\ 4 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 \\ -6 \\ -1 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}$$

**Solution.**

**Note:** In the above example, vectors are in  $\mathbb{R}^n$ ,  $n = 3$ ; the number of vectors  $p = 4$ . As in this example, if  $p > n$  then the vectors *must* be linearly dependent.

**Definition 4.28.** Let  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_p$  be  $p$  vectors in  $\mathbb{R}^n$ . Then  $\text{Span}\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_p\}$  is the collection of all linear combination of  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_p$ , that can be written in the form  $c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \dots + c_p\mathbf{v}_p$ , where  $c_1, c_2, \dots, c_p$  are weights. That is,

$$\text{Span}\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_p\} = \{\mathbf{y} \mid \mathbf{y} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \dots + c_p\mathbf{v}_p\} \quad (4.17)$$

**Example 4.29.** Find the value of  $h$  so that  $\mathbf{c}$  is in  $\text{Span}\{\mathbf{a}, \mathbf{b}\}$ .

$$\mathbf{a} = \begin{bmatrix} 3 \\ -6 \\ 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -6 \\ 4 \\ -3 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 9 \\ h \\ 3 \end{bmatrix}$$

**Solution.**

**True-or-False 4.30.**

- The columns of any  $3 \times 4$  matrix are linearly dependent.
- If  $\mathbf{u}$  and  $\mathbf{v}$  are linearly independent, and if  $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$  is linearly dependent, then  $\mathbf{w} \in \text{Span}\{\mathbf{u}, \mathbf{v}\}$ .
- Two vectors are linearly dependent if and only if they lie on a line through the origin.
- The columns of a matrix  $A$  are linearly independent, if the equation  $A\mathbf{x} = \mathbf{0}$  has the trivial solution.

*Ans:* T,T,T,F

## 4.4. Invertible Matrices

**Definition 4.31.** An  $n \times n$  (square) matrix  $A$  is said to be **invertible (nonsingular)** if there is an  $n \times n$  matrix  $B$  such that  $AB = I_n = BA$ , where  $I_n$  is the identity matrix.

**Note:** In this case,  $B$  is the *unique inverse* of  $A$  denoted by  $A^{-1}$ .  
(Thus  $AA^{-1} = I_n = A^{-1}A$ .)

**Example 4.32.** If  $A = \begin{bmatrix} 2 & 5 \\ -3 & -7 \end{bmatrix}$  and  $B = \begin{bmatrix} -7 & -5 \\ 3 & 2 \end{bmatrix}$ . Find  $AB$  and  $BA$ .

**Solution.**

**Theorem 4.33. (Inverse of an  $n \times n$  matrix,  $n \geq 2$ )** An  $n \times n$  matrix  $A$  is invertible if and only if  $A$  is row equivalent to  $I_n$ ; in this case, any sequence of elementary row operations that reduces  $A$  into  $I_n$  will also reduce  $I_n$  to  $A^{-1}$ .

**Algorithm 4.34. Algorithm to find  $A^{-1}$ :**

- 1) Row reduce the augmented matrix  $[A : I_n]$
- 2) If  $A$  is row equivalent to  $I_n$ , then  $[A : I_n]$  is row equivalent to  $[I_n : A^{-1}]$ . Otherwise  $A$  does not have any inverse.

**Note:** For the system  $Ax = b$ , when  $A$  is invertible,

$$[A : b] \rightarrow \cdots \rightarrow [I_n : x] \Rightarrow x = A^{-1}b. \quad (4.18)$$

**Example 4.35.** Find the inverse of  $A = \begin{bmatrix} 3 & 2 \\ 8 & 5 \end{bmatrix}$ .

**Solution.** You may begin with

$$[A : I_2] = \begin{bmatrix} 3 & 2 & 1 & 0 \\ 8 & 5 & 0 & 1 \end{bmatrix}$$

**Self-study 4.36.** Use pencil-and-paper to find the inverse of  $A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 3 \\ 4 & -3 & 8 \end{bmatrix}$ ,

if it exists.

**Solution.**

When it is implemented:

```

inverse_matrix.m
1  A = [0  1  0
2     1  0  3
3     4 -3  8];
4  I = eye(3);
5
6  AI = [A I];
7  rref(AI)

```

```

Result
1  ans =
2     1.0000         0         0     2.2500    -2.0000     0.7500
3         0     1.0000         0     1.0000         0         0
4         0         0     1.0000    -0.7500     1.0000    -0.2500

```

**Definition 4.37.** Given an  $m \times n$  matrix  $A$ , the **transpose** of  $A$  is the matrix, denoted by  $A^T$ , whose columns are formed from the corresponding rows of  $A$ . That is,

$$A = [a_{ij}] \in \mathbb{R}^{m \times n} \Rightarrow A^T = [a_{ji}] \in \mathbb{R}^{n \times m}. \quad (4.19)$$

**Example 4.38.** If  $A = \begin{bmatrix} 1 & 4 & 8 & 1 \\ 0 & -2 & -1 & 3 \\ 9 & 0 & 0 & 5 \end{bmatrix}$ , then  $A^T =$

**Theorem 4.39. Properties of Invertible Matrices**

a. **(Inverse of a  $2 \times 2$  matrix)** Let  $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ . If  $ad - bc \neq 0$ , then  $A$  is invertible and

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad (4.20)$$

b. If  $A$  is an invertible matrix, then  $A^{-1}$  is also invertible and  $(A^{-1})^{-1} = A$ .

c. If  $A$  and  $B$  are  $n \times n$  invertible matrices then  $AB$  is also invertible and  $(AB)^{-1} = B^{-1}A^{-1}$ .

d. If  $A$  is invertible, then  $A^T$  is also invertible and  $(A^T)^{-1} = (A^{-1})^T$ .

e. If  $A$  is an  $n \times n$  invertible matrix, then for each  $\mathbf{b} \in \mathbb{R}^n$ , the equation  $A\mathbf{x} = \mathbf{b}$  has a unique solution  $\mathbf{x} = A^{-1}\mathbf{b}$ .

**Theorem 4.40. Invertible Matrix Theorem**

Let  $A$  be an  $n \times n$  matrix. Then the following are equivalent.

- a.  **$A$  is an invertible matrix.** (Def: There is  $B$  s.t.  $AB = BA = I$ )
- b.  $A$  is row equivalent to the  $n \times n$  identity matrix.
- c.  $A$  has  $n$  **pivot positions**.
- d. The equation  $Ax = 0$  has only the trivial solution  $x = 0$ .
- e. The columns of  $A$  are **linearly independent**.
- f. The linear transformation  $x \mapsto Ax$  is one-to-one.
- g. The equation  $Ax = b$  has a **unique solution** for each  $b \in \mathbb{R}^n$ .
- h. The columns of  $A$  span  $\mathbb{R}^n$ .
- i. The linear transformation  $x \mapsto Ax$  maps  $\mathbb{R}^n$  onto  $\mathbb{R}^n$ .
- j. There is a matrix  $C \in \mathbb{R}^{n \times n}$  such that  $CA = I$
- k. There is a matrix  $D \in \mathbb{R}^{n \times n}$  such that  $AD = I$
- l.  $A^T$  is invertible and  $(A^T)^{-1} = (A^{-1})^T$ .

More statements will be added in the coming sections; see Theorem 5.10, p.140, and Theorem 5.17, p.142.

**Note:** Let  $A$  and  $B$  be square matrices. If  $AB = I$ , then  $A$  and  $B$  are both invertible, with  $B = A^{-1}$  and  $A = B^{-1}$ .

**Example 4.41.** Use the Invertible Matrix Theorem to decide if  $A$  is invertible:

$$A = \begin{bmatrix} 1 & 0 & -2 \\ 3 & 1 & -2 \\ -5 & -1 & 9 \end{bmatrix}$$

## Exercises for Chapter 4

- 4.1. Find the general solutions of the systems (in **parametric vector form**) whose augmented matrices are given as

$$(a) \begin{bmatrix} 1 & -7 & 0 & 6 & 5 \\ 0 & 0 & 1 & -2 & -3 \\ -1 & 7 & -4 & 2 & 7 \end{bmatrix}$$

$$(b) \begin{bmatrix} 1 & 2 & -5 & -6 & 0 & -5 \\ 0 & 1 & -6 & -3 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

*Ans:* (a)  $\mathbf{x} = [5, 0, -3, 0]^T + x_2[7, 1, 0, 0]^T + x_4[-6, 0, 2, 1]^T$ ;  
 (b)  $\mathbf{x} = [-9, 2, 0, 0, 0]^T + x_3[-7, 6, 1, 0, 0]^T + x_4[0, 3, 0, 1, 0]^T$ .

- 4.2. In the following, we use the notation for matrices in echelon form: the leading entries with  $\blacksquare$ , and any values (including zero) with  $*$ . Suppose each matrix represents the augmented matrix for a system of linear equations. In each case, determine if the system is consistent. If the system is consistent, determine if the solution is unique.

$$(a) \begin{bmatrix} \blacksquare & * & * & * \\ 0 & \blacksquare & * & * \\ 0 & 0 & \blacksquare & * \end{bmatrix}$$

$$(b) \begin{bmatrix} 0 & \blacksquare & * & * & * \\ 0 & 0 & \blacksquare & * & * \\ 0 & 0 & 0 & 0 & \blacksquare \end{bmatrix}$$

$$(c) \begin{bmatrix} \blacksquare & * & * & * & * \\ 0 & 0 & \blacksquare & * & * \\ 0 & 0 & 0 & \blacksquare & * \end{bmatrix}$$

- 4.3. Suppose the coefficient matrix of a system of linear equations has a pivot position in every row. Explain why the system is consistent.
- 4.4. (a) For what values of  $h$  is  $\mathbf{v}_3$  in  $\text{Span}\{\mathbf{v}_1, \mathbf{v}_2\}$ , and (b) for what values of  $h$  is  $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$  linearly dependent? Justify each answer.

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ -3 \\ 2 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} -3 \\ 9 \\ -6 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} 5 \\ -7 \\ h \end{bmatrix}.$$

*Ans:* (a) No  $h$ ; (b) All  $h$

- 4.5. Find the inverses of the matrices, if exist:  $A = \begin{bmatrix} 3 & -4 \\ 7 & -8 \end{bmatrix}$  and  $B = \begin{bmatrix} 1 & -2 & 1 \\ 4 & -7 & 3 \\ -2 & 6 & -4 \end{bmatrix}$

*Ans:*  $B$  is not invertible.

- 4.6. Describe the possible echelon forms of the matrix. Use the notation of Exercise 2 above.
- (a)  $A$  is a  $3 \times 3$  matrix with linearly independent columns.  
 (b)  $A$  is a  $2 \times 2$  matrix with linearly dependent columns.  
 (c)  $A$  is a  $4 \times 2$  matrix,  $A = [\mathbf{a}_1, \mathbf{a}_2]$  and  $\mathbf{a}_2$  is not a multiple of  $\mathbf{a}_1$ .

- 4.7. If  $C$  is  $6 \times 6$  and the equation  $C\mathbf{x} = \mathbf{v}$  is consistent for every  $\mathbf{v} \in \mathbb{R}^6$ , is it possible that for some  $\mathbf{v}$ , the equation  $C\mathbf{x} = \mathbf{v}$  has more than one solution? Why or why not?

*Ans:* No





## CHAPTER 5

# Programming with Linear Algebra

In Chapter 4, we studied **linear algebra basics**. In this chapter, we will consider **popular subjects in linear algebra**, which are applicable for real-world problems through programming.

### Contents of Chapter 5

5.1. Determinants . . . . .	136
5.2. Eigenvalues and Eigenvectors . . . . .	141
5.3. Dot Product, Length, and Orthogonality . . . . .	148
5.4. Vector Norms, Matrix Norms, and Condition Numbers . . . . .	151
5.5. Power Method and Inverse Power Method for Eigenvalues . . . . .	155
Exercises for Chapter 5 . . . . .	162

## 5.1. Determinants

**Definition 5.1.** Let  $A$  be an  $n \times n$  square matrix. Then the **determinant** of  $A$  is a scalar value, denoted by  $\det A$  or  $|A|$ .

1) Let  $A = [a] \in \mathbb{R}^{1 \times 1}$ . Then  $\det A = a$ .

2) Let  $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \in \mathbb{R}^{2 \times 2}$ . Then  $\det A = ad - bc$ .

**Example 5.2.** Let  $A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$ . Consider a linear transformation  $T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  defined by  $T(\mathbf{x}) = A\mathbf{x}$ .

- Find the determinant of  $A$ .
- Determine the image of a rectangle  $R = [0, 2] \times [0, 1]$  under  $T$ .
- Find the area of the image.
- Figure out how  $\det A$ , the area of the rectangle ( $= 2$ ), and the area of the image are related.

**Solution.**

Ans: (c) 12

**Note:** The determinant can be viewed as a **volume scaling factor**.

**Definition 5.3.** Let  $A_{ij}$  be the *submatrix* of  $A$  obtained by deleting row  $i$  and column  $j$  of  $A$ . Then the  $(i, j)$ -**cofactor** of  $A = [a_{ij}]$  is the scalar  $C_{ij}$ , given by

$$C_{ij} = (-1)^{i+j} \det A_{ij}. \quad (5.1)$$

**Definition 5.4.** For  $n \geq 2$ , the **determinant** of an  $n \times n$  matrix  $A = [a_{ij}]$  is given by the following formulas:

1. The *cofactor expansion* across the first row:

$$\det A = a_{11}C_{11} + a_{12}C_{12} + \cdots + a_{1n}C_{1n} \quad (5.2)$$

2. The *cofactor expansion* across the row  $i$ :

$$\det A = a_{i1}C_{i1} + a_{i2}C_{i2} + \cdots + a_{in}C_{in} \quad (5.3)$$

3. The *cofactor expansion* down the column  $j$ :

$$\det A = a_{1j}C_{1j} + a_{2j}C_{2j} + \cdots + a_{nj}C_{nj} \quad (5.4)$$

**Example 5.5.** Find the determinant of  $A = \begin{bmatrix} 1 & 5 & 0 \\ 2 & 4 & -1 \\ 0 & -2 & 0 \end{bmatrix}$ , by expanding

across the first row and down column 3.

**Solution.**

**Note:** If  $A$  is a triangular (upper or lower) matrix, then  $\det A$  is the product of entries on the main diagonal of  $A$ .

**Example 5.6.** Compute the determinant of  $A = \begin{bmatrix} 1 & -2 & 5 & 2 \\ 0 & -6 & -7 & 5 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$ .

**Solution.**

```

_____ determinant.m _____
1  A = [1 -2 5 2; 0 -6 -7 5; 0 0 3 0; 0 0 0 4];
2  det(A)

```

```

_____ Result _____
1  ans =
2  -72

```

**Remark 5.7.** The matrix  $A$  in Example 5.6 has a pivot position in each column  $\Rightarrow$  It is invertible.

### Properties of Determinants

**Theorem 5.8.** Let  $A$  be an  $n \times n$  square matrix.

a) **(Replacement):** If  $B$  is obtained from  $A$  by a row replacement, then  $\det B = \det A$ .

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 3 \\ 0 & -5 \end{bmatrix}$$

b) **(Interchange):** If two rows of  $A$  are interchanged to form  $B$ , then  $\det B = -\det A$ .

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

c) **(Scaling):** If one row of  $A$  is multiplied by  $k$  ( $\neq 0$ ), then  $\det B = k \cdot \det A$ .

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 3 \\ -4 & -2 \end{bmatrix}$$

**Example 5.9.** Compute  $\det A$ , where  $A = \begin{bmatrix} 1 & -4 & 2 \\ -2 & 8 & -9 \\ -1 & 7 & 0 \end{bmatrix}$ , after applying

some elementary row operations.

**Solution.**

**Theorem 5.10. Invertible Matrix Theorem (p.132)**

A square matrix  $A$  is invertible  $\Leftrightarrow (Ax = 0 \Rightarrow x = 0)$

m.  $\det A \neq 0$  (Note:  $\det$  is a volume scaling factor)

**Remark 5.11.** Let  $A$  and  $B$  be  $n \times n$  matrices.

a)  $\det A^T = \det A$ .

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}$$

b)  $\det(AB) = \det A \cdot \det B$ .

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 \\ 4 & 2 \end{bmatrix}; \text{ then } AB = \begin{bmatrix} 13 & 7 \\ 6 & 4 \end{bmatrix}.$$

c) If  $A$  is invertible, then  $\det A^{-1} = \frac{1}{\det A}$ . ( $\because \det I_n = 1$ .)

**Example 5.12.** Suppose the sequence  $5 \times 5$  matrices  $A$ ,  $A_1$ ,  $A_2$ , and  $A_3$  are related by following elementary row operations:

$$A \xrightarrow{R_2 \leftarrow R_2 - 3R_1} A_1 \xrightarrow{R_3 \leftarrow (1/5)R_3} A_2 \xrightarrow{R_4 \leftrightarrow R_5} A_3$$

Find  $\det A$ , if  $A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 & 1 \\ 0 & -2 & 1 & -1 & 1 \\ 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

**Solution.**

## 5.2. Eigenvalues and Eigenvectors

**Definition 5.13.** Let  $A$  be an  $n \times n$  matrix. An **eigenvector** of  $A$  is a **nonzero vector**  $\mathbf{x}$  such that

$$A\mathbf{x} = \lambda\mathbf{x} \quad (5.5)$$

for some scalar  $\lambda$ . In this case, the scalar  $\lambda$  is an **eigenvalue** and  $\mathbf{x}$  is the *corresponding* **eigenvector**.

**Example 5.14.** Is  $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$  an eigenvector of  $\begin{bmatrix} 5 & 2 \\ 3 & 6 \end{bmatrix}$ ? What is the eigenvalue?

**Solution.**

**Example 5.15.** Let  $A = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix}$ . Show that 7 is an eigenvalue of matrix  $A$ , and find the **corresponding eigenvectors**.

**Solution.** *Hint:* Start with  $A\mathbf{x} = 7\mathbf{x}$ . Then  $(A - 7I)\mathbf{x} = \mathbf{0}$ .

**Remark 5.16.** Let  $\lambda$  be an eigenvalue of  $A$ . Then

- (a) The homogeneous system  $(A - \lambda I) \mathbf{x} = \mathbf{0}$  has at least one free variable ( $\because \mathbf{x} \neq \mathbf{0}$ ).
- (b)  $\det(A - \lambda I) = 0$ .

**Theorem 5.17. Invertible Matrix Theorem (p.132)**

A square matrix  $A$  is invertible  $\Leftrightarrow (A\mathbf{x} = \mathbf{0} \Rightarrow \mathbf{x} = \mathbf{0})$

*n.* The number 0 is not an eigenvalue of  $A$

### 5.2.1. Characteristic Equation

**Definition 5.18.** The scalar equation  $\det(A - \lambda I) = 0$  is called the **characteristic equation** of  $A$ ; the polynomial  $p(\lambda) = \det(A - \lambda I)$  is called the **characteristic polynomial** of  $A$ .

- The **solutions of  $\det(A - \lambda I) = 0$**  are the **eigenvalues of  $A$** .

**Example 5.19.** Find the characteristic polynomial, eigenvalues, and corresponding eigenvectors of  $A = \begin{bmatrix} 8 & 2 \\ 3 & 3 \end{bmatrix}$ .

**Solution.**



**Example 5.20.** Find the characteristic polynomial and all eigenvalues of

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 6 & 0 & 5 \\ 0 & 0 & 2 \end{bmatrix}$$

eigenvalues.m

```

1 syms x
2 A = [1 1 0; 6 0 5; 0 0 2];
3
4 polyA = charpoly(A,x)
5 eigenA = solve(polyA)
6 [P,D] = eig(A) % A*P = P*D
7 P*D*inv(P)

```

Results

```

1 polyA =
2 12 - 4*x - 3*x^2 + x^3
3
4 eigenA =
5 -2
6 2
7 3
8
9 P =
10 0.4472 -0.3162 -0.6155
11 0.8944 0.9487 -0.6155
12 0 0 0.4924
13 D =
14 3 0 0
15 0 -2 0
16 0 0 2
17
18 ans =
19 1.0000 1.0000 -0.0000
20 6.0000 0.0000 5.0000
21 0 0 2.0000

```

## 5.2.2. Matrix Similarity and The Diagonalization Theorem

**Definition 5.21.** Let  $A$  and  $B$  be  $n \times n$  matrices. Then,  $A$  is **similar** to  $B$ , if there is an invertible matrix  $P$  such that

$$A = PBP^{-1}, \text{ or equivalently, } P^{-1}AP = B.$$

Writing  $Q = P^{-1}$ , we have  $B = QAQ^{-1}$ . So  $B$  is also similar to  $A$ , and we say simply that  $A$  and  $B$  are *similar*. The map  $A \mapsto P^{-1}AP$  is called a **similarity transformation**.

The next theorem illustrates one use of the characteristic polynomial, and it provides the foundation for several iterative methods that approximate eigenvalues.

**Theorem 5.22.** If  $n \times n$  matrices  $A$  and  $B$  are **similar**, then they have **the same characteristic polynomial** and hence **the same eigenvalues** (with the same multiplicities).

**Proof.**  $B = P^{-1}AP$ . Then,

$$\begin{aligned} B - \lambda I &= P^{-1}AP - \lambda I \\ &= P^{-1}AP - \lambda P^{-1}P \\ &= P^{-1}(A - \lambda I)P, \end{aligned}$$

from which we conclude that

$$\det(B - \lambda I) = \det(P^{-1}) \det(A - \lambda I) \det(P) = \det(A - \lambda I).$$

**Diagonalization**

**Definition 5.23.** An  $n \times n$  matrix  $A$  is said to be **diagonalizable** if there exists an invertible matrix  $P$  and a diagonal matrix  $D$  such that

$$A = PDP^{-1} \quad (\text{or } P^{-1}AP = D) \quad (5.6)$$

That is, diagonalizable matrices are those similar to a diagonal matrix.

**Remark 5.24.** Let  $A$  be diagonalizable, i.e.,  $A = PDP^{-1}$ . Then

$$\begin{aligned} A^2 &= (PDP^{-1})(PDP^{-1}) = PD^2P^{-1} \\ A^k &= PD^kP^{-1} \\ A^{-1} &= PD^{-1}P^{-1} \quad (\text{when } A \text{ is invertible}) \\ \det A &= \det D \end{aligned} \quad (5.7)$$

Diagonalization enables us to compute  $A^k$  and  $\det A$  quickly.

**Self-study 5.25.** Let  $A = \begin{bmatrix} 7 & 2 \\ -4 & 1 \end{bmatrix}$ . Find a formula for  $A^k$ , given that

$$A = PDP^{-1}, \text{ where } P = \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix} \text{ and } D = \begin{bmatrix} 5 & 0 \\ 0 & 3 \end{bmatrix}.$$

**Solution.**

$$\text{Ans: } A^k = \begin{bmatrix} 2 \cdot 5^k - 3^k & 5^k - 3^k \\ 2 \cdot 3^k - 2 \cdot 5^k & 2 \cdot 3^k - 5^k \end{bmatrix}$$

**Theorem 5.26. (The Diagonalization Theorem)**

1. An  $n \times n$  matrix  $A$  is diagonalizable if and only if  $A$  has  $n$  linearly independent eigenvectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ .
2. In fact,  $A = PDP^{-1}$  if and only if columns of  $P$  are  $n$  linearly independent eigenvectors of  $A$ . In this case, the diagonal entries of  $D$  are the corresponding eigenvalues of  $A$ . That is,

$$\begin{aligned}
 P &= [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n], \\
 D &= \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n) = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}, \quad (5.8)
 \end{aligned}$$

where  $A\mathbf{v}_k = \lambda_k\mathbf{v}_k$ ,  $k = 1, 2, \dots, n$ .

**Proof.** Let  $P = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n]$  and  $D = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ , arbitrary matrices. Then,

$$AP = A[\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n] = [A\mathbf{v}_1 \ A\mathbf{v}_2 \ \cdots \ A\mathbf{v}_n], \quad (5.9)$$

while

$$PD = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n] \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} = [\lambda_1\mathbf{v}_1 \ \lambda_2\mathbf{v}_2 \ \cdots \ \lambda_n\mathbf{v}_n]. \quad (5.10)$$

( $\Rightarrow$ ) Now suppose  $A$  is diagonalizable and  $A = PDP^{-1}$ . Then we have  $AP = PD$ ; it follows from (5.9) and (5.10) that

$$[A\mathbf{v}_1 \ A\mathbf{v}_2 \ \cdots \ A\mathbf{v}_n] = [\lambda_1\mathbf{v}_1 \ \lambda_2\mathbf{v}_2 \ \cdots \ \lambda_n\mathbf{v}_n],$$

from which we conclude

$$A\mathbf{v}_k = \lambda_k\mathbf{v}_k, \quad k = 1, 2, \dots, n. \quad (5.11)$$

Furthermore,  $P$  is invertible  $\Rightarrow \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$  is linearly independent.

( $\Leftarrow$ ) It is almost trivial.  $\square$

**Example 5.27.** Diagonalize the following matrix, if possible.

$$A = \begin{bmatrix} 1 & 3 & 3 \\ -3 & -5 & -3 \\ 3 & 3 & 1 \end{bmatrix}$$

**Solution.**

1. Find the eigenvalues of  $A$ .
2. Find three linearly independent eigenvectors of  $A$ .
3. Construct  $P$  from the vectors in step 2.
4. Construct  $D$  from the corresponding eigenvalues.

Check:  $AP = PD$ ?

$$\text{Ans: } \lambda = 1, -2, -2. \mathbf{v}_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}, \mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \mathbf{v}_3 = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

```

_____ diagonalization.m _____
1  A = [1 3 3; -3 -5 -3; 3 3 1];
2  [P,D] = eig(A)    % A*P = P*D
3  P*D*inv(P)

```

```

_____ Results _____
1  P =
2     -0.5774    -0.7876     0.4206
3     0.5774     0.2074    -0.8164
4     -0.5774     0.5802     0.3957
5  D =
6     1.0000         0         0
7         0    -2.0000         0
8         0         0    -2.0000
9
10 ans =
11     1.0000     3.0000     3.0000
12    -3.0000    -5.0000    -3.0000
13     3.0000     3.0000     1.0000

```

**Attention: Eigenvectors corresponding to  $\lambda = -2$**

## 5.3. Dot Product, Length, and Orthogonality

**Definition 5.28.** Let  $\mathbf{u} = [u_1, u_2, \dots, u_n]^T$  and  $\mathbf{v} = [v_1, v_2, \dots, v_n]^T$  are vectors in  $\mathbb{R}^n$ . Then, the **dot product** (or **inner product**) of  $\mathbf{u}$  and  $\mathbf{v}$  is given by

$$\begin{aligned} \mathbf{u} \bullet \mathbf{v} &= \mathbf{u}^T \mathbf{v} = [u_1 \ u_2 \ \cdots \ u_n] \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \\ &= u_1 v_1 + u_2 v_2 + \cdots + u_n v_n = \sum_{k=1}^n u_k v_k. \end{aligned} \tag{5.12}$$

**Note:** In a **matrix-vector multiplication**  $A\mathbf{v}$ , the product of a row of  $A$  and the column vector  $\mathbf{v}$  is a dot product.

**Example 5.29.** Let  $\mathbf{u} = \begin{bmatrix} 1 \\ -2 \\ 2 \end{bmatrix}$  and  $\mathbf{v} = \begin{bmatrix} 3 \\ 2 \\ -4 \end{bmatrix}$ . Find  $\mathbf{u} \bullet \mathbf{v}$ .

**Solution.**

**Theorem 5.30.** Let  $\mathbf{u}$ ,  $\mathbf{v}$ , and  $\mathbf{w}$  be vectors in  $\mathbb{R}^n$ , and  $c$  be a scalar. Then

- $\mathbf{u} \bullet \mathbf{v} = \mathbf{v} \bullet \mathbf{u}$
- $(\mathbf{u} + \mathbf{v}) \bullet \mathbf{w} = \mathbf{u} \bullet \mathbf{w} + \mathbf{v} \bullet \mathbf{w}$
- $(c\mathbf{u}) \bullet \mathbf{v} = c(\mathbf{u} \bullet \mathbf{v}) = \mathbf{u} \bullet (c\mathbf{v})$
- $\mathbf{u} \bullet \mathbf{u} \geq 0$ , and  $\mathbf{u} \bullet \mathbf{u} = 0 \Leftrightarrow \mathbf{u} = \mathbf{0}$

**Definition 5.31.** The **length (norm)** of  $\mathbf{v}$  is nonnegative scalar  $\|\mathbf{v}\|$  defined by

$$\|\mathbf{v}\| = \sqrt{\mathbf{v} \bullet \mathbf{v}} = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} \quad \text{and} \quad \|\mathbf{v}\|^2 = \mathbf{v} \bullet \mathbf{v}. \quad (5.13)$$

**Note:** For any scalar  $c$ ,  $\|c\mathbf{v}\| = |c| \|\mathbf{v}\|$ .

**Example 5.32.** Let  $W$  be a subspace of  $\mathbb{R}^2$  spanned by  $\mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ . Find a **unit vector**  $\mathbf{u}$  that is a basis for  $W$ .

**Solution.**

### Distance in $\mathbb{R}^n$

**Definition 5.33.** For  $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ , the **distance** between  $\mathbf{u}$  and  $\mathbf{v}$  is

$$\text{dist}(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|, \quad (5.14)$$

the length of the vector  $\mathbf{u} - \mathbf{v}$ .

**Example 5.34.** Compute the distance between the vectors  $\mathbf{u} = (7, 1)$  and  $\mathbf{v} = (3, 2)$ .

**Solution.**

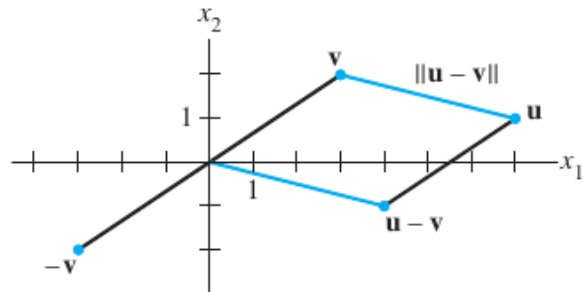


Figure 5.1: The distance between  $\mathbf{u}$  and  $\mathbf{v}$  is the length of  $\mathbf{u} - \mathbf{v}$ .

## Orthogonal Vectors

**Definition 5.35.** Two vectors  $\mathbf{u}$  and  $\mathbf{v}$  in  $\mathbb{R}^n$  are **orthogonal** if  $\mathbf{u} \bullet \mathbf{v} = 0$ .

**Theorem 5.36. The Pythagorean Theorem:** *Two vectors  $\mathbf{u}$  and  $\mathbf{v}$  are orthogonal if and only if*

$$\|\mathbf{u} + \mathbf{v}\|^2 = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2. \quad (5.15)$$

**Proof.** For all  $\mathbf{u}$  and  $\mathbf{v}$  in  $\mathbb{R}^n$ ,

$$\|\mathbf{u} + \mathbf{v}\|^2 = (\mathbf{u} + \mathbf{v}) \bullet (\mathbf{u} + \mathbf{v}) = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 + 2\mathbf{u} \bullet \mathbf{v}. \quad (5.16)$$

Thus,  $\mathbf{u}$  and  $\mathbf{v}$  are orthogonal  $\Leftrightarrow$  (5.15) holds □

**Note:** The **inner product** can be defined as

$$\mathbf{u} \bullet \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta, \quad (5.17)$$

where  $\theta$  is the angle between  $\mathbf{u}$  and  $\mathbf{v}$ .

**Example 5.37.** Use (5.17) to find the angle between  $\mathbf{u}$  and  $\mathbf{v}$ .

(a)  $\mathbf{u} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} -4 \\ 3 \end{bmatrix}$

(b)  $\mathbf{u} = \begin{bmatrix} 1 \\ \sqrt{3} \end{bmatrix}, \mathbf{v} = \begin{bmatrix} -1/2 \\ \sqrt{3}/2 \end{bmatrix}$

**Solution.**



## 5.4. Vector Norms, Matrix Norms, and Condition Numbers

### Vector Norms

**Definition 5.38.** A **norm** (or, **vector norm**) on  $\mathbb{R}^n$  is a function that assigns to each  $\mathbf{x} \in \mathbb{R}^n$  a nonnegative real number  $\|\mathbf{x}\|$  such that the following three properties are satisfied: for all  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$  and  $\lambda \in \mathbb{R}$ ,

$$\begin{aligned} \|\mathbf{x}\| &> 0 \text{ if } \mathbf{x} \neq 0 && \text{(positive definiteness)} \\ \|\lambda\mathbf{x}\| &= |\lambda| \|\mathbf{x}\| && \text{(homogeneity)} \\ \|\mathbf{x} + \mathbf{y}\| &\leq \|\mathbf{x}\| + \|\mathbf{y}\| && \text{(triangle inequality)} \end{aligned} \quad (5.18)$$

**Example 5.39.** The most common norms are

$$\|\mathbf{x}\|_p = \left( \sum_i |x_i|^p \right)^{1/p}, \quad 1 \leq p < \infty, \quad (5.19)$$

which we call the  **$p$ -norms**, and

$$\|\mathbf{x}\|_\infty = \max_i |x_i|, \quad (5.20)$$

which is called the **infinity-norm** or **maximum-norm**.

**Note:** Two of frequently used  $p$ -norms are

$$\|\mathbf{x}\|_1 = \sum_i |x_i|, \quad \|\mathbf{x}\|_2 = \left( \sum_i |x_i|^2 \right)^{1/2} \quad (5.21)$$

The 2-norm is also called the **Euclidean norm**, often denoted by  $\|\cdot\|$ .

**Example 5.40.** Let  $\mathbf{x} = [4, 2, -2, -4, 3]^T$ . Find  $\|\mathbf{x}\|_p$ , for  $p = 1, 2, \infty$ .

**Solution.**

**Note:** In general,  $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1$  for all  $\mathbf{x} \in \mathbb{R}^n$ ; see Exercise 5.5.

## Matrix Norms

**Definition 5.41.** A **matrix norm** on  $m \times n$  matrices is a vector norm on the  $mn$ -dimensional space, satisfying

$$\begin{aligned} \|A\| &\geq 0, \text{ and } \|A\| = 0 \Leftrightarrow A = 0 && \text{(positive definiteness)} \\ \|\lambda A\| &= |\lambda| \|A\| && \text{(homogeneity)} \\ \|A + B\| &\leq \|A\| + \|B\| && \text{(triangle inequality)} \end{aligned} \tag{5.22}$$

**Example 5.42.**  $\|A\|_F \equiv \left( \sum_{i,j} |a_{ij}|^2 \right)^{1/2} = \sqrt{\text{tr}(AA^T)}$  is called the **Frobenius norm**. Here “ $\text{tr}(B)$ ” is the **trace** of a square matrix  $B$ , *the sum of elements on the main diagonal*.

**Definition 5.43.** Once a vector norm  $\|\cdot\|$  has been specified, the **induced matrix norm** is defined by

$$\|A\| = \max_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} = \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|. \tag{5.23}$$

It is also called an **operator norm** or **subordinate norm**.

### Theorem 5.44.

(a) For all operator norms and the Frobenius norm,

$$\|A\mathbf{x}\| \leq \|A\| \|\mathbf{x}\|, \quad \|AB\| \leq \|A\| \|B\|. \tag{5.24}$$

(b)  $\|A\|_1 \equiv \max_{\|\mathbf{x}\|_1=1} \|A\mathbf{x}\|_1 = \max_j \sum_i |a_{ij}|$

(c)  $\|A\|_\infty \equiv \max_{\|\mathbf{x}\|_\infty=1} \|A\mathbf{x}\|_\infty = \max_i \sum_j |a_{ij}|$

(d)  $\|A\|_2 \equiv \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \sqrt{\lambda_{\max}(A^T A)}$ ,  
where  $\lambda_{\max}$  denotes the largest eigenvalue.

(e)  $\|A\|_2 = \|A^T\|_2$ .

(f)  $\|A\|_2 = \max_i |\lambda_i(A)|$ , when  $A^T A = A A^T$  (**normal matrix**).

**Definition 5.45.** Let  $A \in \mathbb{R}^{n \times n}$  be invertible. Then

$$\kappa(A) \equiv \|A\| \|A^{-1}\| \quad (5.25)$$

is called the **condition number** of  $A$ , associated to the matrix norm.

**Example 5.46.** Let  $A = \begin{bmatrix} 1 & 2 & -2 \\ 0 & 1 & 1 \\ 1 & -2 & 2 \end{bmatrix}$ . Then, we have

$$A^{-1} = \frac{1}{8} \begin{bmatrix} 4 & 0 & -4 \\ 1 & 4 & -1 \\ -1 & 4 & 1 \end{bmatrix} \quad \text{and} \quad A^T A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 9 & -7 \\ 0 & -7 & 9 \end{bmatrix}.$$

- Find  $\|A\|_1$ ,  $\|A\|_\infty$ , and  $\|A\|_2$ .
- Compute the  $\ell^1$ -condition number  $\kappa_1(A)$ .

**Solution.**

**Example 5.47.** One may consider the infinity-norm as the limit of  $p$ -norms, as  $p \rightarrow \infty$ .

**Solution.**

### The Induced Matrix 2-Norm of $A$

**Definition 5.48.** The *induced matrix 2-norm* can be defined by

$$\|A\|_2 = \max_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \max_{\|\mathbf{x}\|_2=\|\mathbf{y}\|_2=1} |\mathbf{y}^T A\mathbf{x}|. \quad (5.26)$$

It is also called an **operator 2-norm** or **subordinate 2-norm**.

### Theorem 5.49. Properties of the Matrix 2-Norm

- (a)  $\|A^T\|_2 = \|A\|_2$
- (b)  $\|A^T A\|_2 = \|A\|_2^2$
- (c)  $\|A A^T\|_2 = \|A\|_2^2$

#### Proof.

(a) The claim follows from the fact that  $\mathbf{y}^T A\mathbf{x}$  is a scalar and therefore  $(\mathbf{y}^T A\mathbf{x})^T = \mathbf{x}^T A^T \mathbf{y}$  and  $|\mathbf{y}^T A\mathbf{x}| = |\mathbf{x}^T A^T \mathbf{y}|$ .

(b) Using the **Cauchy-Schwarz inequality**,

$$\begin{aligned} \|A^T A\|_2 &= \max_{\|\mathbf{x}\|_2=\|\mathbf{y}\|_2=1} |\mathbf{y}^T A^T A\mathbf{x}| \\ &\leq \max_{\|\mathbf{x}\|_2=\|\mathbf{y}\|_2=1} \|A\mathbf{y}\|_2 \|A\mathbf{x}\|_2 = \|A\|_2^2. \end{aligned} \quad (5.27)$$

Now, we choose a unit vector  $\mathbf{z}$ , for which  $\|A\mathbf{z}\|_2 = \|A\|_2$ . Then

$$\begin{aligned} \|A\|_2^2 &= \|A\mathbf{z}\|_2^2 = (A\mathbf{z})^T (A\mathbf{z}) = \mathbf{z}^T A^T A \mathbf{z} \\ &\leq \max_{\|\mathbf{x}\|_2=\|\mathbf{y}\|_2=1} |\mathbf{y}^T A^T A\mathbf{x}| \equiv \|A^T A\|_2. \end{aligned} \quad (5.28)$$

(c) Using the results in (a) and (b),

$$\|A\|_2^2 = \|A^T\|_2^2 = \|(A^T)^T A^T\|_2 = \|A A^T\|_2, \quad (5.29)$$

which completes the proof.  $\square$

## 5.5. Power Method and Inverse Power Method for Eigenvalues

### 5.5.1. The Power Method

The **power method** is an **iterative algorithm**:

Given a square matrix  $A \in \mathbb{R}^{n \times n}$ , the algorithm finds a number  $\lambda$ , which is **the largest eigenvalue of  $A$**  (in modulus), and **its corresponding eigenvector  $\mathbf{v}$** .

**Assumption.** To apply the power method, we assume that  $A \in \mathbb{R}^{n \times n}$  has

- $n$  eigenvalues  $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$ ,
- $n$  associated eigenvectors  $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ , which are **linearly independent**, and
- **exactly one eigenvalue** that is largest in magnitude,  $\lambda_1$ :

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \dots \geq |\lambda_n|. \quad (5.30)$$

The power method approximates the largest eigenvalue  $\lambda_1$  and its associated eigenvector  $\mathbf{v}_1$ .

#### Derivation of Power Iteration

- Since eigenvectors  $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$  are linearly independent, any vector  $\mathbf{x} \in \mathbb{R}^n$  can be expressed as

$$\mathbf{x} = \sum_{j=1}^n \beta_j \mathbf{v}_j, \quad (5.31)$$

for some constants  $\{\beta_1, \beta_2, \dots, \beta_n\}$ .

- Multiplying both sides of (5.31) by  $A$  and  $A^2$  gives

$$\begin{aligned} A\mathbf{x} &= A\left(\sum_{j=1}^n \beta_j \mathbf{v}_j\right) = \sum_{j=1}^n \beta_j A\mathbf{v}_j = \sum_{j=1}^n \beta_j \lambda_j \mathbf{v}_j, \\ A^2\mathbf{x} &= A\left(\sum_{j=1}^n \beta_j \lambda_j \mathbf{v}_j\right) = \sum_{j=1}^n \beta_j \lambda_j^2 \mathbf{v}_j. \end{aligned} \quad (5.32)$$

- In general,

$$A^k \mathbf{x} = \sum_{j=1}^n \beta_j \lambda_j^k \mathbf{v}_j, \quad k = 1, 2, \dots, \quad (5.33)$$

which gives

$$A^k \mathbf{x} = \lambda_1^k \cdot \sum_{j=1}^n \beta_j \left(\frac{\lambda_j}{\lambda_1}\right)^k \mathbf{v}_j = \lambda_1^k \cdot \left[ \beta_1 \left(\frac{\lambda_1}{\lambda_1}\right)^k \mathbf{v}_1 + \beta_2 \left(\frac{\lambda_2}{\lambda_1}\right)^k \mathbf{v}_2 + \dots + \beta_n \left(\frac{\lambda_n}{\lambda_1}\right)^k \mathbf{v}_n \right]. \quad (5.34)$$

- For  $j = 2, 3, \dots, n$ , since  $|\lambda_j/\lambda_1| < 1$ , we have  $\lim_{k \rightarrow \infty} |\lambda_j/\lambda_1|^k = 0$ , and

$$\lim_{k \rightarrow \infty} A^k \mathbf{x} = \lim_{k \rightarrow \infty} \lambda_1^k \beta_1 \mathbf{v}_1. \quad (5.35)$$

**Remark 5.50.** The sequence in (5.35) converges to 0 if  $|\lambda_1| < 1$  and diverges if  $|\lambda_1| > 1$ , provided that  $\beta_1 \neq 0$ .

- The entries of  $A^k \mathbf{x}$  will grow with  $k$  if  $|\lambda_1| > 1$  and will go to 0 if  $|\lambda_1| < 1$ .
- In either case, it is hard to decide the largest eigenvalue  $\lambda_1$  and its associated eigenvector  $\mathbf{v}_1$ .
- **To take care of that possibility**, we scale  $A^k \mathbf{x}$  in an appropriate manner to ensure that the limit in (5.35) is finite and nonzero.

**Algorithm 5.51. (The Power Iteration)** Given  $\mathbf{x} \neq 0$ :

**initialization** :  $\mathbf{x}^0 = \mathbf{x}/\|\mathbf{x}\|_\infty$   
**for**  $k = 1, 2, \dots$   
 $\mathbf{y}^k = A\mathbf{x}^{k-1}; \quad \mu_k = \|\mathbf{y}^k\|_\infty$   
 $\mathbf{x}^k = \mathbf{y}^k/\mu_k$   
**end for**

(5.36)

**Claim 5.52.** Let  $\{\mathbf{x}^k, \mu_k\}$  be a sequence produced by the power method. Then,

$$\mathbf{x}^k \rightarrow \mathbf{v}_1, \quad \mu_k \rightarrow |\lambda_1|, \quad \text{as } k \rightarrow \infty. \quad (5.37)$$

More precisely, the power method converges as

$$\mu_k = |\lambda_1| + \mathcal{O}(|\lambda_2/\lambda_1|^k). \quad (5.38)$$

**Example 5.53.** The matrix  $A = \begin{bmatrix} 5 & -2 & 2 \\ -2 & 3 & -4 \\ 2 & -4 & 3 \end{bmatrix}$  has eigenvalues and eigenvectors as follows

$$\text{eig}(A) = \begin{bmatrix} 9 \\ 3 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 0 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix}$$

Verify that the sequence produced by the power method converges to the largest eigenvalue and its associated eigenvector.

**Solution.** The algorithm is implemented in **both Matlab and Python.**

```

power_iteration.m
1  A = [5 -2 2; -2 3 -4; 2 -4 3];
2  [V,D] = eig(A);
3  values = diag(D)';
4  [~,ind] = sort(values,'descend');
5  values = values(ind)
6  V = V(:,ind); V = V./max(abs(V),[],1)
7
8  x = [1 0 0]';
9  fmt = ['k=%2d: x=[',repmat('%.5f ',1,numel(x)-1),'%'.5f], ',...
10      'mu=%.5f (error=%.7f)\n'];
11
12 for k=1:10
13     y = A*x;
14     [~,ind] = max(abs(y)); mu = y(ind);
15     x =y/mu;
16     fprintf(fmt,k,x,mu,abs(values(1)-mu))
17 end

```

```

power_iteration.py
1  import numpy as np;
2  np.set_printoptions(suppress=True)
3
4  A = np.array([[5,-2,2],[-2,3,-4],[2,-4,3]])
5  values, EVectors = np.linalg.eig(A)
6
7  # Sorting eigenvalues: descend
8  idx = values.argsort()[::-1]
9  values = values[idx]; EVectors = EVectors[:,idx]
10 EVectors /= np.max(abs(EVectors),axis=0) #normalize
11

```

```

12 print('evalues=',evalues)
13 print('EVectors=\n',EVectors)
14
15 x = np.array([1,0,0]).T
16 for k in range(10):
17     y = A.dot(x)
18     ind = np.argmax(np.abs(y)); mu = y[ind]
19     x = y/mu
20     print('k=%2d; x=[%.5f, %.5f, %.5f]; mu=%.5f (error=%.7f)'
21           '%(k,*x,mu,np.abs(evalues[0]-mu)) ');

```

The results are the same; here is the output from the Matlab code.

```

----- Output from power_iteration.m -----
1  evalues =
2     9.0000    3.0000   -1.0000
3
4  V =
5     1.0000e+00    1.0000e+00    3.9252e-17
6    -1.0000e+00    5.0000e-01    1.0000e+00
7     1.0000e+00   -5.0000e-01    1.0000e+00
8
9  k= 1: x=[1.00000, -0.40000, 0.40000], mu=5.00000 (error=4.0000000)
10 k= 2: x=[1.00000, -0.72727, 0.72727], mu=6.60000 (error=2.4000000)
11 k= 3: x=[1.00000, -0.89655, 0.89655], mu=7.90909 (error=1.0909091)
12 k= 4: x=[1.00000, -0.96386, 0.96386], mu=8.58621 (error=0.4137931)
13 k= 5: x=[1.00000, -0.98776, 0.98776], mu=8.85542 (error=0.1445783)
14 k= 6: x=[1.00000, -0.99590, 0.99590], mu=8.95102 (error=0.0489796)
15 k= 7: x=[1.00000, -0.99863, 0.99863], mu=8.98358 (error=0.0164159)
16 k= 8: x=[1.00000, -0.99954, 0.99954], mu=8.99452 (error=0.0054820)
17 k= 9: x=[1.00000, -0.99985, 0.99985], mu=8.99817 (error=0.0018284)
18 k=10: x=[1.00000, -0.99995, 0.99995], mu=8.99939 (error=0.0006096)

```

Notice that  $|9 - \mu_k| \approx \frac{1}{3}|9 - \mu_{k-1}|$ , for which  $|\lambda_2/\lambda_1| = \frac{1}{3}$ .



### 5.5.2. The Inverse Power Method

Some applications require to find an eigenvalue of the matrix  $A$ , near a prescribed value  $q$ . The **inverse power method** is a variant of the Power method to solve such a problem.

- We begin with the eigenvalues and eigenvectors of  $(A - qI)^{-1}$ . Let

$$A\mathbf{v}_i = \lambda_i\mathbf{v}_i, \quad i = 1, 2, \dots, n. \quad (5.39)$$

- Then it is easy to see that

$$(A - qI)\mathbf{v}_i = (\lambda_i - q)\mathbf{v}_i. \quad (5.40)$$

Thus, we obtain

$$(A - qI)^{-1}\mathbf{v}_i = \frac{1}{\lambda_i - q}\mathbf{v}_i. \quad (5.41)$$

- That is, when  $q \notin \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ , the eigenvalues of  $(A - qI)^{-1}$  are

$$\frac{1}{\lambda_1 - q}, \frac{1}{\lambda_2 - q}, \dots, \frac{1}{\lambda_n - q}, \quad (5.42)$$

with the same eigenvectors  $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$  of  $A$ .

**Algorithm 5.54. (Inverse Power Method)** Applying the power method to  $(A - qI)^{-1}$  gives the **inverse power method**. Given  $\mathbf{x} \neq \mathbf{0}$ :

**set** :  $\mathbf{x}^0 = \mathbf{x}/\|\mathbf{x}\|_\infty$

**for**  $k = 1, 2, \dots$

$$\mathbf{y}^k = (A - qI)^{-1}\mathbf{x}^{k-1}; \quad \mu_k = \|\mathbf{y}^k\|_\infty \quad (5.43)$$

$$\mathbf{x}^k = \mathbf{y}^k/\mu_k$$

$$\lambda_k = 1/\mu_k + q$$

**end for**

**Note:** All eigenvalues of a square matrix can be found simultaneously by applying the **QR iteration**; see §9.5 below.

**Example 5.55.** The matrix  $A$  is as in Example 5.53:  $A = \begin{bmatrix} 5 & -2 & 2 \\ -2 & 3 & -4 \\ 2 & -4 & 3 \end{bmatrix}$ .

Find the the eigenvalue of  $A$  nearest to  $q = 4$ , using the inverse power method.

**Solution.**

```

inverse_power.m
1  A = [5 -2 2; -2 3 -4; 2 -4 3];
2  [V,D] = eig(A);
3  values = diag(D)';
4  [~,ind] = sort(values,'descend');
5  values = values(ind)
6  V = V(:,ind); V = V./max(abs(V),[],1)
7
8  x = [1 0 0]';
9  fmt = ['k=%2d: x = ',repmat('%.5f, ',1,numel(x)-1), '%.5f], ',...
10       'lambda=%.7f (error = %.7f)\n'];
11
12  q = 4; B = inv(A-q*eye(3));
13  for k=1:10
14     y = B*x;
15     [~,ind] = max(abs(y)); mu = y(ind);
16     x =y/mu;
17     lambda = 1/mu + q;
18     fprintf(fmt,k,x,lambda,abs(values(2)-lambda))
19  end

```

```

inverse_power.py
1  import numpy as np;
2  np.set_printoptions(suppress=True)
3
4  A = np.array([[5,-2,2],[-2,3,-4],[2,-4,3]])
5  values, EVectors = np.linalg.eig(A)
6
7  # Sorting eigenvalues: largest to smallest
8  idx = values.argsort()[::-1]
9  values = values[idx]; EVectors = EVectors[:,idx]
10 EVectors /= np.max(abs(EVectors),axis=0) #normalize
11
12 print('values=',values)
13 print('EVectors=\n',EVectors)
14
15 q = 4; x = np.array([1,0,0]).T

```

```

16 B = np.linalg.inv(A-q*np.identity(3))
17 for k in range(10):
18     y = B.dot(x)
19     ind = np.argmax(np.abs(y)); mu = y[ind]
20     x = y/mu
21     Lambda = 1/mu + q
22     print('k=%2d; x=[%.5f, %.5f, %.5f]; Lambda=%.7f (error=%.7f)'
23           '%(k,*x,Lambda,np.abs(evalues[1]-Lambda)) ');

```

----- Output from inverse\_power.py -----

```

1  evalues= [ 9.  3. -1.]
2  EVectors=
3  [[-1. -1. -0. ]
4   [ 1. -0.5  1. ]
5   [-1.  0.5  1. ]]
6  k= 0; x=[1.00000, 0.66667, -0.66667]; Lambda=2.3333333 (error=0.6666667)
7  k= 1; x=[1.00000, 0.47059, -0.47059]; Lambda=3.1176471 (error=0.1176471)
8  k= 2; x=[1.00000, 0.50602, -0.50602]; Lambda=2.9759036 (error=0.0240964)
9  k= 3; x=[1.00000, 0.49880, -0.49880]; Lambda=3.0047962 (error=0.0047962)
10 k= 4; x=[1.00000, 0.50024, -0.50024]; Lambda=2.9990398 (error=0.0009602)
11 k= 5; x=[1.00000, 0.49995, -0.49995]; Lambda=3.0001920 (error=0.0001920)
12 k= 6; x=[1.00000, 0.50001, -0.50001]; Lambda=2.9999616 (error=0.0000384)
13 k= 7; x=[1.00000, 0.50000, -0.50000]; Lambda=3.0000077 (error=0.0000077)
14 k= 8; x=[1.00000, 0.50000, -0.50000]; Lambda=2.9999985 (error=0.0000015)
15 k= 9; x=[1.00000, 0.50000, -0.50000]; Lambda=3.0000003 (error=0.0000003)

```

**Note:** When  $q = 4$ , eigenvalues of  $(A - qI)^{-1}$  are  $\{1/5, -1, -1/5\}$ .

- The initial vector:  $\mathbf{x}_0 = [1, 0, 0]^T = \frac{1}{3}(\mathbf{v}_1 + \mathbf{v}_2)$ ; see Example 5.53.
- Thus, each iteration must reduce the error **by a factor of 5**.

----- When  $q = 3.1$  -----

```

1  k= 0; x=[1.00000, 0.51282, -0.51282]; Lambda=2.9487179 (error=0.0512821)
2  k= 1; x=[1.00000, 0.49978, -0.49978]; Lambda=3.0008617 (error=0.0008617)
3  k= 2; x=[1.00000, 0.50000, -0.50000]; Lambda=2.9999854 (error=0.0000146)
4  k= 3; x=[1.00000, 0.50000, -0.50000]; Lambda=3.0000002 (error=0.0000002)
5  k= 4; x=[1.00000, 0.50000, -0.50000]; Lambda=3.0000000 (error=0.0000000)

```

See Exercise 5.8.

## Exercises for Chapter 5

5.1. Let  $A = \begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}$ . Write  $5A$ . Is  $\det(5A) = 5\det A$ ?

5.2. Let  $A = \begin{bmatrix} 1 & 1 & -3 \\ 0 & 2 & 8 \\ 2 & 4 & 2 \end{bmatrix}$ .

(a) Find  $\det A$ .

(b) Let  $\mathcal{U} = [0, 1]^3$ , the unit cube. What can you say about  $A(\mathcal{U})$ , the image of  $\mathcal{U}$  under the matrix multiplication by  $A$ .

5.3. Use pencil-and-paper to compute  $\det(B^6)$ , where  $B = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ .

Ans: 64

5.4. A matrix is not always diagonalizable. Let  $A = \begin{bmatrix} 3 & 1 & 0 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{bmatrix}$ . Use  $[P, D] = \text{eig}(A)$  in

Matlab to verify

(a)  $P$  does not have its inverse.

(b)  $AP = PD$ .

5.5. Show that  $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1$  for all  $\mathbf{x} \in \mathbb{R}^n$ .

5.6. The matrix in Example 5.55 has eigenvalues  $\{-6, -3, -1\}$ . We may try to find the eigenvalue of  $A$  nearest to  $q = -3.1$ .

(a) Estimate (mathematically) the convergence speed of the inverse power method.

(b) Verify it by implementing the inverse power method, with  $\mathbf{x}_0 = [0, 1, 0]^T$ .

5.7. Let  $A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & 0 & -1 \\ 0 & 0 & 4 & -2 \\ 0 & -1 & -2 & 4 \end{bmatrix}$ . Use indicated methods to approximate eigenvalues and

their associated eigenvectors of  $A$  within to  $10^{-12}$  accuracy.

(a) The power method, the largest eigenvalue.

(b) The inverse power method, an eigenvalue near  $q = 3$ .

(c) The inverse power method, the smallest eigenvalue.

5.8. What is the *theoretical* error reduction rate for the convergence of the inverse power iteration, when  $q = 3.1$ , shown on p.161.

Ans: 59.

## CHAPTER 6

# Multivariable Calculus

In this chapter, we will learn subjects in multivariable calculus, such as

- The gradient vector
- Optimization: Method of Lagrange multipliers
- The gradient descent method

### Contents of Chapter 6

6.1. Multi-Variable Functions and Their Partial Derivatives . . . . .	164
6.2. Directional Derivatives and the Gradient Vector . . . . .	168
6.3. Optimization: Method of Lagrange Multipliers . . . . .	173
6.4. The Gradient Descent Method . . . . .	181
Exercises for Chapter 6 . . . . .	192

## 6.1. Multi-Variable Functions and Their Partial Derivatives

### 6.1.1. Functions of Several Variables

**Definition 6.1.** A **function of two variables**,  $f$ , is a rule that assigns each ordered pair of real numbers  $(x, y)$  in a set  $D \subset \mathbb{R}^2$  a unique real number denoted by  $f(x, y)$ . The set  $D$  is called the **domain** of  $f$  and its **range** is the set of values that  $f$  takes on, that is,  $\{f(x, y) : (x, y) \in D\}$ .

**Definition 6.2.** Let  $f$  be a function of two variables, and  $z = f(x, y)$ . Then  $x$  and  $y$  are called **independent variables** and  $z$  is called a **dependent variable**.

**Example 6.3.** Let  $f(x, y) = \frac{\sqrt{x + y + 1}}{x - 1}$ . Evaluate  $f(3, 2)$  and give its domain.

**Solution.**

$$\text{Ans: } f(3, 2) = \sqrt{6}/2; D = \{(x, y) : x + y + 1 \geq 0, x \neq 1\}$$

**Example 6.4.** Find the domain and the range of

$$f(x, y) = \sqrt{9 - x^2 - y^2}.$$

**Solution.**

## 6.1.2. First-order Partial Derivatives

**Recall:** A function  $y = f(x)$  is **differentiable** at  $a$  if

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h} \text{ exists.}$$

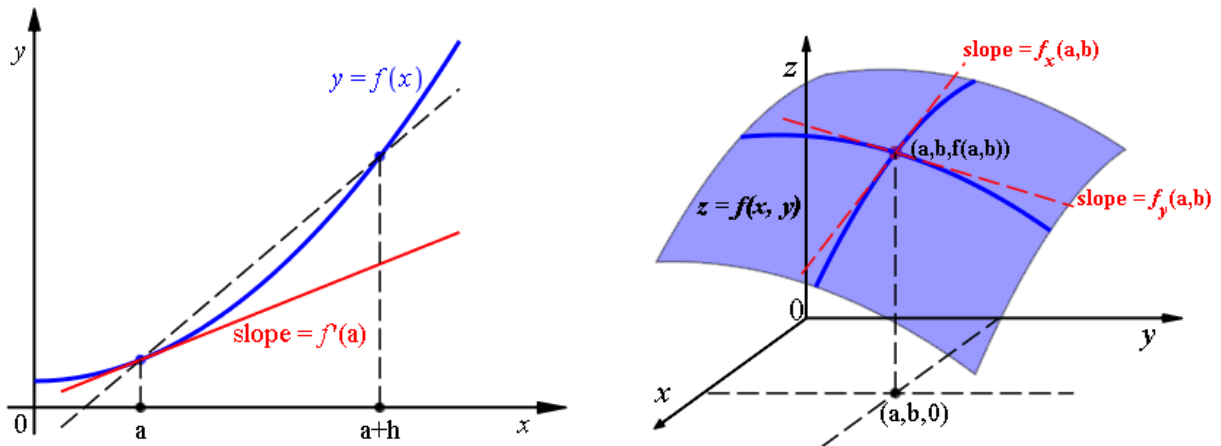


Figure 6.1: Ordinary derivative  $f'(a)$  and partial derivatives  $f_x(a, b)$  and  $f_y(a, b)$ .

$$f_x = \partial f / \partial x$$

Let  $f$  be a function of two variables  $(x, y)$ . Suppose we let only  $x$  vary while keeping  $y$  fixed, say  $y = b$ . Then  $g(x) := f(x, b)$  is a function of a single variable. If  $g$  is differentiable at  $a$ , then we call it the **partial derivative of  $f$  with respect to  $x$  at  $(a, b)$**  and denoted by  $f_x(a, b)$ .

$$\begin{aligned} g'(a) &= \lim_{h \rightarrow 0} \frac{g(a+h) - g(a)}{h} \\ &= \lim_{h \rightarrow 0} \frac{f(a+h, b) - f(a, b)}{h} =: f_x(a, b). \end{aligned} \tag{6.1}$$

$$f_y = \partial f / \partial y$$

Similarly, the **partial derivative of  $f$  with respect to  $y$  at  $(a, b)$** , denoted by  $f_y(a, b)$ , is obtained keeping  $x$  fixed, say  $\bar{x} = \bar{a}$ , and finding the ordinary derivative at  $b$  of  $G(y) := f(a, y)$ :

$$\begin{aligned} G'(b) &= \lim_{h \rightarrow 0} \frac{G(b+h) - G(b)}{h} \\ &= \lim_{h \rightarrow 0} \frac{f(a, b+h) - f(a, b)}{h} =: f_y(a, b). \end{aligned} \tag{6.2}$$

**Example 6.5.** Find  $f_x(0, 0)$ , when  $f(x, y) = \sqrt[3]{x^3 + y^3}$ .

**Solution.** Using the definition,

$$f_x(0, 0) = \lim_{h \rightarrow 0} \frac{f(h, 0) - f(0, 0)}{h}$$

Ans: 1

**Definition 6.6.** If  $f$  is a function of two variables, its **partial derivatives** are the functions  $f_x = \frac{\partial f}{\partial x}$  and  $f_y = \frac{\partial f}{\partial y}$  defined by:

$$\begin{aligned} f_x(x, y) &= \frac{\partial f}{\partial x}(x, y) = \lim_{h \rightarrow 0} \frac{f(x+h, y) - f(x, y)}{h} \quad \text{and} \\ f_y(x, y) &= \frac{\partial f}{\partial y}(x, y) = \lim_{h \rightarrow 0} \frac{f(x, y+h) - f(x, y)}{h}. \end{aligned} \tag{6.3}$$



**Observation 6.7. Partial Derivatives**

- **The partial derivative with respect to  $x$**  represents the slope of the tangent lines to the curve that are parallel to the  $xz$ -plane (i.e. in the direction of  $\langle 1, 0, \cdot \rangle$ ).
- Similarly, **the partial derivative with respect to  $y$**  represents the slope of the tangent lines to the curve that are parallel to the  $yz$ -plane (i.e. in the direction of  $\langle 0, 1, \cdot \rangle$ ).

**Rule for finding Partial Derivatives of  $z = f(x, y)$** 

- To find  $f_x$ , regard  $y$  as a constant and differentiate  $f$  w.r.t.  $x$ .
- To find  $f_y$ , regard  $x$  as a constant and differentiate  $f$  w.r.t.  $y$ .

**Example 6.8.** If  $f(x, y) = x^3 + x^2y^3 - 2y^2$ , find  $f_x(2, 1)$ ,  $f_y(2, 1)$ , and  $f_{xy}(2, 1)$ .  
**Solution.**

$$\text{Ans: } f_x(2, 1) = 16; f_y(2, 1) = 8$$

**Example 6.9. (Functions of Three Variables).** Let  $f(x, y, z) = \sin\left(\frac{xz}{1+y}\right)$ .

Find the first partial derivatives of  $f(x, y, z)$ .

**Solution.**

## 6.2. Directional Derivatives and the Gradient Vector

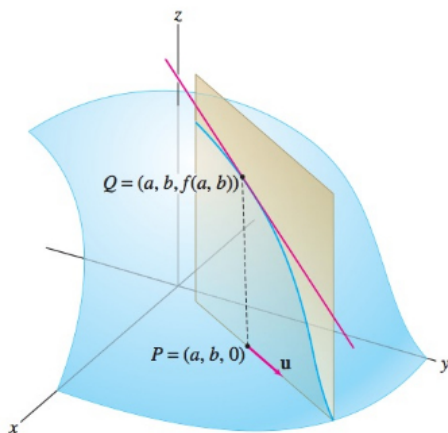


Figure 6.2

**Recall:** For  $z = f(x, y)$ , the partial derivatives  $(f_x, f_y)$  represent the **rates of change of  $z$**  in the  $(x, y)$ -directions, i.e., in the directions of the unit vectors  $(\mathbf{i}, \mathbf{j})$ .

**Note:** It would be nice to be able to find the slope of the tangent line to a surface  $S$  in **the direction of an arbitrary unit vector  $\mathbf{u} = \langle a, b \rangle$** .

**Definition 6.10.** The **directional derivative** of  $f$  at  $\mathbf{x}_0 = (x_0, y_0)$  in the direction of a unit vector  $\mathbf{u} = \langle a, b \rangle$  is

$$D_{\mathbf{u}}f(x_0, y_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + ha, y_0 + hb) - f(x_0, y_0)}{h}, \quad (6.4)$$

if the limit exists.

Note that

$$\begin{aligned} f(x_0 + ha, y_0 + hb) - f(x_0, y_0) &= f(x_0 + ha, y_0 + hb) - f(x_0, y_0 + hb) \\ &\quad + f(x_0, y_0 + hb) - f(x_0, y_0) \end{aligned}$$

Thus

$$\begin{aligned} \frac{f(x_0 + ha, y_0 + hb) - f(x_0, y_0)}{h} &= a \frac{f(x_0 + ha, y_0 + hb) - f(x_0, y_0 + hb)}{ha} \\ &\quad + b \frac{f(x_0, y_0 + hb) - f(x_0, y_0)}{hb}, \end{aligned}$$

which converges to " $a f_x(x_0, y_0) + b f_y(x_0, y_0)$ " as  $h \rightarrow 0$ .

**Theorem 6.11.** If  $f$  is a differentiable function of  $x$  and  $y$ , then  $f$  has a **directional derivative** in the direction of any unit vector  $\mathbf{u} = \langle a, b \rangle$  and

$$\begin{aligned} D_{\mathbf{u}}f(x, y) &= f_x(x, y)a + f_y(x, y)b \\ &= \langle f_x(x, y), f_y(x, y) \rangle \cdot \langle a, b \rangle \\ &= \langle f_x(x, y), f_y(x, y) \rangle \cdot \mathbf{u}. \end{aligned} \quad (6.5)$$

**Example 6.12.** Let  $f(x, y) = x^3 + 2xy + y^4$ . Find the directional derivative  $D_{\mathbf{u}}f(x, y)$ , when  $\mathbf{u}$  is the unit vector given by the angle  $\theta = \frac{\pi}{4}$ . What is  $D_{\mathbf{u}}f(2, 3)$ ?

**Solution.**  $\mathbf{u} = \langle \cos(\pi/4), \sin(\pi/4) \rangle = \langle 1/\sqrt{2}, 1/\sqrt{2} \rangle$ .

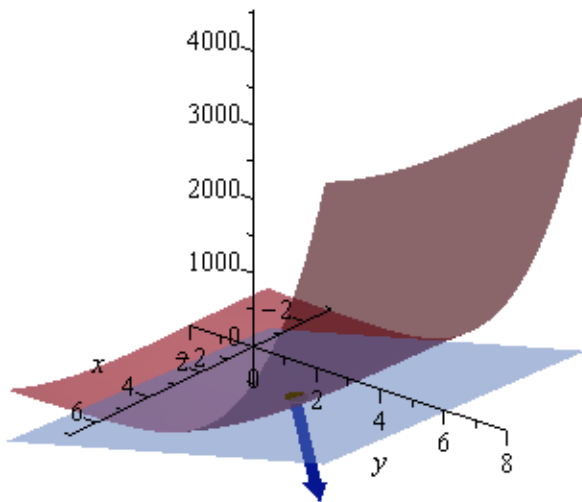


Figure 6.3

*Ans:*  $65\sqrt{2}$

**Self-study** 6.13. Find the directional derivative of  $f(x, y) = x + \sin(xy)$  at the point  $(1, 0)$  in the direction given by the angle  $\theta = \pi/3$ .

**Solution.**

*Ans:*  $(1 + \sqrt{3})/2$

**Example** 6.14. (Functions of Three Variables).

If  $f(x, y, z) = x^2 - 2y^2 + z^4$ , find the directional derivative of  $f$  at  $(1, 3, 1)$  in the direction of  $\mathbf{v} = \langle 2, -2, -1 \rangle$ .

**Solution.**

*Ans:* 8

## The Gradient Vector

**Definition 6.15.** Let  $f$  be a differentiable function of two variables  $x$  and  $y$ . Then the **gradient** of  $f$  is the vector function

$$\nabla f(x, y) = \langle f_x(x, y), f_y(x, y) \rangle = \frac{\partial f}{\partial x} \mathbf{i} + \frac{\partial f}{\partial y} \mathbf{j}. \quad (6.6)$$

**Example 6.16.** If  $f(x, y) = \sin(x) + e^{xy}$ , find  $\nabla f(x, y)$  and  $\nabla f(0, 1)$ .

**Solution.**

*Ans:*  $\langle 2, 0 \rangle$

**Remark 6.17.** With this notation of the gradient vector, we can rewrite

$$D_{\mathbf{u}}f(x, y) = \nabla f(x, y) \cdot \mathbf{u} = f_x(x, y)a + f_y(x, y)b, \quad \text{where } \mathbf{u} = \langle a, b \rangle. \quad (6.7)$$

**Example 6.18.** Find the directional derivative of  $f(x, y) = x^2y^3 - 4y$  at the point  $(2, -1)$  and in the direction of the vector  $\mathbf{v} = \langle 3, 4 \rangle$ .

**Solution.**

*Ans:* 4

### Maximizing the Directional Derivative

**Note:** Let  $\theta$  is the angle between  $\nabla f$  and  $\mathbf{u}$ . Then

$$D_{\mathbf{u}}f = \nabla f \cdot \mathbf{u} = |\nabla f| |\mathbf{u}| \cos \theta = |\nabla f| \cos \theta,$$

of which the maximum occurs when  $\theta = 0$ .

**Theorem 6.19.** Let  $f$  be a differentiable function of two or three variables. Then

$$\max_{\mathbf{u}} D_{\mathbf{u}}f(\mathbf{x}) = |\nabla f(\mathbf{x})| \quad (6.8)$$

and it occurs when  $\mathbf{u}$  has the same direction as  $\nabla f(\mathbf{x})$ .

**Example 6.20.** Let  $f(x, y) = xe^y$ .

- Find the rate of change of  $f$  at  $P(1, 0)$  in the direction from  $P$  to  $Q(-1, 2)$ .
- In what direction does  $f$  have the maximum rate of change? What is the maximum rate of change?

**Solution.**

Ans: (a) 0; (b)  $\sqrt{2}$

**Remark 6.21.** Let  $\mathbf{u} = \frac{\nabla f(\mathbf{x})}{|\nabla f(\mathbf{x})|}$ , the unit vector in the gradient direction. Then

$$D_{\mathbf{u}}f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u} = \nabla f(\mathbf{x}) \cdot \frac{\nabla f(\mathbf{x})}{|\nabla f(\mathbf{x})|} = |\nabla f(\mathbf{x})|. \quad (6.9)$$

This implies that the directional derivative is maximized in the gradient direction.

**Claim 6.22.** The **gradient direction** is the direction where the function changes fastest, more precisely, **increases fastest!**

## 6.3. Optimization: Method of Lagrange Multipliers

**Recall:** (Claim 6.22) The **gradient direction** is the direction where the function changes fastest, more precisely, **increases fastest!**

### Level Curves

**Example 6.23.** Consider the **unit circle**, the circle of radius 1:

$$F(x, y) = x^2 + y^2 = 1. \quad (6.10)$$

What can you say about the gradient of  $F$ ,  $\nabla F$ ?

**Solution.** The curve can be parametrized as

$$\mathbf{r}(t) = \langle x(t), y(t) \rangle = \langle \cos t, \sin t \rangle, \quad 0 \leq t \leq 2\pi. \quad (6.11)$$

- Apply the Chain Rule to have

$$\frac{d}{dt}F = F_x \frac{dx}{dt} + F_y \frac{dy}{dt} = 0,$$

and therefore

$$\nabla F \cdot \langle -\sin t, \cos t \rangle = 0. \quad (6.12)$$

- Note that  $\langle -\sin t, \cos t \rangle = \mathbf{r}'(t)$  is the tangential direction to the unit circle. Thus  $\nabla F$  must be normal to the curve.
- Indeed,

$$\nabla F = \langle 2x, 2y \rangle, \quad (6.13)$$

which is normal to the curve and the fastest increasing direction.  $\square$

**Claim 6.24.** Given a level curve  $F(\mathbf{x}) = k$ , the gradient vector  $\nabla F(\mathbf{x})$  is **normal** to the curve and pointing the **fastest increasing direction**.

### 6.3.1. Optimization Problems with Equality Constraints

We first consider Lagrange's method to solve the problem of the form

$$\max_{\mathbf{x}} f(\mathbf{x}) \quad \text{subj.to} \quad g(\mathbf{x}) = c. \quad (6.14)$$

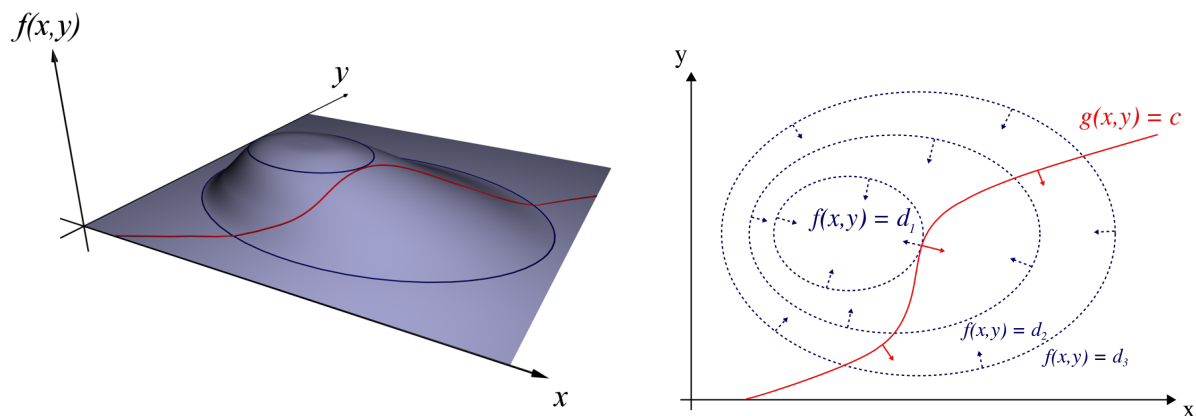


Figure 6.4: The method of Lagrange multipliers in  $\mathbb{R}^2$ :  $\nabla f \parallel \nabla g$ , at maximum.

**Strategy 6.25. (Method of Lagrange multipliers).** For the maximum and minimum values of  $f(x, y, z)$  **subject to**  $g(x, y, z) = c$ ,

(a) Find all values of  $(x, y, z)$  and  $\lambda$  such that

$$\nabla f(x, y, z) = \lambda \nabla g(x, y, z) \quad \text{and} \quad g(x, y, z) = c. \quad (6.15)$$

(b) Evaluate  $f$  at all these points, to find the maximum and minimum.



**Example 6.26.** A topless rectangular box is made from  $12\text{m}^2$  of cardboard. Find the dimensions of the box that maximizes the volume of the box.

**Solution.** Maximize  $V = xyz$  subj.to  $2xz + 2yz + xy = 12$ .

*Ans: 4 ( $x = y = 2z = 2$ )*

**Example 6.27.** Find the extreme values of  $f(x, y) = x^2 + 2y^2$  on the circle  $x^2 + y^2 = 1$ .

**Solution.**  $\nabla f = \lambda \nabla g \implies \begin{bmatrix} 2x \\ 4y \end{bmatrix} = \lambda \begin{bmatrix} 2x \\ 2y \end{bmatrix}$ . Therefore, 
$$\begin{cases} 2x = 2x \lambda & \textcircled{1} \\ 4y = 2y \lambda & \textcircled{2} \\ x^2 + y^2 = 1 & \textcircled{3} \end{cases}$$

From  $\textcircled{1}$ ,  $x = 0$  or  $\lambda = 1$ .

*Ans: min:  $f(\pm 1, 0) = 1$ ; max:  $f(0, \pm 1) = 2$*

**Example 6.28. (Continuation of Example 6.27)**

Find the extreme values of  $f(x, y) = x^2 + 2y^2$  on the disk  $x^2 + y^2 \leq 1$ .

**Solution.** *Hint:* You may use Lagrange multipliers when  $x^2 + y^2 = 1$ .

*Ans:* min:  $f(0, 0) = 0$ ;  $f(0, \pm 1) = 2$

**Remark 6.29. The Method of Lagrange Multipliers**

- **(Geometric Formula)** For the optimization problem (6.14):

$$\max_{\mathbf{x}} f(\mathbf{x}) \quad \text{subj.to} \quad g(\mathbf{x}) = c, \quad (6.16)$$

the method finds values of  $\mathbf{x}$  and  $\lambda$  such that

$$\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x}) \quad \text{and} \quad g(\mathbf{x}) = c. \quad (6.17)$$

- **(Interpretation by Calculus)** It can be interpreted as follows: Find the **critical points** of

$$\mathcal{L}(\mathbf{x}, \lambda) \stackrel{\text{def}}{=} f(\mathbf{x}) - \lambda(g(\mathbf{x}) - c). \quad (6.18)$$

Indeed,

$$\begin{aligned} \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda) &= \nabla f(\mathbf{x}) - \lambda \nabla g(\mathbf{x}), \\ \frac{\partial}{\partial \lambda} \mathcal{L}(\mathbf{x}, \lambda) &= g(\mathbf{x}) - c. \end{aligned} \quad (6.19)$$

By equating the right-side with zero, we obtain (6.17).  $\square$

The function  $\mathcal{L}(\mathbf{x}, \lambda)$  is called the **Lagrangian** for the problem (6.16).

### 6.3.2. Optimization Problems with Inequality Constraints

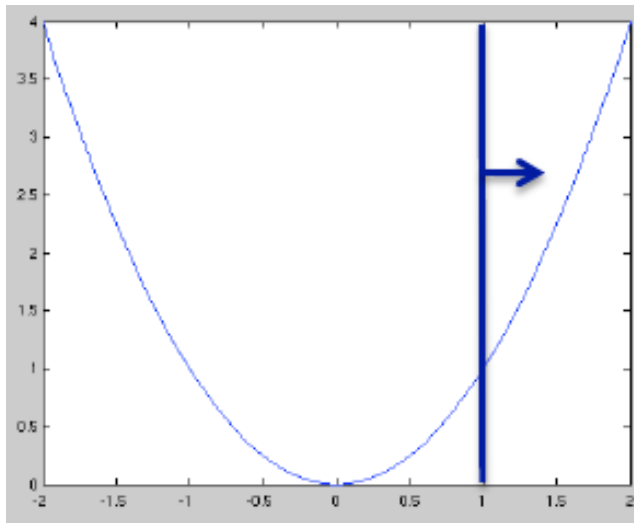


Figure 6.5:  $\min_x x^2$  subj.to  $x \geq 1$ .

For simplicity, consider

$$\min_x x^2 \quad \text{subj.to } x \geq 1. \quad (6.20)$$

Rewriting the constraint

$$x - 1 \geq 0,$$

we formulate the **Lagrangian**:

$$\mathcal{L}(x, \alpha) = x^2 - \alpha(x - 1). \quad (6.21)$$

Now, consider

$$\min_x \max_{\alpha} \mathcal{L}(x, \alpha) \quad \text{subj.to } \alpha \geq 0. \quad (6.22)$$

**Claim 6.30.** The minimization problem (6.20) is equivalent to the **minimax problem** (6.22).

**Proof.** ① Let  $x > 1$ .  $\Rightarrow \max_{\alpha \geq 0} \{-\alpha(x - 1)\} = 0$  and  $\alpha^* = 0$ . Thus,

$$\mathcal{L}(x, \alpha) = x^2. \quad (\text{original objective})$$

② Let  $x = 1$ .  $\Rightarrow \max_{\alpha \geq 0} \{-\alpha(x - 1)\} = 0$  and  $\alpha$  is arbitrary. Thus, again,

$$\mathcal{L}(x, \alpha) = x^2. \quad (\text{original objective})$$

③ Let  $x < 1$ .  $\Rightarrow \max_{\alpha \geq 0} \{-\alpha(x - 1)\} = \infty$ . However,  $\min_x$  won't make this happen! ( $\min_x$  is fighting  $\max_{\alpha}$ ) That is, when  $x < 1$ , the objective  $\mathcal{L}(x, \alpha)$  becomes huge as  $\alpha$  grows; then,  $\min_x$  will push  $x \nearrow 1$  or increase it to become  $x \geq 1$ . In other words,  $\min_x$  **forces**  $\max_{\alpha}$  **to behave, so constraints will be satisfied.**  $\square$

**Remark 6.31. A Formal Argument for the Equivalence**

Let  $f(x) = x^2$  and  $f^*$  be the optimal value of the problem (6.20):

$$\min_x f(x) \quad \text{subj.to } x \geq 1. \quad (6.23)$$

Then it is clear to see

$$\begin{aligned} x \geq 1 &\Rightarrow \mathcal{L}(x, \alpha) = f(x) - \alpha(x - 1) \leq f(x) \Rightarrow \max_{\alpha \geq 0} \mathcal{L}(x, \alpha) = f(x) \\ x \not\geq 1 &\Rightarrow \max_{\alpha \geq 0} \mathcal{L}(x, \alpha) = \infty. \end{aligned} \quad (6.24)$$

Thus

$$\min_x \max_{\alpha \geq 0} \mathcal{L}(x, \alpha) = \min_{x \geq 1} f(x) = f^*, \quad (6.25)$$

where  $\min_x$  does not require  $x$  to satisfy  $x \geq 1$ .

The above analysis implies that the (original) minimization problem (6.23) is equivalent to the **minimax problem**.

**Recall:** The minimax problem (6.22), which is equivalent to the (original) primal problem:

$$\min_x \max_{\alpha} \mathcal{L}(x, \alpha) \quad \text{subj.to} \quad \alpha \geq 0, \quad (\text{Primal}) \quad (6.26)$$

where

$$\mathcal{L}(x, \alpha) = x^2 - \alpha(x - 1).$$

**Definition 6.32.** The dual problem of (6.26) is formulated by swapping  $\min_x$  and  $\max_{\alpha}$  as follows:

$$\max_{\alpha} \min_x \mathcal{L}(x, \alpha) \quad \text{subj.to} \quad \alpha \geq 0, \quad (\text{Dual}) \quad (6.27)$$

In the **maximin problem**, the term  $\min_x \mathcal{L}(x, \alpha)$  is called the **Lagrange dual function** and the Lagrange multiplier  $\alpha$  is also called the **dual variable**.

**How to solve it.** For the Lagrange dual function  $\min_x \mathcal{L}(x, \alpha)$ , the minimum occurs where the gradient is equal to zero.

$$\frac{d}{dx} \mathcal{L}(x, \alpha) = 2x - \alpha = 0 \Rightarrow x = \frac{\alpha}{2}. \quad (6.28)$$

Plugging this to  $\mathcal{L}(x, \alpha)$ , we have

$$\mathcal{L}(x, \alpha) = \left(\frac{\alpha}{2}\right)^2 - \alpha\left(\frac{\alpha}{2} - 1\right) = \alpha - \frac{\alpha^2}{4}.$$

We can rewrite the dual problem (6.27) as

$$\max_{\alpha \geq 0} \left[ \alpha - \frac{\alpha^2}{4} \right]. \quad (\text{Dual}) \quad (6.29)$$

$\Rightarrow$  **the maximum is 1 when  $\alpha^* = 2$**  (for the dual problem).

Plugging  $\alpha = \alpha^*$  into (6.28) to get  $x^* = 1$ . Or, using the Lagrangian objective, we have

$$\mathcal{L}(x, \alpha) = x^2 - 2(x - 1) = (x - 1)^2 + 1. \quad (6.30)$$

$\Rightarrow$  **the minimum is 1 when  $x^* = 1$**  (for the primal problem).  $\square$

### Multiple Constraints

Consider the problem of the form

$$\max_{\mathbf{x}} f(\mathbf{x}) \quad \text{subj.to} \quad g(\mathbf{x}) = c \quad \text{and} \quad h(\mathbf{x}) = d. \quad (6.31)$$

Then, at extrema we must have

$$\nabla f \in \text{Plane}(\nabla g, \nabla h) := \{c_1 \nabla g + c_2 \nabla h\}. \quad (6.32)$$

Thus (6.31) can be solved by finding all values of  $(x, y, z)$  and  $(\lambda, \mu)$  such that

$$\begin{aligned} \nabla f(x, y, z) &= \lambda \nabla g(x, y, z) + \mu \nabla h(x, y, z) \\ g(x, y, z) &= c \\ h(x, y, z) &= d \end{aligned} \quad (6.33)$$

**Example 6.33.** Find the maximum value of the function  $f(x, y, z) = z$  on the curve of the intersection of the cone  $2x^2 + 2y^2 = z^2$  and the plane  $x + y + z = 4$ .

**Solution.** Letting  $g = 2x^2 + 2y^2 - z^2 = 0$  ④ and  $h = x + y + z - 4 = 0$  ⑤, we have

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} 4x \\ 4y \\ -2z \end{bmatrix} + \mu \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \implies \begin{cases} 0 = 4\lambda x + \mu & \text{①} \\ 0 = 4\lambda y + \mu & \text{②} \\ 1 = -2\lambda z + \mu & \text{③} \end{cases}$$

From ① and ②, we conclude  $x = y$ ; using ④, we have  $z = \pm 2x$ .

## 6.4. The Gradient Descent Method

### 6.4.1. Introduction to the Gradient Descent Method

**Problem 6.34. (Minimization Problem)** Let  $\Omega \subset \mathbb{R}^n$ ,  $n \geq 1$ . Given a real-valued function  $f : \Omega \rightarrow \mathbb{R}$ , the general problem of finding the value that minimizes  $f$  is formulated as follows.

$$\min_{\mathbf{x} \in \Omega} f(\mathbf{x}). \quad (6.34)$$

In this context,  $f$  is the **objective function** (sometimes referred to as **loss function** or **cost function**).  $\Omega \subset \mathbb{R}^n$  is the **domain** of the function (also known as the **constraint set**).

In this section, we solve the minimization problem (6.34) **iteratively** as follows: Given an initial guess  $\mathbf{x}_0 \in \mathbb{R}^n$ , find successive approximations  $\mathbf{x}_k \in \mathbb{R}^n$  of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \gamma_k \mathbf{p}_k, \quad k = 0, 1, \dots, \quad (6.35)$$

where  $\mathbf{p}_k$  is the **search direction** and  $\gamma_k > 0$  is the **step length**.

**Note:** The **gradient descent method** is also known as the **steepest descent method** or the **Richardson's method**.

- Recall that we would solve the minimization problem (6.34) using iterative algorithms of the form (6.35).

### Derivation of the GD method

- Given  $\mathbf{x}_{k+1}$  as in (6.35), we have by **Taylor's formula**: for some  $\xi$ ,

$$\begin{aligned} f(\mathbf{x}_{k+1}) &= f(\mathbf{x}_k + \gamma_k \mathbf{p}_k) \\ &= f(\mathbf{x}_k) + \gamma_k f'(\mathbf{x}_k) \cdot \mathbf{p}_k + \frac{\gamma_k^2}{2} \mathbf{p}_k \cdot f''(\xi) \mathbf{p}_k. \end{aligned} \quad (6.36)$$

- Assume that  $f''$  is bounded. Then

$$f(\mathbf{x}_{k+1}) = f(\mathbf{x}_k) + \gamma_k f'(\mathbf{x}_k) \cdot \mathbf{p}_k + \mathcal{O}(\gamma_k^2), \quad \text{as } \gamma_k \rightarrow 0.$$

- **The Goal**: To find  $\mathbf{p}_k$  and  $\gamma_k$  such that

$$f(\mathbf{x}_{k+1}) < f(\mathbf{x}_k), \quad (6.37)$$

which can be achieved if

$$f'(\mathbf{x}_k) \cdot \mathbf{p}_k < 0 \quad (6.38)$$

and either  $\gamma_k$  is sufficiently small or  $f''(\xi)$  is nonnegative.

- **Choice**: Let  $f'(\mathbf{x}_k) \neq 0$ . If we choose

$$\mathbf{p}_k = -f'(\mathbf{x}_k), \quad (6.39)$$

then

$$f'(\mathbf{x}_k) \cdot \mathbf{p}_k = -\|f'(\mathbf{x}_k)\|^2 < 0, \quad (6.40)$$

which satisfies (6.38) and therefore (6.37).

- **Summary**: In the GD method, the search direction is the **negative gradient**, the steepest descent direction.



### The Gradient Descent Method in 1D

**Algorithm 6.35.** Consider the minimization problem in 1D:

$$\min_x f(x), \quad x \in S, \quad (6.41)$$

where  $S$  is a closed interval in  $\mathbb{R}$ . Then its gradient descent method reads

$$x_{k+1} = x_k - \gamma f'(x_k). \quad (6.42)$$

**Picking the step length  $\gamma$ :** Assume that the step length was chosen to be independent of  $n$ , although one can play with other choices as well. The question is how to select  $\gamma$  in order to make the best gain of the method. To turn the right-hand side of (6.42) into a more manageable form, we invoke Taylor's Theorem:<sup>1</sup>

$$f(x+t) = f(x) + t f'(x) + \int_x^{x+t} (x+t-s) f''(s) ds. \quad (6.43)$$

Assuming that  $|f''(s)| \leq L$ , we have

$$f(x+t) \leq f(x) + t f'(x) + \frac{t^2}{2} L.$$

Now, letting  $x = x_k$  and  $t = -\gamma f'(x_k)$  reads

$$\begin{aligned} f(x_{k+1}) &= f(x_k - \gamma f'(x_k)) \\ &\leq f(x_k) - \gamma f'(x_k) f'(x_k) + \frac{1}{2} L [\gamma f'(x_k)]^2 \\ &= f(x_k) - [f'(x_k)]^2 \left( \gamma - \frac{L}{2} \gamma^2 \right). \end{aligned} \quad (6.44)$$

The gain (learning) from the method occurs when

$$\gamma - \frac{L}{2} \gamma^2 > 0 \quad \Rightarrow \quad 0 < \gamma < \frac{2}{L}, \quad (6.45)$$

and it will be best when  $\gamma - \frac{L}{2} \gamma^2$  is maximal. This happens at the point

$$\gamma = \frac{1}{L}. \quad (6.46)$$

<sup>1</sup>**Taylor's Theorem with integral remainder:** Suppose  $f \in C^{n+1}[a, b]$  and  $x_0 \in [a, b]$ . Then, for every  $x \in [a, b]$ ,  $f(x) = \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!} (x-x_0)^k + R_n(x)$ , where  $R_n(x) = \frac{1}{n!} \int_{x_0}^x (x-s)^n f^{(n+1)}(s) ds$ .

Thus an effective **gradient descent method** (6.42) can be written as

$$x_{k+1} = x_k - \gamma f'(x_k) = x_k - \frac{1}{L} f'(x_k) = x_k - \frac{1}{\max |f''(x)|} f'(x_k). \quad (6.47)$$

Furthermore, it follows from (6.44) that for  $\gamma = 1/L$ ,

$$f(x_{k+1}) \leq f(x_k) - \frac{\gamma}{2} [f'(x_k)]^2. \quad (6.48)$$

**Remark 6.36. (Convergence of gradient descent method).**

Thus it is obvious that the method defines a sequence of points  $\{x_k\}$  along which  $\{f(x_k)\}$  decreases.

- If  $f$  is bounded from below and the level sets of  $f$  are bounded,  $\{f(x_k)\}$  converges; so does  $\{x_k\}$ . That is, there is a point  $\hat{x}$  such that

$$\lim_{n \rightarrow \infty} x_k = \hat{x}. \quad (6.49)$$

- Now, we can rewrite (6.48) as

$$[f'(x_k)]^2 \leq 2L [f(x_k) - f(x_{k+1})]. \quad (6.50)$$

Since  $f(x_k) - f(x_{k+1}) \rightarrow 0$ , also  $f'(x_k) \rightarrow 0$ .

- When  $f'$  is continuous, using (6.49) reads

$$f'(\hat{x}) = \lim_{n \rightarrow \infty} f'(x_k) = 0, \quad (6.51)$$

which implies that the limit  $\hat{x}$  is a **critical point**.

- The method thus generally finds a critical point but that could still be a local minimum or a saddle point. Which it is cannot be decided at this level of analysis.  $\square$

## 6.4.2. The Gradient Descent Method in Multi-Dimensions

**Example 6.37. (Rosenbrock function).** For example, the **Rosenbrock function** in the two-dimensional (2D) space is defined as<sup>2</sup>

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2. \quad (6.52)$$

Use the GD method to find the minimizer, starting with  $\mathbf{x}_0 = (-1, 2)$ .

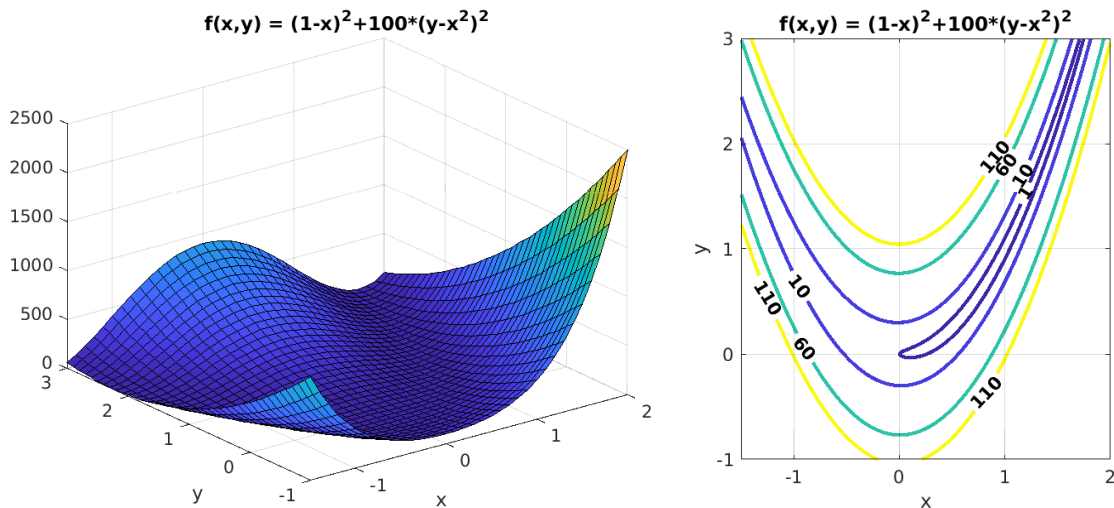


Figure 6.6: Plots of the Rosenbrock function  $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ .

```

rosenbrock_2D_GD.py
1 import numpy as np; import time
2
3 itmax = 10000; tol = 1.e-7; gamma = 1/500
4 x0 = np.array([-1., 2.])
5
6 def rosen(x):
7     return (1.-x[0])**2+100*(x[1]-x[0]**2)**2
8
9 def rosen_grad(x):
10    h = 1.e-5;
11    g1 = ( rosen([x[0]+h,x[1]]) - rosen([x[0]-h,x[1]]) )/(2*h)
12    g2 = ( rosen([x[0],x[1]+h]) - rosen([x[0],x[1]-h]) )/(2*h)

```

<sup>2</sup>The Rosenbrock function in 3D is given as  $f(x, y, z) = [(1 - x)^2 + 100(y - x^2)^2] + [(1 - y)^2 + 100(z - y^2)^2]$ , which has exactly one minimum at  $(1, 1, 1)$ . Similarly, one can define the Rosenbrock function in general  $N$ -dimensional spaces, for  $N \geq 4$ , by adding one more component for each enlarged dimension.

That is,  $f(\mathbf{x}) = \sum_{i=1}^{N-1} [(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2]$ , where  $\mathbf{x} = [x_1, x_2, \dots, x_N] \in \mathbb{R}^N$ . See Wikipedia ([https://en.wikipedia.org/wiki/Rosenbrock\\_function](https://en.wikipedia.org/wiki/Rosenbrock_function)) for details.

```

13     return np.array([g1,g2])
14
15 # Now, GD iteration begins
16 if __name__ == '__main__':
17     t0 = time.time()
18     x=x0
19     for it in range(itmax):
20         corr = gamma*rosen_grad(x)
21         x = x - corr
22         if np.linalg.norm(corr)<tol: break
23     print('GD Method: it = %d; E-time = %.4f' %(it+1,time.time()-t0))
24     print(x)

```

Output

```

1 GD Method: it = 7687; E-time = 0.0521
2 [0.99994416 0.99988809]

```

### The Choice of Step Size and Line Search

**Note:** The convergence of the gradient descent method can be extremely **sensitive to the choice of step size**. It often requires to choose the step size **adaptively**: the step size would better be chosen small in regions of large variability of the gradient, while in regions with small variability we would like to take it large.

**Strategy 6.38. Backtracking Line Search** procedures allow to select a step size depending on the current iterate and the gradient. In this procedure, we select an initial (optimistic) step size  $\gamma_k$  and evaluate the following inequality (known as **sufficient decrease condition**):

$$f(\mathbf{x}_{k+1}) = f(\mathbf{x}_k - \gamma_k \nabla f(\mathbf{x}_k)) \leq f(\mathbf{x}_k) - \frac{\gamma_k}{2} \|\nabla f(\mathbf{x}_k)\|^2. \quad (6.53)$$

If this inequality is verified, the current step size is kept. If not, the step size is divided by 2 (or any number larger than 1) repeatedly until (6.53) is verified. To get a better understanding, refer to (6.48) on p. 184.

The gradient descent algorithm with backtracking line search then becomes

**Algorithm 6.39. (The Gradient Descent Algorithm, with Backtracking Line Search).**

```

input: initial guess  $\mathbf{x}_0$ , step size  $\gamma > 0$ ;
for  $k = 0, 1, 2, \dots$  do
    initial step size estimate  $\gamma_k$ ;
    while (TRUE) do
        if  $f(\mathbf{x}_k - \gamma_k \nabla f(\mathbf{x}_k)) \leq f(\mathbf{x}_k) - \frac{\gamma_k}{2} \|\nabla f(\mathbf{x}_k)\|^2$ 
            break;
        else  $\gamma_k = \gamma_k/2$ ;
    end while
     $\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla f(\mathbf{x}_k)$ ;
end for
return  $\mathbf{x}_{k+1}$ ;

```

(6.54)

**Remark 6.40.** Incorporated with

- either **a line search**
- or **partial updates**,

the **gradient descent method** is *the* major computational algorithm for various machine learning tasks.

**Note:** The gradient descent method with partial updates is called the **stochastic gradient descent (SGD)** method.

### 6.4.3. The Gradient Descent Method for Positive Definite Linear Systems

**Definition 6.41.** A matrix  $A = (a_{ij}) \in \mathbb{R}^{n \times n}$  is said to be **positive definite** if

$$\mathbf{x}^T A \mathbf{x} = \sum_{i,j=1}^n x_i a_{ij} x_j > 0, \quad \forall \mathbf{x} \in \mathbb{R}^n, \quad \mathbf{x} \neq 0. \quad (6.55)$$

**Theorem 6.42.** Let  $A \in \mathbb{R}^{n \times n}$  be symmetric. Then  $A$  is positive definite if and only if all eigenvalues of  $A$  are positive.

**Remark 6.43.** Let  $A \in \mathbb{R}^{n \times n}$  be symmetric positive definite and consider

$$A \mathbf{x} = \mathbf{b}. \quad (6.56)$$

Then the algebraic system admits a unique solution  $\mathbf{x} \in \mathbb{R}^n$ , which is equivalently characterized by

$$\mathbf{x} = \arg \min_{\boldsymbol{\eta} \in \mathbb{R}^n} f(\boldsymbol{\eta}), \quad f(\boldsymbol{\eta}) = \frac{1}{2} \boldsymbol{\eta} \cdot A \boldsymbol{\eta} - \mathbf{b} \cdot \boldsymbol{\eta}. \quad (6.57)$$

For the algebraic system (6.56), **Krylov subspace methods** update the iterates as follows.

Given an initial guess  $\mathbf{x}_0 \in \mathbb{R}^n$ , find successive approximations  $\mathbf{x}_k \in \mathbb{R}^n$  of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \quad k = 0, 1, \dots, \quad (6.58)$$

where  $\mathbf{p}_k$  is the **search direction** and  $\alpha_k > 0$  is the **step length**.

- Different methods differ in the choice of the search direction and the step length.
- In this subsection, we focus on the **gradient descent method**.
- For other Krylov subspace methods, see e.g. [1, 7].

**Derivation of the GD Method for (6.57)**

- We denote the **gradient** and **Hessian** of  $f$  by  $f'$  and  $f''$ , respectively:

$$f'(\eta) = A\eta - \mathbf{b}, \quad f''(\eta) = A. \quad (6.59)$$

- Given  $\mathbf{x}_{k+1}$  as in (6.58), we have by Taylor's formula

$$\begin{aligned} f(\mathbf{x}_{k+1}) &= f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \\ &= f(\mathbf{x}_k) + \alpha_k f'(\mathbf{x}_k) \cdot \mathbf{p}_k + \frac{\alpha_k^2}{2} \mathbf{p}_k \cdot f''(\boldsymbol{\xi}) \mathbf{p}_k \\ &= f(\mathbf{x}_k) + \alpha_k f'(\mathbf{x}_k) \cdot \mathbf{p}_k + \frac{\alpha_k^2}{2} \mathbf{p}_k \cdot A \mathbf{p}_k. \end{aligned} \quad (6.60)$$

- Since  $A$  is bounded,

$$f(\mathbf{x}_{k+1}) = f(\mathbf{x}_k) + \alpha_k f'(\mathbf{x}_k) \cdot \mathbf{p}_k + \mathcal{O}(\alpha_k^2), \quad \text{as } \alpha_k \rightarrow 0.$$

- **The goal** is to find  $\mathbf{p}_k$  and  $\alpha_k$  such that

$$f(\mathbf{x}_{k+1}) < f(\mathbf{x}_k), \quad (6.61)$$

which can be achieved if

$$f'(\mathbf{x}_k) \cdot \mathbf{p}_k < 0 \quad (6.62)$$

and either  $\alpha_k$  is sufficiently small or  $A = f''(\boldsymbol{\xi})$  is nonnegative.

- **Choice:** When  $f'(\mathbf{x}_k) \neq 0$ , (6.62) holds true, if we choose:

$$\mathbf{p}_k = -f'(\mathbf{x}_k) = \mathbf{b} - A\mathbf{x}_k =: \mathbf{r}_k \quad (6.63)$$

That is, the search direction is the **negative gradient**, the **steepest descent direction**.

### Optimal step length

We may determine  $\alpha_k$  such that

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = \min_{\alpha} f(\mathbf{x}_k + \alpha \mathbf{p}_k), \quad (6.64)$$

in which case  $\alpha_k$  is said to be **optimal**.

If  $\alpha_k$  is optimal, then

$$\begin{aligned} 0 &= \left. \frac{d}{d\alpha} f(\mathbf{x}_k + \alpha \mathbf{p}_k) \right|_{\alpha=\alpha_k} = f'(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \cdot \mathbf{p}_k \\ &= (A(\mathbf{x}_k + \alpha_k \mathbf{p}_k) - \mathbf{b}) \cdot \mathbf{p}_k \\ &= (A\mathbf{x}_k - \mathbf{b}) \cdot \mathbf{p}_k + \alpha_k \mathbf{p}_k \cdot A\mathbf{p}_k. \end{aligned} \quad (6.65)$$

So,

$$\alpha_k = \frac{\mathbf{r}_k \cdot \mathbf{p}_k}{\mathbf{p}_k \cdot A\mathbf{p}_k}. \quad (6.66)$$

#### Algorithm 6.44. (GD Algorithm)

Select  $\mathbf{x}_0, \varepsilon$ ;

$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ ;

Do  $k = 0, 1, \dots$

$$\alpha_k = \|\mathbf{r}_k\|^2 / \mathbf{r}_k \cdot A\mathbf{r}_k; \quad (\text{GD1}) \quad (6.67)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{r}_k; \quad (\text{GD2})$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k A\mathbf{r}_k; \quad (\text{GD3})$$

if  $\|\mathbf{r}_{k+1}\| < \varepsilon \|\mathbf{r}_0\|$ , stop;

End Do

**Note:** The equation in (GD3) is equivalent to

$$\mathbf{r}_{k+1} = \mathbf{b} - A\mathbf{x}_{k+1}; \quad (6.68)$$

see Exercise 6.4, p.192.



**Recall: (Definition 5.45)** Let  $A \in \mathbb{R}^{n \times n}$  be invertible. Then

$$\kappa(A) \equiv \|A\| \|A^{-1}\| \quad (6.69)$$

is called the **condition number** of  $A$ , associated to the matrix norm.

**Remark 6.45.** When  $A$  is symmetric positive definite (SPD), the condition number becomes

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}, \quad (A : \text{SPD}) \quad (6.70)$$

where  $\lambda_{\min}$  and  $\lambda_{\max}$  are the minimum and the maximum eigenvalues of  $A$ , respectively.

**Theorem 6.46. (Convergence of the GD method):** *The GD method converges, satisfying*

$$\|\mathbf{x} - \mathbf{x}_k\| \leq \left(1 - \frac{1}{\kappa(A)}\right)^k \|\mathbf{x} - \mathbf{x}_0\|. \quad (6.71)$$

*Thus, the number of iterations required to reduce the error by a factor of  $\varepsilon$  is in the order of the condition number of  $A$ :*

$$k \geq \kappa(A) \log \frac{1}{\varepsilon}. \quad (6.72)$$

## Exercises for Chapter 6

6.1. Find the partial derivatives of the functions.

(a)  $z = y \cos(xy)$

(b)  $f(u, v) = (uv - v^3)^2$

(c)  $w = \ln(x + 2y + 3z)$

(d)  $u = \sin(x_1^2 + x_2^2 + \cdots + x_n^2)$

Ans: (d)  $\partial u / \partial x_i = 2x_i \cdot \cos(x_1^2 + x_2^2 + \cdots + x_n^2)$

6.2. Use Lagrange multipliers to find extreme values of the function subject to the given constraint.

(a)  $f(x, y) = xy; \quad x^2 + 4y^2 = 2$

(b)  $f(x, y) = x + y + 2z; \quad x^2 + y^2 + z^2 = 6$

Ans: max:  $f(1, 1, 2) = 6$ ; min:  $f(-1, -1, -2) = -6$

6.3. Use the **method of Lagrange multipliers** to solve the problem.

$$\min_{x,y} x_1^2 + x_2^2, \quad \text{subj.to} \begin{cases} x_1 \geq 1 \\ x_2 \geq 2 \end{cases} \quad (6.73)$$

**Hint:** You may start with the **Lagrangian**

$$\mathcal{L}(x_1, x_2, \alpha_1, \alpha_2) = x_1^2 + x_2^2 - \alpha_1(x_1 - 1) - \alpha_2(x_2 - 2), \quad \alpha_1, \alpha_2 \geq 0, \quad (6.74)$$

and consider the dual problem  $\max_{\alpha \geq 0} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha)$ , where  $\alpha = (\alpha_1, \alpha_2)$  and  $\mathbf{x} = (x_1, x_2)$ .

Then

$$\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha) = \begin{bmatrix} 2x_1 - \alpha_1 \\ 2x_2 - \alpha_2 \end{bmatrix} = 0 \Rightarrow \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \alpha_1/2 \\ \alpha_2/2 \end{bmatrix}. \quad (6.75)$$

Ans: 5

6.4. When the **boundary-value problem**

$$\begin{cases} -u_{xx} = -2, & 0 < x < 4 \\ u_x(0) = 0, & u(4) = 16 \end{cases} \quad (6.76)$$

is discretized by the second-order finite difference method with  $h = 1$ , the algebraic system reads  $A\mathbf{x} = \mathbf{b}$ , where

$$A = \begin{bmatrix} 2 & -2 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -2 \\ -2 \\ -2 \\ 14 \end{bmatrix} \quad (6.77)$$

and the exact solution is  $\mathbf{x} = [0, 1, 4, 9]^T$ .

(a) Find the condition number of  $A$ .

(b) Prove that (GD3) in the GD algorithm, Algorithm 6.44, is equivalent to 6.68. Why do we consider such a manipulation?

(c) Implement the GD algorithm to find a numerical solution in 6-digit accuracy.

## CHAPTER 7

# Least-Squares and Regression Analysis

### Contents of Chapter 7

7.1. The Least-Squares Problem . . . . .	194
7.2. Regression Analysis . . . . .	198
7.3. Scene Analysis with Noisy Data: Weighted Least-Squares and RANSAC . . . . .	204
Exercises for Chapter 7 . . . . .	209

## 7.1. The Least-Squares Problem

**Definition 7.1.** For a given dataset  $\{(x_i, y_i)\}$ , let a continuous function  $p(x)$  be constructed.

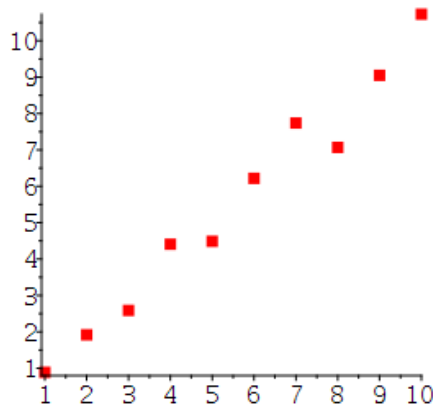
- (a)  $p$  is an **interpolation** if it passes (interpolates) all the data points.
- (b)  $p$  is an **approximation** if it approximates (represents) the data points.

Dataset, in Maple

```

1 with(LinearAlgebra): with(CurveFitting):
2 n := 100: roll := rand(-n..n):
3 m := 10: xy := Matrix(m, 2):
4 for i to m do
5     xy[i, 1] := i;
6     xy[i, 2] := i + roll()/n;
7 end do:
8 plot(xy,color=red, style=point, symbol=solidbox, symbolsize=20);

```



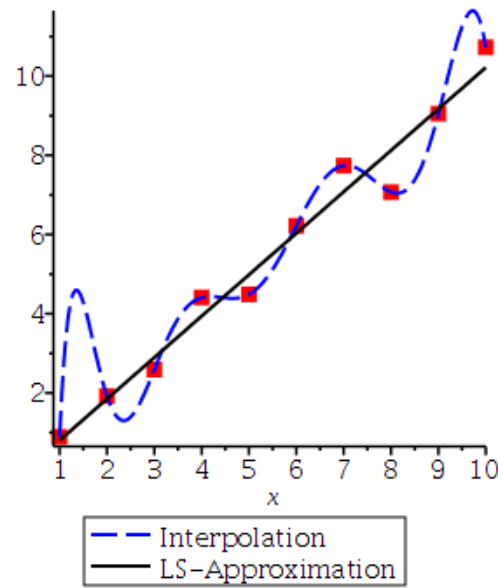
$p := \text{PolynomialInterpolation}(xy, x);$

$$\frac{1507}{12096000} x^9 - \frac{27977}{4032000} x^8 + \frac{36751}{224000} x^7 - \frac{68851}{32000} x^6 + \frac{1975291}{115200} x^5$$

$$- \frac{5471791}{64000} x^4 + \frac{199836041}{756000} x^3 - \frac{486853651}{1008000} x^2 + \frac{39227227}{84000} x - \frac{1771}{10}$$

$L := \text{CurveFitting}[\text{LeastSquares}](xy, x);$

$$-\frac{121}{500} + \frac{523}{500} x$$



**Note:** Interpolation may be too oscillatory to be useful.  
Furthermore, it may not be defined.

### The Least-Squares (LS) Problem

**Note:** Let  $A$  is an  $m \times n$  matrix. Then  $Ax = \mathbf{b}$  may have no solution, particularly when  $m > n$ . In real-world,

- $m \gg n$ , where  $m$  represents the number of data points and  $n$  denotes the dimension of the points
- Need to find a best solution for  $Ax \approx \mathbf{b}$

**Definition 7.2.** Let  $A \in \mathbb{R}^{m \times n}$ ,  $m \geq n$ , and  $\mathbf{b} \in \mathbb{R}^m$ . The **least-squares problem** is to find  $\hat{\mathbf{x}} \in \mathbb{R}^n$  which minimizes  $\|Ax - \mathbf{b}\|_2$ :

$$\begin{aligned} \hat{\mathbf{x}} &= \arg \min_{\mathbf{x}} \|Ax - \mathbf{b}\|_2, \\ &\text{or, equivalently,} \\ \hat{\mathbf{x}} &= \arg \min_{\mathbf{x}} \|Ax - \mathbf{b}\|_2^2, \end{aligned} \tag{7.1}$$

where  $\hat{\mathbf{x}}$  is called a **least-squares solution** of  $Ax = \mathbf{b}$ .

### The Method of Normal Equations

**Theorem 7.3.** *The set of LS solutions of  $A\mathbf{x} = \mathbf{b}$  coincides with the nonempty set of solutions of the **normal equations***

$$A^T A\mathbf{x} = A^T \mathbf{b}. \quad (7.2)$$

#### Method of Calculus

Let  $\mathcal{J}(\mathbf{x}) = \|A\mathbf{x} - \mathbf{b}\|^2 = (A\mathbf{x} - \mathbf{b})^T(A\mathbf{x} - \mathbf{b})$  and  $\hat{\mathbf{x}}$  a minimizer of  $\mathcal{J}(\mathbf{x})$ .

- Then we must have

$$\nabla_{\mathbf{x}} \mathcal{J}(\hat{\mathbf{x}}) = \left. \frac{\partial \mathcal{J}(\mathbf{x})}{\partial \mathbf{x}} \right|_{\mathbf{x}=\hat{\mathbf{x}}} = \mathbf{0}. \quad (7.3)$$

- Let's compute the gradient of  $\mathcal{J}$ .

$$\begin{aligned} \frac{\partial \mathcal{J}(\mathbf{x})}{\partial \mathbf{x}} &= \frac{\partial ((A\mathbf{x} - \mathbf{b})^T(A\mathbf{x} - \mathbf{b}))}{\partial \mathbf{x}} \\ &= \frac{\partial (\mathbf{x}^T A^T A\mathbf{x} - 2\mathbf{x}^T A^T \mathbf{b} + \mathbf{b}^T \mathbf{b})}{\partial \mathbf{x}} \\ &= 2A^T A\mathbf{x} - 2A^T \mathbf{b}. \end{aligned} \quad (7.4)$$

- By setting the last term to zero, we obtain normal equations.

**Remark 7.4.** Theorem 7.3 implies that LS solutions of  $A\mathbf{x} = \mathbf{b}$  are solutions of the normal equations  $A^T A\hat{\mathbf{x}} = A^T \mathbf{b}$ .

- When  $A^T A$  is **not invertible**, the normal equations have either no solution or infinitely many solutions.
- So, data acquisition is important, to make it invertible.

**Theorem 7.5. (Method of Normal Equations)** Let  $A \in \mathbb{R}^{m \times n}$ ,  $m \geq n$ . The following statements are logically equivalent:

- The equation  $Ax = b$  has a **unique** LS solution for each  $b \in \mathbb{R}^m$ .
- The matrix  $A^T A$  is invertible.
- Columns of  $A$  are linearly independent.

When these statements hold true, the **unique** LS solution  $\hat{x}$  is given by

$$\hat{x} = (A^T A)^{-1} A^T b. \quad (7.5)$$

**Definition 7.6.**  $A^+ := (A^T A)^{-1} A^T$  is called the **pseudoinverse** of  $A$ .

**Example 7.7.** Describe all least squares solutions of the equation  $Ax = b$ , given

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 1 \\ 3 \\ 8 \\ 2 \end{bmatrix}.$$

**Solution.** Let's try to solve the problem with pencil-and-paper.

```

least_squares.m
1 A = [1 1 0; 0 1 0; 0 0 1; 1 0 1];
2 b = [1; 3; 8; 2];
3 x = (A'*A)\(A'*b)

```

*Ans:*  $x = [-4, 4, 7]^T$

## 7.2. Regression Analysis

**Definition 7.8. Regression analysis** is a set of statistical methods used to estimate relationships between **one or more independent variables** and a **dependent variable**.

### 7.2.1. Regression line

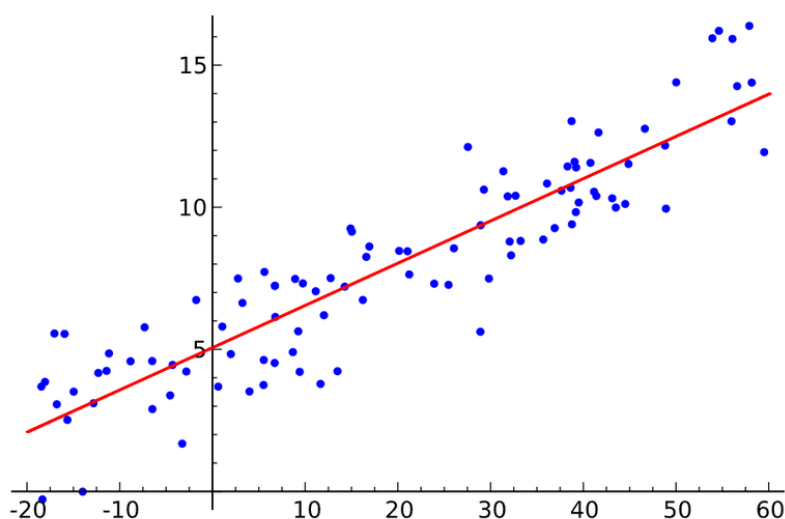


Figure 7.1: A least-squares regression line.

**Definition 7.9.** Suppose a set of experimental data points are given as

$$(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$$

such that the graph is close to a line. We (*may and must*) determine a line

$$y = \beta_0 + \beta_1 x \tag{7.6}$$

that is **as close as possible to the data points**. Then this line is called the **least-squares line**; it is also called the **regression line** of  $y$  on  $x$  and  $\beta_0, \beta_1$  are called **regression coefficients**.



### Calculation of Least-Squares Lines

Consider a least-squares (LS) model of the form  $y = \beta_0 + \beta_1 x$ , for a given data set  $\{(x_i, y_i) \mid i = 1, 2, \dots, m\}$ .

- Then

Predicted $y$ -value	=	Observed $y$ -value	
$\beta_0 + \beta_1 x_1$	=	$y_1$	
$\beta_0 + \beta_1 x_2$	=	$y_2$	(7.7)
$\vdots$		$\vdots$	
$\beta_0 + \beta_1 x_m$	=	$y_m$	

- It can be equivalently written as

$$X\boldsymbol{\beta} = \mathbf{y}, \quad (7.8)$$

where

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

Here we call  $X$  the **design matrix**,  $\boldsymbol{\beta}$  the **parameter vector**, and  $\mathbf{y}$  the **observation vector**.

- Thus the LS solution can be determined by solving the normal equations:

$$X^T X \boldsymbol{\beta} = X^T \mathbf{y}, \quad (7.9)$$

provided that  $X^T X$  is invertible.

- The normal equations for the regression line read

$$\begin{bmatrix} m & \Sigma x_i \\ \Sigma x_i & \Sigma x_i^2 \end{bmatrix} \boldsymbol{\beta} = \begin{bmatrix} \Sigma y_i \\ \Sigma x_i y_i \end{bmatrix}. \quad (7.10)$$

**Remark 7.10. (Pointwise construction of the normal equations)**

The normal equations for the regression line in (7.10) can be rewritten as

$$\sum_{i=1}^m \begin{bmatrix} 1 & x_i \\ x_i & x_i^2 \end{bmatrix} \boldsymbol{\beta} = \sum_{i=1}^m \begin{bmatrix} y_i \\ x_i y_i \end{bmatrix}. \quad (7.11)$$

- The pointwise construction of the normal equation is convenient when either points are first to be searched or weights are applied depending on the point location.
- The idea is applicable for other regression models as well.

**Example 7.11.** Find the equation  $y = \beta_0 + \beta_1 x$  of least-squares line that best fits the given points:

$(-1, 0), (0, 1), (1, 2), (2, 4)$

**Solution.**

### 7.2.2. Least-squares fitting of other curves

**Remark 7.12.** Consider a regression model of the form

$$y = \beta_0 + \beta_1 x + \beta_2 x^2,$$

for a given data set  $\{(x_i, y_i) \mid i = 1, 2, \dots, m\}$ . Then

Predicted $y$ -value	Observed $y$ -value	
$\beta_0 + \beta_1 x_1 + \beta_2 x_1^2$	$=$	$y_1$
$\beta_0 + \beta_1 x_2 + \beta_2 x_2^2$	$=$	$y_2$
$\vdots$	$=$	$\vdots$
$\beta_0 + \beta_1 x_m + \beta_2 x_m^2$	$=$	$y_m$

(7.12)

It can be equivalently written as

$$X\boldsymbol{\beta} = \mathbf{y}, \quad (7.13)$$

where

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

Now, it can be solved through *normal equations*:

$$X^T X \boldsymbol{\beta} = \begin{bmatrix} \Sigma 1 & \Sigma x_i & \Sigma x_i^2 \\ \Sigma x_i & \Sigma x_i^2 & \Sigma x_i^3 \\ \Sigma x_i^2 & \Sigma x_i^3 & \Sigma x_i^4 \end{bmatrix} \boldsymbol{\beta} = \begin{bmatrix} \Sigma y_i \\ \Sigma x_i y_i \\ \Sigma x_i^2 y_i \end{bmatrix} = X^T \mathbf{y} \quad (7.14)$$

**Self-study 7.13.** Find an LS curve of the form  $y = \beta_0 + \beta_1 x + \beta_2 x^2$  that best fits the given points:

$(0, 1), (1, 1), (1, 2), (2, 3)$ .

**Solution.**

*Ans:*  $y = 1 + 0.5x^2$

### 7.2.3. Nonlinear regression: Linearization

**Strategy 7.14.** For nonlinear models, a **change of variables** can be applied to get a linear model.

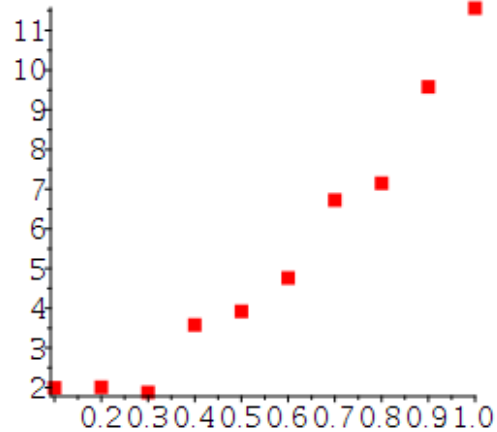
Model	Change of Variables	Linearization
$y = A + \frac{B}{x}$	$\tilde{x} = \frac{1}{x}, \tilde{y} = y$	$\Rightarrow \tilde{y} = A + B\tilde{x}$
$y = \frac{1}{A + Bx}$	$\tilde{x} = x, \tilde{y} = \frac{1}{y}$	$\Rightarrow \tilde{y} = A + B\tilde{x}$
$y = Ce^{Dx}$	$\tilde{x} = x, \tilde{y} = \ln y$	$\Rightarrow \tilde{y} = \ln C + D\tilde{x}$
$y = \frac{1}{A + B \ln x}$	$\tilde{x} = \ln x, \tilde{y} = \frac{1}{y}$	$\Rightarrow \tilde{y} = A + B\tilde{x}$

(7.15)

The above table contains just a few examples of linearization; for other nonlinear models, use your imagination and creativity.

**Example 7.15.** Find the best fitting curve of the form  $y = ce^{dx}$  for the data

0.1	1.9940
0.2	2.0087
0.3	1.8770
0.4	3.5783
0.5	3.9203
0.6	4.7617
0.7	6.7246
0.8	7.1491
0.9	9.5777
1.0	11.5625



**Solution.** Applying the natural log function ( $\ln$ ) to  $y = ce^{dx}$  gives

$$\ln y = \ln c + dx. \quad (7.16)$$

Using the change of variables

$$Y = \ln y, \quad a_0 = \ln c, \quad a_1 = d, \quad X = x,$$

the equation (7.16) reads

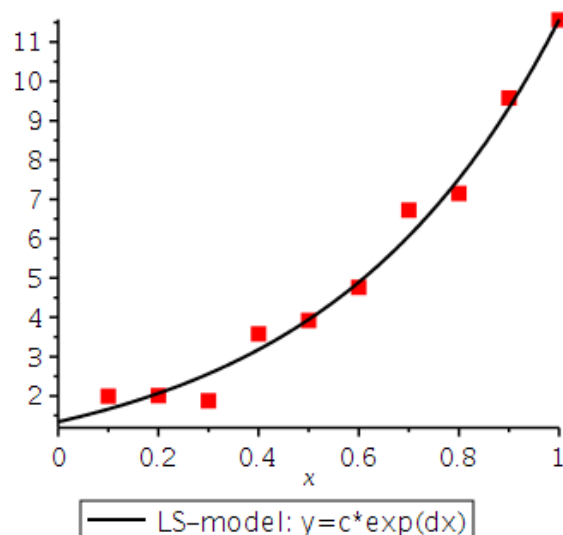
$$Y = a_0 + a_1X, \quad (7.17)$$

for which one can apply the **linear LS procedure**.

```

Linearized regression, in Maple
1 # The transformed data
2 xlny := Matrix(m, 2):
3 for i to m do
4     xlny[i, 1] := xy[i, 1];
5     xlny[i, 2] := ln(xy[i, 2]);
6 end do:
7
8 # The linear LS
9 L := CurveFitting[LeastSquares](xlny, x, curve = b*x + a);
10     0.295704647799999 + 2.1530740654363654 x
11
12 # Back to the original parameters
13 c := exp(0.295704647799999) = 1.344073123
14 d := 2.15307406543637:
15
16 # The desired nonlinear model
17 c*exp(d*x);
18     1.344073123 exp(2.15307406543637 x)

```



## 7.3. Scene Analysis with Noisy Data: Weighted Least-Squares and RANSAC

**Note:** **Scene analysis** is concerned with the interpretation of acquired data in terms of predefined models. It consists of 2 subproblems:

1. Finding the best model (**classification problem**)
2. Computing the best parameter values (**parameter estimation problem**)

- **Traditional parameter estimation techniques**, such as *least-squares* (LS), optimize the model to **all data points**.
  - Those techniques are simple averaging methods, based on the **smoothing assumption**: *There will always be good data points enough to smooth out any gross deviation.*
- However, in many interesting parameter estimation problems, **the smoothing assumption does not hold**; that is, the dataset may involve gross errors such as noise.
  - Thus, in order to obtain more reliable model parameters, there must be **internal mechanisms** to determine which points are matching to the model (**inliers**) and which points are false matches (**outliers**).

### 7.3.1. Weighted Least-Squares

**Definition 7.16.** When certain data points are more important or more reliable than the others, one may try to compute the coefficient vector with larger weights on more reliable data points. The **weighted least-squares method** is an LS method which involves a **weight matrix**  $W$ , often given as a diagonal matrix

$$W = \text{diag}(w_1, w_2, \dots, w_m), \quad (7.18)$$

which can be decided either *manually* or *automatically*.

**Algorithm 7.17. (Weighted Least-Squares)**

- Given data  $\{(x_i, y_i)\}$ ,  $1 \leq i \leq m$ , the best-fitting curve can be found by solving an over-determined algebraic system (7.8):

$$X\beta = y. \quad (7.19)$$

- When a weight matrix is applied, the above system can be written as

$$WX\beta = Wy. \quad (7.20)$$

- Thus its **weighted normal equations** read

$$X^T W X \beta = X^T W y. \quad (7.21)$$

**Example 7.18.** Given data, find the LS line with and without a weight. When a weight is applied, weigh the first and the last data point by 1/4.

$$xy := \begin{bmatrix} 1. & 2. & 3. & 4. & 5. & 6. & 7. & 8. & 9. & 10. \\ 5.89 & 1.92 & 2.59 & 4.41 & 4.49 & 6.22 & 7.74 & 7.07 & 9.05 & 5.7 \end{bmatrix}^T$$

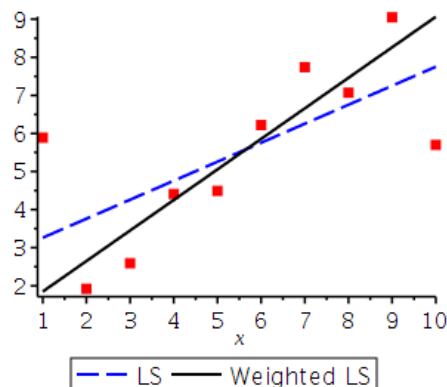
**Solution.**

## Weighted-LS

```

1 LS := CurveFitting[LeastSquares](xy, x);
2   2.7639999999999967 + 0.49890909090909125 x
3 WLS := CurveFitting[LeastSquares](xy, x,
4   weight = [1/4,1,1,1,1,1,1,1,1,1/4]);
5   1.0466694879390623 + 0.8019424460431653 x

```



### 7.3.2. RANdOm SAmple Consensus (RANSAC)

The **random sample consensus (RANSAC)** is one of the most powerful tools for the reconstruction of ground structures from point cloud observations in many applications. The algorithm utilizes **iterative search techniques** for a set of inliers, to find a proper model for given data.

**Algorithm 7.19. (RANSAC)** (Fischler-Bolles, 1981) [5]

*Input:* Measurement set  $\mathbf{X} = \{\mathbf{x}_i\}$ , the error tolerance  $\tau_e$ , the stopping threshold  $\eta$ , and the maximum number of iterations  $N$ .

1. Select randomly a **minimum point set**  $S$ , required to determine a hypothesis.
2. Generate a **hypothesis**  $\mathbf{p} = g(S)$ .
3. Compute the hypothesis **consensus set**, fitting within the error tolerance  $\tau_e$ :  

$$\mathcal{C} = \text{inlier}(\mathbf{X}, \mathbf{p}, \tau_e)$$
4. If  $|\mathcal{C}| \geq \gamma = \eta|\mathbf{X}|$ , then re-estimate a hypothesis  $\mathbf{p} = g(\mathcal{C})$  and stop.
5. Otherwise, repeat steps 1–4 (maximum of  $N$  times).

**Example 7.20.** Let's set a hypothesis for a **regression line**.

1. **Minimum point set**  $S$ : a set of **two points**,  $(x_1, y_1)$  and  $(x_2, y_2)$ .
2. **Hypothesis**  $\mathbf{p}$ :  $y = a + bx$  ( $\Rightarrow a + bx - y = 0$ )

$$y = b(x - x_1) + y_1 = a + bx \Leftrightarrow b = \frac{y_2 - y_1}{x_2 - x_1}, \quad a = y_1 - bx_1.$$

3. **Consensus set**  $\mathcal{C}$ :

$$\mathcal{C} = \left\{ (x_i, y_i) \in \mathbf{X} \mid d = \frac{|a + bx_i - y_i|}{\sqrt{b^2 + 1}} \leq \tau_e \right\} \quad (7.22)$$



**Note: Implementation of RANSAC**

- **Step 2:** A hypothesis  $\mathbf{p}$  is the set of model parameters, rather than the model itself.
- **Step 3:** The consensus set can be represented more conveniently by considering  $\mathcal{C}$  as an index array. That is,

$$\mathcal{C}(i) = \begin{cases} 1 & \text{if } \mathbf{x}_i \in \mathcal{C} \\ 0 & \text{if } \mathbf{x}_i \notin \mathcal{C} \end{cases} \quad (7.23)$$

See `inlier.m` implemented for Exercise 7.2, p. 209.

**Remark 7.21.** The “inlier” function in Step 3 collects points whose distance from the model,  $f(\mathbf{p})$ , is not larger than  $\tau_e$ . Thus, the distance can be interpreted as an **automatic weighting mechanism**. Indeed, for each point  $\mathbf{x}_i$ ,

$$\text{dist}(f(\mathbf{p}), \mathbf{x}_i) \begin{cases} \leq \tau_e, & \text{then } w_i = 1 \\ > \tau_e, & \text{then } w_i = 0 \end{cases} \quad (7.24)$$

Then the re-estimation in Step 4,  $\mathbf{p} = g(\mathcal{C})$ , can be seen as an parameter estimation  $\mathbf{p} = g(\mathbf{X})$  with the corresponding weight matrix  $W = \{w_1, w_2, \dots, w_m\}$ .

**Note: The Basic RANSAC Algorithm**

- It is an **iterative search** method for a set of inliers which may produce presumably accurate model parameters.
- It is simple to implement and efficient. However, it is problematic and often erroneous.
- The **main disadvantage of RANSAC** is that RANSAC is **unrepeatable**; it may yield different results in each run so that none of the results can be optimal.

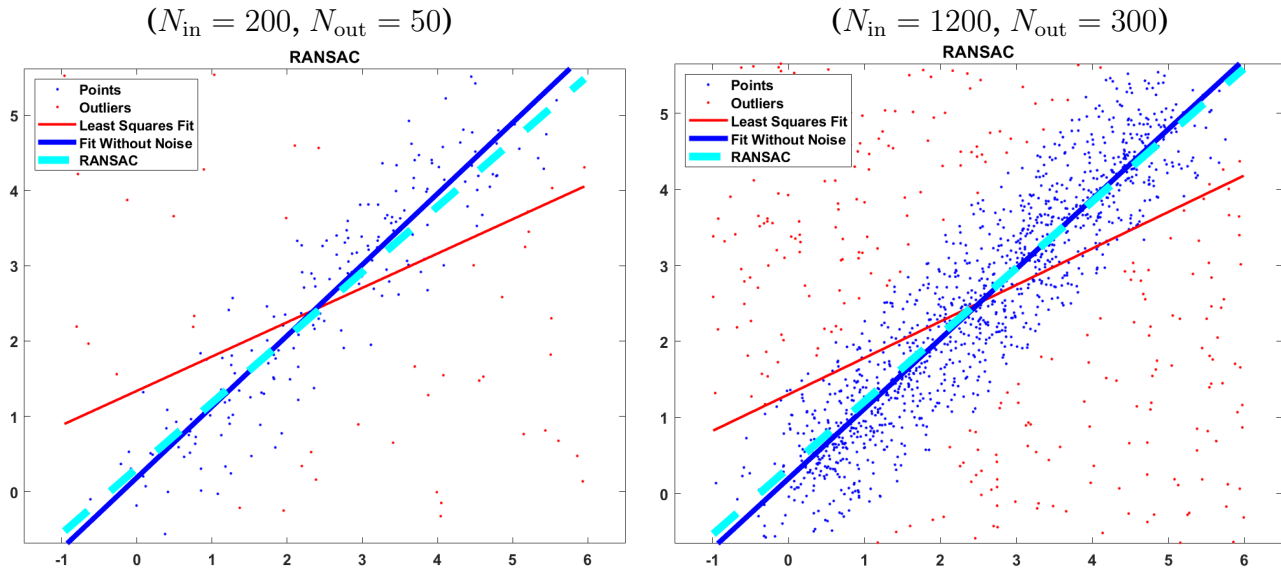


Figure 7.2: The RANSAC for linear-type synthetic datasets.

Table 7.1: The RANSAC: model fitting  $y = a_0 + a_1x$ . The algorithm runs 1000 times for each dataset to find the standard deviation of the error:  $\sigma(a_0 - \hat{a}_0)$  and  $\sigma(a_1 - \hat{a}_1)$ .

Data	$\sigma(a_0 - \hat{a}_0)$	$\sigma(a_1 - \hat{a}_1)$	E-time (sec)
1	0.1156	0.0421	0.0156
2	0.1101	0.0391	0.0348

### **RANSAC is neither repeatable nor optimal.**

In order to overcome the drawback, various variants have been studied in the literature. For variants, see e.g.,

- Maximum Likelihood Estimation Sample Consensus (MLESC) [12]
- Progressive Sample Consensus (PROSAC) [3]
- Recursive RANSAC (R-RANSAC) [8]

Nonetheless, RANSAC remains a prevailing algorithm for finding inliers.

## Exercises for Chapter 7

### 7.1. Given data

$x_i$	0.2	0.4	0.6	0.8	1.	1.2	1.4	1.6	1.8	2.
$y_i$	1.88	2.13	1.76	2.78	3.23	3.82	6.13	7.22	6.66	9.07

- Plot the data (scattered point plot)
- Decide what curve fits the data best.
- Use the **method of normal equations** to find the LS solution.
- Plot the curves superposed over the point plot.

### 7.2. This problem uses the data in Example 7.18, p.205.

- Implement the method of normal equations for the least-squares regression to find the best-fitting line.
- The RANSAC, Algorithm 7.19 is implemented for you below. Use the code to analyze the performance of the RANSAC.
  - Set  $\tau_e = 1$ ,  $\gamma = \eta|\mathbf{X}| = 8$ , and  $N = 100$ .
  - Run `ransac2` 100 times to get the minimum, maximum, and average number of iterations for the RANSAC to find an acceptable hypothesis consensus set.
- Plot the best-fitting lines found from (a) and (b), superposed over the data.

```

                                get_hypothesis_WLS.m
1  function p = get_hypothesis_WLS(X,C)
2  % Get hypothesis p, with C being used as weights
3  % Output: p = [a,b], where y= a+b*x
4
5  m = size(X,1);
6
7  A = [ones(m,1) X(:,1)];
8  A = A.*C;    %A = bsxfun(@times,A,C);
9  r = X(:,2).*C;
10
11 p = ((A'*A)\(A'*r))';

```

```

                                inlier.m
1  function C = inlier(X,p,tau_e)
2  % Input: p=[a,b] s.t. a+b*x-y=0
3
4  m = size(X,1);
5  C = zeros(m,1);
6
7  a = p(1); b=p(2);
8  factor = 1./sqrt(b^2+1);
9  for i=1:m
10     xi = X(i,1); yi = X(i,2);
11     dist = abs(a+b*xi-yi)*factor; %distance from point to line
12     if dist<=tau_e, C(i)=1; end
13 end

```

```

                                ransac2.m
1  function [p,C,iter] = ransac2(X,tau_e,gamma,N)
2  % Input:   X = {(x_i,y_i)}
3  %         tau_e: the error tolerance
4  %         gamma = eta*|X|
5  %         N: the maximum number of iterations
6  % Output:  p = [a,b], where y= a+b*x
7
8  %%-----
9  [m,n] = size(X);
10 if n>m, X=X'; [m,n] = size(X); end
11
12 for iter = 1:N
13     % step 1
14     s1 = randi([1 m]); s2 = randi([1 m]);
15     while s1==s2, s2 = randi([1 m]); end
16     S = [X(s1,:);X(s2,:)];
17     % step 2
18     p = get_hypothesis_WLS(S,[1;1]);
19     % step 3
20     C = inlier(X,p,tau_e);
21     % step 4
22     if sum(C)>=gamma
23         p = get_hypothesis_WLS(X,C);
24         break;
25     end
26 end

```

## CHAPTER 8

# Python Basics

### Contents of Chapter 8

8.1. Why Python? . . . . .	212
8.2. Python Essentials in 30 Minutes . . . . .	215
8.3. Zeros of a Polynomial in Python . . . . .	221
8.4. Python Classes . . . . .	225
Exercises for Chapter 8 . . . . .	232

## 8.1. Why Python?

**Note:** A good programming language must be **easy to learn and use** and **flexible and reliable**.

### Advantages of Python

**Python** has the following characteristics.

- Easy to learn and use
- Flexible and reliable
- Extensively used in **Data Science**
- Handy for **Web Development** purposes
- Having **Vast Libraries** support
- Among the **fastest-growing** programming languages in the tech industry

### Disadvantage of Python

Python is an interpreted and dynamically-typed language. The line-by-line execution of code, built with a high flexibility, most likely leads to **slow execution**. **Python scripts are way slow!**

### **Remark** 8.1. Speed up Python Programs

- Use **numpy** and **scipy** for all mathematical operations.
  - Always use **built-in functions** wherever possible.
- 
- **Cython**: It is designed as **a C-extension for Python**, which is developed **for users not familiar with C**. **A good choice!**
  - You may create and import your own **C/C++/Fortran-modules** into Python. If you extend Python with pieces of **compiled modules**, then the resulting code is easily **100× faster than Python scripts**. **The Best Choice!**

## How to call C/C++/Fortran from Python

Functions in C/C++/Fortran can be compiled using the shell script.

```

1  Compile-f90-c-cpp
2  #!/usr/bin/bash
3  LIB_F90='lib_f90'
4  LIB_GCC='lib_gcc'
5  LIB_GPP='lib_gpp'
6
7  ### Compiling: f90
8  f2py3 -c --f90flags='-O3' -m $LIB_F90 *.f90
9
10 ### Compiling: C (PIC: position-independent code)
11 gcc -fPIC -O3 -shared -o $LIB_GCC.so *.c
12
13 ### Compiling: C++
14 g++ -fPIC -O3 -shared -o $LIB_GPP.so *.cpp
```

The **shared objects** (\*.so) can be imported to the **Python wrap-up**.

```

1  Python Wrap-up
2  #!/usr/bin/python3
3  import numpy as np
4  import ctypes, time
5  from lib_py3 import *
6  from lib_f90 import *
7  lib_gcc = ctypes.CDLL("./lib_gcc.so")
8  lib_gpp = ctypes.CDLL("./lib_gpp.so")
9
10 ### For C/C++ -----
11 # e.g., lib_gcc.CFUNCTION(double array,double array,int,int)
12 #     returns a double value.
13 #-----
14 IN_ddii = [np.ctypeslib.ndpointer(dtype=np.double),
15            np.ctypeslib.ndpointer(dtype=np.double),
16            ctypes.c_int, ctypes.c_int] #input type
17 OUT_d = ctypes.c_double #output type
18
19 lib_gcc.CFUNCTION.argtypes = IN_ddii
20 lib_gcc.CFUNCTION.restype = OUT_d
21
22 result = lib_gcc.CFUNCTION(x,y,n,m)
```

- The library `numpy` is designed for a **Matlab-like implementation**.
- Python can be used as a convenient **desktop calculator**.
  - First, set a startup environment
  - Use Python as a desktop calculator

```

1  #.bashrc: export PYTHONSTARTUP=~/.python_startup.py
2  #.cshrc:  setenv PYTHONSTARTUP ~/.python_startup.py
3  #-----
4  print("\t^[[1;33m~/.python_startup.py")
5
6  import numpy as np; import sympy as sym
7  import numpy.linalg as la; import matplotlib.pyplot as plt
8  print("\tnp=numpy; la=numpy.linalg; plt=matplotlib.pyplot; sym=sympy")
9
10 from numpy import zeros,ones
11 print("\tzeros,ones, from numpy")
12
13 import random
14 from sympy import *
15 x,y,z,t = symbols('x,y,z,t');
16 print("\tfrom sympy import *; x,y,z,t = symbols('x,y,z,t')")
17
18 print("\t^[[1;37mTo see details: dir() or dir(np)^[[m")

```

```

[Thu Jan 12] python [Thu Jan 12] vi gradient-descent-method.tex
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux /home/skim/Books/Programming-Machine-Learning-Lecture
Type "help", "copyright", "credits" or "license" for more information.
~/python_startup.py
np=numpy; la=numpy.linalg; plt=matplotlib.pyplot; sym=sympy
zeros,ones, from numpy
from sympy import *; x,y,z,t = symbols('x,y,z,t')
To see details: dir() or dir(np)
>>>

```

Figure 8.1: Python startup.



## 8.2. Python Essentials in 30 Minutes

### Key Features of Python

- Python is a **simple, readable, open source** programming language which is easy to learn.
- It is an **interpreted** language, not a compiled language.
- In Python, **variables are untyped**; i.e., there is no need to define the data type of a variable while declaring it.
- Python supports **object-oriented programming** models.
- It is **platform-independent** and easily extensible and embeddable.
- It has a **huge standard library** with lots of modules and packages.
- Python is a **high level language** as it is easy to use because of simple syntax, **powerful** because of its rich libraries and extremely versatile.

### Programming Features

- Python has **no support pointers**.
- Python codes are stored with **.py** extension.
- **Indentation**: Python uses indentation to define a block of code.
  - A **code block** (body of a function, loop, etc.) starts with indentation and ends with the first unindented line.
  - The amount of indentation is up to the user, but it must be consistent throughout that block.
- **Comments**:
  - The hash (#) symbol is used to start writing a comment.
  - **Multi-line comments**: Python uses triple quotes, either ''' or """.

## Python Essentials

- **Sequence datatypes:** list, tuple, string
  - **[list]:** defined using square brackets (and commas)  

```
>>> li = ["abc", 14, 4.34, 23]
```
  - **(tuple):** defined using parentheses (and commas)  

```
>>> tu = (23, (4,5), 'a', 4.1, -7)
```
  - **"string":** defined using quotes (" , ' , or """)  

```
>>> st = 'Hello World'  
>>> st = "Hello World"  
>>> st = """This is a multi-line string  
... that uses triple quotes."""
```
- **Retrieving elements**  

```
>>> li[0]  
'abc'  
>>> tu[1],tu[2],tu[-2]  
((4, 5), 'a', 4.1)  
>>> st[25:36]  
'ng\nthat use'
```
- **Slicing**  

```
>>> tu[1:4] # be aware  
((4, 5), 'a', 4.1)
```
- **The + and \* operators**  

```
>>> [1, 2, 3]+[4, 5, 6,7]  
[1, 2, 3, 4, 5, 6, 7]  
>>> "Hello" + " " + 'World'  
Hello World  
>>> (1,2,3)*3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

- **Reference semantics**

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

**Be aware with copying lists and numpy arrays!**

- **numpy, range, and iteration**

```
>>> range(8)
[0, 1, 2, 3, 4, 5, 6, 7]
>>> import numpy as np
>>> for k in range(np.size(li)):
...     li[k]
... <Enter>
'abc'
14
4.34
23
```

- **numpy array and deepcopy**

```
>>> from copy import deepcopy
>>> A = np.array([1,2,3])
>>> B = A
>>> C = deepcopy(A)
>>> A *= 4
>>> B
array([ 4,  8, 12])
>>> C
array([1, 2, 3])
```

## Frequently used Python Rules

```
_____ frequently_used_rules.py _____
1  ## Multi-line statement
2  a = 1 + 2 + 3 + 4 + 5 + \
3      6 + 7 + 8 + 9 + 10
4  b = (1 + 2 + 3 + 4 + 5 +
5      6 + 7 + 8 + 9 + 10) #inside (), [], or {}
6  print(a,b)
7  # Output: 55 55
8
9  ## Multiple statements in a single line using ";"
10 a = 1; b = 2; c = 3
11
12 ## Docstrings in Python
13 def double(num):
14     """Function to double the value"""
15     return 2*num
16 print(double.__doc__)
17 # Output: Function to double the value
18
19 ## Assigning multiple values to multiple variables
20 a, b, c = 1, 2, "Hello"
21 ## Swap
22 b, c = c, b
23 print(a,b,c)
24 # Output: 1 Hello 2
25
26 ## Data types in Python
27 a = 5; b = 2.1
28 print("type of (a,b)", type(a), type(b))
29 # Output: type of (a,b) <class 'int'> <class 'float'>
30
31 ## Python Set: 'set' object is not subscriptable
32 a = {5,2,3,1,4}; b = {1,2,2,3,3,3}
33 print("a=",a,"b=",b)
34 # Output: a= {1, 2, 3, 4, 5} b= {1, 2, 3}
```

```
35
36 ## Python Dictionary
37 d = {'key1':'value1', 'Seth':22, 'Alex':21}
38 print(d['key1'],d['Alex'],d['Seth'])
39 # Output: value1 21 22
40
41 ## Output Formatting
42 x = 5.1; y = 10
43 print('x = %d and y = %d' %(x,y))
44 print('x = %f and y = %d' %(x,y))
45 print('x = {} and y = {}'.format(x,y))
46 print('x = {1} and y = {0}'.format(x,y))
47 # Output: x = 5 and y = 10
48 #           x = 5.100000 and y = 10
49 #           x = 5.1 and y = 10
50 #           x = 10 and y = 5.1
51
52 print("x=",x,"y=",y, sep="#",end="&\n")
53 # Output: x=#5.1#y=#10&
54
55 ## Python Interactive Input
56 C = input('Enter any: ')
57 print(C)
58 # Output: Enter any: Starkville
59 #           Starkville
```

## Looping and Functions

**Example 8.2.** Compose a Python function which returns cubes of natural numbers.

**Solution.**

```

_____ get_cubes.py _____
1  def get_cubes(num):
2      cubes = []
3      for i in range(1,num+1):
4          value = i**3
5          cubes.append(value)
6      return cubes
7
8  if __name__ == '__main__':
9      num = input('Enter a natural number: ')
10     cubes = get_cubes(int(num))
11     print(cubes)

```

**Remark 8.3.** `get_cubes.py`

- Lines 8-11 are added for the function to be called directly. That is,
 

```
[Sun Nov.05] python get_cubes.py
Enter a natural number: 6
[1, 8, 27, 64, 125, 216]
```
- When `get_cubes` is called from another function, the last four lines will not be executed.

```

_____ call_get_cubes.py _____
1  from get_cubes import *
2
3  cubes = get_cubes(8)
4  print(cubes)

```

Execusion

```

1  [Sun Nov.05] python call_get_cubes.py
2  [1, 8, 27, 64, 125, 216, 343, 512]

```

## 8.3. Zeros of a Polynomial in Python

In this section, we will implement **a Python code** for zeros of a polynomial and **compare it with a Matlab code**.

**Recall:** Let's begin with recalling how to find zeros of a polynomial, presented in §3.6.

- **Remark 3.69:** When the Newton's method is applied for finding an approximate zero of  $P(x)$ , the iteration reads

$$x_n = x_{n-1} - \frac{P(x_{n-1})}{P'(x_{n-1})}. \quad (8.1)$$

Thus both  $P(x)$  and  $P'(x)$  must be evaluated in each iteration.

- **Strategy 3.70:** **The derivative  $P'(x)$  can be evaluated by using the Horner's method with the same efficiency.** Indeed, differentiating (3.90)

$$P(x) = (x - x_0)Q(x) + P(x_0)$$

reads

$$P'(x) = Q(x) + (x - x_0)Q'(x). \quad (8.2)$$

Thus

$$P'(x_0) = Q(x_0). \quad (8.3)$$

That is, the evaluation of  $Q$  at  $x_0$  becomes the desired quantity  $P'(x_0)$ .

**Example 8.4. (Revisit of Example 3.73, p. 108)**

Let  $P(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$ . Use the Newton's method and the Horner's method to implement a code and find an approximate zero of  $P$  near 3.

**Solution.** First, let's try to use built-in functions.

```

zeros_of_poly_built_in.py
1  import numpy as np
2
3  coeff = [1, -4, 7, -5, -2]
4  P = np.poly1d(coeff)
5  Pder = np.polyder(P)
6
7  print(P)
8  print(Pder)
9  print(np.roots(P))
10 print(P(3), Pder(3))

```

```

Output
1      4      3      2
2  1 x - 4 x + 7 x - 5 x - 2
3      3      2
4  4 x - 12 x + 14 x - 5
5  [ 2. +0.j  1.1378411+1.52731225j  1.1378411-1.52731225j  -0.2756822+0.j ]
6  19 37

```

**Observation 8.5.** We will see:

Python programming is **as easy and simple as** Matlab programming.

- In particular, **numpy** is developed for **Matlab-like implementation, with enhanced convenience**.
- Numpy is used extensively in most of scientific Python packages: SciPy, Pandas, Matplotlib, scikit-learn, ...



Now, we implement **a code in Python** for Newton-Horner method to find an approximate zero of  $P$  near 3.

```

----- Zeros-Polynomials-Newton-Horner.py -----
1  def horner(A,x0):
2      """ input:  A = [a_n,...,a_1,a_0]
3          output: p,d = P(x0),DP(x0) = horner(A,x0) """
4      n = len(A)
5      p = A[0]; d = 0
6
7      for i in range(1,n):
8          d = p + x0*d
9          p = A[i] +x0*p
10     return p,d
11
12     def newton_horner(A,x0,tol,itmax):
13         """ input:  A = [a_n,...,a_1,a_0]
14             output: x: P(x)=0 """
15         x=x0
16         for it in range(1,itmax+1):
17             p,d = horner(A,x)
18             h = -p/d;
19             x = x + h;
20             if(abs(h)<tol): break
21         return x,it
22
23     if __name__ == '__main__':
24         coeff = [1, -4, 7, -5, -2]; x0 = 3
25         tol = 10**(-12); itmax = 1000
26         x,it =newton_horner(coeff,x0,tol,itmax)
27         print("newton_horner: x0=%g; x=%g, in %d iterations" %(x0,x,it))

```

----- Execution -----

```

1  [Sat Jul.23] python Zeros-Polynomials-Newton-Horner.py
2  newton_horner: x0=3; x=2, in 7 iterations

```

**Note:** The above Python code must be compared with the Matlab code in §3.6:

\_\_\_\_\_ horner.m \_\_\_\_\_

```

1 function [p,d] = horner(A,x0)
2 % input: A = [a_0,a_1,...,a_n]
3 % output: p=P(x0), d=P'(x0)
4
5 n = size(A(:),1);
6 p = A(n); d=0;
7
8 for i = n-1:-1:1
9     d = p + x0*d;
10    p = A(i) +x0*p;
11 end

```

\_\_\_\_\_ newton\_horner.m \_\_\_\_\_

```

1 function [x,it] = newton_horner(A,x0,tol,itmax)
2 % input: A = [a_0,a_1,...,a_n]; x0: initial for P(x)=0
3 % outpue: x: P(x)=0
4
5 x = x0;
6 for it=1:itmax
7     [p,d] = horner(A,x);
8     h = -p/d;
9     x = x + h;
10    if(abs(h)<tol), break; end
11 end

```

\_\_\_\_\_ Call\_newton\_horner.m \_\_\_\_\_

```

1 a = [-2 -5 7 -4 1];
2 x0=3;
3 tol = 10^-12; itmax=1000;
4 [x,it] = newton_horner(a,x0,tol,itmax);
5 fprintf(" newton_horner: x0=%g; x=%g, in %d iterations\n",x0,x,it)
6     Result: newton_horner: x0=3; x=2, in 7 iterations

```

## 8.4. Python Classes

### **Remark 8.6. Object-Oriented Programming (OOP)**

Classes are a key concept in the object-oriented programming.

**Classes provide a means of bundling data and functionality together.**

- A **class** is a user-defined template or prototype from which real-world objects are created.
- The major merit of using classes is on the **sharing mechanism** between functions/methods and objects.
  - **Initialization** and the **sharing boundaries** must be declared clearly and conveniently.
- A class tells us
  - what data an object should have,
  - what are the initial/default values of the data, and
  - what methods are associated with the object to take actions on the objects using their data.
- An object is an **instance** of a class, and creating an object from a class is called **instantiation**.

In the following, we would build a simple class, as Dr. Xu did in [14, Appendix B.5]; you will learn how to **initiate, refine, and use classes**.

## Initiation of a Class

```

Polynomial_01.py
1 class Polynomial():
2     """A class of polynomials"""
3
4     def __init__(self,coefficient):
5         """Initialize coefficient attribute of a polynomial."""
6         self.coeff = coefficient
7
8     def degree(self):
9         """Find the degree of a polynomial"""
10        return len(self.coeff)-1
11
12 if __name__ == '__main__':
13     p2 = Polynomial([1,2,3])
14     print(p2.coeff)      # a variable; output: [1, 2, 3]
15     print(p2.degree()) # a method;   output: 2

```

- **Lines 1-2:** define a class called Polynomial with a docstring.
  - The parentheses in the class definition are empty because we create this class from scratch.
- **Lines 4-10:** define two functions, `__init__()` and `degree()`. A function in a class is called a **method**.
  - **The `__init__()` method** is a special method for initialization; it is called the `__init__()` **constructor**.
  - **The `self` Parameter and Its Sharing**
    - \* The `self` parameter is required and must come first before the other parameters in each method.
    - \* The variable `self.coeff` (**prefixed with `self`**) is **available** to every method and is **accessible** by any objects created from the class. (Variables prefixed with `self` are called **attributes**.)
    - \* We do not need to provide arguments for `self`.
- **Line 13:** The line `p2 = Polynomial([1,2,3])` creates an object `p2` (a polynomial  $x^2 + 2x + 3$ ), by passing the coefficient list `[1,2,3]`.
  - When Python reads this line, it calls the method `__init__()` in the class Polynomial and creates the object named `p2` that represents this particular polynomial  $x^2 + 2x + 3$ .

**Refinement of the Polynomial class**

```

Polynomial_02.py
1 class Polynomial():
2     """A class of polynomials"""
3
4     count = 0      #Polynomial.count
5
6     def __init__(self):
7         """Initialize coefficient attribute of a polynomial."""
8         self.coeff = [1]
9         Polynomial.count += 1
10
11    def __del__(self):
12        """Delete a polynomial object"""
13        Polynomial.count -= 1
14
15    def degree(self):
16        """Find the degree of a polynomial"""
17        return len(self.coeff)-1
18
19    def evaluate(self,x):
20        """Evaluate a polynomial."""
21        n = self.degree(); eval = []
22        for xi in x:
23            p = self.coeff[0]      #Horner's method
24            for k in range(1,n+1): p = self.coeff[k]+ xi*p
25            eval.append(p)
26        return eval
27
28    if __name__ == '__main__':
29        poly1 = Polynomial()
30        print('poly1, default coefficients:', poly1.coeff)
31        poly1.coeff = [1,2,-3]
32        print('poly1, coefficients after reset:', poly1.coeff)
33        print('poly1, degree:', poly1.degree())
34
35        poly2 = Polynomial(); poly2.coeff = [1,2,3,4,-5]
36        print('poly2, coefficients after reset:', poly2.coeff)
37        print('poly2, degree:', poly2.degree())
38
39        print('number of created polynomials:', Polynomial.count)
40        del poly1
41        print('number of polynomials after a deletion:', Polynomial.count)
42        print('poly2.evaluate([-1,0,1,2]):', poly2.evaluate([-1,0,1,2]))

```

- **Line 4: (Global Variable)** The variable `count` is a **class attribute** of `Polynomial`.
  - It belongs to the class but not a particular object.
  - All objects of the class share this same variable (`Polynomial.count`).
- **Line 8: (Initialization)** Initializes the class attribute `self.coeff`.
  - Every object or class attribute in a class needs an initial value.
  - One can set a **default value** for an object attribute in the `__init__()` constructor; and we do not have to include a parameter for that attribute. See Lines 29 and 35.
- **Lines 11-13: (Deletion of Objects)** Define the `__del__()` method in the class for the deletion of objects. See Line 40.
  - `del` is a built-in function which deletes variables and objects.
- **Lines 19-28: (Add Methods)** Define another method called `evaluate`, which uses the *Horner's method*. See Example 8.4, p.222.

## Output

```
1 poly1, default coefficients: [1]
2 poly1, coefficients after reset: [1, 2, -3]
3 poly1, degree: 2
4 poly2, coefficients after reset: [1, 2, 3, 4, -5]
5 poly2, degree: 4
6 number of created polynomials: 2
7 number of polynomials after a deletion: 1
8 poly2.evaluate([-1,0,1,2]): [-7, -5, 5, 47]
```

## Inheritance

**Note:** If we want to write a class that is just *a specialized version of another class*, we do not need to write the class from scratch.

- We call the specialized class a **child class** and the other general class a **parent class**.
- The child class can inherit all the attributes and methods from the parent class.
  - It can also define its own special attributes and methods or even overrides methods of the parent class.

Classes can import functions implemented earlier, to define methods.

```

Classes.py
1  from util_Poly import *
2
3  class Polynomial():
4      """A class of polynomials"""
5
6      def __init__(self,coefficient):
7          """Initialize coefficient attribute of a polynomial."""
8          self.coeff = coefficient
9
10     def degree(self):
11         """Find the degree of a polynomial"""
12         return len(self.coeff)-1
13
14     class Quadratic(Polynomial):
15         """A class of quadratic polynomial"""
16
17         def __init__(self,coefficient):
18             """Initialize the coefficient attributes ."""
19             super().__init__(coefficient)
20             self.power_decrease = 1
21
22         def roots(self):
23             return roots_Quad(self.coeff,self.power_decrease)
24
25         def degree(self):
26             return 2

```

- **Line 1:** Imports functions implemented earlier.
- **Line 14:** We must include the name of the parent class in the parentheses of the definition of the child class (to indicate the parent-child relation for inheritance).
- **Line 19:** The `super()` function is to give an child object all the attributes defined in the parent class.
- **Line 20:** An additional child class attribute `self.power_decrease` is initialized.
- **Lines 22-23:** define a new method called `roots`, reusing a function implemented earlier.
- **Lines 25-26:** The method `degree()` overrides the parent's method.

```

_____ util_Poly.py _____
1 def roots_Quad(coeff,power_decrease):
2     a,b,c = coeff
3     if power_decrease != 1:
4         a,c = c,a
5     discriminant = b**2-4*a*c
6     r1 = (-b+discriminant**0.5)/(2*a)
7     r2 = (-b-discriminant**0.5)/(2*a)
8     return [r1,r2]

```

```

_____ call_Quadratic.py _____
1 from Classes import *
2
3 quad1 = Quadratic([2,-3,1])
4 print('quad1, roots:',quad1.roots())
5 quad1.power_decrease = 0
6 print('roots when power_decrease = 0:',quad1.roots())

```

```

_____ Output _____
1 quad1, roots: [1.0, 0.5]
2 roots when power_decrease = 0: [2.0, 1.0]

```



### Final Remarks on Python Implementation

- A proper **modularization** must precede implementation, as for other programming languages.
- Classes are used quite frequently.
  - You do not have to use classes for small projects.
- Try to use classes **smartly**.  
Quite often, they add unnecessary complications and their methods are *hardly* applicable directly for other projects.
  - You may implement **stand-alone functions** to import.
  - This strategy enhances **reusability** of functions.  
For example, the function `roots_Quad` defined in `util_Poly.py` (page 230) can be used directly for other projects.
  - Afterwards, you will get **your own utility functions**; using them, you can complete various programming tasks effectively.

## Exercises for Chapter 8

*You should use Python for the following problems.*

- 8.1. Use nested for loops to assign entries of a  $5 \times 5$  matrix  $A$  such that  $A[i, j] = ij$ .
- 8.2. The variable  $d$  is initially equal to 1. Use a while loop to keep dividing  $d$  by 2 until  $d < 10^{-6}$ .
  - (a) Determine how many divisions are made.
  - (b) Verify your result by algebraic derivation.

**Note:** A while loop has not been considered in the lecture. However, you can figure it out easily by yourself.

- 8.3. Write a function that takes as input a list of values and returns the largest value. Do this without using the Python `max()` function; you should combine a for loop and an if statement.
  - (a) Produce a random list of size 10-20 to verify your function.
- 8.4. Let  $P_4(x) = 2x^4 - 5x^3 - 11x^2 + 20x + 10$ . Solve the following.
  - (a) Plot  $P_4$  over the interval  $[-3, 4]$ .
  - (b) Find all zeros of  $P_4$ , modifying `Zeros-Polynomials-Newton-Horner.py`, p.222.
  - (c) Add markers for the zeros to the plot.
  - (d) Find all roots of  $P_4'(x) = 0$ .
  - (e) Add markers for the zeros of  $P_4'$  to the plot.

**Hint:** For plotting, you may import: “import matplotlib.pyplot as plt” then use `plt.plot()`. You will see the Python plotting is quite similar to Matlab plotting.

## CHAPTER 9

# Vector Spaces and Orthogonality

### Contents of Chapter 9

9.1. Subspaces of $\mathbb{R}^n$ . . . . .	234
9.2. Orthogonal Sets and Orthogonal Matrix . . . . .	238
9.3. Orthogonal Projections . . . . .	243
9.4. The Gram-Schmidt Process and QR Factorization . . . . .	248
9.5. QR Iteration for Finding Eigenvalues . . . . .	253
Exercises for Chapter 9 . . . . .	257

## 9.1. Subspaces of $\mathbb{R}^n$

**Definition 9.1.** A **subspace** of  $\mathbb{R}^n$  is any set  $H$  in  $\mathbb{R}^n$  that has three properties:

- The **zero vector** is in  $H$ .
- For each  $u$  and  $v$  in  $H$ , the sum  $u + v$  is in  $H$ .
- For each  $u$  in  $H$  and each scalar  $c$ , the vector  $cu$  is in  $H$ .

That is,  $H$  is **closed under linear combinations**.

**Remark 9.2.**  $\mathbb{R}^n$ , with the *standard addition* and *scalar multiplication*, is a **vector space**.

### Example 9.3.

- A line through the origin in  $\mathbb{R}^2$  is a subspace of  $\mathbb{R}^2$ .
- Any plane through the origin in  $\mathbb{R}^3$  is a subspace of  $\mathbb{R}^3$ .

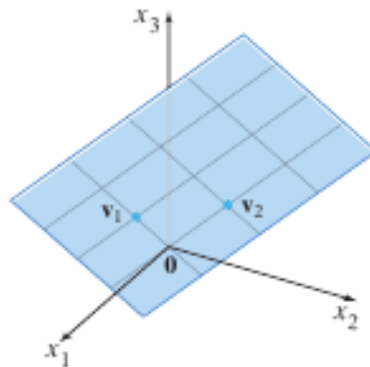


Figure 9.1:  $\text{Span}\{v_1, v_2\}$  as a plane through the origin.

- Let  $v_1, v_2, \dots, v_p \in \mathbb{R}^n$ . Then  $\text{Span}\{v_1, v_2, \dots, v_p\}$  is a subspace of  $\mathbb{R}^n$ .
- Every spanned set in  $\mathbb{R}^n$  is a subspace.

Given a matrix  $A \in \mathbb{R}^{m \times n}$ , we may consider the subspace spanned by the column vectors of  $A$ .

**Definition 9.4.** Let  $A$  be an  $m \times n$  matrix.

The **column space** of  $A$  is the set ( $Col A$ ) of all linear combinations of columns of  $A$ . That is, if  $A = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n]$ , then

$$Col A = \{\mathbf{u} \mid \mathbf{u} = c_1 \mathbf{a}_1 + c_2 \mathbf{a}_2 + \cdots + c_n \mathbf{a}_n\}, \quad (9.1)$$

where  $c_1, c_2, \dots, c_n$  are scalars.  $Col A$  is a subspace of  $\mathbb{R}^m$ .

**Example 9.5.** Let  $A = \begin{bmatrix} 1 & -3 & -4 \\ -4 & 6 & -2 \\ -3 & 7 & 6 \end{bmatrix}$  and  $\mathbf{b} = \begin{bmatrix} 3 \\ 3 \\ -4 \end{bmatrix}$ . Determine whether

$\mathbf{b}$  is in the column space of  $A$ ,  $Col A$ .

**Solution.** *Clue:* ①  $\mathbf{b} \in Col A$

$\Leftrightarrow$  ②  $\mathbf{b}$  is a linear combinations of columns of  $A$

$\Leftrightarrow$  ③  $A\mathbf{x} = \mathbf{b}$  is consistent

$\Leftrightarrow$  ④  $[A \ \mathbf{b}]$  has a solution

**Definition 9.6.** Let  $A$  be an  $m \times n$  matrix. The **null space** of  $A$ ,  $Nul A$ , is the set of all solutions of the homogeneous system  $A\mathbf{x} = \mathbf{0}$ .

**Theorem 9.7.**  $Nul A$  is a subspace of  $\mathbb{R}^n$ .

**Proof.**

### Basis for a Subspace

**Definition 9.8.** A **basis** for a subspace  $H$  in  $\mathbb{R}^n$  is a set of vectors that

1. is *linearly independent*, and
2. *spans*  $H$ .

**Remark 9.9.**

1.  $\left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\}$  is a basis for  $\mathbb{R}^2$ .

2. Let  $\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ ,  $\mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ ,  $\dots$ ,  $\mathbf{e}_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$ . Then  $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$  is called the **standard basis** for  $\mathbb{R}^n$ .

**Example 9.10.** Find a basis for the column space of the matrix

$$B = \begin{bmatrix} 1 & 0 & -3 & 5 & 0 \\ 0 & 1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

**Solution.** Observation:  $\mathbf{b}_3 = -3\mathbf{b}_1 + 2\mathbf{b}_2$  and  $\mathbf{b}_4 = 5\mathbf{b}_1 - \mathbf{b}_2$ .

**Theorem 9.11.** In general, non-pivot columns are linear combinations of preceding pivot columns. Thus the pivot columns of a matrix  $A$  forms a basis for  $\text{Col } A$ .

**Example 9.12.** Find bases for the column space and the null space of the matrix

$$A = \begin{bmatrix} -3 & 6 & -1 & 1 \\ 1 & -2 & 2 & 3 \\ 2 & -4 & 5 & 8 \end{bmatrix}.$$

**Solution.**  $A \sim \begin{bmatrix} 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

*Ans: Col A = {a<sub>1</sub>, a<sub>3</sub>}*

**Theorem 9.13.** A basis for  $\text{Nul } A$  can be obtained from the parametric vector form of solutions of  $Ax = 0$ . That is, suppose that the solutions of  $Ax = 0$  reads

$$\mathbf{x} = x_1 \mathbf{u}_1 + x_2 \mathbf{u}_2 + \cdots + x_k \mathbf{u}_k,$$

where  $x_1, x_2, \dots, x_k$  correspond to free variables. Then, a basis for  $\text{Nul } A$  is  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k\}$ .

**Theorem 9.14. (Rank Theorem)** Let  $A \in \mathbb{R}^{m \times n}$ . Then

$$\begin{aligned} \dim \text{Col } A + \dim \text{Nul } A &= \text{rank } A + \text{nullity } A = n \\ &= (\text{the number of columns in } A) \end{aligned}$$

Here, “ $\dim \text{Nul } A$ ” is called the **nullity** of  $A$ :  $\text{nullity } A$

## 9.2. Orthogonal Sets and Orthogonal Matrix

**Definition 9.15.** A set of vectors  $S = \{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\}$  in  $\mathbb{R}^n$  is said to be an **orthogonal set** if each pair of distinct vectors from the set is orthogonal. That is,

$$\mathbf{u}_i \bullet \mathbf{u}_j = 0, \quad \text{for } i \neq j.$$

If the vectors in  $S$  are **nonzero**, then  $S$  is linearly independent and therefore forms a basis for the subspace spanned by  $S$ .

**Definition 9.16.** An **orthogonal basis** for a subspace  $W$  of  $\mathbb{R}^n$  is a basis for  $W$  that is also an orthogonal set.

The following theorem shows one of reasons why orthogonality is a useful property in vector spaces and matrix algebra.

**Theorem 9.17.** Let  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\}$  be an orthogonal basis for a subspace  $W$  of  $\mathbb{R}^n$ . For each  $\mathbf{y}$  in  $W$ , the weights in the linear combination

$$\mathbf{y} = c_1 \mathbf{u}_1 + c_2 \mathbf{u}_2 + \dots + c_p \mathbf{u}_p \quad (9.2)$$

are given by

$$c_j = \frac{\mathbf{y} \bullet \mathbf{u}_j}{\mathbf{u}_j \bullet \mathbf{u}_j} \quad (j = 1, 2, \dots, p). \quad (9.3)$$

**Proof.**  $\mathbf{y} \bullet \mathbf{u}_j = (c_1 \mathbf{u}_1 + c_2 \mathbf{u}_2 + \dots + c_p \mathbf{u}_p) \bullet \mathbf{u}_j = c_j \mathbf{u}_j \bullet \mathbf{u}_j$ , from which we can conclude (9.3).  $\square$

**Example 9.18.** Consider a set of vectors  $S = \{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ , where

$$\mathbf{u}_1 = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}, \quad \mathbf{u}_2 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}, \quad \text{and } \mathbf{u}_3 = \begin{bmatrix} -5 \\ -2 \\ 1 \end{bmatrix}.$$

(a) Is  $S$  orthogonal? (b) Express the vector  $\mathbf{y} = [11, 0, -5]^T$  as a linear combination of the vectors in  $S$ .

**Solution.**

$$\text{Ans: } \mathbf{y} = \mathbf{u}_1 - 2\mathbf{u}_2 - 2\mathbf{u}_3.$$



### An Orthogonal Projection

**Note:** Given a nonzero vector  $\mathbf{u}$  in  $\mathbb{R}^n$ , consider the problem of decomposing a vector  $\mathbf{y} \in \mathbb{R}^n$  into sum of two vectors, one a multiple of  $\mathbf{u}$  and the other orthogonal to  $\mathbf{u}$ . Let

$$\mathbf{y} = \hat{\mathbf{y}} + \mathbf{z}, \quad \hat{\mathbf{y}} // \mathbf{u} \text{ and } \mathbf{z} \perp \mathbf{u}.$$

Let  $\hat{\mathbf{y}} = \alpha \mathbf{u}$ . Then

$$0 = \mathbf{z} \bullet \mathbf{u} = (\mathbf{y} - \alpha \mathbf{u}) \bullet \mathbf{u} = \mathbf{y} \bullet \mathbf{u} - \alpha \mathbf{u} \bullet \mathbf{u}.$$

Thus  $\alpha = \mathbf{y} \bullet \mathbf{u} / \mathbf{u} \bullet \mathbf{u}$ .

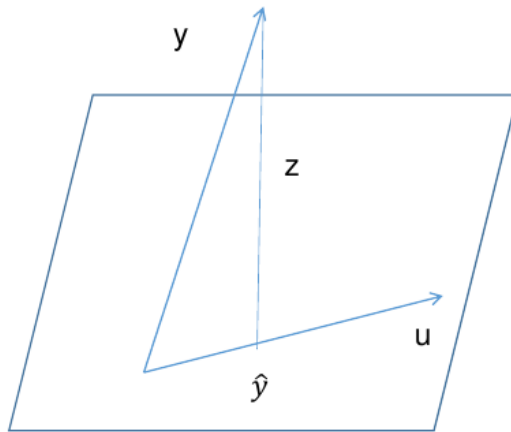


Figure 9.2: Orthogonal projection:  $\mathbf{y} = \hat{\mathbf{y}} + \mathbf{z}$ .

**Definition 9.19.** Given a nonzero vector  $\mathbf{u}$  in  $\mathbb{R}^n$ , for  $\mathbf{y} \in \mathbb{R}^n$ , let

$$\mathbf{y} = \hat{\mathbf{y}} + \mathbf{z}, \quad \hat{\mathbf{y}} // \mathbf{u} \text{ and } \mathbf{z} \perp \mathbf{u}. \quad (9.4)$$

Then

$$\hat{\mathbf{y}} = \alpha \mathbf{u} = \frac{\mathbf{y} \bullet \mathbf{u}}{\mathbf{u} \bullet \mathbf{u}} \mathbf{u}, \quad \mathbf{z} = \mathbf{y} - \hat{\mathbf{y}}. \quad (9.5)$$

The vector  $\hat{\mathbf{y}}$  is called the **orthogonal projection of  $\mathbf{y}$  onto  $\mathbf{u}$** , and  $\mathbf{z}$  is called the **component of  $\mathbf{y}$  orthogonal to  $\mathbf{u}$** . Let  $L = \text{Span}\{\mathbf{u}\}$ . Then we denote

$$\hat{\mathbf{y}} = \frac{\mathbf{y} \bullet \mathbf{u}}{\mathbf{u} \bullet \mathbf{u}} \mathbf{u} = \text{proj}_L \mathbf{y}, \quad (9.6)$$

which is called the **orthogonal projection of  $\mathbf{y}$  onto  $L$** .

**Example 9.20.** Let  $\mathbf{y} = \begin{bmatrix} 7 \\ 6 \end{bmatrix}$  and  $\mathbf{u} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$ .

- Find the orthogonal projection of  $\mathbf{y}$  onto  $\mathbf{u}$ .
- Write  $\mathbf{y}$  as the sum of two orthogonal vectors, one in  $L = \text{Span}\{\mathbf{u}\}$  and one orthogonal to  $\mathbf{u}$ .
- Find the distance from  $\mathbf{y}$  to  $L$ .

**Solution.**

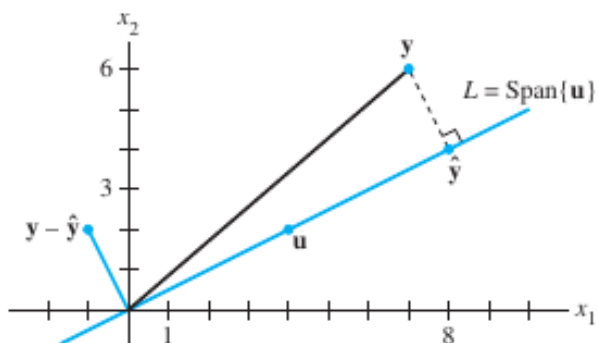


Figure 9.3: The orthogonal projection of  $\mathbf{y}$  onto  $L = \text{Span}\{\mathbf{u}\}$ .

**Orthonormal Sets**

**Definition 9.21.** A set  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\}$  is an **orthonormal set**, if it is an orthogonal set of *unit* vectors. If  $W$  is the subspace spanned by such a set, then  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\}$  is an **orthonormal basis** for  $W$ , since the set is automatically linearly independent.

**Example 9.22.** In Example 9.18, p. 238, we know  $\mathbf{v}_1 = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$ ,  $\mathbf{v}_2 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$ ,

and  $\mathbf{v}_3 = \begin{bmatrix} -5 \\ -2 \\ 1 \end{bmatrix}$  form an *orthogonal* basis for  $\mathbb{R}^3$ . Find the corresponding *orthonormal* basis.

**Solution.**

**Theorem 9.23.** An  $m \times n$  matrix  $U$  has orthonormal columns if and only if  $U^T U = I$ .

**Proof.** To simplify notation, we suppose that  $U$  has only three columns:  $U = [\mathbf{u}_1 \ \mathbf{u}_2 \ \mathbf{u}_3]$ ,  $\mathbf{u}_i \in \mathbb{R}^m$ . Then

$$U^T U = \begin{bmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \mathbf{u}_3^T \end{bmatrix} [\mathbf{u}_1 \ \mathbf{u}_2 \ \mathbf{u}_3] = \begin{bmatrix} \mathbf{u}_1^T \mathbf{u}_1 & \mathbf{u}_1^T \mathbf{u}_2 & \mathbf{u}_1^T \mathbf{u}_3 \\ \mathbf{u}_2^T \mathbf{u}_1 & \mathbf{u}_2^T \mathbf{u}_2 & \mathbf{u}_2^T \mathbf{u}_3 \\ \mathbf{u}_3^T \mathbf{u}_1 & \mathbf{u}_3^T \mathbf{u}_2 & \mathbf{u}_3^T \mathbf{u}_3 \end{bmatrix}.$$

Thus,  $U$  has orthonormal columns  $\Leftrightarrow U^T U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ . The proof of the general case is essentially the same.  $\square$

**Theorem 9.24.** Let  $U$  be an  $m \times n$  matrix with orthonormal columns, and let  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ . Then

- (a)  $\|U\mathbf{x}\| = \|\mathbf{x}\|$  (length preservation)  
 (b)  $(U\mathbf{x}) \bullet (U\mathbf{y}) = \mathbf{x} \bullet \mathbf{y}$  (dot product preservation)  
 (c)  $(U\mathbf{x}) \bullet (U\mathbf{y}) = 0 \Leftrightarrow \mathbf{x} \bullet \mathbf{y} = 0$  (orthogonality preservation)

**Proof.**

Theorems 9.23 and 9.24 are particularly useful when applied to square matrices.

**Definition 9.25.** An **orthogonal matrix** is a square matrix  $U$  such that  $U^T = U^{-1}$ , i.e.,

$$U \in \mathbb{R}^{n \times n} \quad \text{and} \quad U^T U = I. \quad (9.7)$$

Let's generate a **random orthogonal matrix** and test it.

```

1  orthogonal_matrix.m
2  n = 4;
3  [Q,~] = qr(rand(n));
4  U = Q;
5
6  disp("U =");    disp(U)
7  disp("U'*U ="); disp(U'*U)
8
9  x = rand([n,1]);
10 fprintf("\nx' ="); disp(x')
11 fprintf("||x||_2 ="); disp(norm(x,2))
12 fprintf("||U*x||_2="); disp(norm(U*x,2))

```

```

1  Output
2  U =
3  -0.5332    0.4892    0.6519    0.2267
4  -0.5928   -0.7162    0.1668   -0.3284
5  -0.0831    0.4507   -0.0991   -0.8833
6  -0.5978    0.2112   -0.7331    0.2462
7  U'*U =
8  1.0000   -0.0000         0   -0.0000
9  -0.0000    1.0000    0.0000    0.0000
10         0    0.0000    1.0000   -0.0000
11 -0.0000    0.0000   -0.0000    1.0000
12 x' =    0.4218    0.9157    0.7922    0.9595
13 ||x||_2 =    1.6015
14 ||U*x||_2=    1.6015

```

## 9.3. Orthogonal Projections

**Definition 9.26.** Let  $W$  be a subspace of  $\mathbb{R}^n$ .

- A vector  $\mathbf{z} \in \mathbb{R}^n$  is said to be **orthogonal** to the subspace  $W$  if  $\mathbf{z} \bullet \mathbf{w} = 0$  for all  $\mathbf{w} \in W$ .
- The set of all vectors  $\mathbf{z}$  that are orthogonal to  $W$  is called the **orthogonal complement** of  $W$  and is denoted by  $W^\perp$  (and read as “ $W$  perpendicular” or simply “ $W$  perp”). That is,

$$W^\perp = \{\mathbf{z} \mid \mathbf{z} \bullet \mathbf{w} = 0, \forall \mathbf{w} \in W\}. \quad (9.8)$$

**Example 9.27.** Let  $W$  be a plane through the origin in  $\mathbb{R}^3$ , and let  $L$  be the line through the origin and perpendicular to  $W$ .

If  $\mathbf{z} \in L$  and  $\mathbf{w} \in W$ , then

$$\mathbf{z} \bullet \mathbf{w} = 0.$$

See Figure 9.4.

In fact,  $L$  consists of all vectors that are orthogonal to the  $\mathbf{w}$ 's in  $W$ , and  $W$  consists of all vectors orthogonal to the  $\mathbf{z}$ 's in  $L$ . That is,

$$L = W^\perp \quad \text{and} \quad W = L^\perp.$$

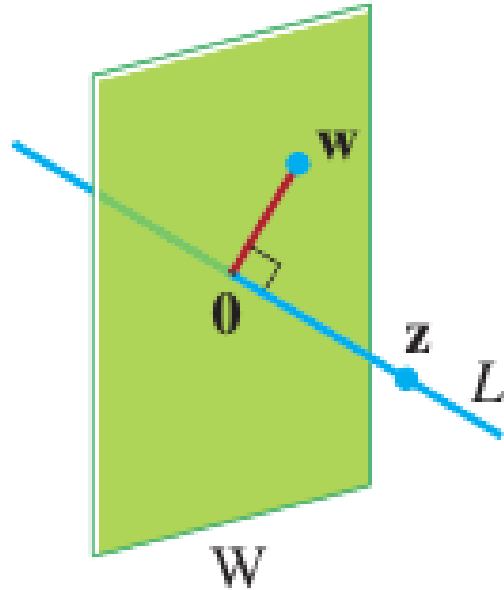


Figure 9.4: A plane and line through the origin as orthogonal complements.

**Remark 9.28.** Let  $W$  be a subspace of  $\mathbb{R}^n$ .

1. A vector  $\mathbf{x}$  is in  $W^\perp \Leftrightarrow \mathbf{x}$  is orthogonal to every vector in a set that spans  $W$ .
2.  $W^\perp$  is a subspace of  $\mathbb{R}^n$ .

**Recall: (Definition 9.19, § 9.2)** Given a nonzero vector  $\mathbf{u}$  in  $\mathbb{R}^n$ , for  $\mathbf{y} \in \mathbb{R}^n$ , let

$$\mathbf{y} = \hat{\mathbf{y}} + \mathbf{z}, \quad \hat{\mathbf{y}} // \mathbf{u} \text{ and } \mathbf{z} \perp \mathbf{u}. \quad (9.9)$$

Then

$$\hat{\mathbf{y}} = \alpha \mathbf{u} = \frac{\mathbf{y} \cdot \mathbf{u}}{\mathbf{u} \cdot \mathbf{u}} \mathbf{u}, \quad \mathbf{z} = \mathbf{y} - \hat{\mathbf{y}}. \quad (9.10)$$

The vector  $\hat{\mathbf{y}}$  is called the *orthogonal projection* of  $\mathbf{y}$  onto  $\mathbf{u}$ , and  $\mathbf{z}$  is called the component of  $\mathbf{y}$  orthogonal to  $\mathbf{u}$ . Let  $L = \text{Span}\{\mathbf{u}\}$ . Then we denote

$$\hat{\mathbf{y}} = \frac{\mathbf{y} \cdot \mathbf{u}}{\mathbf{u} \cdot \mathbf{u}} \mathbf{u} = \text{proj}_L \mathbf{y}, \quad (9.11)$$

which is called the orthogonal projection of  $\mathbf{y}$  onto  $L$ .

We generalize this orthogonal projection to subspaces.

**Theorem 9.29. (The Orthogonal Decomposition Theorem)** Let  $W$  be a subspace of  $\mathbb{R}^n$ . Then each  $\mathbf{y} \in \mathbb{R}^n$  can be written *uniquely* in the form

$$\mathbf{y} = \hat{\mathbf{y}} + \mathbf{z}, \quad (9.12)$$

where  $\hat{\mathbf{y}} \in W$  and  $\mathbf{z} \in W^\perp$ . In fact, if  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\}$  is an orthogonal basis for  $W$ , then

$$\begin{aligned} \hat{\mathbf{y}} &= \text{proj}_W \mathbf{y} = \frac{\mathbf{y} \cdot \mathbf{u}_1}{\mathbf{u}_1 \cdot \mathbf{u}_1} \mathbf{u}_1 + \frac{\mathbf{y} \cdot \mathbf{u}_2}{\mathbf{u}_2 \cdot \mathbf{u}_2} \mathbf{u}_2 + \dots + \frac{\mathbf{y} \cdot \mathbf{u}_p}{\mathbf{u}_p \cdot \mathbf{u}_p} \mathbf{u}_p, \\ \mathbf{z} &= \mathbf{y} - \hat{\mathbf{y}}. \end{aligned} \quad (9.13)$$

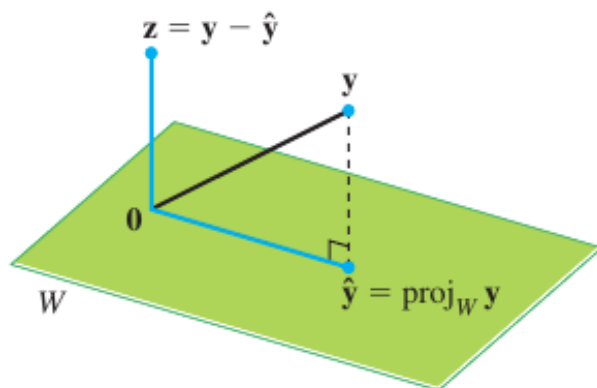


Figure 9.5: Orthogonal projection of  $\mathbf{y}$  onto  $W$ .

**Example 9.30.** Let  $\mathbf{u}_1 = \begin{bmatrix} 2 \\ 5 \\ -1 \end{bmatrix}$ ,  $\mathbf{u}_2 = \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}$ , and  $\mathbf{y} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ . Observe that  $\{\mathbf{u}_1, \mathbf{u}_2\}$  is an orthogonal basis for  $W = \text{Span}\{\mathbf{u}_1, \mathbf{u}_2\}$ .

- (a) Write  $\mathbf{y}$  as the sum of a vector in  $W$  and a vector orthogonal to  $W$ .  
 (b) Find the distance from  $\mathbf{y}$  to  $W$ .

**Solution.**  $\mathbf{y} = \hat{\mathbf{y}} + \mathbf{z} \Rightarrow \hat{\mathbf{y}} = \frac{\mathbf{y} \cdot \mathbf{u}_1}{\mathbf{u}_1 \cdot \mathbf{u}_1} \mathbf{u}_1 + \frac{\mathbf{y} \cdot \mathbf{u}_2}{\mathbf{u}_2 \cdot \mathbf{u}_2} \mathbf{u}_2$  and  $\mathbf{z} = \mathbf{y} - \hat{\mathbf{y}}$ .

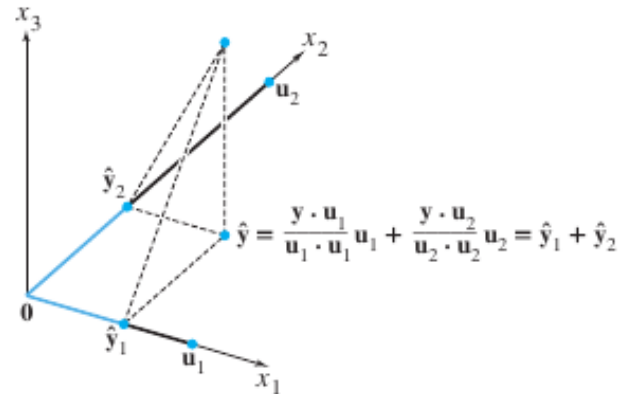


Figure 9.6: A geometric interpretation of the orthogonal projection.

**Remark 9.31. (Properties of Orthogonal Decomposition)**

Let  $\mathbf{y} = \hat{\mathbf{y}} + \mathbf{z}$ , where  $\hat{\mathbf{y}} \in W$  and  $\mathbf{z} \in W^\perp$ . Then

1.  $\hat{\mathbf{y}}$  is called the **orthogonal projection** of  $\mathbf{y}$  onto  $W$  ( $= \text{proj}_W \mathbf{y}$ )
2.  $\hat{\mathbf{y}}$  is the **closest point** to  $\mathbf{y}$  in  $W$ .  
(in the sense  $\|\mathbf{y} - \hat{\mathbf{y}}\| \leq \|\mathbf{y} - \mathbf{v}\|$ , for all  $\mathbf{v} \in W$ )
3.  $\hat{\mathbf{y}}$  is called the **best approximation** to  $\mathbf{y}$  by elements of  $W$ .
4. If  $\mathbf{y} \in W$ , then  $\text{proj}_W \mathbf{y} = \mathbf{y}$ .

**Proof.** 2. For an arbitrary  $\mathbf{v} \in W$ ,  $\mathbf{y} - \mathbf{v} = (\mathbf{y} - \hat{\mathbf{y}}) + (\hat{\mathbf{y}} - \mathbf{v})$ , where  $(\hat{\mathbf{y}} - \mathbf{v}) \in W$ . Thus, by the *Pythagorean theorem*,

$$\|\mathbf{y} - \mathbf{v}\|^2 = \|\mathbf{y} - \hat{\mathbf{y}}\|^2 + \|\hat{\mathbf{y}} - \mathbf{v}\|^2,$$

which implies that  $\|\mathbf{y} - \mathbf{v}\| \geq \|\mathbf{y} - \hat{\mathbf{y}}\|$ .  $\square$

**Self-study 9.32.** Find the closest point to  $\mathbf{y}$  in the subspace  $\text{Span}\{\mathbf{u}_1, \mathbf{u}_2\}$  and hence find the distance from  $\mathbf{y}$  to  $W$ .

$$\mathbf{y} = \begin{bmatrix} 3 \\ -1 \\ 1 \\ 13 \end{bmatrix}, \quad \mathbf{u}_1 = \begin{bmatrix} 1 \\ -2 \\ -1 \\ 2 \end{bmatrix}, \quad \mathbf{u}_2 = \begin{bmatrix} -4 \\ 1 \\ 0 \\ 3 \end{bmatrix}$$

**Solution.**



**Theorem 9.33.** If  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\}$  is an *orthonormal basis* for a subspace  $W$  of  $\mathbb{R}^n$ , then

$$\text{proj}_W \mathbf{y} = (\mathbf{y} \bullet \mathbf{u}_1) \mathbf{u}_1 + (\mathbf{y} \bullet \mathbf{u}_2) \mathbf{u}_2 + \dots + (\mathbf{y} \bullet \mathbf{u}_p) \mathbf{u}_p. \quad (9.14)$$

If  $U = [\mathbf{u}_1 \ \mathbf{u}_2 \ \dots \ \mathbf{u}_p]$ , then

$$\text{proj}_W \mathbf{y} = UU^T \mathbf{y}, \quad \text{for all } \mathbf{y} \in \mathbb{R}^n. \quad (9.15)$$

Thus the orthogonal projection can be viewed as a **matrix transformation**.

**Proof.** Notice that

$$\begin{aligned} & (\mathbf{y} \bullet \mathbf{u}_1) \mathbf{u}_1 + (\mathbf{y} \bullet \mathbf{u}_2) \mathbf{u}_2 + \dots + (\mathbf{y} \bullet \mathbf{u}_p) \mathbf{u}_p \\ &= (\mathbf{u}_1^T \mathbf{y}) \mathbf{u}_1 + (\mathbf{u}_2^T \mathbf{y}) \mathbf{u}_2 + \dots + (\mathbf{u}_p^T \mathbf{y}) \mathbf{u}_p \\ &= U(U^T \mathbf{y}). \end{aligned}$$

**Example 9.34.** Let  $\mathbf{y} = \begin{bmatrix} 7 \\ 9 \end{bmatrix}$ ,  $\mathbf{u}_1 = \begin{bmatrix} 1/\sqrt{10} \\ -3/\sqrt{10} \end{bmatrix}$ , and  $W = \text{Span}\{\mathbf{u}_1\}$ .

- (a) Let  $U$  be the  $2 \times 1$  matrix whose only column is  $\mathbf{u}_1$ . Compute  $U^T U$  and  $UU^T$ .
- (b) Compute  $\text{proj}_W \mathbf{y} = (\mathbf{y} \bullet \mathbf{u}_1) \mathbf{u}_1$  and  $UU^T \mathbf{y}$ .

**Solution.**

$$\text{Ans: (a) } UU^T = \frac{1}{10} \begin{bmatrix} 1 & -3 \\ -3 & 9 \end{bmatrix} \quad \text{(b) } \begin{bmatrix} -2 \\ 6 \end{bmatrix}$$

## 9.4. The Gram-Schmidt Process and QR Factorization

**Note:** The **Gram-Schmidt process** is an algorithm to produce an **orthogonal or orthonormal basis** for any nonzero subspace of  $\mathbb{R}^n$ .

**Example 9.35.** Let  $W = \text{Span}\{\mathbf{x}_1, \mathbf{x}_2\}$ , where  $\mathbf{x}_1 = \begin{bmatrix} 3 \\ 6 \\ 0 \end{bmatrix}$  and  $\mathbf{x}_2 = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$ . Find

an orthogonal basis for  $W$ .

**Main idea: Orthogonal projection**

$$\begin{cases} \mathbf{x}_1 \\ \mathbf{x}_2 \end{cases} \Rightarrow \begin{cases} \mathbf{x}_1 \\ \mathbf{x}_2 = \alpha\mathbf{x}_1 + \mathbf{v}_2 \end{cases} \Rightarrow \begin{cases} \mathbf{v}_1 = \mathbf{x}_1 \\ \mathbf{v}_2 = \mathbf{x}_2 - \alpha\mathbf{x}_1 \end{cases}$$

where  $\mathbf{x}_1 \bullet \mathbf{v}_2 = 0$ . Then  $W = \text{Span}\{\mathbf{x}_1, \mathbf{x}_2\} = \text{Span}\{\mathbf{v}_1, \mathbf{v}_2\}$ .

**Solution.**

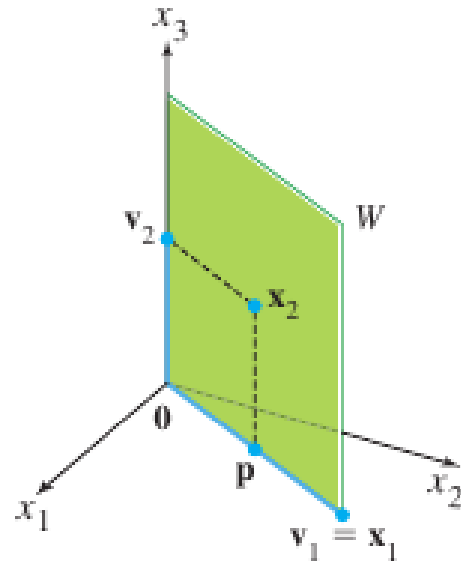


Figure 9.7: Construction of an orthogonal basis  $\{\mathbf{v}_1, \mathbf{v}_2\}$ .

**Theorem 9.36. (The Gram-Schmidt Process)** Given a basis  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_p\}$  for a nonzero subspace  $W$  of  $\mathbb{R}^n$ , define

$$\begin{aligned} \mathbf{v}_1 &= \mathbf{x}_1 \\ \mathbf{v}_2 &= \mathbf{x}_2 - \frac{\mathbf{x}_2 \bullet \mathbf{v}_1}{\mathbf{v}_1 \bullet \mathbf{v}_1} \mathbf{v}_1 \\ \mathbf{v}_3 &= \mathbf{x}_3 - \frac{\mathbf{x}_3 \bullet \mathbf{v}_1}{\mathbf{v}_1 \bullet \mathbf{v}_1} \mathbf{v}_1 - \frac{\mathbf{x}_3 \bullet \mathbf{v}_2}{\mathbf{v}_2 \bullet \mathbf{v}_2} \mathbf{v}_2 \\ &\vdots \\ \mathbf{v}_p &= \mathbf{x}_p - \frac{\mathbf{x}_p \bullet \mathbf{v}_1}{\mathbf{v}_1 \bullet \mathbf{v}_1} \mathbf{v}_1 - \frac{\mathbf{x}_p \bullet \mathbf{v}_2}{\mathbf{v}_2 \bullet \mathbf{v}_2} \mathbf{v}_2 - \dots - \frac{\mathbf{x}_p \bullet \mathbf{v}_{p-1}}{\mathbf{v}_{p-1} \bullet \mathbf{v}_{p-1}} \mathbf{v}_{p-1} \end{aligned} \quad (9.16)$$

Then  $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_p\}$  is an **orthogonal basis** for  $W$ . In addition,

$$\text{Span}\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\} = \text{Span}\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}, \quad \text{for } 1 \leq k \leq p. \quad (9.17)$$

**Remark 9.37.** For the result of the Gram-Schmidt process, define

$$\mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}, \quad \text{for } 1 \leq k \leq p. \quad (9.18)$$

Then  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\}$  is an **orthonormal basis** for  $W$ . In practice, it is often implemented with the **normalized Gram-Schmidt process**.

**Example 9.38.** Find an *orthonormal basis* for  $W = \text{Span}\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$ , where

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 1 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} -2 \\ 2 \\ 1 \\ 0 \end{bmatrix}, \quad \text{and } \mathbf{x}_3 = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1 \end{bmatrix}.$$

**Solution.**

### QR Factorization of Matrices

**Theorem 9.39. (The QR Factorization)** *If  $A$  is an  $m \times n$  matrix with linearly independent columns, then  $A$  can be factored as*

$$A = QR, \quad (9.19)$$

where

- $Q$  is an  $m \times n$  matrix whose columns are orthonormal.
- $R$  is an  $n \times n$  upper triangular invertible matrix with positive entries on its diagonal.

**Proof.** The columns of  $A$  form a basis  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$  for  $W = \text{Col } A$ .

1. Construct an **orthonormal basis**  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n\}$  for  $W$  (the **Gram-Schmidt process**). Set

$$Q \stackrel{\text{def}}{=} [\mathbf{u}_1 \ \mathbf{u}_2 \ \cdots \ \mathbf{u}_n]. \quad (9.20)$$

2. (**Expression**) Since  $\text{Span}\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\} = \text{Span}\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k\}$ ,  $1 \leq k \leq n$ , there are constants  $r_{1k}, r_{2k}, \dots, r_{kk}$  such that

$$\mathbf{x}_k = r_{1k}\mathbf{u}_1 + r_{2k}\mathbf{u}_2 + \cdots + r_{kk}\mathbf{u}_k + 0 \cdot \mathbf{u}_{k+1} + \cdots + 0 \cdot \mathbf{u}_n. \quad (9.21)$$

We may assume that  $r_{kk} > 0$ . (If  $r_{kk} < 0$ , multiply both  $r_{kk}$  and  $\mathbf{u}_k$  by  $-1$ .)

3. Let  $\mathbf{r}_k = [r_{1k}, r_{2k}, \dots, r_{kk}, 0, \dots, 0]^T$ . Then

$$\mathbf{x}_k = Q\mathbf{r}_k \quad (9.22)$$

4. Define

$$R \stackrel{\text{def}}{=} [\mathbf{r}_1 \ \mathbf{r}_2 \ \cdots \ \mathbf{r}_n]. \quad (9.23)$$

Then we see  $A = [\mathbf{x}_1 \ \mathbf{x}_2 \ \cdots \ \mathbf{x}_n] = [Q\mathbf{r}_1 \ Q\mathbf{r}_2 \ \cdots \ Q\mathbf{r}_n] = QR$ .  $\square$

The QR Factorization is summarized as follows.

**Algorithm 9.40. (QR Factorization)** Let  $A = [\mathbf{x}_1 \ \mathbf{x}_2 \ \cdots \ \mathbf{x}_n]$ .

- Apply the Gram-Schmidt process to obtain an orthonormal basis  $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n\}$ .
- Then, as in (9.21),

$$\begin{aligned} \mathbf{x}_1 &= (\mathbf{u}_1 \bullet \mathbf{x}_1) \mathbf{u}_1 \\ \mathbf{x}_2 &= (\mathbf{u}_1 \bullet \mathbf{x}_2) \mathbf{u}_1 + (\mathbf{u}_2 \bullet \mathbf{x}_2) \mathbf{u}_2 \\ \mathbf{x}_3 &= (\mathbf{u}_1 \bullet \mathbf{x}_3) \mathbf{u}_1 + (\mathbf{u}_2 \bullet \mathbf{x}_3) \mathbf{u}_2 + (\mathbf{u}_3 \bullet \mathbf{x}_3) \mathbf{u}_3 \\ &\vdots \\ \mathbf{x}_n &= \sum_{j=1}^n (\mathbf{u}_j \bullet \mathbf{x}_n) \mathbf{u}_j. \end{aligned} \quad (9.24)$$

- Thus

$$A = [\mathbf{x}_1 \ \mathbf{x}_2 \ \cdots \ \mathbf{x}_n] = QR \quad (9.25)$$

implies that

$$\begin{aligned} Q &= [\mathbf{u}_1 \ \mathbf{u}_2 \ \cdots \ \mathbf{u}_n], \\ R &= \begin{bmatrix} \mathbf{u}_1 \bullet \mathbf{x}_1 & \mathbf{u}_1 \bullet \mathbf{x}_2 & \mathbf{u}_1 \bullet \mathbf{x}_3 & \cdots & \mathbf{u}_1 \bullet \mathbf{x}_n \\ 0 & \mathbf{u}_2 \bullet \mathbf{x}_2 & \mathbf{u}_2 \bullet \mathbf{x}_3 & \cdots & \mathbf{u}_2 \bullet \mathbf{x}_n \\ 0 & 0 & \mathbf{u}_3 \bullet \mathbf{x}_3 & \cdots & \mathbf{u}_3 \bullet \mathbf{x}_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \mathbf{u}_n \bullet \mathbf{x}_n \end{bmatrix} = Q^T A. \end{aligned} \quad (9.26)$$

- In practice, the coefficients  $r_{ij} = \mathbf{u}_i \bullet \mathbf{x}_j$ ,  $i < j$ , can be saved during the (normalized) Gram-Schmidt process.

**Example 9.41.** Find the QR factorization for  $A = \begin{bmatrix} 4 & -1 \\ 3 & 2 \end{bmatrix}$ .

**Solution.**

$$\text{Ans: } Q = \begin{bmatrix} 0.8 & -0.6 \\ 0.6 & 0.8 \end{bmatrix} \quad R = \begin{bmatrix} 5 & 0.4 \\ 0 & 2.2 \end{bmatrix}$$

### Alternative Calculations of Least-Squares Solutions

**Recall: (Theorem 7.5, p.197)** Let  $A \in \mathbb{R}^{m \times n}$ ,  $m \geq n$  and  $\text{rank}(A) = n$ . Then the equation  $Ax = b$  has a **unique** LS solution for each  $b \in \mathbb{R}^m$ :

$$A^T Ax = A^T b \Rightarrow \hat{x} = (A^T A)^{-1} A^T b,$$

which is the **Method of Normal Equations**. The matrix

$$A^+ := (A^T A)^{-1} A^T \tag{9.27}$$

is called the **pseudoinverse** of  $A$ .

**Theorem 9.42.** Given an  $m \times n$  matrix  $A$  with linearly independent columns, let  $A = QR$  be a **QR factorization** of  $A$ , as in Algorithm 9.40. Then, for each  $b \in \mathbb{R}^m$ , the equation  $Ax = b$  has a unique LS solution, given by

$$\hat{x} = R^{-1} Q^T b. \tag{9.28}$$

**Proof.** Let  $A = QR$ . Then the pseudoinverse of  $A$ :

$$\begin{aligned} (A^T A)^{-1} A^T &= ((QR)^T QR)^{-1} (QR)^T = (R^T Q^T QR)^{-1} R^T Q^T \\ &= R^{-1} (R^T)^{-1} R^T Q^T = R^{-1} Q^T, \end{aligned} \tag{9.29}$$

which completes the proof. □

**Self-study 9.43.** Find the LS solution of  $Ax = b$  for

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 1 & 1 & 0 \\ 1 & 1 & 2 \\ 1 & 3 & 3 \end{bmatrix} \text{ and } b = \begin{bmatrix} 3 \\ 5 \\ 7 \\ -3 \end{bmatrix}, \text{ where } A = QR = \begin{bmatrix} 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 \\ 1/2 & -1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 \end{bmatrix} \begin{bmatrix} 2 & 4 & 5 \\ 0 & 2 & 3 \\ 0 & 0 & 2 \end{bmatrix}$$

**Solution.**

$$\text{Ans: } Q^T b = (6, -6, 4) \text{ and } \hat{x} = (10, -6, 2)$$

## 9.5. QR Iteration for Finding Eigenvalues

**Algorithm 9.44. (QR Iteration)** Let  $A \in \mathbb{R}^{n \times n}$ .

**set**  $A_0 = A$  and  $U_0 = I$

**for**  $k = 1, 2, \dots$  **do**

(a)  $A_{k-1} = Q_k R_k$ ; % QR factorization

(b)  $A_k = R_k Q_k$ ; (9.30)

(c)  $U_k = U_{k-1} Q_k$ ; % Update transformation matrix

**end for**

**set**  $T := A_\infty$  and  $U := U_\infty$

**Remark 9.45.** It follows from (a) and (b) of Algorithm 9.44 that

$$A_k = R_k Q_k = Q_k^T A_{k-1} Q_k, \quad (9.31)$$

and therefore

$$\begin{aligned} A_k &= R_k Q_k = Q_k^T A_{k-1} Q_k = Q_k^T Q_{k-1}^T A_{k-2} Q_{k-1} Q_k = \dots \\ &= Q_k^T Q_{k-1}^T \dots Q_1^T A_0 \underbrace{Q_1 Q_2 \dots Q_k}_{U_k} \end{aligned} \quad (9.32)$$

The above converges to

$$T = U^T A U \quad (9.33)$$

**Claim 9.46.**

- Algorithm 9.44 produces an **upper triangular matrix**  $T$ , with its diagonals being **eigenvalues** of  $A$ , and an orthogonal matrix  $U$  such that

$$A = U T U^T, \quad (9.34)$$

which is called the **Schur decomposition** of  $A$ .

- If  $A$  is symmetric**, then  $T$  becomes a diagonal matrix of **eigenvalues** of  $A$  and  $U$  is the collection of corresponding **eigenvectors**.

**Example 9.47.** Let  $A = \begin{bmatrix} 3 & 1 & 3 \\ 1 & 6 & 4 \\ 6 & 7 & 8 \end{bmatrix}$  and  $B = \begin{bmatrix} 4 & -1 & 1 \\ -1 & 3 & -2 \\ 1 & -2 & 3 \end{bmatrix}$ . Apply the QR algorithm, Algorithm 9.44, to find their Schur decompositions.

**Solution.** You will solve this example once more implementing the QR iteration algorithm in **Python**; see Exercise 9.7.

```

                                qr_iteration.m
1  function [T,U,iter] = qr_iteration(A)
2  % It produces the Schur decomposition: A = U*T*U^T
3  %   T: upper triangular, with diagonals being eigenvalues of A
4  %   U: orthogonal
5  % Once A is symmetric,
6  %   T becomes diagonal && U contains eigenvectors of A
7
8  T = A; U = eye(size(A));
9
10 % for stopping
11 D0 = diag(T); change = 1;
12 tol = 10^-15; iter=0;
13
14 %%-----
15 while change>tol
16     [Q,R] = qr(T);
17     T = R*Q;
18     U = U*Q;
19
20     % for stopping
21     iter= iter+1;
22     D=diag(T); change=norm(D-D0); D0=D;
23     %if iter<=8, fprintf('A_%d =\n',iter); disp(T); end
24 end

```



We may call it as

```

_____ call_qr_iteration.m _____
1  A =[3 1 3; 1 6 4; 6 7 8];
2  [T1,U1,iter1] = qr_iteration(A)
3  U1*T1*U1'
4  [V1,D1] = eig(A)
5
6  B =[4 -1 1; -1 3 -2; 1 -2 3];
7  [T2,U2,iter2] = qr_iteration(B)
8  U2*T2*U2'
9  [V2,D2] = eig(B)

```

```

_____ A _____
1  T1 =
2     13.8343    1.0429   -4.0732
3     0.0000    3.3996    0.5668
4     0.0000   -0.0000   -0.2339
5  U1 =
6     0.2759   -0.5783   -0.7677
7     0.4648    0.7794   -0.4201
8     0.8414   -0.2409    0.4838
9  iter1 =
10     26
11 ans =
12     3.0000    1.0000    3.0000
13     1.0000    6.0000    4.0000
14     6.0000    7.0000    8.0000
15
16 %---- [V1,D1] = eig(A)
17 V1 =
18    -0.2759   -0.5630    0.6029
19    -0.4648   -0.3805   -0.7293
20    -0.8414    0.7337    0.3234
21 D1 =
22    13.8343         0         0
23         0   -0.2339         0
24         0         0    3.3996

```

```

_____ B _____
1  T2 =
2     6.0000   -0.0000    0.0000
3    -0.0000    3.0000   -0.0000
4     0.0000   -0.0000    1.0000
5  U2 =
6     0.5774    0.8165   -0.0000
7    -0.5774    0.4082    0.7071
8     0.5774   -0.4082    0.7071
9  iter2 =
10     28
11 ans =
12     4.0000   -1.0000    1.0000
13    -1.0000    3.0000   -2.0000
14     1.0000   -2.0000    3.0000
15
16 %---- [V2,D2] = eig(B)
17 V2 =
18    -0.0000    0.8165    0.5774
19     0.7071    0.4082   -0.5774
20     0.7071   -0.4082    0.5774
21 D2 =
22     1.0000         0         0
23         0    3.0000         0
24         0         0    6.0000

```

### Convergence Check

```

1 % A =[3 1 3; 1 6 4; 6 7 8];
2 A_1 =
3     12.0652     4.4464     4.2538
4     3.3054     5.0728     0.9038
5     0.2644     0.0192    -0.1380
6 A_2 =
7     13.6436     2.0123    -4.1057
8     0.9772     3.5927     0.1704
9     0.0050    -0.0063    -0.2362
10 A_3 =
11     13.8038     1.2893     4.0854
12     0.2459     3.4300    -0.4708
13     0.0001    -0.0005    -0.2338
14 A_4 =
15     13.8279     1.1035    -4.0765
16     0.0607     3.4060     0.5430
17     0.0000    -0.0000    -0.2339
18 A_5 =
19     13.8328     1.0578     4.0740
20     0.0149     3.4011    -0.5609
21     0.0000    -0.0000    -0.2339
22 A_6 =
23     13.8339     1.0465    -4.0734
24     0.0037     3.3999     0.5653
25     0.0000    -0.0000    -0.2339
26 A_7 =
27     13.8342     1.0438     4.0733
28     0.0009     3.3997    -0.5664
29     0.0000    -0.0000    -0.2339
30 A_8 =
31     13.8343     1.0431    -4.0732
32     0.0002     3.3996     0.5667
33     0.0000    -0.0000    -0.2339

```

```

1 % B =[4 -1 1; -1 3 -2; 1 -2 3];
2 A_1 =
3     5.0000    -1.3765    -0.3244
4    -1.3765     3.8421     0.6699
5    -0.3244     0.6699     1.1579
6 A_2 =
7     5.6667    -0.9406     0.0640
8    -0.9406     3.3226    -0.1580
9     0.0640    -0.1580     1.0108
10 A_3 =
11     5.9091    -0.5141    -0.0112
12    -0.5141     3.0899     0.0454
13    -0.0112     0.0454     1.0010
14 A_4 =
15     5.9767    -0.2631     0.0019
16    -0.2631     3.0232    -0.0145
17     0.0019    -0.0145     1.0001
18 A_5 =
19     5.9942    -0.1323    -0.0003
20    -0.1323     3.0058     0.0048
21    -0.0003     0.0048     1.0000
22 A_6 =
23     5.9985    -0.0663     0.0001
24    -0.0663     3.0015    -0.0016
25     0.0001    -0.0016     1.0000
26 A_7 =
27     5.9996    -0.0331    -0.0000
28    -0.0331     3.0004     0.0005
29    -0.0000     0.0005     1.0000
30 A_8 =
31     5.9999    -0.0166     0.0000
32    -0.0166     3.0001    -0.0002
33     0.0000    -0.0002     1.0000

```

**Exercises for Chapter 9**

9.1. Suppose  $y$  is orthogonal to  $u$  and  $v$ . Prove that  $y$  is orthogonal to every  $w$  in  $\text{Span}\{u, v\}$ .

9.2. Let  $u_1 = \begin{bmatrix} 3 \\ -3 \\ 0 \end{bmatrix}$ ,  $u_2 = \begin{bmatrix} 2 \\ 2 \\ -1 \end{bmatrix}$ ,  $u_3 = \begin{bmatrix} 1 \\ 1 \\ 4 \end{bmatrix}$ , and  $x = \begin{bmatrix} 5 \\ -3 \\ 1 \end{bmatrix}$ .

(a) Check if  $\{u_1, u_2, u_3\}$  is an orthogonal basis for  $\mathbb{R}^3$ .

(b) Express  $x$  as a linear combination of  $\{u_1, u_2, u_3\}$ .

*Ans:*  $x = \frac{4}{3}u_1 + \frac{1}{3}u_2 + \frac{1}{3}u_3$

9.3. Let  $U$  and  $V$  be  $n \times n$  orthogonal matrices. Prove that  $UV$  is an orthogonal matrix.

**Hint:** See Definition 9.25, where  $U^{-1} = U^T \Leftrightarrow U^T U = I$ .

9.4. Find the best approximation to  $z$  by vectors of the form  $c_1v_1 + c_2v_2$ .

(a)  $z = \begin{bmatrix} 3 \\ -1 \\ 1 \\ 13 \end{bmatrix}$ ,  $v_1 = \begin{bmatrix} 1 \\ -2 \\ -1 \\ 2 \end{bmatrix}$ ,  $v_2 = \begin{bmatrix} -4 \\ 1 \\ 0 \\ 3 \end{bmatrix}$       (b)  $z = \begin{bmatrix} 3 \\ -7 \\ 2 \\ 3 \end{bmatrix}$ ,  $v_1 = \begin{bmatrix} 2 \\ -1 \\ -3 \\ 1 \end{bmatrix}$ ,  $v_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ -1 \end{bmatrix}$

*Ans:* (a)  $\hat{z} = 3v_1 + v_2$

9.5. Find an orthogonal basis for the column space of the matrix

$$\begin{bmatrix} -1 & 6 & 6 \\ 3 & -8 & 3 \\ 1 & -2 & 6 \\ 1 & -4 & -3 \end{bmatrix}$$

*Ans:*  $v_3 = (1, 1, -3, 1)$

9.6. Implement a code for the problem. Let  $A =$

$$\begin{bmatrix} -10 & 13 & 7 & -11 \\ 2 & 1 & -5 & 3 \\ -6 & 3 & 13 & -3 \\ 16 & -16 & -2 & 5 \\ 2 & 1 & -5 & -7 \end{bmatrix}$$

(a) Use the Gram-Schmidt process to produce an orthogonal basis for the column space of  $A$ .

(b) Use Algorithm 9.40 to produce a QR factorization of  $A$ .

(c) Apply the QR iteration to find eigenvalues of  $A(1:4, 1:4)$ .

*Ans:* (a)  $v_4 = (0, 5, 0, 0, -5)$

9.7. Solve Example 9.47 by implementing the QR iteration algorithm in *Python*; you may use `qr_iteration.m`, p.254.



## CHAPTER 10

# Introduction to Machine Learning

In this chapter, you will learn:

- What machine learning (ML) is
- Popular ML classifiers
- Scikit-Learn: A Python ML library
- A machine learning modelcode

The chapter is a brief introduction to ML. I hope it would be useful.

### Contents of Chapter 10

10.1. What is Machine Learning? . . . . .	260
10.2. Binary Classifiers . . . . .	265
10.3. Popular Machine Learning Classifiers . . . . .	275
10.4. Neural Networks . . . . .	283
10.5. Scikit-Learn: A Python Machine Learning Library . . . . .	292
A Machine Learning Modelcode . . . . .	296
Exercises for Chapter 10 . . . . .	301

## 10.1. What is Machine Learning?

### The Three Tasks (T3)

Most real-world problems are expressed as

$$f(\mathbf{x}) = y, \quad (10.1)$$

where  $f$  is an **operation**,  $\mathbf{x}$  denotes the **input**, and  $y$  is the **output**.

1. Known:  $(f, \mathbf{x}) \Rightarrow y$ : simple to get
2. Known:  $(f, y) \Rightarrow \mathbf{x}$ : solve the equation (10.1) (10.2)
3. Known:  $(\mathbf{x}, y) \Rightarrow f$ : approximated; Machine Learning

### Definition 10.1. Machine Learning (ML)

- **ML algorithms** are algorithms that can learn from **data** (input) and produce **functions/models** (output).
- **Machine learning** is the science of getting machines to act, without functions/models being explicitly programmed to do so.

**Example 10.2.** There are different types of ML:

- **Supervised learning**: e.g., classification, regression  $\Leftarrow$  Labeled data
- **Unsupervised learning**: e.g., clustering  $\Leftarrow$  No labels
- **Reinforcement learning**: e.g., chess engine  $\Leftarrow$  Reward system

The most popular type is **supervised learning**.

### A Belief in Machine Learning

**The average is correct (at least, acceptable)**

- **“Guess The Weight of the Ox”** Competition
  - Questioner: **Francis Galton** – a cousin of Charles Darwin
  - A county fair, Plymouth, MA, 1907
  - The annual West of England Fat Stock and Poultry Exhibition
- 800 people gathered: **mean=1197 lbs; real=1198 lbs**

## Supervised Learning

**Assumption.** Given a data set  $\{(x_i, y_i)\}$ , where  $y_i$  are labels, there exists a relation  $f : X \rightarrow Y$ .

**Supervised learning:**

$$\begin{cases} \text{Given : A training data } \{(x_i, y_i) \mid i = 1, \dots, N\} \\ \text{Find : } \hat{f} : X \rightarrow Y, \text{ a good approximation to } f \end{cases} \quad (10.3)$$

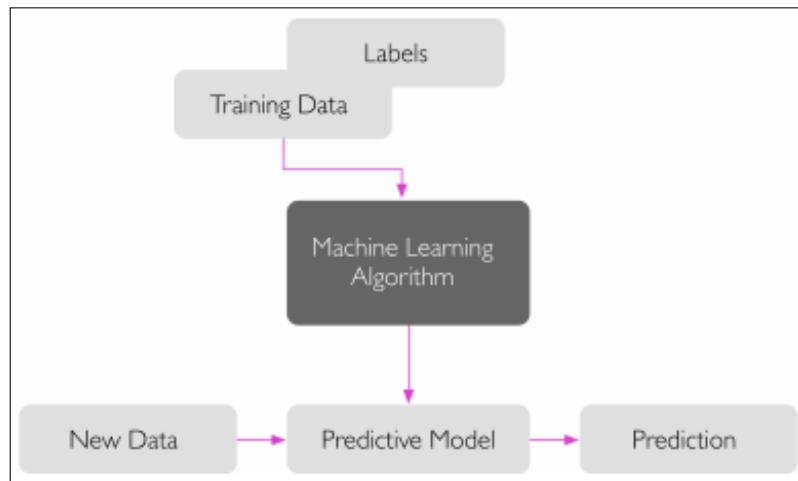


Figure 10.1: Supervised learning and prediction.

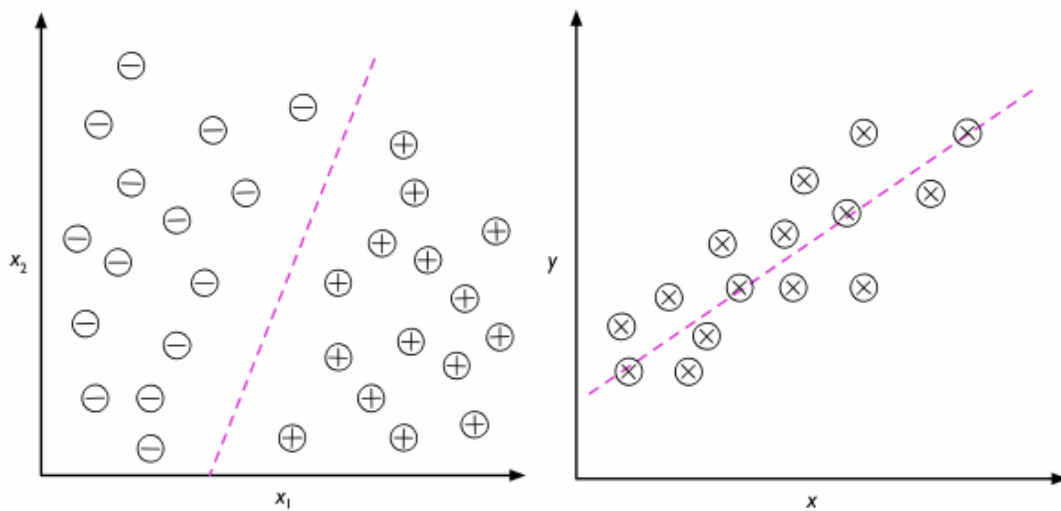


Figure 10.2: Classification and regression.

## Unsupervised Learning

### Note:

- **In supervised learning**, we know the right answer beforehand when we train our model, and **in reinforcement learning**, we define a measure of reward for particular actions by the agent.
- **In unsupervised learning**, however, we are dealing with unlabeled data or data of unknown structure. Using unsupervised learning techniques, we are able **to explore the structure of our data** to extract meaningful information, without the guidance of a known outcome variable or reward function.
- **Clustering** is an exploratory data analysis technique that allows us to organize a pile of information into **meaningful subgroups** (clusters) without having any prior knowledge of their group memberships.

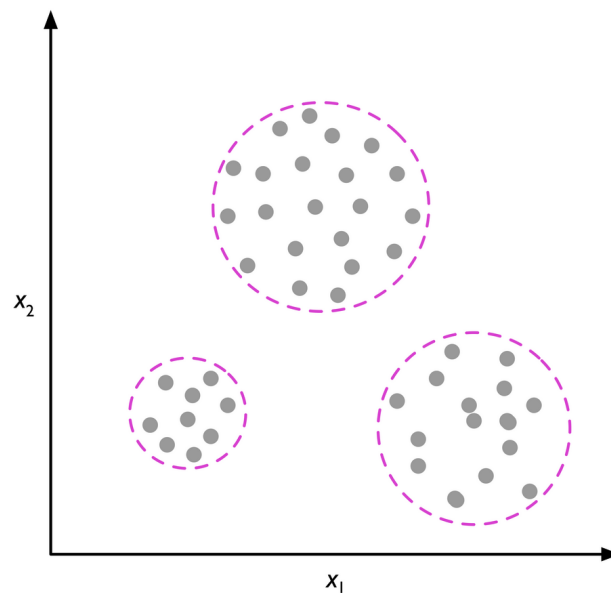


Figure 10.3: Clustering.

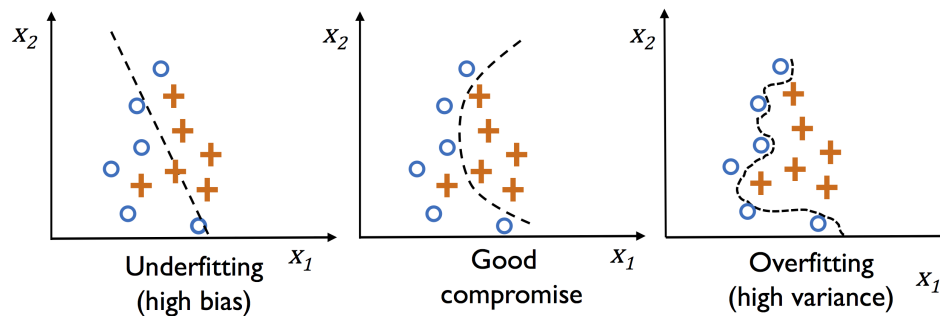


## Why is ML not Always Simple?

### Major Issues in ML

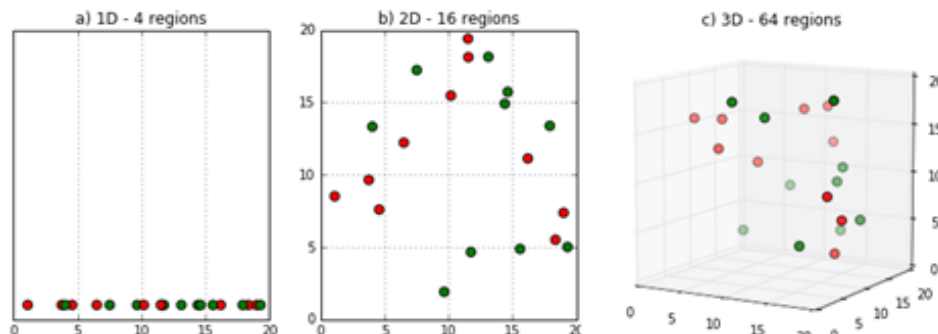
#### 1. **Overfitting:** Fitting training data *too tightly*

- **Difficulties:** Accuracy drops significantly for **test data**
- **Remedies:**
  - More training data (often, impossible)
  - Early stopping; feature selection
  - Regularization; ensembling (multiple classifiers)



#### 2. **Curse of Dimensionality:** The feature space becomes increasingly sparse for an increasing number of dimensions (of a fixed-size training dataset)

- **Difficulties:** Larger error, more computation time; Data points **appear equidistant** from all the others
- **Remedies**
  - More training data (often, impossible)
  - **Dimensionality reduction** (e.g., Feature selection, PCA)



### 3. **Multiple Local Minima Problem**

Training often involves minimizing an objective function.

- **Difficulties:** Larger error, unrepeatability
- **Remedies**
  - Gaussian sailing; regularization
  - **Careful access to the data** (e.g., mini-batch)

### 4. **Interpretability:**

Although ML has come very far, researchers still **don't know exactly how some algorithms (e.g., deep nets) work**.

- If we don't know how training nets actually work, how do we make any **real progress**?

### 5. **One-Shot Learning:**

We still haven't been able to achieve one-shot learning. *Traditional gradient-based networks need a huge amount of data*, and are often in the form of **extensive iterative training**.

- Instead, we should find a way to **enable neural networks to learn, using just a few examples**.

## 10.2. Binary Classifiers

### General Remarks

A **binary classifier** is a function which can decide whether or not an input vector belongs to a specific class (e.g., spam/ham).

- Binary classification often refers to those classification tasks that have **two class labels**. (**two-class classification**)
- It is a **type of linear classifier**, i.e. a classification algorithm that **makes its predictions based on a linear predictor function**.

**Examples:** Perceptron [10], Adaline, Logistic Regression, Support Vector Machine [4]

Binary classifiers are **artificial neurons**.

**Note:** **Neurons** are **interconnected nerve cells**, involved in the **processing and transmitting** of chemical and electrical signals. Such a nerve cell can be described as a simple logic gate with binary outputs;

- multiple signals arrive at the dendrites,
- they are integrated into the cell body,
- and if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

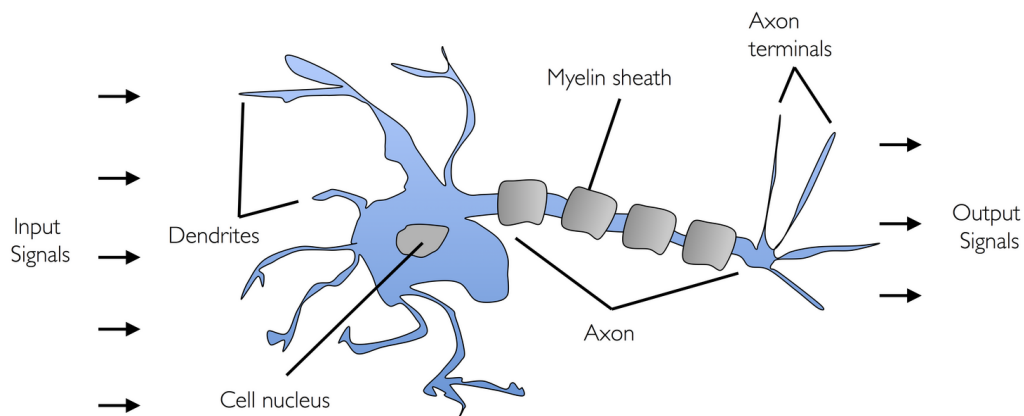


Figure 10.4: A schematic description of a neuron.

**Definition 10.3.** Let  $\{(\mathbf{x}^{(i)}, y^{(i)})\}$  be labeled data, with  $\mathbf{x}^{(i)} \in \mathbb{R}^d$  and  $y^{(i)} \in \{0, 1\}$ . A **binary classifier** finds a **hyperplane** in  $\mathbb{R}^d$  that separates data points  $X = \{\mathbf{x}^{(i)}\}$  to two classes:

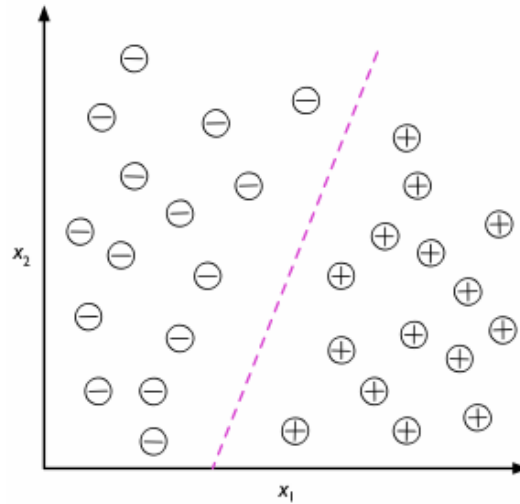


Figure 10.5: A binary classifier, finding a **hyperplane** in  $\mathbb{R}^d$ .

**Observation 10.4. Binary Classifiers**

- The **labels**  $\{0, 1\}$  are chosen for simplicity.
- A **hyperplane** can be formulated by a normal vector  $\mathbf{w} \in \mathbb{R}^d$  and a shift (bias)  $w_0$ :

$$z = \mathbf{w}^T \mathbf{x} + w_0. \quad (10.4)$$

- To **learn  $\mathbf{w}$  and  $w_0$** , you may formulate a **cost function** to minimize, as the **Sum of Squared Errors (SSE)**:

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2} \sum_i \left( y^{(i)} - \phi(\mathbf{w}^T \mathbf{x}^{(i)} + w_0) \right)^2, \quad (10.5)$$

where  $\phi(z)$  is an **activation function**.

### 10.2.1. The Perceptron Algorithm

The **perceptron** is a binary classifier of supervised learning.

- 1957: Perceptron algorithm is invented by **Frank Rosenblatt**, Cornell Aeronautical Laboratory
  - Built on work of Hebb (1949)
  - Improved by Widrow-Hoff (1960): Adaline
- 1960: Perceptron Mark 1 Computer – hardware implementation
- 1970's: Learning methods for two-layer neural networks

**Definition 10.5.** We can pose the **perceptron** as a **binary classifier**, in which we refer to our two classes as 1 (positive class) and  $-1$  (negative class) for simplicity.

- **Input values:**  $\mathbf{x} = (x_1, x_2, \dots, x_d)^T$
- **Weight vector:**  $\mathbf{w} = (w_1, w_2, \dots, w_d)^T$
- **Net input:**  $z = w_1x_1 + w_2x_2 + \dots + w_dx_d$
- **Activation function:**  $\phi(z)$ , defined by

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise,} \end{cases} \quad (10.6)$$

where  $\theta$  is a threshold.

For simplicity, we can bring the threshold  $\theta$  in (10.6) to the left side of the equation; define a weight-zero as  $w_0 = -\theta$  and reformulate as

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise,} \end{cases} \quad z = \mathbf{w}^T \mathbf{x} = w_0 + w_1x_1 + \dots + w_dx_d. \quad (10.7)$$

In the ML literature, the variable  $w_0$  is called the **bias**.

**The equation  $w_0 + w_1x_1 + \dots + w_dx_d = 0$  represents a hyperplane in  $\mathbb{R}^d$ , while  $w_0$  decides the intercept.**

### The Perceptron Learning Rule

The whole idea behind the **Rosenblatt's thresholded perceptron model** is to use a reductionist approach to mimic how a single neuron in the brain works: it either fires or it doesn't.

#### Algorithm 10.6. Rosenblatt's Initial Perceptron Rule

1. Initialize the weights to 0 or small random numbers.
2. For each training sample  $\mathbf{x}^{(i)}$ ,
  - (a) Compute the output value  $\hat{y}^{(i)}$  ( $:= \phi(\mathbf{w}^T \mathbf{x}^{(i)})$ ).
  - (b) Update the weights.

The update of the weight vector  $\mathbf{w}$  can be more formally written as:

$$\begin{aligned} \mathbf{w} &= \mathbf{w} + \Delta \mathbf{w}, & \Delta \mathbf{w} &= \eta (y^{(i)} - \hat{y}^{(i)}) \mathbf{x}^{(i)}, \\ w_0 &= w_0 + \Delta w_0, & \Delta w_0 &= \eta (y^{(i)} - \hat{y}^{(i)}), \end{aligned} \quad (10.8)$$

where  $\eta$  is the **learning rate**,  $0 < \eta < 1$ ,  $y^{(i)}$  is the true class label of the  $i$ -th training sample, and  $\hat{y}^{(i)}$  denotes the predicted class label.

#### Remark 10.7. A simple thought experiment for the perceptron learning rule:

- Let the perceptron predict the class label correctly. Then  $y^{(i)} - \hat{y}^{(i)} = 0$  so that the weights remain unchanged.
- Let the perceptron make a wrong prediction. Then

$$\Delta w_j = \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} = \pm 2 \eta x_j^{(i)}$$

so that the weight  $w_j$  is pushed towards the direction of the positive or negative target class, respectively.

**Perceptron for Iris Dataset**

```

perceptron.py
1 import numpy as np
2
3 class Perceptron():
4     def __init__(self, xdim, epoch=10, learning_rate=0.01):
5         self.epoch = epoch
6         self.learning_rate = learning_rate
7         self.weights = np.zeros(xdim + 1)
8
9     def activate(self, x):
10        net_input = np.dot(x,self.weights[1:])+self.weights[0]
11        return 1 if (net_input > 0) else 0
12
13    def fit(self, Xtrain, ytrain):
14        for k in range(self.epoch):
15            for x, y in zip(Xtrain, ytrain):
16                yhat = self.activate(x)
17                self.weights[1:] += self.learning_rate*(y-yhat)*x
18                self.weights[0] += self.learning_rate*(y-yhat)
19
20    def predict(self, Xtest):
21        yhat=[]
22        #for x in Xtest: yhat.append(self.activate(x))
23        [yhat.append(self.activate(x)) for x in Xtest]
24        return yhat
25
26    def score(self, Xtest, ytest):
27        count=0;
28        for x, y in zip(Xtest, ytest):
29            if self.activate(x)==y: count+=1
30        return count/len(ytest)
31
32    #-----
33    def fit_and_fig(self, Xtrain, ytrain):
34        wgts_all = []
35        for k in range(self.epoch):
36            for x, y in zip(Xtrain, ytrain):
37                yhat = self.activate(x)
38                self.weights[1:] += self.learning_rate*(y-yhat)*x
39                self.weights[0] += self.learning_rate*(y-yhat)
40                if k==0: wgts_all.append(list(self.weights))
41        return np.array(wgts_all)

```

```

----- Iris_perceptron.py -----
1 import numpy as np; import matplotlib.pyplot as plt
2 from sklearn.model_selection import train_test_split
3 from sklearn import datasets; #print(dir(datasets))
4 np.set_printoptions(suppress=True)
5 from perceptron import Perceptron
6
7 #-----
8 data_read = datasets.load_iris(); #print(data_read.keys())
9 X = data_read.data;
10 y = data_read.target
11 targets = data_read.target_names; features = data_read.feature_names
12
13 N,d = X.shape; nclass=len(set(y));
14 print('N,d,nclass=',N,d,nclass)
15
16 #---- Take 2 classes in 2D -----
17 X2 = X[y<=1]; y2 = y[y<=1];
18 X2 = X2[:,[0,2]]
19
20 #---- Train and Test -----
21 Xtrain, Xtest, ytrain, ytest = train_test_split(X2, y2,
22         random_state=None, train_size=0.7e0)
23 clf = Perceptron(X2.shape[1],epoch=2)
24 #clf.fit(Xtrain, ytrain);
25 wgts_all = clf.fit_and_fig(Xtrain, ytrain);
26 accuracy = clf.score(Xtest, ytest); print('accuracy =', accuracy)
27 #yhat = clf.predict(Xtest);

```

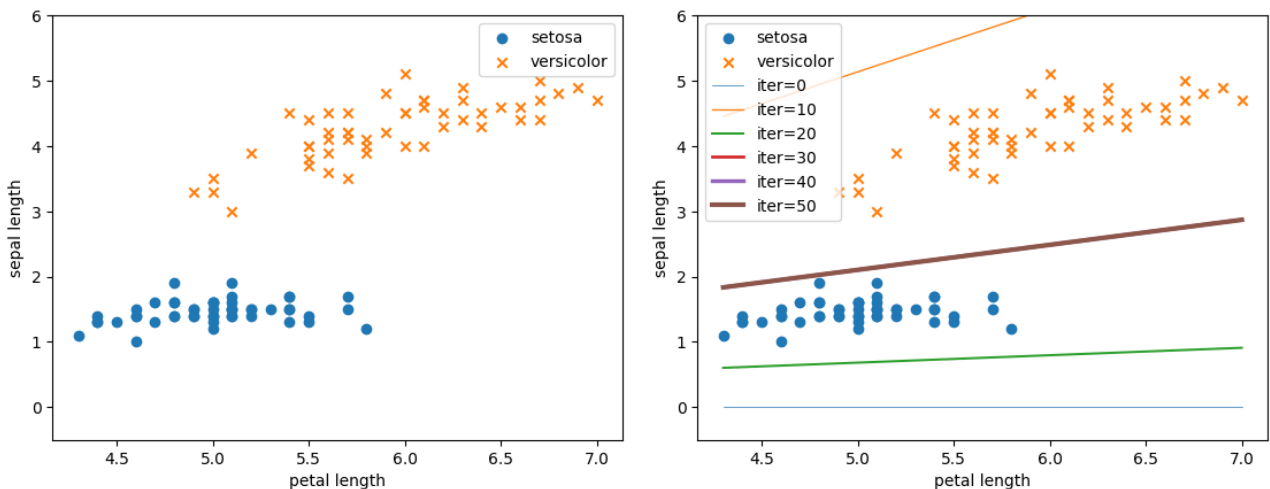


Figure 10.6: A part of Iris data (left) and the convergence of Perceptron iteration (right).



## 10.2.2. Adaline: ADaptive LInear NEuron

- (Widrow & Hoff, 1960)
- Weights are updated based on linear activation: e.g.,

$$\phi(\mathbf{w}^T \mathbf{x}) = \mathbf{w}^T \mathbf{x}.$$

That is,  $\phi$  is the **identity function**.

- Adaline algorithm is particularly interesting because it illustrates the key concept of defining and minimizing **continuous cost functions**, which will **lay the groundwork for understanding more advanced machine learning algorithms** for classification, such as logistic regression and support vector machines, as well as regression models.
- **Continuous cost functions allow the ML optimization to incorporate advanced mathematical techniques such as calculus.**

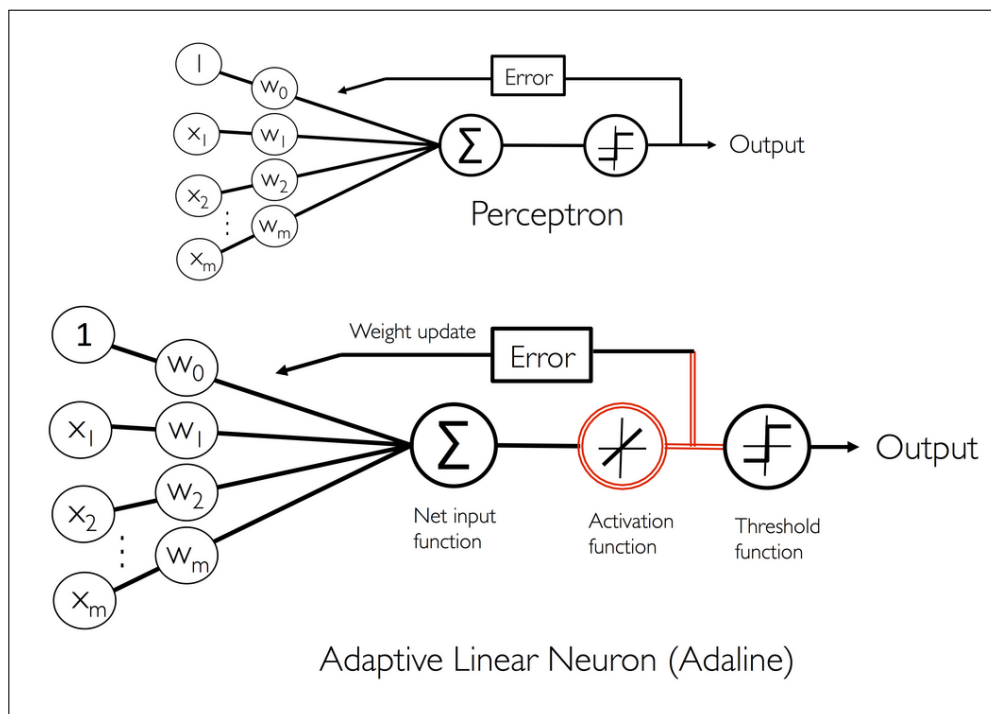


Figure 10.7: Perceptron vs. Adaline

**Algorithm 10.8. Adaline Learning:**

Given a dataset  $\{(\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, 2, \dots, N\}$ , learn the weights  $\mathbf{w}$  and bias  $b = w_0$ :

- **Activation function:**  $\phi(z) = z$  (i.e., identity activation)
- **Cost function:** the SSE

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{2} \sum_{i=1}^N \left( y^{(i)} - \phi(z^{(i)}) \right)^2, \quad (10.9)$$

where  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)} + b$  and  $\phi = I$ , the identity.

The dominant algorithm for the minimization of the cost function is the the Gradient Descent Method.

**Algorithm 10.9. The Gradient Descent Method** uses  $-\nabla \mathcal{J}$  for the **search direction** (update direction):

$$\begin{aligned} \mathbf{w} &= \mathbf{w} + \Delta \mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}, b), \\ b &= b + \Delta b = b - \eta \nabla_b \mathcal{J}(\mathbf{w}, b), \end{aligned} \quad (10.10)$$

where  $\eta > 0$  is the **step length** (learning rate).

**Computation of  $\nabla \mathcal{J}$  for Adaline:**

The partial derivatives of the cost function  $\mathcal{J}$  w.r.to  $w_j$  and  $b$  read

$$\begin{aligned} \frac{\partial \mathcal{J}(\mathbf{w}, b)}{\partial w_j} &= - \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}, \\ \frac{\partial \mathcal{J}(\mathbf{w}, b)}{\partial b} &= - \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right). \end{aligned} \quad (10.11)$$

Thus, with  $\phi = I$ ,

$$\begin{aligned} \Delta \mathbf{w} &= -\eta \nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}, b) = \eta \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}, \\ \Delta b &= -\eta \nabla_b \mathcal{J}(\mathbf{w}, b) = \eta \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right). \end{aligned} \quad (10.12)$$

**You will modify** `perceptron.py` **for Adaline**; an implementation issue is considered in Exercise 10.2, p.301.

## Hyperparameters

**Definition 10.10.** In ML, a **hyperparameter** is a parameter whose value is set before the learning process begins. Thus it is an **algorithmic parameter**. Examples are

- The learning rate ( $\eta$ )
- The number of maximum epochs/iterations ( $n\_iter$ )

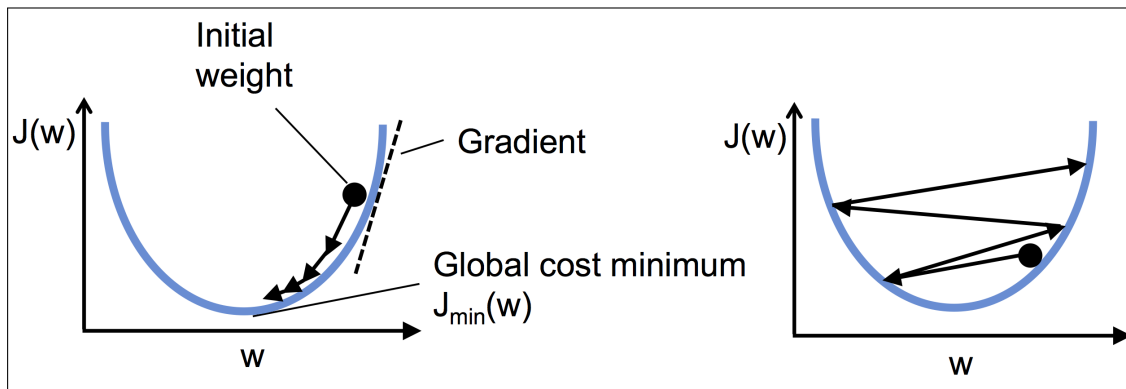


Figure 10.8: Well-chosen learning rate vs. a large learning rate

**Note:** There are effective searching schemes to set the learning rate  $\eta$  automatically.

**Multi-class Classification**

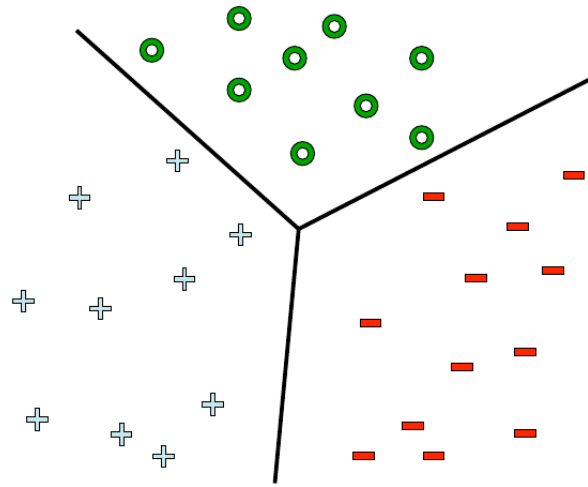


Figure 10.9: Classification for three classes.

**One-versus-all (one-versus-rest) classification**

<p>Figure 10.10: Three weights: <math>w_-</math>, <math>w_+</math>, and <math>w_0</math>.</p>	<p><b>Learning:</b> learn 3 classifiers</p> <ul style="list-style-type: none"> <li>• <math>-</math> vs <math>\{o, +\} \Rightarrow</math> weights <math>w_-</math></li> <li>• <math>+</math> vs <math>\{o, -\} \Rightarrow</math> weights <math>w_+</math></li> <li>• <math>o</math> vs <math>\{+, -\} \Rightarrow</math> weights <math>w_0</math></li> </ul> <p><b>Prediction:</b> for a new data sample <math>\mathbf{x}</math>,</p> $\hat{y} = \arg \max_{i \in \{-, +, o\}} \phi(\mathbf{w}_i^T \mathbf{x}).$
---	--

OvA (OvR) is readily applicable for classification of general  $n$  classes,  $n \geq 2$ .

## 10.3. Popular Machine Learning Classifiers

### **Remark 10.11. (The standard logistic sigmoid function)**

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x} \quad (10.13)$$

- The standard logistic function is the solution of the simple first-order non-linear ordinary differential equation

$$\frac{d}{dx}y = y(1 - y), \quad y(0) = \frac{1}{2}. \quad (10.14)$$

- It can be verified easily as

$$\sigma'(x) = \frac{e^x(1 + e^x) - e^x \cdot e^x}{(1 + e^x)^2} = \frac{e^x}{(1 + e^x)^2} = \sigma(x)(1 - \sigma(x)). \quad (10.15)$$

- $\sigma'$  is even:  $\sigma'(-x) = \sigma'(x)$ .
- **Rotational symmetry** about  $(0, 1/2)$ :

$$\sigma(x) + \sigma(-x) = \frac{1}{1 + e^{-x}} + \frac{1}{1 + e^x} = \frac{2 + e^x + e^{-x}}{2 + e^x + e^{-x}} \equiv 1. \quad (10.16)$$

- $\int \sigma(x) dx = \int \frac{e^x}{1 + e^x} dx = \ln(1 + e^x)$ , which is known as the **softplus function** in artificial neural networks. It is a smooth approximation of the the **rectifier** (an activation function) defined as

$$f(x) = x^+ = \max(x, 0). \quad (10.17)$$

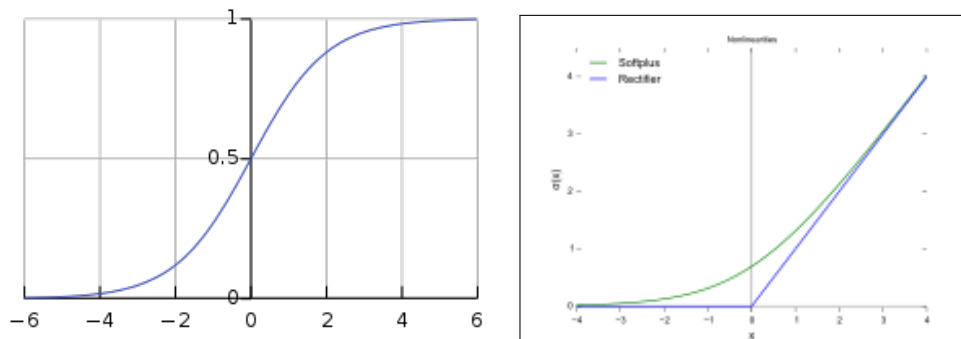


Figure 10.11: **Popular activation functions**: (left) The standard logistic sigmoid function and (right) the **rectifier** and **softplus** function.

### 10.3.1. Logistic Regression

**Logistic regression is a probabilistic model.**

- **Logistic regression maximizes the likelihood of the parameter  $w$** ; in realization, it is similar to **Adaline**.
- Only the difference is the **activation function** (the sigmoid function), as illustrated in the figure:

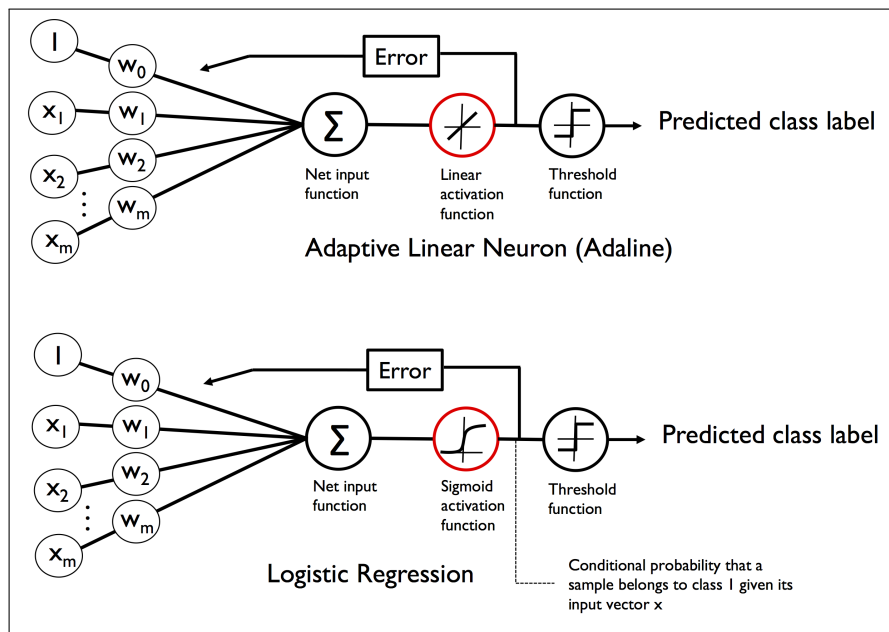


Figure 10.12: Adaline vs. Logistic regression.

- **The prediction** (the output of the sigmoid function) is interpreted as the **probability** of a particular sample belonging to class 1,

$$\phi(z) = p(y = 1 | \mathbf{x}; \mathbf{w}), \quad (10.18)$$

given its features  $x$  parameterized by the weights  $w$ ,  $z = \mathbf{w}^T \mathbf{x} + b$ .

### Derivation of the Logistic Cost Function

- Assume that the individual samples in our dataset are **independent** of one another. Then we can define the **likelihood**  $L$  as

$$\begin{aligned} L(\mathbf{w}) &= P(\mathbf{y}|\mathbf{x}; \mathbf{w}) = \prod_{i=1}^N P(y^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) \\ &= \prod_{i=1}^N \left( \phi(z^{(i)}) \right)^{y^{(i)}} \left( 1 - \phi(z^{(i)}) \right)^{1-y^{(i)}}, \end{aligned} \quad (10.19)$$

where  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)} + b$ .

- In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood function**:

$$\ell(\mathbf{w}) = \ln(L(\mathbf{w})) = \sum_{i=1}^N \left[ y^{(i)} \ln \left( \phi(z^{(i)}) \right) + (1 - y^{(i)}) \ln \left( 1 - \phi(z^{(i)}) \right) \right]. \quad (10.20)$$

#### Algorithm 10.12. Logistic Regression Learning:

From data  $\{(\mathbf{x}^{(i)}, y^{(i)})\}$ , learn the weights  $\mathbf{w}$  and bias  $b$ , with

- **Activation function:**  $\phi(z) = \sigma(z)$ , the logistic sigmoid function
- **Cost function:** The likelihood is maximized.

Based on the log-likelihood, we define the **logistic cost function** to be minimized:

$$\mathcal{J}(\mathbf{w}, b) = \sum_i \left[ -y^{(i)} \ln \left( \phi(z^{(i)}) \right) - (1 - y^{(i)}) \ln \left( 1 - \phi(z^{(i)}) \right) \right], \quad (10.21)$$

where  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)} + b$ .

### Computation of $\nabla \mathcal{J}$ for Logistic Regression:

Let's start by calculating the partial derivative of the logistic cost function with respect to the  $j$ -th weight,  $w_j$ :

$$\frac{\partial \mathcal{J}(\mathbf{w}, b)}{\partial w_j} = \sum_i \left[ -y^{(i)} \frac{1}{\phi(z^{(i)})} + (1 - y^{(i)}) \frac{1}{1 - \phi(z^{(i)})} \right] \frac{\partial \phi(z^{(i)})}{\partial w_j}, \quad (10.22)$$

where, using  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$  and (10.15),

$$\frac{\partial \phi(z^{(i)})}{\partial w_j} = \phi'(z^{(i)}) \frac{\partial z^{(i)}}{\partial w_j} = \phi(z^{(i)}) (1 - \phi(z^{(i)})) x_j^{(i)}.$$

Thus, it follows from the above and (10.22) that

$$\begin{aligned} \frac{\partial \mathcal{J}(\mathbf{w}, b)}{\partial w_j} &= \sum_i \left[ -y^{(i)} (1 - \phi(z^{(i)})) + (1 - y^{(i)}) \phi(z^{(i)}) \right] x_j^{(i)} \\ &= - \sum_i \left[ y^{(i)} - \phi(z^{(i)}) \right] x_j^{(i)} \end{aligned}$$

and therefore

$$\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}) = - \sum_i \left[ y^{(i)} - \phi(z^{(i)}) \right] \mathbf{x}^{(i)}. \quad (10.23)$$

Similarly, one can get

$$\nabla_b \mathcal{J}(\mathbf{w}) = - \sum_i \left[ y^{(i)} - \phi(z^{(i)}) \right]. \quad (10.24)$$

**Algorithm 10.13.** Gradient descent learning for Logistic Regression is formulated as

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad b := b + \Delta b, \quad (10.25)$$

where  $\eta > 0$  is the **step length** (learning rate) and

$$\begin{aligned} \Delta \mathbf{w} &= -\eta \nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}, b) = \eta \sum_i \left[ y^{(i)} - \phi(z^{(i)}) \right] \mathbf{x}^{(i)}, \\ \Delta b &= -\eta \nabla_b \mathcal{J}(\mathbf{w}, b) = \eta \sum_i \left[ y^{(i)} - \phi(z^{(i)}) \right]. \end{aligned} \quad (10.26)$$

**Note:** The above gradient descent rule for Logistic Regression is of the same form as that of Adaline; see (10.12) on p. 272. Only the difference is the activation function  $\phi$ .



### 10.3.2. Support Vector Machine

- **Support vector machine (SVM)**, developed in 1995 by Cortes-Vapnik [4], can be considered as an extension of the Perceptron/Adaline, *which maximizes the margin*.
- The **rationale** behind having decision boundaries with large margins is that they tend to have a **lower generalization error**, whereas **models with small margins are more prone to overfitting**.

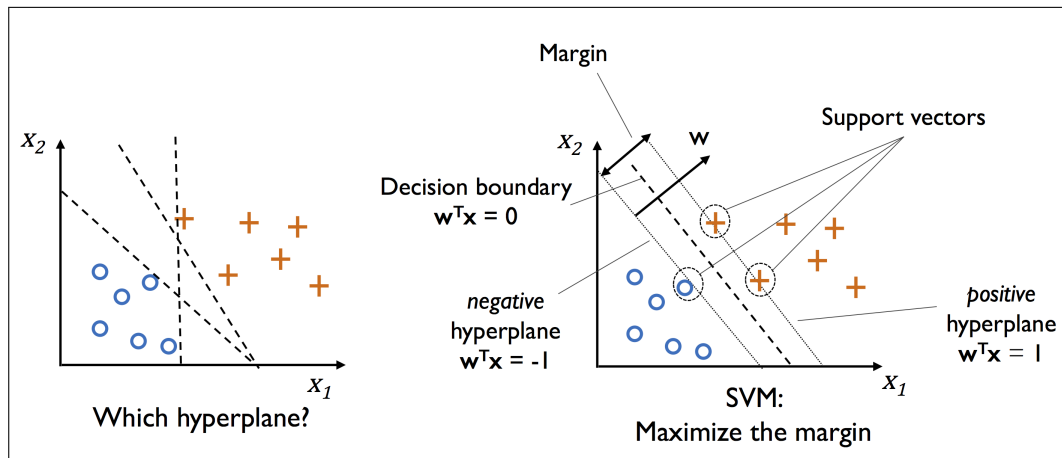


Figure 10.13: Linear support vector machine.

To find an optimal hyperplane that maximizes the margin, let's begin with considering the **positive** and **negative** hyperplanes that are parallel to the decision boundary:

$$\begin{aligned} w_0 + \mathbf{w}^T \mathbf{x}_+ &= 1, \\ w_0 + \mathbf{w}^T \mathbf{x}_- &= -1. \end{aligned} \quad (10.27)$$

where  $\mathbf{w} = [w_1, w_2, \dots, w_d]^T$ . If we subtract those two linear equations from each other, then we have

$$\mathbf{w} \cdot (\mathbf{x}_+ - \mathbf{x}_-) = 2$$

and therefore

$$\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot (\mathbf{x}_+ - \mathbf{x}_-) = \frac{2}{\|\mathbf{w}\|}. \quad (10.28)$$

**Note:**  $\mathbf{w} = [w_1, w_2, \dots, w_d]^T$  is a **normal vector** to the decision boundary (a hyperplane) so that the left side of (10.28) is the distance between the positive and negative hyperplanes.

Maximizing the distance (margin) is equivalent to minimizing its reciprocal  $\frac{1}{2}\|\mathbf{w}\|$ , or minimizing  $\frac{1}{2}\|\mathbf{w}\|^2$ .

**Problem 10.14.** The **linear SVM** is formulated as

$$\min_{\mathbf{w}, w_0} \frac{1}{2} \|\mathbf{w}\|^2, \quad \text{subject to} \quad (10.29)$$

$$\begin{cases} w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \geq 1 & \text{if } y^{(i)} = 1, \\ w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \leq -1 & \text{if } y^{(i)} = -1. \end{cases}$$

The minimization problem in (10.29) can be solved by the **method of Lagrange multipliers**; See Appendices A.1, A.2, and A.3.

**Remark 10.15.** The constraints in Problem 10.14 can be written as

$$y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 \geq 0, \quad \forall i. \quad (10.30)$$

- The beauty of linear SVM is that if the data is linearly separable, there is a unique global minimum value.
- An ideal SVM analysis should produce a hyperplane that completely separates the vectors (cases) into two non-overlapping classes.
- However, perfect separation may not be possible, or it may result in a model with so many cases that the model does not classify correctly.
- There are variations of the SVM:
  - **soft-margin classification**, for noisy data
  - **nonlinear SVMs**, kernel methods

### 10.3.3. $k$ -Nearest Neighbors

The  $k$ -nearest neighbor ( $k$ -NN) classifier is a typical example of a **lazy learner**.

- It is called *lazy* not because of its apparent simplicity, but because it **doesn't learn a discriminative function** from the training data, but memorizes the training dataset instead.
- Analysis of the training data is **delayed until a query is made** to the system.

**Algorithm 10.16.** ( $k$ -NN algorithm). The algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number  $k$  and a distance metric.
2. For the new sample, find the  $k$ -nearest neighbors.
3. Assign the class label by majority vote.

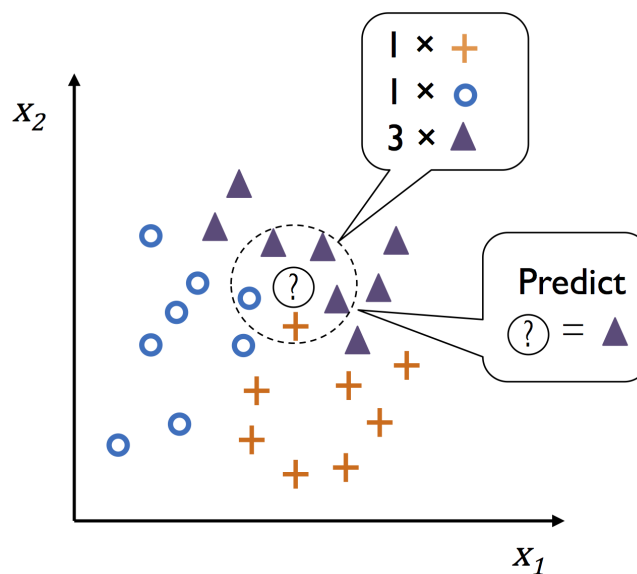


Figure 10.14: Illustration for how a new data point (?) is assigned the triangle class label, based on majority voting, when  $k = 5$ .

***k*-NN: pros and cons**

- Since it is memory-based, the classifier **immediately adapts** as we collect new training data.
- **(Prediction Cost)** The **computational complexity** for classifying new samples grows linearly with the number of samples in the training dataset in the worst-case scenario.<sup>a</sup>
- Furthermore, we can't discard training samples since *no training step* is involved. Thus, **storage space** can become a challenge if we are working with large datasets.

<sup>a</sup>J. H. Friedman, J. L. Bentley, and R.A. Finkel (1977). *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, ACM transactions on Mathematical Software (TOMS), **3**, no. 3, pp.209–226. The algorithm in the article is called the **KD-tree**.

***k*-NN: what to choose *k* and a distance metric?**

- The **right choice of *k* is crucial** to find a good balance between overfitting and underfitting.  
(For `sklearn.neighbors.KNeighborsClassifier`, default `n_neighbors = 5`.)
- We also choose a distance metric that is appropriate for the features in the dataset. (e.g., the simple Euclidean distance, along with data standardization)
- Alternatively, we can choose the **Minkowski distance**:

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|_p \stackrel{\text{def}}{=} \left( \sum_{i=1}^m |x_i - z_i|^p \right)^{1/p}. \quad (10.31)$$

(For `sklearn.neighbors.KNeighborsClassifier`, default `p = 2`.)

## 10.4. Neural Networks

**Recall:** The **Perceptron** (or, Adaline, Logistic Regression) is **the simplest artificial neuron** that makes decisions by **weighting up** evidence.

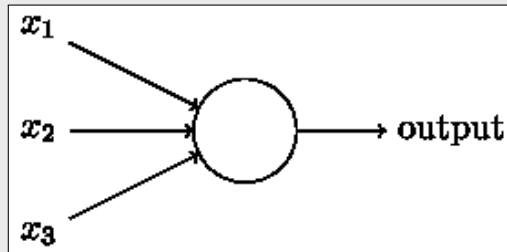


Figure 10.15: A simplest artificial neuron.

### Complex Neural Networks

- Obviously, a simple artificial neuron is not a complete model of human decision-making!
- However, they can be used as **building blocks** for more complex neural networks.

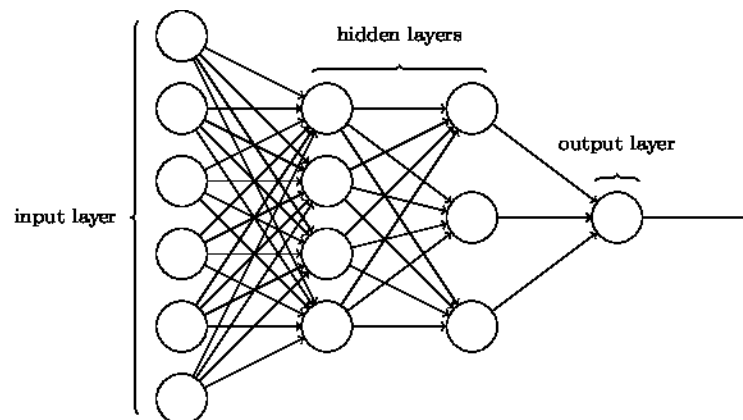


Figure 10.16: A complex neural network.

### 10.4.1. A Simple Network to Classify Hand-written Digits: MNIST Dataset

- The problem of recognizing hand-written digits has two components: segmentation and classification.

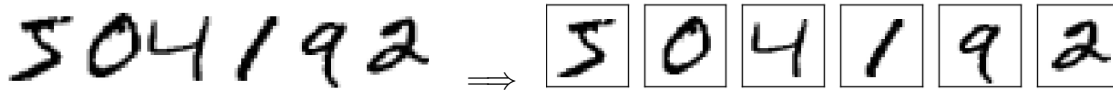


Figure 10.17: Segmentation.

- We'll focus on algorithmic components for the classification of individual digits.

#### MNIST dataset:

A modified subset of two datasets collected by NIST (US National Institute of Standards and Technology):

- Its first part contains 60,000 images (for training)
- The second part is 10,000 images (for test), each of which is in  $28 \times 28$  grayscale pixels

#### A Simple Neural Network

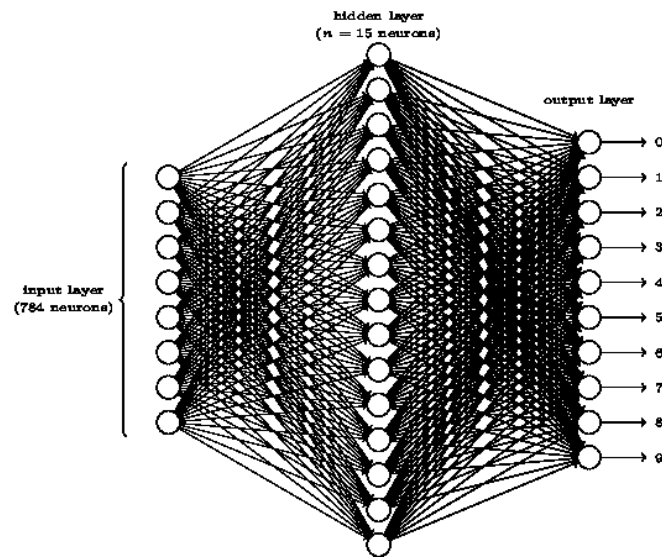


Figure 10.18: A sigmoid network **having a single hidden layer**.

### What the Neural Network Will Do: An Interpretation

- Let's concentrate on **the first output neuron**, the one that is trying to decide whether or not the input digit is a **0**.
- It does this by weighing up evidence from the hidden layer of neurons.

#### • What are those hidden neurons doing?

- Let's suppose **for the sake of argument** that **the first neuron** in the hidden layer may detect whether or not the input image contains



It can do this **by heavily weighting the corresponding pixels, and lightly weighting the other pixels.**

- Similarly, let's suppose that **the second, third, and fourth neurons** in the hidden layer detect whether or not the input image contains



- These four parts together make up a 0 image:



- Thus, if all four of these hidden neurons are firing, then we can conclude that the digit is a 0.

#### **Remark** 10.17. **Explainable AI (XAI)**

- The above is a common interpretation for neural networks.
- It *hardly* helps end users understand XAI
- Neural networks may have been designed **ineffectively**.

### Learning with Gradient Descent

- **Data set**  $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}, i = 1, 2, \dots, N$   
(e.g., if an image  $\mathbf{x}^{(k)}$  depicts a 2, then  $\mathbf{y}^{(k)} = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)^T$ .)

- **Cost function**

$$C(\mathbf{W}, B) = \frac{1}{2N} \sum_i \|\mathbf{y}^{(i)} - \mathbf{a}(\mathbf{x}^{(i)})\|^2, \quad (10.32)$$

where  $\mathbf{W}$  denotes the collection of all weights in the network,  $B$  all the biases, and  $\mathbf{a}(\mathbf{x}^{(i)})$  is the vector of outputs from the network when  $\mathbf{x}^{(i)}$  is input.

- **Gradient descent method**

$$\begin{bmatrix} \mathbf{W} \\ B \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{W} \\ B \end{bmatrix} + \begin{bmatrix} \Delta \mathbf{W} \\ \Delta B \end{bmatrix}, \quad (10.33)$$

where

$$\begin{bmatrix} \Delta \mathbf{W} \\ \Delta B \end{bmatrix} = -\eta \begin{bmatrix} \nabla_{\mathbf{W}} C \\ \nabla_B C \end{bmatrix}.$$

**Note:** To compute the gradient  $\nabla C$ , we need to compute the gradients  $\nabla C_{\mathbf{x}^{(i)}}$  separately for each training input,  $\mathbf{x}^{(i)}$ , and then average them:

$$\nabla C = \frac{1}{N} \sum_i \nabla C_{\mathbf{x}^{(i)}}. \quad (10.34)$$

Unfortunately, when the number of training inputs is very large, it can take a long time, and learning thus occurs slowly. An idea called **stochastic gradient descent** can be used to speed up learning.



### Stochastic Gradient Descent

The idea is to estimate the gradient  $\nabla C$  by computing  $\nabla C_{\mathbf{x}^{(i)}}$  for a **small sample of randomly chosen training inputs**. By averaging over this small sample, it turns out that we can quickly get a good estimate of the true gradient  $\nabla C$ ; this helps speed up gradient descent, and thus learning.

- Pick out a small number of randomly chosen training inputs ( $m \ll N$ ):

$$\tilde{\mathbf{x}}^{(1)}, \tilde{\mathbf{x}}^{(2)}, \dots, \tilde{\mathbf{x}}^{(m)},$$

which we refer to as a **mini-batch**.

- Average  $\nabla C_{\tilde{\mathbf{x}}^{(k)}}$  to approximate the gradient  $\nabla C$ . That is,

$$\frac{1}{m} \sum_{k=1}^m \nabla C_{\tilde{\mathbf{x}}^{(k)}} \approx \nabla C \stackrel{\text{def}}{=} \frac{1}{N} \sum_i \nabla C_{\mathbf{x}^{(i)}}. \quad (10.35)$$

- For classification of hand-written digits for the MNIST dataset, you may choose: `batch_size = 10`.

**Note:** In practice, you can implement the stochastic gradient descent as follows. **For an epoch**,

- Shuffle the dataset
- For each  $m$  samples (selected from the beginning), update  $(W, B)$  using the approximate gradient (10.35).

## 10.4.2. Implementation for MNIST Digits Dataset [9]

```

network.py
-----
1  """
2  network.py      (by Michael Nielsen)
3  ~~~~~
4  A module to implement the stochastic gradient descent learning
5  algorithm for a feedforward neural network. Gradients are calculated
6  using backpropagation. """
7  ##### Libraries
8  # Standard library
9  import random
10 # Third-party libraries
11 import numpy as np
12
13 class Network(object):
14     def __init__(self, sizes):
15         """The list ``sizes`` contains the number of neurons in the
16         respective layers of the network. For example, if the list
17         was [2, 3, 1] then it would be a three-layer network, with the
18         first layer containing 2 neurons, the second layer 3 neurons,
19         and the third layer 1 neuron. """
20
21         self.num_layers = len(sizes)
22         self.sizes = sizes
23         self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
24         self.weights = [np.random.randn(y, x)
25                         for x, y in zip(sizes[:-1], sizes[1:])]
26
27     def feedforward(self, a):
28         """Return the output of the network if ``a`` is input."""
29         for b, w in zip(self.biases, self.weights):
30             a = sigmoid(np.dot(w, a)+b)
31         return a
32
33     def SGD(self, training_data, epochs, mini_batch_size, eta,
34            test_data=None):
35         """Train the neural network using mini-batch stochastic
36         gradient descent. The ``training_data`` is a list of tuples
37         ``(x, y)`` representing the training inputs and the desired
38         outputs. """
39
40         if test_data: n_test = len(test_data)
41         n = len(training_data)
42         for j in xrange(epochs):
43             random.shuffle(training_data)
44             mini_batches = [
45                 training_data[k:k+mini_batch_size]
46                 for k in xrange(0, n, mini_batch_size)]

```

```

47         for mini_batch in mini_batches:
48             self.update_mini_batch(mini_batch, eta)
49         if test_data:
50             print "Epoch {0}: {1} / {2}".format(
51                 j, self.evaluate(test_data), n_test)
52         else:
53             print "Epoch {0} complete".format(j)
54
55     def update_mini_batch(self, mini_batch, eta):
56         """Update the network's weights and biases by applying
57         gradient descent using backpropagation to a single mini batch.
58         The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
59         is the learning rate."""
60         nabla_b = [np.zeros(b.shape) for b in self.biases]
61         nabla_w = [np.zeros(w.shape) for w in self.weights]
62         for x, y in mini_batch:
63             delta_nabla_b, delta_nabla_w = self.backprop(x, y)
64             nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
65             nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
66         self.weights = [w-(eta/len(mini_batch))*nw
67                         for w, nw in zip(self.weights, nabla_w)]
68         self.biases = [b-(eta/len(mini_batch))*nb
69                        for b, nb in zip(self.biases, nabla_b)]
70
71     def backprop(self, x, y):
72         """Return a tuple ``(nabla_b, nabla_w)`` representing the
73         gradient for the cost function C_x. ``nabla_b`` and
74         ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
75         to ``self.biases`` and ``self.weights``."""
76         nabla_b = [np.zeros(b.shape) for b in self.biases]
77         nabla_w = [np.zeros(w.shape) for w in self.weights]
78         # feedforward
79         activation = x
80         activations = [x] #list to store all the activations, layer by layer
81         zs = [] # list to store all the z vectors, layer by layer
82         for b, w in zip(self.biases, self.weights):
83             z = np.dot(w, activation)+b
84             zs.append(z)
85             activation = sigmoid(z)
86             activations.append(activation)
87         # backward pass
88         delta = self.cost_derivative(activations[-1], y) * \
89             sigmoid_prime(zs[-1])
90         nabla_b[-1] = delta
91         nabla_w[-1] = np.dot(delta, activations[-2].transpose())
92
93         for l in xrange(2, self.num_layers):

```

```

94         z = zs[-1]
95         sp = sigmoid_prime(z)
96         delta = np.dot(self.weights[-1+1].transpose(), delta) * sp
97         nabla_b[-1] = delta
98         nabla_w[-1] = np.dot(delta, activations[-1-1].transpose())
99         return (nabla_b, nabla_w)
100
101     def evaluate(self, test_data):
102         test_results = [(np.argmax(self.feedforward(x)), y)
103                         for (x, y) in test_data]
104         return sum(int(x == y) for (x, y) in test_results)
105
106     def cost_derivative(self, output_activations, y):
107         """Return the vector of partial derivatives \partial C_x /
108         \partial a for the output activations."""
109         return (output_activations-y)
110
111     ##### Miscellaneous functions
112     def sigmoid(z):
113         return 1.0/(1.0+np.exp(-z))
114
115     def sigmoid_prime(z):
116         return sigmoid(z)*(1-sigmoid(z))

```

The code is executed using

```

_____ Run_network.py _____
1  import mnist_loader
2  training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
3
4  import network
5  n_neurons = 20
6  net = network.Network([784 , n_neurons, 10])
7
8  n_epochs, batch_size, eta = 30, 10, 3.0
9  net.SGD(training_data , n_epochs, batch_size, eta, test_data = test_data)

```

```
len(training_data)=50000, len(validation_data)=10000, len(test_data)=10000
```

## Validation Accuracy

	Validation Accuracy
1	Epoch 0: 9006 / 10000
2	Epoch 1: 9128 / 10000
3	Epoch 2: 9202 / 10000
4	Epoch 3: 9188 / 10000
5	Epoch 4: 9249 / 10000
6	...
7	Epoch 25: 9356 / 10000
8	Epoch 26: 9388 / 10000
9	Epoch 27: 9407 / 10000
10	Epoch 28: 9410 / 10000
11	Epoch 29: 9428 / 10000

### Accuracy Comparisons

- scikit-learn's SVM classifier using the default settings: 9435/10000
- A well-tuned SVM:  $\approx 98.5\%$
- Well-designed **Convolutional NN (CNN)**:  
9979/10000 (**only 21 missed!**)

**Note:** For **well-designed neural networks**, the performance is close to **human-equivalent**, and is **arguably better**, since quite a few of the MNIST images are difficult even for humans to recognize with confidence, e.g.,

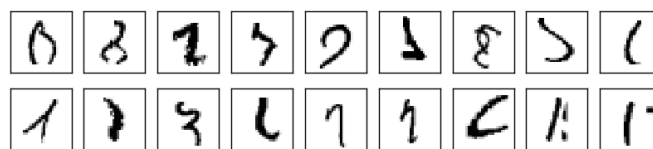


Figure 10.19: MNIST images difficult even for humans to recognize.

### XAI $\approx$ (Neural Network Design)

- The above neural network **converges slowly**. **Why?**
- Can we design **a new (explainable) form of neural network**
  - which converges in 2-3 epochs
  - to a model showing a better accuracy?

## 10.5. Scikit-Learn: A Python Machine Learning Library

**Scikit-learn** is one of the most useful and robust libraries for machine learning.

- It provides various tools for **machine learning** and **statistical modeling**, including
  - preprocessing,
  - classification, regression, clustering, and
  - ensemble methods.
- This library is built upon Numpy, SciPy, Matplotlib, and Pandas.

- **Prerequisites:** The following are required/recommended, before we start using scikit-learn.
  - Python 3
  - Numpy, SciPy, Matplotlib
  - Pandas (data analysis)
  - Seaborn (visualization)
- **Scikit-Learn Installation:** For example (on Ubuntu),

```
pip install -U scikit-learn
sudo apt-get install python3-sklearn python3-sklearn-lib
```

### Five Main Steps, in Machine Learning

1. Selection of features
2. Choosing a performance metric
3. Choosing a classifier and optimization algorithm
4. Evaluating the performance of the model
5. Tuning the algorithm

### In practice:

- Each algorithm has its own characteristics and is based on certain assumptions.
- **No Free Lunch Theorem:** No single classifier works best across all possible scenarios.
- **Best Model:** It is always recommended that you compare the performance of **at least a handful of different learning algorithms** to select the **best model** for the particular problem.

### Why Scikit-Learn?

- Nice documentation and usability
  - The library covers most machine-learning tasks:
    - Preprocessing modules
    - Algorithms
    - Analysis tools
  - **Robust Model:** Given a dataset, you may
    - (a) Compare algorithms
    - (b) Build an ensemble model
  - Scikit-learn **scales** to most data problems
- ⇒ **Easy-to-use, convenient, and powerful enough**

**A Simple Example Code**

iris\_sklearn.py

```

1  #-----
2  # Load Data
3  #-----
4  from sklearn import datasets
5  # dir(datasets); load_iris, load_digits, load_breast_cancer, load_wine, ...
6
7  iris = datasets.load_iris()
8
9  feature_names = iris.feature_names
10 target_names = iris.target_names
11 print("## feature names:", feature_names)
12 print("## target names :", target_names)
13 print("## set(iris.target):", set(iris.target))
14
15 #-----
16 # Create "model instances"
17 #-----
18 from sklearn.linear_model import LogisticRegression
19 from sklearn.neighbors import KNeighborsClassifier
20
21 LR = LogisticRegression(max_iter = 1000)
22 KNN = KNeighborsClassifier(n_neighbors = 5)
23
24 #-----
25 # Split, Train, and Test
26 #-----
27 import numpy as np
28 from sklearn.model_selection import train_test_split
29
30 X = iris.data; y = iris.target
31 iter = 100; Acc = np.zeros([iter,2])
32
33 for i in range(iter):
34     X_train, X_test, y_train, y_test = train_test_split(
35         X, y, test_size=0.3, random_state=i, stratify=y)
36     LR.fit(X_train, y_train); Acc[i,0] = LR.score(X_test, y_test)
37     KNN.fit(X_train, y_train); Acc[i,1] = KNN.score(X_test, y_test)
38
39 acc_mean = np.mean(Acc,axis=0)
40 acc_std = np.std(Acc,axis=0)
41 print('## iris.Accuracy.LR : %.4f +- %.4f' %(acc_mean[0],acc_std[0]))
42 print('## iris.Accuracy.KNN: %.4f +- %.4f' %(acc_mean[1],acc_std[1]))
43

```



```
44 #-----  
45 # New Sample ---> Predict  
46 #-----  
47 sample = [[5, 3, 2, 4],[4, 3, 3, 6]];  
48 print('## New sample =',sample)  
49  
50 predL = LR.predict(sample); predK = KNN.predict(sample)  
51 print("  ## sample.LR.predict :",target_names[predL])  
52 print("  ## sample.KNN.predict:",target_names[predK])
```

----- Output -----

```
1 ## feature names: ['sepal length (cm)', 'sepal width (cm)',  
2   'petal length (cm)', 'petal width (cm)']  
3 ## target names : ['setosa' 'versicolor' 'virginica']  
4 ## set(iris.target): {0, 1, 2}  
5 ## iris.Accuracy.LR : 0.9631 +- 0.0240  
6 ## iris.Accuracy.KNN: 0.9658 +- 0.0202  
7 ## New sample = [[5, 3, 2, 4], [4, 3, 3, 6]]  
8   ## sample.LR.predict : ['setosa' 'virginica']  
9   ## sample.KNN.predict: ['versicolor' 'virginica']
```

**In Scikit-Learn**, particularly with **ensembling**, you can finish most machine learning tasks **conveniently and easily**.

# A Machine Learning Modelcode: Scikit-Learn Comparisons and Ensembling

In machine learning, you can write a code **easily and effectively** using the following **modelcode**. It is also useful for **algorithm comparisons and ensembling**. You may download

<https://skim.math.msstate.edu/LectureNotes/data/Machine-Learning-Modelcode.PY.tar>.

```

Machine_Learning_Model.py
1  import numpy as np; import pandas as pd; import time
2  import seaborn as sbn; import matplotlib.pyplot as plt
3  from sklearn.model_selection import train_test_split
4  from sklearn import datasets
5  np.set_printoptions(suppress=True)
6
7  #=====
8  # Upload a Dataset: print(dir(datasets))
9  # load_iris, load_wine, load_breast_cancer, ...
10 #=====
11 data_read = datasets.load_iris(); #print(data_read.keys())
12
13 X = data_read.data
14 y = data_read.target
15 dataname = data_read.filename
16 targets = data_read.target_names
17 features = data_read.feature_names
18
19 #-----
20 # SETTING
21 #-----
22 N,d = X.shape; nclass=len(set(y));
23 print('DATA: N, d, nclass =',N,d,nclass)
24 rtrain = 0.7e0; run = 50; CompEnsm = 2;
25
26 def multi_run(clf,X,y,rtrain,run):
27     t0 = time.time(); acc = np.zeros([run,1])
28     for it in range(run):
29         Xtrain, Xtest, ytrain, ytest = train_test_split(
30             X, y, train_size=rtrain, random_state=it, stratify = y)
31         clf.fit(Xtrain, ytrain);
32         acc[it] = clf.score(Xtest, ytest)
33     etime = time.time()-t0
34     return np.mean(acc)*100, np.std(acc)*100, etime # accmean,acc_std,etime

```

```

35
36 #=====
37 # My Classifier
38 #=====
39 from myclf import * # My Classifier = MyCLF()
40 if 'MyCLF' in locals():
41     accmean, acc_std, etime = multi_run(MyCLF(mode=1),X,y,rtrain,run)
42
43     print('%s: MyCLF()      : Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
44           %(dataname,accmean,acc_std,etime/run))
45
46 #=====
47 # Scikit-learn Classifiers, for Comparisions && Ensembling
48 #=====
49 if CompEnsm >= 1:
50     exec(open("sklearn_classifiers.py").read())

```

```

----- myclf.py -----
1 import numpy as np
2 from sklearn.base import BaseEstimator, ClassifierMixin
3 from sklearn.tree import DecisionTreeClassifier
4
5 class MyCLF(BaseEstimator, ClassifierMixin): #a child class
6     def __init__(self, mode=0, learning_rate=0.01):
7         self.mode = mode
8         self.learning_rate = learning_rate
9         self.clf = DecisionTreeClassifier(max_depth=5)
10        if self.mode==1: print('MyCLF() = %s' %(self.clf))
11
12        def fit(self, X, y):
13            self.clf.fit(X, y)
14
15        def predict(self, X):
16            return self.clf.predict(X)
17
18        def score(self, X, y):
19            return self.clf.score(X, y)

```

**Note:** Replace `DecisionTreeClassifier()` with your own classier.

- The classifier must be implemented as **a child class** if it is used in ensembling.

```

sklearn_classifiers.py
1  #=====
2  # Required: X, y, multi_run [dataname, rtrain, run, CompEnsm]
3  #=====
4  from sklearn.preprocessing import StandardScaler
5  from sklearn.datasets import make_moons, make_circles, make_classification
6  from sklearn.neural_network import MLPClassifier
7  from sklearn.neighbors import KNeighborsClassifier
8  from sklearn.linear_model import LogisticRegression
9  from sklearn.svm import SVC
10 from sklearn.gaussian_process import GaussianProcessClassifier
11 from sklearn.gaussian_process.kernels import RBF
12 from sklearn.tree import DecisionTreeClassifier
13 from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
14 from sklearn.naive_bayes import GaussianNB
15 from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
16 from sklearn.ensemble import VotingClassifier
17
18 #-----
19 classifiers = [
20     LogisticRegression(max_iter = 1000),
21     KNeighborsClassifier(5),
22     SVC(kernel="linear", C=0.5),
23     SVC(gamma=2, C=1),
24     RandomForestClassifier(max_depth=5, n_estimators=50, max_features=1),
25     MLPClassifier(hidden_layer_sizes=[100], activation='logistic',
26                   alpha=0.5, max_iter=1000),
27     AdaBoostClassifier(),
28     GaussianNB(),
29     QuadraticDiscriminantAnalysis(),
30     GaussianProcessClassifier(),
31 ]
32 names = [
33     "Logistic-Regr",
34     "KNeighbors-5 ",
35     "SVC-Linear  ",
36     "SVC-RBF    ",
37     "Random-Forest",
38     "MLPClassifier",
39     "AdaBoost   ",
40     "Naive-Bayes ",
41     "QDA       ",
42     "Gaussian-Proc",
43 ]

```

```

44 #-----
45 if dataname is None: dataname = 'No-dataname';
46 if run      is None: run      = 50;
47 if rtrain   is None: rtrain   = 0.7e0;
48 if CompEnsm is None: CompEnsm = 2;
49
50 #=====
51 print('===== Comparision: Scikit-learn Classifiers =====')
52 #=====
53 import os;
54 acc_max=0; Acc_CLF = np.zeros([len(classifiers),1]);
55
56 for k, (name, clf) in enumerate(zip(names, classifiers)):
57     accmean, acc_std, etime = multi_run(clf,X,y,rtrain,run)
58
59     Acc_CLF[k] = accmean
60     if accmean>acc_max: acc_max,alname = accmean,name
61     print('%s: %s: Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
62           %(os.path.basename(dataname),name,accmean,acc_std,etime/run))
63 print('-----')
64 print('sklearn classifiers Acc: (mean,max) = (%.2f,%.2f)%%; Best = %s'
65       %(np.mean(Acc_CLF),acc_max,alname))
66
67 if CompEnsm <2: quit()
68 #=====
69 print('===== Ensembling: SKlearn Classifiers =====')
70 #=====
71 names = [x.rstrip() for x in names]
72 popped_clf = []
73 popped_clf.append(names.pop(9)); classifiers.pop(9); #Gaussian Proc
74 popped_clf.append(names.pop(7)); classifiers.pop(7); #Naive Bayes
75 popped_clf.append(names.pop(6)); classifiers.pop(6); #AdaBoost
76 popped_clf.append(names.pop(4)); classifiers.pop(4); #Random Forest
77 popped_clf.append(names.pop(0)); classifiers.pop(0); #Logistic Regr
78 #print('popped_clf=',popped_clf[::-1])
79
80 CLFs = [(name, clf) for name, clf in zip(names, classifiers)]
81 #if 'MyCLF' in locals(): CLFs += [('MyCLF',MyCLF())]
82 EnCLF = VotingClassifier(estimators=CLFs, voting='hard')
83 accmean, acc_std, etime = multi_run(EnCLF,X,y,rtrain,run)
84
85 print('EnCLF =',[lis[0] for lis in CLFs])
86 print('%s: Ensemble CLFs: Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
87       %(os.path.basename(dataname),accmean,acc_std,etime/run))

```

## Output

```

1 DATA: N, d, nclass = 150 4 3
2 MyCLF() = DecisionTreeClassifier(max_depth=5)
3 iris.csv: MyCLF() : Acc.(mean,std) = (94.53,3.12)%; E-time= 0.00074
4 ===== Comparision: Scikit-learn Classifiers =====
5 iris.csv: Logistic-Regr: Acc.(mean,std) = (96.13,2.62)%; E-time= 0.01035
6 iris.csv: KNeighbors-5 : Acc.(mean,std) = (96.49,1.99)%; E-time= 0.00176
7 iris.csv: SVC-Linear : Acc.(mean,std) = (97.60,2.26)%; E-time= 0.00085
8 iris.csv: SVC-RBF : Acc.(mean,std) = (96.62,2.10)%; E-time= 0.00101
9 iris.csv: Random-Forest: Acc.(mean,std) = (94.84,3.16)%; E-time= 0.03647
10 iris.csv: MLPClassifier: Acc.(mean,std) = (98.58,1.32)%; E-time= 0.20549
11 iris.csv: AdaBoost : Acc.(mean,std) = (94.40,2.64)%; E-time= 0.04119
12 iris.csv: Naive-Bayes : Acc.(mean,std) = (95.11,3.20)%; E-time= 0.00090
13 iris.csv: QDA : Acc.(mean,std) = (97.64,2.06)%; E-time= 0.00085
14 iris.csv: Gaussian-Proc: Acc.(mean,std) = (95.64,2.63)%; E-time= 0.16151
15 -----
16 sklearn classifiers Acc: (mean,max) = (96.31,98.58)%; Best = MLPClassifier
17 ===== Ensembling: SKlearn Classifiers =====
18 EnCLF = ['KNeighbors-5', 'SVC-Linear', 'SVC-RBF', 'MLPClassifier', 'QDA']
19 iris.csv: Ensemble CLFs: Acc.(mean,std) = (97.60,1.98)%; E-time= 0.22272

```

**Ensembling:**

**You may stack **the best** and **its siblings** of other options.**

## Exercises for Chapter 10

### 10.1. Machine Learning Modelcode

- (a) Search the database to get at least five datasets.  
(You may try “`print(dir(datasets))`”.)
- (b) Run the Machine Learning Modelcode, p. 296, to compare the performances of 10 selected classifiers.

### 10.2. Modify `perceptron.py`, p. 269, to get a code for Adaline.

- For a given training dataset, Adaline converges to a unique *weights*, while Perceptron does not.
- Note that the correction terms are accumulated from all data points in each iteration. As a consequence, the learning rate  $\eta$  may be chosen smaller as the number of points increases.

**Implementation:** In order to overcome the problem, you may scale the correction terms by the number of data points.

- Redefine the **cost function** (10.9):

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^N (y^{(i)} - \phi(z^{(i)}))^2. \quad (10.36)$$

where  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)} + b$  and  $\phi = I$ , the identity.

- Then the correction terms in (10.12) become correspondingly

$$\begin{aligned} \Delta \mathbf{w} &= \eta \frac{1}{N} \sum_i (y^{(i)} - \phi(z^{(i)})) \mathbf{x}^{(i)}, \\ \Delta b &= \eta \frac{1}{N} \sum_i (y^{(i)} - \phi(z^{(i)})). \end{aligned} \quad (10.37)$$





# CHAPTER 11

## Principal Component Analysis

### Contents of Chapter 11

11.1. Principal Component Analysis . . . . .	304
11.2. Singular Value Decomposition . . . . .	315
11.3. Applications of the SVD to LS Problems . . . . .	321
Exercises for Chapter 11 . . . . .	329

## 11.1. Principal Component Analysis

**Definition 11.1.** **Principal component analysis (PCA)** is the process of computing and using the **principal components** to perform a **change of basis** on the data, sometimes with only the first few principal components and ignoring the rest.

### The PCA, in a Nutshell

- The PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of *possibly correlated variables* into a set of *linearly uncorrelated variables* called the **principal components**.
- The **orthogonal axes** of the new subspace can be interpreted as the **directions of maximum variance** given the constraint that the new feature axes are orthogonal to each other:

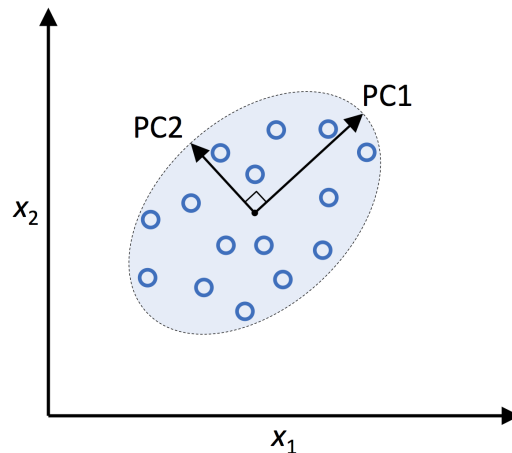


Figure 11.1: Principal components.

- It can be shown that the **principal directions** are **eigenvectors of the data's covariance matrix**.
- The PCA directions are highly **sensitive to data scaling**, and we need to standardize the features prior to PCA, particularly when the features were measured on different scales and we want to assign equal importance to all features.

### 11.1.1. The covariance matrix

**Definition 11.2.** **Variance** measures the variation of a single random variable, whereas **covariance** is a measure of the joint variability of two random variables. Let the random variable pair  $(x, y)$  take on the values  $\{(x_i, y_i) \mid i = 1, 2, \dots, n\}$ , with equal probabilities  $p_i = 1/n$ . Then

- The formula for **variance** of  $x$  is given by

$$\sigma_x^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2, \quad (11.1)$$

where  $\bar{x}$  is the mean of  $x$  values.

- The **covariance**  $\sigma(x, y)$  of two random variables  $x$  and  $y$  is given by

$$\text{cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}). \quad (11.2)$$

**Remark 11.3.** In reality, **data** are saved in a matrix  $X \in \mathbb{R}^{n \times d}$ :

- each of the  $n$  rows represents a different data point, and
- each of the  $d$  columns gives a particular kind of feature.

Thus,  $d$  describes the dimension of data points and also can be considered as the number of random variables.

**Definition 11.4.** The **covariance matrix** of a data matrix  $X \in \mathbb{R}^{n \times d}$  is a square matrix  $C \in \mathbb{R}^{d \times d}$ , whose  $(i, j)$ -entry is the covariance of the  $i$ -th column and the  $j$ -th column of  $X$ . That is,

$$C = [C_{ij}] \in \mathbb{R}^{d \times d}, \quad C_{ij} = \text{cov}(X_i, X_j). \quad (11.3)$$

**Example 11.5.** Let  $\hat{X}$  be the data  $X$  subtracted by the mean column-wisely:  $\hat{X} = X - E[X]$ . Then the covariance matrix of  $X$  reads

$$C = \frac{1}{n} \hat{X}^T \hat{X} = \frac{1}{n} (X - E[X])^T (X - E[X]), \quad (11.4)$$

for which the scaling factor  $1/n$  is often ignored in reality.  $\square$

**Example 11.6.** Generate a synthetic data  $X$  in 2D to find its **covariance matrix** and **principal directions**.

**Solution.**

```

util_Covariance.py
1  import numpy as np
2
3  # Generate data
4  def generate_data(n):
5      # Normally distributed around the origin
6      x = np.random.normal(0,1, n)
7      y = np.random.normal(0,1, n)
8      S = np.vstack((x, y)).T
9      # Transform
10     sx, sy = 1, 3;
11     Scale = np.array([[sx, 0], [0, sy]])
12     theta = 0.25*np.pi; c,s = np.cos(theta), np.sin(theta)
13     Rot = np.array([[c, -s], [s, c]]).T #T, due to right multiplication
14
15     return S.dot(Scale).dot(Rot) +[5,2]
16
17 # Covariance
18 def cov(x, y):
19     xbar, ybar = x.mean(), y.mean()
20     return np.sum((x - xbar)*(y - ybar))/len(x)
21
22 # Covariance matrix
23 def cov_matrix(X):
24     return np.array([[cov(X[:,0], X[:,0]), cov(X[:,0], X[:,1])], \
25                     [cov(X[:,1], X[:,0]), cov(X[:,1], X[:,1])]])

```

```

Covariance.py
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from util_Covariance import *
4
5  # Generate data
6  n = 200
7  X = generate_data(n)
8  print('Generated data: X.shape =', X.shape)
9
10 # Covariance matrix
11 C = cov_matrix(X)

```

```
12 print('C:\n',C)
13
14 # Principal directions
15 eVal, eVec = np.linalg.eig(C)
16 xbar,ybar = np.mean(X,0)
17 print('eVal:\n',eVal); print('eVec:\n',eVec)
18 print('np.mean(X, 0) =',xbar,ybar)
19
20 # Plotting
21 plt.style.use('ggplot')
22 plt.scatter(X[:, 0],X[:, 1],c='#00a0c0',s=10)
23 plt.axis('equal');
24 plt.title('Generated Data')
25 plt.savefig('py-data-generated.png')
26
27 for e, v in zip(eVal, eVec.T):
28     plt.plot([0,2*np.sqrt(e)*v[0]]+xbar,\
29             [0,2*np.sqrt(e)*v[1]]+ybar, 'r-', lw=2)
30 plt.title('Principal Directions')
31 plt.savefig('py-data-principal-directions.png')
32 plt.show()
```



Figure 11.2: Synthetic data and its principal directions (right).

Output

```

1  Generated data: X.shape = (200, 2)
2  C:
3  [[ 5.10038723 -4.15289232]
4  [-4.15289232  4.986776   ]]
5  eVal:
6  [9.19686242  0.89030081]
7  eVec:
8  [[ 0.71192601  0.70225448]
9  [-0.70225448  0.71192601]]
10 np.mean(X, 0) = 4.986291809096116  2.1696690114181947

```

### Observation 11.7. Covariance Matrix.

- **Symmetry:** The covariance matrix  $C$  is **symmetric** so that it is **diagonalizable**. (See §5.2.2, p.144.) That is,

$$C = UDU^{-1}, \quad (11.5)$$

where  $D$  is a diagonal matrix of eigenvalues of  $C$  and  $U$  is the corresponding eigenvectors of  $C$  such that  $U^T U = I$ . (Such a square matrix  $U$  is called an **orthogonal matrix**.)

- **Principal directions:** The principal directions are **eigenvectors of the data's covariance matrix**.
- **Minimum volume enclosing ellipsoid (MVEE):** The **PCA** can be viewed as **fitting a  $d$ -dimensional ellipsoid** to the data, where each axis of the ellipsoid represents one of **principal directions**.
  - If some axis of the ellipsoid is small, then the variance along that axis is also small.

### 11.1.2. Computation of principal components

- Consider a **data matrix**  $X \in \mathbb{R}^{n \times d}$ :
  - each of the  $n$  rows represents a different data point,
  - each of the  $d$  columns gives a particular kind of feature, and
  - each column has zero empirical mean (e.g., after standardization).
- Our goal is to find an **orthogonal** weight matrix  $W \in \mathbb{R}^{d \times d}$  such that

$$Z = XW, \quad (11.6)$$

where  $Z \in \mathbb{R}^{n \times d}$  is call the **score matrix**. Columns of  $Z$  represent principal components of  $X$ .

#### First weight vector $w_1$ : the first column of $W$ :

In order to maximize variance of  $z_1$ , the first weight vector  $w_1$  should satisfy

$$\begin{aligned} w_1 &= \arg \max_{\|w\|=1} \|z_1\|^2 = \arg \max_{\|w\|=1} \|Xw\|^2 \\ &= \arg \max_{\|w\|=1} w^T X^T X w = \arg \max_{w \neq 0} \frac{w^T X^T X w}{w^T w}, \end{aligned} \quad (11.7)$$

where the quantity to be maximized can be recognized as a **Rayleigh quotient**.

**Theorem 11.8.** For a **positive semidefinite matrix** (such as  $X^T X$ ), the maximum of the Rayleigh quotient is the same as the largest eigenvalue of the matrix, which occurs when  $w$  is the corresponding eigenvector, i.e.,

$$w_1 = \arg \max_{w \neq 0} \frac{w^T X^T X w}{w^T w} = \frac{v_1}{\|v_1\|}, \quad (X^T X)v_1 = \lambda_1 v_1, \quad (11.8)$$

where  $\lambda_1$  is the largest eigenvalue of  $X^T X \in \mathbb{R}^{d \times d}$ .

**Example 11.9.** With  $w_1$  found, the **first principal component** of a data vector  $x^{(i)}$ , the  $i$ -th row of  $X$ , is then given as a score  $z_1^{(i)} = x^{(i)} \cdot w_1$ .

**Further weight vectors  $\mathbf{w}_k$ :**

The  $k$ -th weight vector can be found by ① subtracting the first  $(k - 1)$  principal components from  $X$ :

$$\hat{X}_k := X - \sum_{i=1}^{k-1} X \mathbf{w}_i \mathbf{w}_i^T, \quad (11.9)$$

and then ② finding the weight vector which extracts the maximum variance from this new data matrix

$$\mathbf{w}_k = \arg \max_{\|\mathbf{w}\|=1} \|\hat{X}_k \mathbf{w}\|^2, \quad (11.10)$$

which turns out to give the **remaining eigenvectors of  $X^T X$** .

**Remark 11.10.** The principal components transformation can also be associated with the **singular value decomposition (SVD)** of  $X$ :

$$X = U \Sigma V^T, \quad (11.11)$$

where

$U$  :  $n \times d$  orthogonal (the **left singular vectors** of  $X$ .)

$\Sigma$  :  $d \times d$  diagonal (the **singular values** of  $X$ .)

$V$  :  $d \times d$  orthogonal (the **right singular vectors** of  $X$ .)

- The matrix  $\Sigma$  explicitly reads

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_d), \quad (11.12)$$

where  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_d \geq 0$ .

- In terms of this factorization, the matrix  $X^T X$  reads

$$X^T X = (U \Sigma V^T)^T U \Sigma V^T = V \Sigma U^T U \Sigma V^T = V \Sigma^2 V^T. \quad (11.13)$$

- Comparing with the **eigenvector factorization** of  $X^T X$ , we conclude
  - the right singular vectors  $V \cong$  the eigenvectors of  $X^T X \Rightarrow V \cong W$
  - (the square of singular values of  $X$ ) = (the eigenvalues of  $X^T X$ )  
 $\Rightarrow \sigma_j^2 = \lambda_j, j = 1, 2, \dots, d$ .



**Summary 11.11. (Computation of Principal Components)**

1. Computer the **singular value decomposition** (SVD) of  $X$ :

$$X = U\Sigma V^T. \quad (11.14)$$

2. Set

$$W = V. \quad (11.15)$$

Then the score matrix, the set of **principal components**, is

$$\begin{aligned} Z &= XW = XV = U\Sigma V^T V = U\Sigma \\ &= [\sigma_1 \mathbf{u}_1 | \sigma_2 \mathbf{u}_2 | \cdots | \sigma_d \mathbf{u}_d]. \end{aligned} \quad (11.16)$$

\* The SVD will be discussed in §11.2.

**11.1.3. Dimensionality reduction: Data compression**

- The transformation  $Z = XW$  maps a data vector  $\mathbf{x}^{(i)} \in \mathbb{R}^d$  to a new space of  $d$  variables which are now uncorrelated.
- However, not all the principal components need to be kept.
- Keeping only the first  $k$  principal components, produced by using only the first  $k$  eigenvectors of  $X^T X$  ( $k \ll d$ ), gives the **truncated score matrix**:

$$Z_k := X W_k = U\Sigma V^T W_k = U\Sigma_k, \quad (11.17)$$

where  $Z_k \in \mathbb{R}^{n \times k}$ ,  $W_k \in \mathbb{R}^{d \times k}$ , and

$$\Sigma_k := \text{diag}(\sigma_1, \dots, \sigma_k, 0, \dots, 0). \quad (11.18)$$

- It follows from (11.17) that the corresponding **truncated data matrix** reads

$$X_k = Z_k W_k^T = U \Sigma_k W_k^T = U \Sigma_k W^T = U \Sigma_k V^T. \quad (11.19)$$

**Quesitons.** How can we choose  $k$ ? &

Is the difference  $\|X - X_k\|$  (that we truncated) small?

**Claim 11.12.** It follows from (11.11) and (11.19) that

$$\begin{aligned}\|X - X_k\|_2 &= \|U \Sigma V^T - U \Sigma_k V^T\|_2 \\ &= \|U(\Sigma - \Sigma_k)V^T\|_2 \\ &= \|\Sigma - \Sigma_k\|_2 = \sigma_{k+1},\end{aligned}\tag{11.20}$$

where  $\|\cdot\|_2$  is the induced matrix  $L^2$ -norm.

**Remark 11.13.** Efficient algorithms exist to compute the SVD of  $X$  without having to form the matrix  $X^T X$ , so computing the SVD is now the standard way to carry out the PCA. See [6, 13].

### Image Compression

- **Dyadic Decomposition:** The data matrix  $X \in \mathbb{R}^{m \times n}$  is expressed as a sum of rank-1 matrices:

$$X = U \Sigma V^T = \sum_{i=1}^n \sigma_i \mathbf{u}_i \mathbf{v}_i^T,\tag{11.21}$$

where

$$V = [\mathbf{v}_1, \dots, \mathbf{v}_n], \quad U = [\mathbf{u}_1, \dots, \mathbf{u}_n].$$

- **Approximation:**  $X$  can be approximated as

$$X \approx X_k := U \Sigma_k V^T = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^T\tag{11.22}$$

is **closest to**  $X$  among matrices of rank  $\leq k$ , and

$$\|X - X_k\|_2 = \sigma_{k+1}.$$

- It only takes  $n \cdot k + m \cdot k = (m + n) \cdot k$  words to store  $[\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k]$  and  $[\sigma_1 \mathbf{u}_1, \sigma_2 \mathbf{u}_2, \dots, \sigma_k \mathbf{u}_k]$ , from which we can reconstruct  $X_k$ .
- We use  $X_k$  as our compressed images, stored using  $(m + n) \cdot k$  words.

A Matlab code to demonstrate the SVD compression of images:

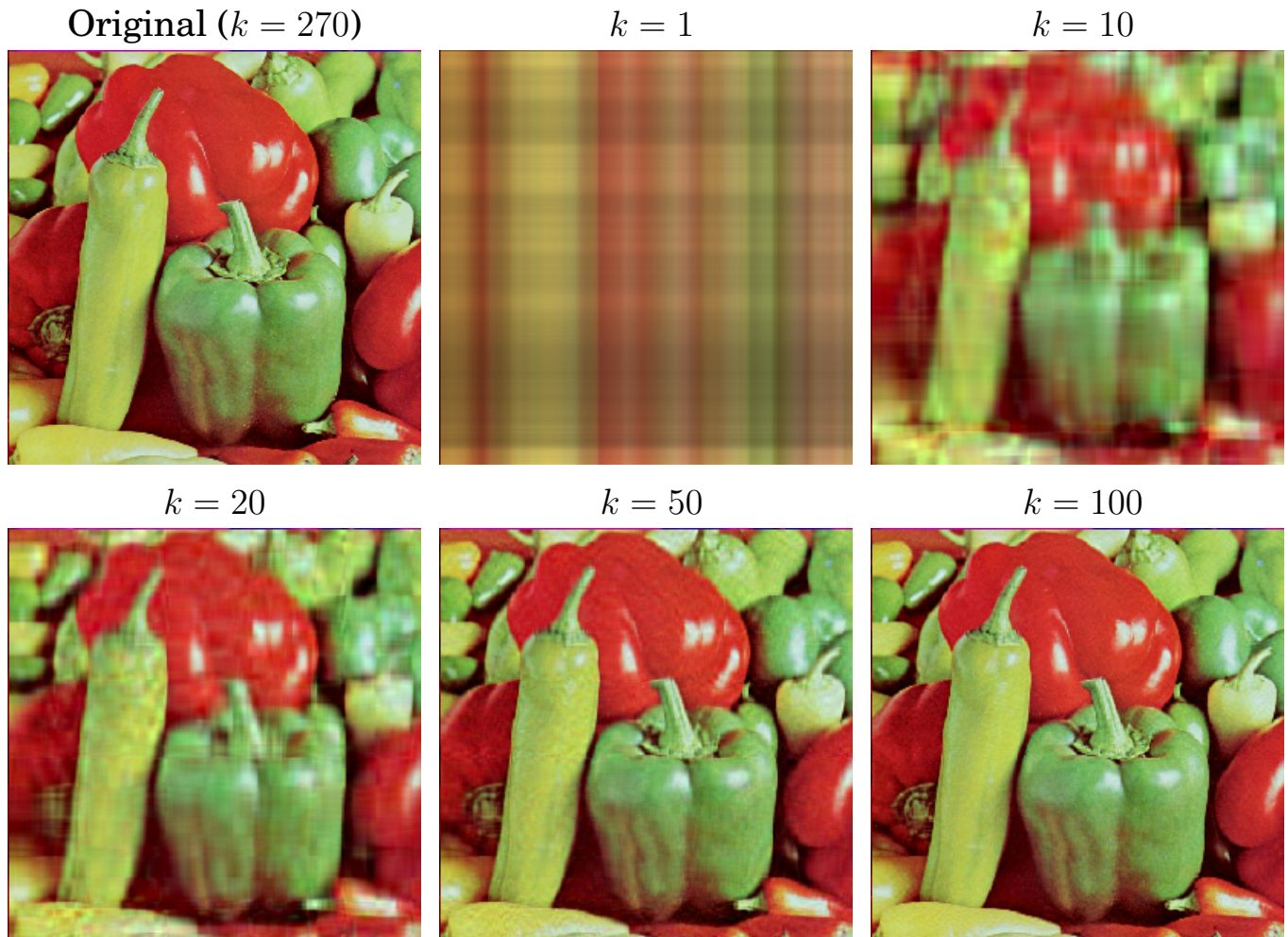
```

peppers_compress.m
1  img = imread('Peppers.png'); [m,n,d]=size(img);
2  [U,S,V] = svd(reshape(im2double(img),m, []));
3  %%---- select k <= p=min(m,n)
4  k = 20;
5  img_k = U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
6  img_k = reshape(img_k,m,n,d);
7  figure, imshow(img_k)

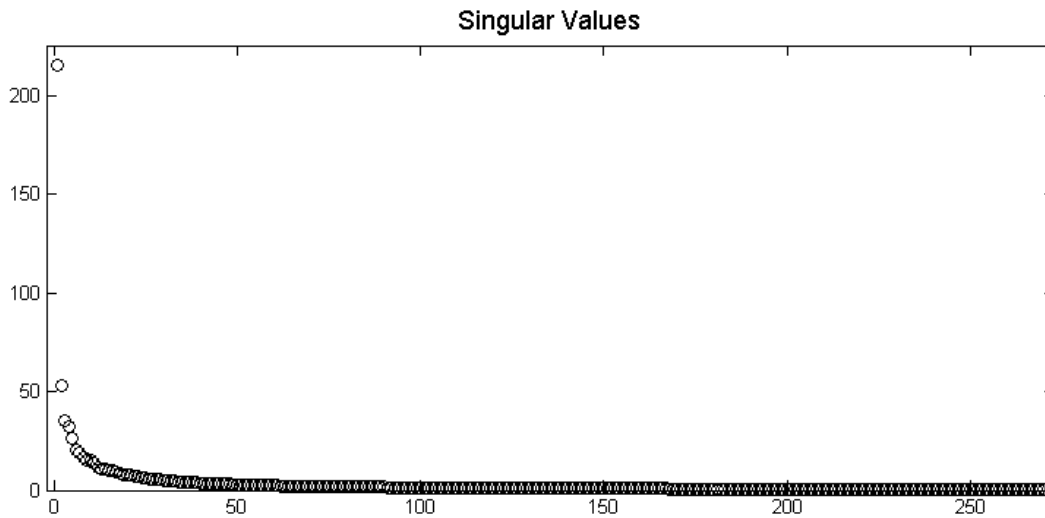
```

The "Peppers" image is in  $[270, 270, 3] \in \mathbb{R}^{270 \times 810}$ .

### Image compression using $k$ singular values



### Peppers: Singular values



### Peppers: Compression quality

$$\text{PSNR (dB)} = \begin{cases} 13.7 & \text{when } k = 1, \\ 20.4 & \text{when } k = 10, \\ 23.7 & \text{when } k = 20, \\ 29.0 & \text{when } k = 50, \\ 32.6 & \text{when } k = 100, \\ 37.5 & \text{when } k = 150, \end{cases}$$

where **PSNR** is the “Peak Signal-to-Noise Ratio.”

**Peppers Storage:** It requires  $(m + n) \cdot k$  words.

For example, when  $k = 50$ ,

$$(m + n) \cdot k = (270 + 810) \cdot 50 = \boxed{54,000}, \quad (11.23)$$

which is approximately **a quarter** the full storage space

$$270 \times 270 \times 3 = \boxed{218,700}.$$

## 11.2. Singular Value Decomposition

Here we will deal with the SVD in detail.

**Theorem 11.14. (SVD Theorem).** Let  $A \in \mathbb{R}^{m \times n}$  with  $m \geq n$ . Then we can write

$$A = U \Sigma V^T, \quad (11.24)$$

where  $U \in \mathbb{R}^{m \times n}$  and satisfies  $U^T U = I$ ,  $V \in \mathbb{R}^{n \times n}$  and satisfies  $V^T V = I$ , and  $\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_n)$ , where

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0.$$

**Remark 11.15.** The matrices are illustrated pictorially as

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} U \end{bmatrix} \begin{bmatrix} \Sigma \end{bmatrix} \begin{bmatrix} V^T \end{bmatrix}, \quad (11.25)$$

where

$U$  :  $m \times n$  orthogonal (the **left singular vectors** of  $A$ .)

$\Sigma$  :  $n \times n$  diagonal (the **singular values** of  $A$ .)

$V$  :  $n \times n$  orthogonal (the **right singular vectors** of  $A$ .)

- For some  $r \leq n$ , the singular values may satisfy

$$\underbrace{\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r}_{\text{nonzero singular values}} > \sigma_{r+1} = \dots = \sigma_n = 0. \quad (11.26)$$

In this case,  $\text{rank}(A) = r$ .

- If  $m < n$ , the **SVD** is defined by considering  $A^T$ .

**Proof. (of Theorem 11.14)** Use induction on  $m$  and  $n$ : we assume that the  $SVD$  exists for  $(m-1) \times (n-1)$  matrices, and prove it for  $m \times n$ . We assume  $A \neq 0$ ; otherwise we can take  $\Sigma = 0$  and let  $U$  and  $V$  be arbitrary orthogonal matrices.

- The basic step occurs when  $n = 1$  ( $m \geq n$ ). We let  $A = U\Sigma V^T$  with  $U = A/\|A\|_2$ ,  $\Sigma = \|A\|_2$ ,  $V = 1$ .
- For the induction step, choose  $\mathbf{v}$  so that

$$\|\mathbf{v}\|_2 = 1 \text{ and } \|A\|_2 = \|A\mathbf{v}\|_2 > 0.$$

- Let  $\mathbf{u} = \frac{A\mathbf{v}}{\|A\mathbf{v}\|_2}$ , which is a unit vector. Choose  $\tilde{U}$ ,  $\tilde{V}$  such that

$$U = [\mathbf{u} \ \tilde{U}] \in \mathbb{R}^{m \times n} \text{ and } V = [\mathbf{v} \ \tilde{V}] \in \mathbb{R}^{n \times n}$$

are orthogonal.

- Now, we write

$$U^T A V = \begin{bmatrix} \mathbf{u}^T \\ \tilde{U}^T \end{bmatrix} \cdot A \cdot [\mathbf{v} \ \tilde{V}] = \begin{bmatrix} \mathbf{u}^T A \mathbf{v} & \mathbf{u}^T A \tilde{V} \\ \tilde{U}^T A \mathbf{v} & \tilde{U}^T A \tilde{V} \end{bmatrix}$$

Since

$$\begin{aligned} \mathbf{u}^T A \mathbf{v} &= \frac{(A\mathbf{v})^T (A\mathbf{v})}{\|A\mathbf{v}\|_2} = \frac{\|A\mathbf{v}\|_2^2}{\|A\mathbf{v}\|_2} = \|A\mathbf{v}\|_2 = \|A\|_2 \equiv \sigma, \\ \tilde{U}^T A \mathbf{v} &= \tilde{U}^T \mathbf{u} \|A\mathbf{v}\|_2 = 0, \end{aligned}$$

we have

$$U^T A V = \begin{bmatrix} \sigma & 0 \\ 0 & U_1 \Sigma_1 V_1^T \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix} \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix}^T,$$

or equivalently

$$A = \left( U \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix} \right) \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \left( V \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix} \right)^T. \quad (11.27)$$

Equation (11.27) is our desired decomposition.  $\square$

### 11.2.1. Algebraic interpretation of the SVD

Let  $\text{rank}(A) = r$ . let the SVD of  $A$  be  $A = U \Sigma V^T$ , with

$$\begin{aligned} U &= [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_n], \\ \Sigma &= \mathbf{diag}(\sigma_1, \sigma_2, \cdots, \sigma_n), \\ V &= [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n], \end{aligned}$$

and  $\sigma_r$  be the **smallest** positive singular value. Since

$$A = U \Sigma V^T \iff AV = U \Sigma V^T V = U \Sigma,$$

we have

$$\begin{aligned} AV &= A[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n] = [A\mathbf{v}_1 \quad A\mathbf{v}_2 \quad \cdots \quad A\mathbf{v}_n] \\ &= [\mathbf{u}_1 \quad \cdots \quad \mathbf{u}_r \quad \cdots \quad \mathbf{u}_n] \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & \ddots & & \\ & & & & 0 & \\ & & & & & \ddots \\ & & & & & & 0 \end{bmatrix} \\ &= [\sigma_1 \mathbf{u}_1 \quad \cdots \quad \sigma_r \mathbf{u}_r \quad \mathbf{0} \quad \cdots \quad \mathbf{0}]. \end{aligned} \tag{11.28}$$

Therefore,

$$A = U \Sigma V^T \iff \begin{cases} A\mathbf{v}_j = \sigma_j \mathbf{u}_j, & j = 1, 2, \cdots, r \\ A\mathbf{v}_j = \mathbf{0}, & j = r + 1, \cdots, n \end{cases} \tag{11.29}$$

Similarly, starting from  $A^T = V \Sigma U^T$ ,

$$A^T = V \Sigma U^T \iff \begin{cases} A^T \mathbf{u}_j = \sigma_j \mathbf{v}_j, & j = 1, 2, \cdots, r \\ A^T \mathbf{u}_j = \mathbf{0}, & j = r + 1, \cdots, n \end{cases} \tag{11.30}$$

**Summary 11.16.** It follows from (11.29) and (11.30) that

- $(\mathbf{v}_j, \sigma_j^2)$ ,  $j = 1, 2, \dots, r$ , are eigenvector-eigenvalue pairs of  $A^T A$ .

$$A^T A \mathbf{v}_j = A^T (\sigma_j \mathbf{u}_j) = \sigma_j^2 \mathbf{v}_j, \quad j = 1, 2, \dots, r. \quad (11.31)$$

So, the singular values play the role of eigenvalues.

- Similarly, we have

$$A A^T \mathbf{u}_j = A (\sigma_j \mathbf{v}_j) = \sigma_j^2 \mathbf{u}_j, \quad j = 1, 2, \dots, r. \quad (11.32)$$

- Equation (11.31) gives how to find the **singular values**  $\{\sigma_j\}$  and the **right singular vectors**  $V$ , while (11.29) shows a way to compute the **left singular vectors**  $U$ .
- **(Dyadic decomposition)** The matrix  $A \in \mathbb{R}^{m \times n}$  can be expressed as

$$A = \sum_{j=1}^n \sigma_j \mathbf{u}_j \mathbf{v}_j^T. \quad (11.33)$$

When  $\text{rank}(A) = r \leq n$ ,

$$A = \sum_{j=1}^r \sigma_j \mathbf{u}_j \mathbf{v}_j^T. \quad (11.34)$$

This property has been utilized for various approximations and applications, e.g., by dropping singular vectors corresponding to *small* singular values.



### 11.2.2. Computation of the SVD

For  $A \in \mathbb{R}^{m \times n}$ , the procedure is as follows.

1. Form  $A^T A$  ( $A^T A$  – **covariance matrix** of  $A$ ).
2. Find the eigen-decomposition of  $A^T A$  by orthogonalization process, i.e.,  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ ,

$$A^T A = V \Lambda V^T,$$

where  $V = [\mathbf{v}_1 \ \dots \ \mathbf{v}_n]$  is orthogonal, i.e.,  $V^T V = I$ .

3. Sort the eigenvalues according to their magnitude and let

$$\sigma_j = \sqrt{\lambda_j}, \quad j = 1, 2, \dots, n.$$

4. Form the  $U$  matrix as follows,

$$\mathbf{u}_j = \frac{1}{\sigma_j} A \mathbf{v}_j, \quad j = 1, 2, \dots, r.$$

If necessary, pick up the remaining columns of  $U$  so it is orthogonal. (These additional columns must be in  $\text{Null}(AA^T)$ .)

$$5. A = U \Sigma V^T = [\mathbf{u}_1 \ \dots \ \mathbf{u}_r \ \dots \ \mathbf{u}_n] \text{diag}(\sigma_1, \dots, \sigma_r, 0, \dots, 0) \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_n^T \end{bmatrix}$$

**Lemma 11.17.** *Let  $A \in \mathbb{R}^{n \times n}$  be symmetric. Then (a) all the eigenvalues of  $A$  are real and (b) eigenvectors corresponding to distinct eigenvalues are orthogonal.*

**Example 11.18.** Find the SVD for  $A = \begin{bmatrix} 1 & 2 \\ -2 & 1 \\ 3 & 2 \end{bmatrix}$ .

**Solution.**

$$1. A^T A = \begin{bmatrix} 14 & 6 \\ 6 & 9 \end{bmatrix}.$$

2. Solving  $\det(A^T A - \lambda I) = 0$  gives the eigenvalues of  $A^T A$

$$\lambda_1 = 18 \text{ and } \lambda_2 = 5,$$

of which corresponding eigenvectors are

$$\tilde{\mathbf{v}}_1 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, \tilde{\mathbf{v}}_2 = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \implies V = \begin{bmatrix} \frac{3}{\sqrt{13}} & -\frac{2}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

3.  $\sigma_1 = \sqrt{\lambda_1} = \sqrt{18} = 3\sqrt{2}$ ,  $\sigma_2 = \sqrt{\lambda_2} = \sqrt{5}$ . So

$$\Sigma = \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix}$$

$$4. \mathbf{u}_1 = \frac{1}{\sigma_1} A \mathbf{v}_1 = \frac{1}{\sqrt{18}} A \begin{bmatrix} \frac{3}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} \end{bmatrix} = \frac{1}{\sqrt{18}} \frac{1}{\sqrt{13}} \begin{bmatrix} 7 \\ -4 \\ 13 \end{bmatrix} = \begin{bmatrix} \frac{7}{\sqrt{234}} \\ -\frac{4}{\sqrt{234}} \\ \frac{13}{\sqrt{234}} \end{bmatrix}$$

$$\mathbf{u}_2 = \frac{1}{\sigma_2} A \mathbf{v}_2 = \frac{1}{\sqrt{5}} A \begin{bmatrix} \frac{-2}{\sqrt{13}} \\ \frac{3}{\sqrt{13}} \end{bmatrix} = \frac{1}{\sqrt{5}} \frac{1}{\sqrt{13}} \begin{bmatrix} 4 \\ 7 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{4}{\sqrt{65}} \\ \frac{7}{\sqrt{65}} \\ 0 \end{bmatrix}.$$

$$5. A = U \Sigma V^T = \begin{bmatrix} \frac{7}{\sqrt{234}} & \frac{4}{\sqrt{65}} \\ -\frac{4}{\sqrt{234}} & \frac{7}{\sqrt{65}} \\ \frac{13}{\sqrt{234}} & 0 \end{bmatrix} \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{13}} & \frac{2}{\sqrt{13}} \\ -\frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

## 11.3. Applications of the SVD to LS Problems

**Recall: (Definition 7.2, p. 195):** Let  $A \in \mathbb{R}^{m \times n}$ ,  $m \geq n$ , and  $\mathbf{b} \in \mathbb{R}^m$ . The **least-squares problem** is to find  $\hat{\mathbf{x}} \in \mathbb{R}^n$  which minimizes  $\|A\mathbf{x} - \mathbf{b}\|_2$ :

$$\begin{aligned}\hat{\mathbf{x}} &= \arg \min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2, \\ &\text{or, equivalently,} \\ \hat{\mathbf{x}} &= \arg \min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2^2,\end{aligned}\tag{11.35}$$

where  $\hat{\mathbf{x}}$  called a **least-squares solution** of  $A\mathbf{x} = \mathbf{b}$ .

**Note: When  $A^T A$  is invertible,** the equation  $A\mathbf{x} = \mathbf{b}$  has a unique LS solution for each  $\mathbf{b} \in \mathbb{R}^m$  (Theorem 7.5). It can be solved by the **method of normal equations**; the **unique** LS solution  $\hat{\mathbf{x}}$  is given by

$$\hat{\mathbf{x}} = (A^T A)^{-1} A^T \mathbf{b}.\tag{11.36}$$

**Recall: (Definition 7.6, p. 197):**  $(A^T A)^{-1} A^T$  is called the **pseudoinverse** of  $A$ . Let  $A = U\Sigma V^T$  be the SVD of  $A$ . Then

$$(A^T A)^{-1} A^T = V\Sigma^{-1}U^T \stackrel{\text{def}}{=} A^+.\tag{11.37}$$

**Example 11.19.** Find the **pseudoinverse** of  $A = \begin{bmatrix} 1 & 2 \\ -2 & 1 \\ 3 & 2 \end{bmatrix}$ .

**Solution.** From Example 11.18, p.320, we have

$$A = U\Sigma V^T = \begin{bmatrix} \frac{7}{\sqrt{234}} & \frac{4}{\sqrt{65}} \\ -\frac{4}{\sqrt{234}} & \frac{7}{\sqrt{65}} \\ \frac{13}{\sqrt{234}} & 0 \end{bmatrix} \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{13}} & \frac{2}{\sqrt{13}} \\ -\frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

Thus,

$$\begin{aligned}A^+ &= V\Sigma^{-1}U^T = \begin{bmatrix} \frac{3}{\sqrt{13}} & -\frac{2}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{18}} & 0 \\ 0 & \frac{1}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{7}{\sqrt{234}} & -\frac{4}{\sqrt{234}} & \frac{13}{\sqrt{234}} \\ \frac{4}{\sqrt{65}} & \frac{7}{\sqrt{65}} & 0 \end{bmatrix} \\ &= \begin{bmatrix} -\frac{1}{30} & -\frac{4}{15} & \frac{1}{6} \\ \frac{11}{45} & \frac{13}{45} & \frac{1}{9} \end{bmatrix}.\end{aligned}$$

**Question.** What if  $A^T A$  is not invertible? Although it is invertible, what if the **hypothesis space** is either too big or too small?

### Solving LS Problems by the SVD

Let  $A \in \mathbb{R}^{m \times n}$ ,  $m > n$ , with  $\text{rank}(A) = k \leq n$ .

- Suppose that the SVD of  $A$  is given, that is,

$$A = U\Sigma V^T.$$

- Since  $U$  and  $V$  are  $\ell^2$ -norm preserving, we have

$$\|Ax - \mathbf{b}\| = \|U\Sigma V^T \mathbf{x} - \mathbf{b}\| = \|\Sigma V^T \mathbf{x} - U^T \mathbf{b}\|. \quad (11.38)$$

- Define  $\mathbf{z} = V^T \mathbf{x}$  and  $\mathbf{c} = U^T \mathbf{b}$ . Then

$$\|Ax - \mathbf{b}\| = \left( \sum_{i=1}^k (\sigma_i z_i - c_i)^2 + \sum_{i=k+1}^n c_i^2 \right)^{1/2}. \quad (11.39)$$

- Thus the norm is minimized when  $z$  is chosen with

$$z_i = \begin{cases} c_i/\sigma_i, & \text{when } i \leq k, \\ \text{arbitrary,} & \text{otherwise.} \end{cases} \quad (11.40)$$

- After determining  $\mathbf{z}$ , one can find the solution as

$$\hat{\mathbf{x}} = V\mathbf{z}. \quad (11.41)$$

Then the least-squares error reads

$$\min_{\mathbf{x}} \|Ax - \mathbf{b}\| = \left( \sum_{i=k+1}^n c_i^2 \right)^{1/2} \quad (11.42)$$

**Strategy 11.20.** When  $\mathbf{z}$  is obtained as in (11.40), it is better to **choose zero** for the “arbitrary” part:

$$\mathbf{z} = [c_1/\sigma_1, c_2/\sigma_2, \dots, c_k/\sigma_k, 0, \dots, 0]^T. \quad (11.43)$$

In this case,  $\mathbf{z}$  can be written as

$$\mathbf{z} = \Sigma_k^+ \mathbf{c} = \Sigma_k^+ U^T \mathbf{b}, \quad (11.44)$$

where

$$\Sigma_k^+ = [1/\sigma_1, 1/\sigma_2, \dots, 1/\sigma_k, 0, \dots, 0]^T. \quad (11.45)$$

Thus the corresponding LS solution reads

$$\hat{\mathbf{x}} = V\mathbf{z} = V\Sigma_k^+ U^T \mathbf{b}. \quad (11.46)$$

Note that  $\hat{\mathbf{x}}$  involves **no components of the null space** of  $A$ ;  **$\hat{\mathbf{x}}$  is unique in this sense.**

**Remark 11.21.**

- **When  $\text{rank}(A) = k = n$ :** It is easy to see that

$$V\Sigma_k^+ U^T = V\Sigma^{-1} U^T, \quad (11.47)$$

which is the **pseudoinverse** of  $A$ .

- **When  $\text{rank}(A) = k < n$ :**  $A^T A$  not invertible. However,

$$A_k^+ := V\Sigma_k^+ U^T \quad (11.48)$$

plays the role of the pseudoinverse of  $A$ . Thus we will call it the  **$k$ -th pseudoinverse** of  $A$ .

**Note:** For some LS applications, although  $\text{rank}(A) = n$ , the  $k$ -th pseudoinverse  $A_k^+$ , with a small  $k < n$ , may give more reliable solutions.

**Example 11.22.** Generate a synthetic dataset in 2D to find **least-squares** solutions, using

- (a) the method of normal equations and
- (b) the SVD with various numbers of principal components.

**Solution.** Here we implement a Matlab code. You will redo it in Python; see **Exercise 11.2**.

```

----- util.m -----
1  classdef util,
2  methods(Static)
3      %-----
4      function data = get_data(npt,bx,sigma)
5          data = zeros(npt,2);
6          data(:,1) = rand(npt,1)*bx;
7          data(:,2) = max(bx/3,2*data(:,1)-bx);
8
9          r = randn(npt,1)*sigma; theta = randn(npt,1)*pi;
10         noise = r.*[cos(theta),sin(theta)];
11         data = data+noise;
12     end % indentation is not required, but an extra 'end' is.
13     %-----
14     function mysave(gcf,filename)
15         exportgraphics(gcf,filename,'Resolution',100)
16         fprintf('saved: %s\n',filename)
17     end
18     %-----
19     function A = get_A(data,n)
20         npt = size(data,1);
21         A = ones(npt,n);
22         for j=2:n
23             A(:,j) = A(:,j-1).*data;
24         end
25     end
26     %-----
27     function Y = predict_Y(X,coeff,S_mean,S_std)
28         n = numel(coeff);
29         if nargin==2, S_mean=zeros(1,n); S_std=ones(1,n); end
30         A = util.get_A(X,:),n);
31         Y = ((A-S_mean)./S_std)*coeff;
32     end
33 end,end

```

**Note:** In Matlab, you can **save multiple functions in a file**, using `classdef` and `methods(Static)`.

- The functions will be called as `class_name.function_name()`.
- **Lines 12, 17, 25, 32:** The extra 'end' is required for Matlab to distinguish functions without ambiguity.
  - You may put the extra 'end' also for stand-alone functions.
- **Line 29:** A Matlab function can be implemented so that you may call the function without some arguments using default arguments.
- **Line 30:** See how to call a class function from another function.

```

1  function [sol_PCA,S_mean,S_std] = pca_regression(A,b,npc)
2  % input: npc = the number of principal components
3
4  %% Standardization
5  %%-----
6  S_mean = mean(A); S_std = std(A);
7  if S_std(1)==0, S_std(1)=1/S_mean(1); S_mean(1)=0; end
8  AS = (A-S_mean)./S_std;
9
10 %% SVD regression, using the pseudoinverse
11 %%-----
12 [U,S,V] = svd(AS,'econ');
13 S1 = diag(S); % a column vector
14 C1 = zeros(size(S1));
15 C1(1:npc) = 1./S1(1:npc);
16 C = diag(C1); % a matrix
17
18 sol_PCA = V*C*U'*b;
19 end

```

**Note:** The standardization variables are included in **output** to be used for the prediction.

- **Line 7:** Note that  $A(:,1)=1$  so that its std must be 0.
- **Lines 13 and 16:** The function `diag()` toggles between a column vector and a diagonal matrix.
- **Line 19:** The function puts an extra 'end' at the end.

```

                                Regression_Analysis.m
1  clear all; close all;
2
3  %%-----
4  %% Setting
5  %%-----
6  regen_data = 0;  %=1, regenerate the synthetic data
7  poly_n = 9;
8  npt=300; bx=5.0; sigma=0.50;  %for synthetic data
9  datafile = 'synthetic-data.txt';
10
11 %%-----
12 %% Data: Generation and Read
13 %%-----
14 if regen_data || ~isfile(datafile)
15     DATA = util.get_data(npt,bx,sigma);
16     writematrix(DATA, datafile);
17     fprintf('%s: re-generated.\n',datafile)
18 end
19 DATA = readmatrix(datafile,"Delimiter","",");
20
21 %%-----
22 %% The system: A x = b
23 %%-----
24 A = util.get_A(DATA(:,1),poly_n+1);
25 b = DATA(:,2);
26
27 %%-----
28 %% Method of Noral Equations
29 %%-----
30 sol_NE = (A'*A)\(A'*b);
31 figure,
32 plot(DATA(:,1),DATA(:,2),'k.', 'MarkerSize',8);
33 axis tight; hold on
34 yticks(1:5); ax = gca; ax.FontSize=13; %ax.GridAlpha=0.25
35 title(sprintf('Synthetic Data: npt = %d',npt),'fontsize',13)
36 util.mysave(gcf, 'data-synthetic.png');
37 x=linspace(min(DATA(:,1)),max(DATA(:,1)),51);
38 plot(x,util.predict_Y(x,sol_NE),'r-', 'linewidth',2);
39 Pn = ['P_',int2str(poly_n)];
40 legend('data',Pn, 'location','best','fontsize',13)
41 TITLE0=sprintf('Method of NE: npt = %d',npt);
42 title(TITLE0,'fontsize',13)
43 hold off

```



```

44     util.mysave(gcf,'data-synthetic-sol-NE.png');
45
46     %%-----
47     %% PCA Regression
48     %%-----
49     for npc=1:size(A,2);
50         [sol_PCA,S_mean,S_std] = pca_regression(A,b,npc);
51         figure,
52         plot(DATA(:,1),DATA(:,2),'k.','MarkerSize',8);
53         axis tight; hold on
54         yticks(1:5); ax = gca; ax.FontSize=13; %ax.GridAlpha=0.25
55         x=linspace(min(DATA(:,1)),max(DATA(:,1)),51);
56         plot(x,util.predict_Y(x,sol_PCA,S_mean,S_std),'r-','linewidth',2);
57         Pn = ['P_',int2str(poly_n)];
58         legend('data',Pn, 'location','best','fontsize',13)
59         TITLE0=sprintf('Method of PC: npc = %d',npc);
60         title(TITLE0,'fontsize',13)
61         hold off
62         savefile = sprintf('data-sol-PCA-npc-%02d.png',npc);
63         util.mysave(gcf,savefile);
64     end

```

**Note:** Regression\_Analysis is the main function. The code is simple; the complication is due to plotting.

- **Lines 6, 14-19:** Data is read from a datafile.
  - Setting `regen_data = 1` will regenerate the datafile.

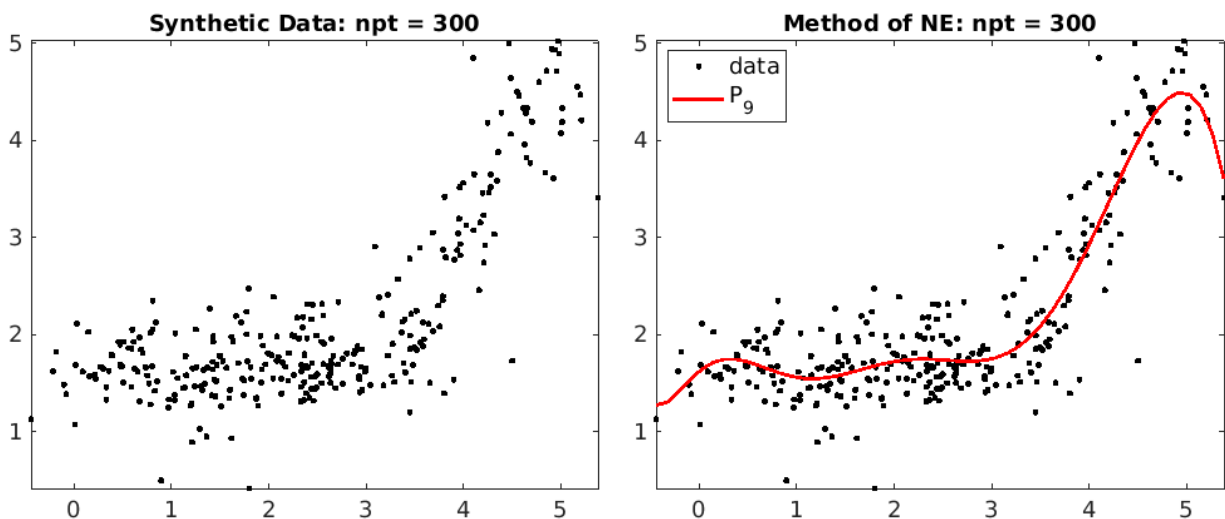


Figure 11.3: The synthetic data and the LS solution  $P_9(x)$ , overfitted.

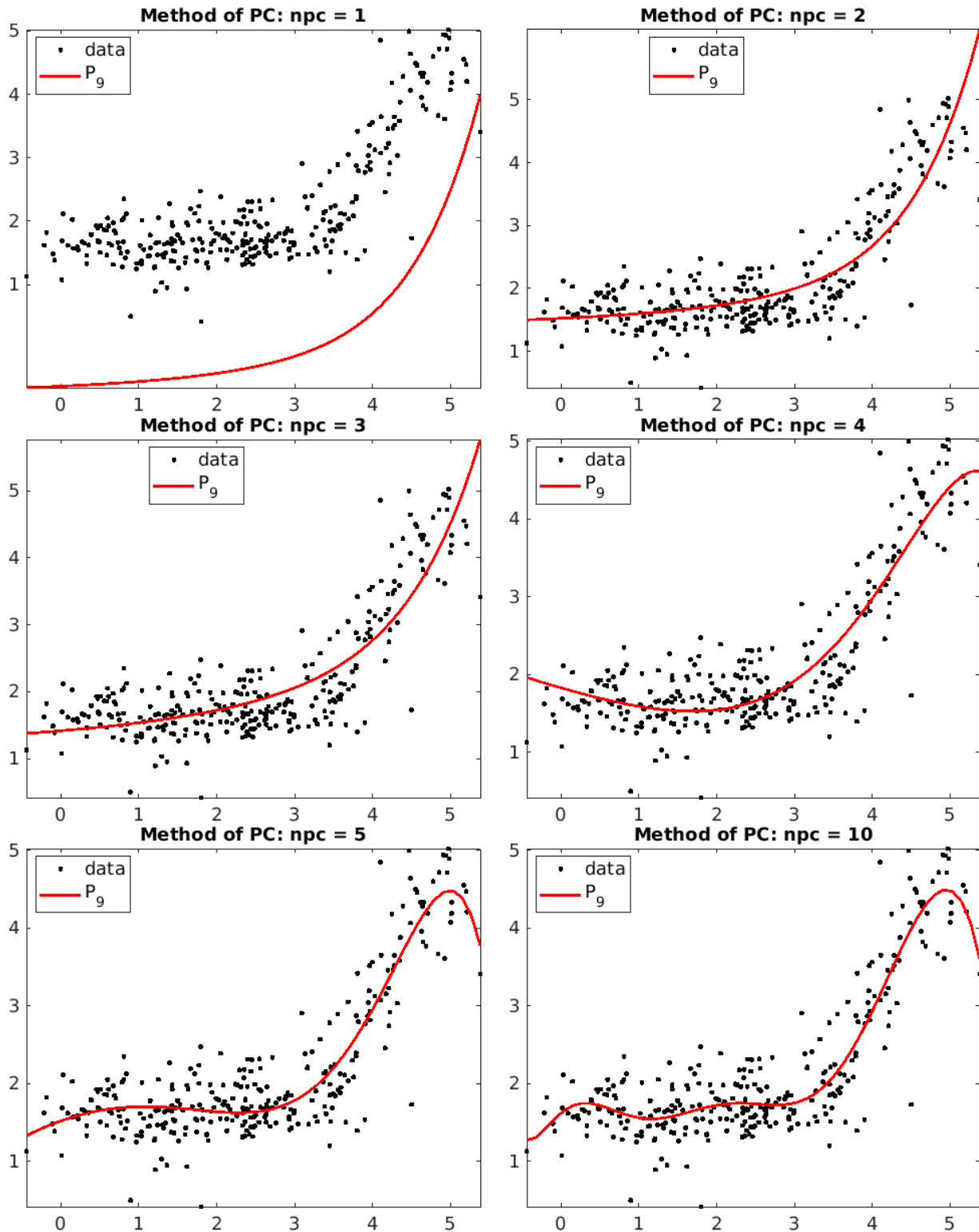


Figure 11.4: PCA regression of the data, with various numbers of principal components. The best regression is achieved when  $\text{npc} = 3$ .

## Exercises for Chapter 11

11.1. Download `wine.data` from the UCI database:

<https://archive.ics.uci.edu/ml/machine-learning-databases/wine/>

The data is extensively used in the Machine Learning community. The first column of the data is the label and the others are features of three different kinds of wines.

(a) Add lines to the code given, to verify (11.20), p.312. For example, set  $k = 5$ .

```

----- Wine_data.py -----
1  import numpy as np
2  from numpy import diag,dot
3  from scipy.linalg import svd,norm
4  import matplotlib.pyplot as plt
5
6  data = np.loadtxt('wine.data', delimiter=',')
7  X = data[:,1:]; y = data[:,0]
8
9  #-----
10 # Standardization
11 #-----
12 X_mean, X_std = np.mean(X,axis=0), np.std(X,axis=0)
13 XS = (X - X_mean)/X_std
14
15 #-----
16 # SVD
17 #-----
18 U, s, VT = svd(XS)
19 if U.shape[0]==U.shape[1]:
20     U = U[:, :len(s)] # cut the nonnecessary
21 Sigma = diag(s)      # transform to a matrix
22 print('U:',U.shape, 'Sigma:',Sigma.shape, 'VT:',VT.shape)

```

### Note:

- **Line 12:** `np.mean` and `np.std` are applied, with the option `axis=0`, to get the quantities column-by-column *vertically*. Thus `X_mean` and `X_std` are row vectors.
- **Line 18:** In Python, `svd` produces `[U, s, VT]`, where  $VT = V^T$ . If you would like to get  $V$ , then  $V = VT.T$ .

11.2. Implement the code in Example 11.22, in **Python**.

- Report your complete code.
- Attached figures as in Figures 11.3 and 11.4.

**Clue:** The major reason that a class is used in the Matlab code in Example 11.22 is to combine multiple functions to be saved in a file. In Python, you do not have to use a class to save multiple functions in a file. You may start with the following.

```

----- util.py -----
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def get_data(npt,bx,sigma):
5      data = np.zeros([npt,2]);
6      data[:,0] = np.random.uniform(0,1,npt)*bx;
7      data[:,1] = np.maximum(bx/3,2*data[:,0]-bx);
8      r = np.random.normal(0,1,npt)*sigma;
9      theta = np.random.normal(0,1,npt)*np.pi;
10     noise = np.column_stack((r*np.cos(theta),r*np.sin(theta)));
11     data += noise;
12     return data
13
14 def mysave(filename):
15     plt.savefig(filename,bbox_inches='tight')
16     print('saved:',filename)
17
18 # Add other functions

```

```

----- Regression_Analysis.py -----
1  import numpy as np
2  import numpy.linalg as la
3  import matplotlib.pyplot as plt
4  from os.path import exists
5  import util
6
7  ##-----
8  ## Setting
9  ##-----
10     regen_data = 1;  #==1, regenerate the synthetic data
11     poly_n = 9;
12     npt=300; bx=5.0; sigma=0.50;  #for synthetic data
13     datafile = 'synthetic-data.txt';
14     plt.style.use('ggplot')
15
16     ##-----
17     ## Data: Generation and Read
18     ##-----
19     if regen_data or not exists(datafile):
20         DATA = util.get_data(npt,bx,sigma);
21         np.savetxt(datafile,DATA,delimiter=',');

```

```
22     print('%s: re-generated.' %(datafile))
23
24     DATA = np.loadtxt(datafile, delimiter=',')
25
26     plt.figure() # initiate a new plot
27     plt.scatter(DATA[:,0],DATA[:,1],s=8,c='k')
28     plt.title('Synthetic Data: npc = '+ str(npc))
29     util.mysave('data-synthetic-py.png')
30     #plt.show()
31
32     ##-----
33     ## The system: A x = b
34     ##-----
```

**Note:** The semi-colons (;) are not necessary in Python nor harmful; they are included from copy-and-paste of Matlab lines. The **ggplot** style emulates “ggplot”, a popular plotting package for R. When `Regression_Analysis.py` is executed, you will have a saved image:

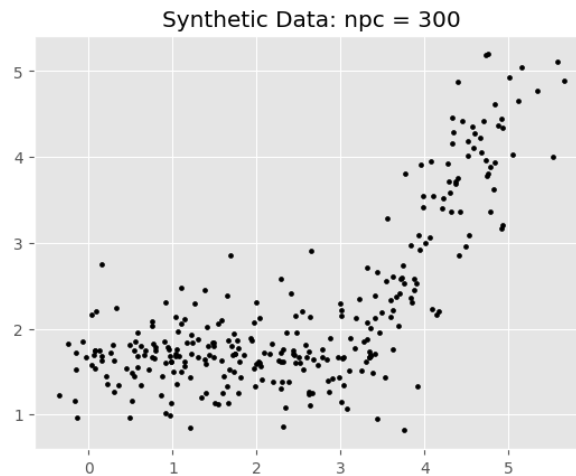


Figure 11.5: data-synthetic-py.png



# APPENDIX **A**

## **Appendices**

### **Contents of Chapter A**

A.1. Optimization: Primal and Dual Problems . . . . .	334
A.2. Weak Duality, Strong Duality, and Complementary Slackness . . . . .	338
A.3. Geometric Interpretation of Duality . . . . .	342
A.4. Rank-One Matrices and Structure Tensors . . . . .	349
A.5. Boundary-Effects in Convolution Functions in Matlab and Python SciPy . . . . .	353
A.6. From Python, Call C, C++, and Fortran . . . . .	357

## A.1. Optimization: Primal and Dual Problems

### A.1.1. The Lagrangian

**Problem A.1.** Consider a **general optimization problem** of the form

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subj.to} \quad & h_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, m \quad (\text{Primal}) \\ & q_j(\mathbf{x}) = 0, \quad j = 1, \dots, p \end{aligned} \quad (\text{A.1.1})$$

We define its **Lagrangian**  $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \rightarrow \mathbb{R}$  as

$$\begin{aligned} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) &= f(\mathbf{x}) + \sum_{i=1}^m \alpha_i h_i(\mathbf{x}) + \sum_{j=1}^p \beta_j q_j(\mathbf{x}) \\ &= f(\mathbf{x}) + \boldsymbol{\alpha} \cdot \mathbf{h}(\mathbf{x}) + \boldsymbol{\beta} \cdot \mathbf{q}(\mathbf{x}), \end{aligned} \quad (\text{A.1.2})$$

where  $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_m) \geq 0$  and  $\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_p)$  are **Lagrange multipliers**.

**Definition A.2.** The set of points that satisfy the constraints,

$$\mathcal{C} \stackrel{\text{def}}{=} \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{h}(\mathbf{x}) \leq 0 \text{ and } \mathbf{q}(\mathbf{x}) = 0\}, \quad (\text{A.1.3})$$

is called the **feasible set**.

**Lemma A.3.** For each  $\mathbf{x}$  in the feasible set  $\mathcal{C}$ ,

$$f(\mathbf{x}) = \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}), \quad \mathbf{x} \in \mathcal{C}. \quad (\text{A.1.4})$$

The maximum is taken iff  $\boldsymbol{\alpha}$  satisfies

$$\alpha_i h_i(\mathbf{x}) = 0, \quad i = 1, \dots, m. \quad (\text{A.1.5})$$

**Proof.** When  $\mathbf{x} \in \mathcal{C}$ , we have  $\mathbf{h}(\mathbf{x}) \leq 0$  and  $\mathbf{q}(\mathbf{x}) = 0$  and therefore

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = f(\mathbf{x}) + \boldsymbol{\alpha} \cdot \mathbf{h}(\mathbf{x}) + \boldsymbol{\beta} \cdot \mathbf{q}(\mathbf{x}) \leq f(\mathbf{x})$$

Clearly, the last inequality becomes equality iff (A.1.5) holds.  $\square$



**Remark A.4.** Recall  $\mathcal{L}(\mathbf{x}, \alpha, \beta) = f(\mathbf{x}) + \alpha \cdot \mathbf{h}(\mathbf{x}) + \beta \cdot \mathbf{q}(\mathbf{x})$  and  $\mathcal{C} = \{\mathbf{x} \mid \mathbf{h}(\mathbf{x}) \leq 0 \text{ and } \mathbf{q}(\mathbf{x}) = 0\}$ . It is not difficult to see

$$\max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta) = \infty, \quad \mathbf{x} \notin \mathcal{C}. \quad (\text{A.1.6})$$

**Theorem A.5.** Let  $f^*$  be the optimal value of the primal problem (A.1.1):

$$f^* = \min_{\mathbf{x} \in \mathcal{C}} f(\mathbf{x}).$$

Then  $f^*$  satisfies

$$f^* = \min_{\mathbf{x}} \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta). \quad (\text{A.1.7})$$

**Note:** The minimum in (A.1.7) **does not require**  $\mathbf{x}$  in  $\mathcal{C}$ .

**Proof.** For  $\mathbf{x} \in \mathcal{C}$ , it follows from (A.1.4) that

$$f^* = \min_{\mathbf{x} \in \mathcal{C}} f(\mathbf{x}) = \min_{\mathbf{x} \in \mathcal{C}} \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta). \quad (\text{A.1.8})$$

When  $\mathbf{x} \notin \mathcal{C}$ , since (A.1.6) holds, we have

$$\min_{\mathbf{x} \notin \mathcal{C}} \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta) = \infty. \quad (\text{A.1.9})$$

The assertion (A.1.7) follows from (A.1.8) and (A.1.9).  $\square$

### Summary A.6. Primal Problem

The primal problem (A.1.1) is equivalent to the **minimax problem**

$$\min_{\mathbf{x}} \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta), \quad (\text{Primal}) \quad (\text{A.1.10})$$

where

$$\mathcal{L}(\mathbf{x}, \alpha, \beta) = f(\mathbf{x}) + \alpha \cdot \mathbf{h}(\mathbf{x}) + \beta \cdot \mathbf{q}(\mathbf{x}).$$

Here the minimum does **not** require  $\mathbf{x}$  in the feasible set  $\mathcal{C}$ .

## A.1.2. Lagrange Dual Problem

Given a Lagrangian  $\mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ , we define its **Lagrange dual function** as

$$\begin{aligned} g(\boldsymbol{\alpha}, \boldsymbol{\beta}) &\stackrel{\text{def}}{=} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \\ &= \min_{\mathbf{x}} \{f(\mathbf{x}) + \boldsymbol{\alpha} \cdot \mathbf{h}(\mathbf{x}) + \boldsymbol{\beta} \cdot \mathbf{q}(\mathbf{x})\}. \end{aligned} \quad (\text{A.1.11})$$

**Claim A.7. Lower Bound Property**

$$g(\boldsymbol{\alpha}, \boldsymbol{\beta}) \leq f^*, \quad \text{for } \boldsymbol{\alpha} \geq 0. \quad (\text{A.1.12})$$

**Proof.** Let  $\boldsymbol{\alpha} \geq 0$ . Then for  $\mathbf{x} \in \mathcal{C}$ ,

$$f(\mathbf{x}) \geq \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \geq \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = g(\boldsymbol{\alpha}, \boldsymbol{\beta}).$$

Minimizing over all feasible points  $\mathbf{x}$  gives  $f^* \geq g(\boldsymbol{\alpha}, \boldsymbol{\beta})$ .  $\square$

**Definition A.8.** Given primal problem (A.1.1), we define its **Lagrange dual problem** as

$$\begin{aligned} &\max_{\boldsymbol{\alpha}, \boldsymbol{\beta}} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \\ &\text{subj.to } \boldsymbol{\alpha} \geq 0 \end{aligned} \quad \text{(Dual)} \quad (\text{A.1.13})$$

Thus the dual problem is a **maximin problem**.

**Remark A.9.** It is clear to see from the definition, the optimal value of the dual problem, named as  $g^*$ , satisfies

$$g^* = \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}). \quad (\text{A.1.14})$$

**Although the primal problem is not convex, the dual problem is always convex** (actually, concave).

**Theorem A.10.** *The dual problem (A.1.13) is a **convex optimization problem**. Thus **it is easy to optimize**.*

**Proof.** From the definition,

$$g(\alpha, \beta) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha, \beta) = \min_{\mathbf{x}} \{f(\mathbf{x}) + \alpha \cdot h(\mathbf{x}) + \beta \cdot q(\mathbf{x})\},$$

which can be viewed as pointwise infimum of **affine functions** of  $\alpha$  and  $\beta$ . Thus it is concave. Hence the dual problem is a **concave maximization problem**, which is a convex optimization problem.  $\square$

**Summary A.11.** Given the optimization problem (A.1.1):

- It is equivalent to the **minimax problem**

$$\min_{\mathbf{x}} \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta), \quad (\text{Primal}) \quad (\text{A.1.15})$$

where the **Lagrangian** is defined as

$$\mathcal{L}(\mathbf{x}, \alpha, \beta) = f(\mathbf{x}) + \alpha \cdot h(\mathbf{x}) + \beta \cdot q(\mathbf{x}). \quad (\text{A.1.16})$$

- Its dual problem is a **maximin problem**

$$\max_{\alpha \geq 0, \beta} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha, \beta), \quad (\text{Dual}) \quad (\text{A.1.17})$$

and the **dual function** is defined as

$$g(\alpha, \beta) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha, \beta). \quad (\text{A.1.18})$$

- **The Lagrangian and Duality**

- The **Lagrangian** is a lower bound of the objective function.

$$f(\mathbf{x}) \geq \mathcal{L}(\mathbf{x}, \alpha, \beta), \quad \text{for } \mathbf{x} \in \mathcal{C}, \quad \alpha \geq 0. \quad (\text{A.1.19})$$

- The **dual function** is a lower bound of the the primal optimal.

$$g(\alpha, \beta) \leq f^*. \quad (\text{A.1.20})$$

- The dual problem is a **convex optimization problem**.

## A.2. Weak Duality, Strong Duality, and Complementary Slackness

**Recall:** For an **optimization problem** of the form

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subj.to} \quad & h_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, m \\ & q_j(\mathbf{x}) = 0, \quad j = 1, \dots, p \end{aligned} \quad \text{(Primal)} \quad \text{(A.2.1)}$$

the **Lagrangian**  $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \rightarrow \mathbb{R}$  is defined as

$$\begin{aligned} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) &= f(\mathbf{x}) + \sum_{i=1}^m \alpha_i h_i(\mathbf{x}) + \sum_{j=1}^p \beta_j q_j(\mathbf{x}) \\ &= f(\mathbf{x}) + \boldsymbol{\alpha} \cdot \mathbf{h}(\mathbf{x}) + \boldsymbol{\beta} \cdot \mathbf{q}(\mathbf{x}), \end{aligned} \quad \text{(A.2.2)}$$

where  $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_m) \geq 0$  and  $\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_p)$  are **Lagrange multipliers**.

- The problem (A.2.1) is equivalent to the **minimax problem**

$$\min_{\mathbf{x}} \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}). \quad \text{(Primal)} \quad \text{(A.2.3)}$$

- Its dual problem is a **maximin problem**

$$\max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}), \quad \text{(Dual)} \quad \text{(A.2.4)}$$

and the **dual function** is defined as

$$g(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}). \quad \text{(A.2.5)}$$

### A.2.1. Weak Duality

**Theorem A.12.** *The dual problem yields a lower bound for the primal problem. That is, the minimax  $f^*$  is greater or equal to the maximin  $g^*$ :*

$$f^* = \min_{\mathbf{x}} \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta) \geq \max_{\alpha \geq 0, \beta} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha, \beta) = g^* \quad (\text{A.2.6})$$

**Proof.** Let  $\mathbf{x}^*$  be the minimizer, the primal optimal. Then

$$\mathcal{L}(\mathbf{x}, \alpha, \beta) \geq \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha, \beta) = \mathcal{L}(\mathbf{x}^*, \alpha, \beta), \quad \forall \mathbf{x}, \alpha \geq 0, \beta.$$

Let  $(\alpha^*, \beta^*)$  be the maximizer, the dual optimal. Then

$$\mathcal{L}(\mathbf{x}, \alpha^*, \beta^*) = \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta) \geq \mathcal{L}(\mathbf{x}, \alpha, \beta), \quad \forall \mathbf{x}, \alpha \geq 0, \beta.$$

It follows from the two inequalities that for all  $\mathbf{x}, \alpha \geq 0, \beta$ ,

$$\mathcal{L}(\mathbf{x}, \alpha^*, \beta^*) = \max_{\alpha \geq 0, \beta} \mathcal{L}(\mathbf{x}, \alpha, \beta) \geq \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \alpha, \beta) = \mathcal{L}(\mathbf{x}^*, \alpha, \beta). \quad (\text{A.2.7})$$

Notice that the left side depends on  $\mathbf{x}$ , while the right side is a function of  $(\alpha, \beta)$ . The inequality holds true for all  $\mathbf{x}, \alpha \geq 0, \beta$ .

$\Rightarrow$  We may take  $\min_{\mathbf{x}}$  and  $\max_{\alpha \geq 0, \beta}$  respectively to the left side and the right side, to conclude (A.2.6).  $\square$

**Definition A.13. Weak and Strong Duality**

- (a) It always holds true that  $f^* \geq g^*$ , called as **weak duality**.
- (b) In some problems, we actually have  $f^* = g^*$ , which is called **strong duality**.

## A.2.2. Strong Duality

### **Theorem** A.14. **Slater's Theorem**

If the primal is a convex problem, and there exists at least one strictly feasible  $\tilde{\mathbf{x}}$ , satisfying the **Slater's condition**:

$$h(\tilde{\mathbf{x}}) < 0 \quad \text{and} \quad q(\tilde{\mathbf{x}}) = 0, \quad (\text{A.2.8})$$

then strong duality holds.

A conception having close relationship with strong duality is the duality gap.

**Definition** A.15. Given primal feasible  $\mathbf{x}$  and dual feasible  $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ , the quantity

$$f(\mathbf{x}) - g(\boldsymbol{\alpha}, \boldsymbol{\beta}) = f(\mathbf{x}) - \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \quad (\text{A.2.9})$$

is called the **duality gap**.

From the weak duality, we have

$$f(\mathbf{x}) - g(\boldsymbol{\alpha}, \boldsymbol{\beta}) \geq f^* - g^* \geq 0$$

Furthermore, we declare a sufficient and necessary condition for duality gap equal to 0.

**Proposition** A.16. With  $\mathbf{x}$ ,  $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ , the duality gap equals to 0 iff

- (a)  $\mathbf{x}$  is the primal optimal solution,
- (b)  $(\boldsymbol{\alpha}, \boldsymbol{\beta})$  is the dual optimal solution, and
- (c) the strong duality holds.

**Proof.** From definitions and the weak duality, we have

$$f(\mathbf{x}) \geq f^* \geq g^* \geq g(\boldsymbol{\alpha}, \boldsymbol{\beta}).$$

The duality gap equals to 0, iff the three inequalities become equalities.  $\square$

### A.2.3. Complementary Slackness

Assume that strong duality holds,  $\mathbf{x}^*$  is the primal optimal, and  $(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)$  is the dual optimal. Then

$$\begin{aligned}
 f(\mathbf{x}^*) = g(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*) &\stackrel{\text{def}}{=} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*) \\
 &= \min_{\mathbf{x}} \left\{ f(\mathbf{x}) + \sum_{i=1}^m \alpha_i^* h_i(\mathbf{x}) + \sum_{j=1}^p \beta_j^* q_j(\mathbf{x}) \right\} \\
 &\leq f(\mathbf{x}^*) + \sum_{i=1}^m \alpha_i^* h_i(\mathbf{x}^*) + \sum_{j=1}^p \beta_j^* q_j(\mathbf{x}^*) \\
 &\leq f(\mathbf{x}^*),
 \end{aligned} \tag{A.2.10}$$

hence two inequalities hold with equality.

- The primal optima  $\mathbf{x}^*$  minimizes  $\mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)$ .
- The **complementary slackness** holds:

$$\alpha_i^* h_i(\mathbf{x}^*) = 0, \quad \text{for all } i = 1, \dots, m, \tag{A.2.11}$$

which implies that

$$\alpha_i^* > 0 \Rightarrow h_i(\mathbf{x}^*) = 0, \quad h_i(\mathbf{x}^*) < 0 \Rightarrow \alpha_i^* = 0. \tag{A.2.12}$$

**Note: Complementary slackness** says that

- If a dual variable is greater than zero (slack/loose), then the corresponding primal constraint must be an equality (tight.)
- If the primal constraint is slack, then the corresponding dual variable is tight (or zero).

**Remark A.17.** **Complementary slackness** is key to designing **primal-dual algorithms**. The basic idea is

1. Start with a feasible dual solution  $\alpha$ .
2. Attempt to find primal feasible  $\mathbf{x}$  such that  $(\mathbf{x}, \alpha)$  satisfy complementary slackness.
3. If Step 2 succeeded, we are done; otherwise the misfit on  $\mathbf{x}$  gives a way to modify  $\alpha$ . Repeat.

## A.3. Geometric Interpretation of Duality

**Recall:** For an **optimization problem** of the form

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subj.to} \quad & h_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, m \\ & q_j(\mathbf{x}) = 0, \quad j = 1, \dots, p \end{aligned} \quad \text{(Primal)} \quad \text{(A.3.1)}$$

the **Lagrangian**  $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \rightarrow \mathbb{R}$  is defined as

$$\begin{aligned} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) &= f(\mathbf{x}) + \sum_{i=1}^m \alpha_i h_i(\mathbf{x}) + \sum_{j=1}^p \beta_j q_j(\mathbf{x}) \\ &= f(\mathbf{x}) + \boldsymbol{\alpha} \cdot \mathbf{h}(\mathbf{x}) + \boldsymbol{\beta} \cdot \mathbf{q}(\mathbf{x}), \end{aligned} \quad \text{(A.3.2)}$$

where  $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_m) \geq 0$  and  $\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_p)$  are **Lagrange multipliers**.

- The problem (A.3.1) is equivalent to the **minimax problem**

$$\min_{\mathbf{x}} \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}). \quad \text{(Primal)} \quad \text{(A.3.3)}$$

- Its dual problem is a **maximin problem**

$$\max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}), \quad \text{(Dual)} \quad \text{(A.3.4)}$$

and the **dual function** is defined as

$$g(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}). \quad \text{(A.3.5)}$$

**Definition A.18.** Given a primal problem (A.2.1), we define its **epigraph (supergraph)** as

$$\mathcal{A} = \{(r, s, t) \mid \mathbf{h}(\mathbf{x}) \leq r, \quad \mathbf{q}(\mathbf{x}) = s, \quad f(\mathbf{x}) \leq t, \text{ for some } \mathbf{x} \in \mathbb{R}^n\}. \quad \text{(A.3.6)}$$



**Geometric-interpretation A.19.** Here are the geometric interpretation of several key values.

(a)  $f^*$  is the lowest projection of  $\mathcal{A}$  to the  $t$ -axis:

$$\begin{aligned} f^* &= \min\{t \mid \mathbf{h}(\mathbf{x}) \leq \mathbf{0}, \mathbf{q}(\mathbf{x}) = \mathbf{0}, f(\mathbf{x}) \leq t\} \\ &= \min\{t \mid (\mathbf{0}, \mathbf{0}, t) \in \mathcal{A}\}. \end{aligned} \quad (\text{A.3.7})$$

(b)  $g(\boldsymbol{\alpha}, \boldsymbol{\beta})$  is the intersection of the  $t$ -axis and a hyperplane of normal vector  $(\boldsymbol{\alpha}, \boldsymbol{\beta}, 1)$ :

$$\begin{aligned} g(\boldsymbol{\alpha}, \boldsymbol{\beta}) &\stackrel{\text{def}}{=} \min_{\mathbf{x}} \{f(\mathbf{x}) + \boldsymbol{\alpha} \cdot \mathbf{h}(\mathbf{x}) + \boldsymbol{\beta} \cdot \mathbf{q}(\mathbf{x})\} \\ &= \min \{(\boldsymbol{\alpha}, \boldsymbol{\beta}, 1)^T(\mathbf{r}, \mathbf{s}, t) \mid (\mathbf{r}, \mathbf{s}, t) \in \mathcal{A}\}. \end{aligned} \quad (\text{A.3.8})$$

This is referred to as a **nonvertical supporting hyperplane**, because the last component of the normal vector is nonzero (it is 1).

(c)  $g^*$  is the highest intersection of the  $t$ -axis and all nonvertical supporting hyperplanes of  $\mathcal{A}$ . Notice that  $\alpha \geq 0$  holds true for each nonvertical supporting hyperplane of  $\mathcal{A}$ .

From the geometric interpretation of  $f^*$  and  $g^*$ , we actually have an equivalent geometric statement of strong duality:

**Theorem A.20.** *The strong duality holds, iff there exists a nonvertical supporting hyperplane of  $\mathcal{A}$  passing through  $(0, 0, f^*)$ .*

**Proof.** From weak duality  $f^* \geq g^*$ , the intersection of the  $t$ -axis and a nonvertical supporting hyperplane cannot exceed  $(0, 0, f^*)$ . The strong duality holds, i.e.,  $f^* = g^*$ , iff  $(0, 0, f^*)$  is just the highest intersection, meaning that there exists a nonvertical supporting hyperplane of  $\mathcal{A}$  passing through  $(0, 0, f^*)$ .  $\square$

**Example A.21.** Solve a simple inequality-constrained **convex problem**

$$\begin{aligned} \min_x \quad & x^2 + 1 \\ \text{subj.to} \quad & x \geq 1. \end{aligned} \tag{A.3.9}$$

**Solution.** A code is implemented **to draw a figure**, shown at the end of the solution.

- **Lagrangian:** The inequality constraint can be written as  $-x + 1 \leq 0$ . Thus the **Lagrangian** reads

$$\begin{aligned} \mathcal{L}(x, \alpha) &= x^2 + 1 + \alpha(-x + 1) = \mathbf{x^2 - \alpha x + \alpha + 1} \\ &= \left(x - \frac{\alpha}{2}\right)^2 - \frac{\alpha^2}{4} + \alpha + 1, \end{aligned} \tag{A.3.10}$$

and therefore the **dual function** reads (when  $x = \alpha/2$ )

$$g(\alpha) = \min_x \mathcal{L}(x, \alpha) = -\frac{\alpha^2}{4} + \alpha + 1. \tag{A.3.11}$$

**Remark A.22. The Solution of  $\min_x \mathcal{L}(x, \alpha)$**

- We may obtain it by applying a **calculus technique**:

$$\frac{\partial}{\partial x} \mathcal{L}(x, \alpha) = 2x - \alpha = 0, \tag{A.3.12}$$

and therefore  $x = \alpha/2$  and (A.3.11) follows.

Equation (A.3.12) is one of the **Karush-Kuhn-Tucker (KKT) conditions**, the **first-order necessary conditions**, which **defines the relationship between the primal variable ( $x$ ) and the dual variable ( $\alpha$ )**.

- Using the KKT condition, (A.3.11) defines the dual function  $g(\alpha)$  as a function of the dual variable ( $\alpha$ ).
- The dual function  $g(\alpha)$  is **concave**, while the Lagrangian is an affine function of  $\alpha$ .

- **Epigraph:** For the convex problem (A.3.9), its epigraph is defined as

$$\mathcal{A} = \{(r, t) \mid -x + 1 \leq r, \quad x^2 + 1 \leq t, \quad \text{for } x \in \mathbb{R}\}. \quad (\text{A.3.13})$$

To find the edge of the epigraph, we replace inequalities with equalities:

$$-x + 1 = r, \quad x^2 + 1 = t \quad (\text{A.3.14})$$

and define  $t$  as a function of  $r$ :

$$t = x^2 + 1 = (-r + 1)^2 + 1. \quad (\text{A.3.15})$$

See Figure A.1, where the shaded region is the **epigraph** of the problem.

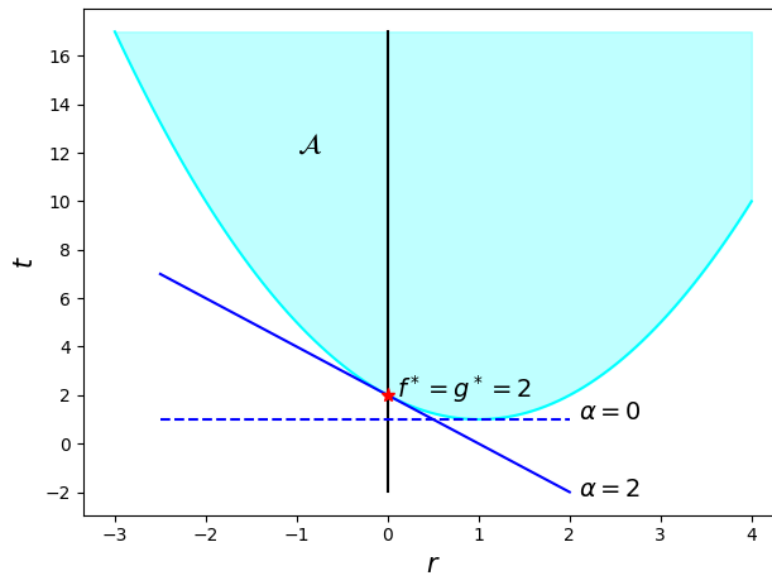


Figure A.1: The epigraph of the convex problem (A.3.9), the shaded region, and strong duality.

**The Primal Optimal:** For a feasible point  $x$ ,

$$-x + 1 \leq 0 \quad \Rightarrow \quad r = -x + 1 \leq 0.$$

Thus the left side of the  $t$ -axis in  $\mathcal{A}$  corresponds to the feasible set; it follows from (A.3.15) that

$$f^* = \min \{t \mid (0, t) \in \mathcal{A}\} = 2. \quad (\text{A.3.16})$$

- **Nonvertical Supporting Hyperplanes:** For the convex problem, it follows from a Geometric-interpretation (A.3.8) that

$$g(\alpha) = \min_{(r,t) \in \mathcal{A}} \{\alpha r + t\}. \quad (\text{A.3.17})$$

For each  $(r, t)$ , the above reads

$$\alpha r + t = g(\alpha) = -\frac{\alpha^2}{4} + \alpha + 1,$$

where (A.3.11) is used. Thus we can define a family of **nonvertical supporting hyperplanes** as

$$t = -\alpha r - \frac{\alpha^2}{4} + \alpha + 1, \quad (\text{A.3.18})$$

which is a line in the  $(r, t)$ -coordinates for a fixed  $\alpha$ . Figure A.1 depicts two of the lines:  $\alpha = 0$  and  $\alpha = 2$ .

- **Strong Duality:** Note that on the  $t$ -axis ( $r = 0$ ), (A.3.18) reads

$$t = -\frac{\alpha^2}{4} + \alpha + 1 = -\frac{1}{4}(\alpha - 2)^2 + 2, \quad (\text{A.3.19})$$

of which the maximum  $g^* = 2$  when  $\alpha = 2$ . Thus we can conclude

$$f^* = g^* = 2; \quad (\text{A.3.20})$$

**strong duality** holds for the convex problem.  $\square$

```

----- duality_convex.py -----
1  import numpy as np
2  from matplotlib import pyplot as plt
3
4  # Convex: min f(x), s.t. x >= 1 (i.e., -x+1 <= 0)
5  def f(x): return x**2+1
6  def g(r,alpha): return -alpha*r+(-alpha**2/4+alpha+1)
7
8  #--- Epigraph: t= f(r)
9  #-----
10 r = np.linspace(-3,4,100); x = -r+1
11 t = f(x); mint = t.min(); maxt = t.max()
12
13 plt.fill_between(r,t,maxt,color='cyan',alpha=0.25)
14 plt.plot(r,t,color='cyan')
15 plt.xlabel(r'$r$',fontsize=15); plt.ylabel(r'$t$',fontsize=15)
16 plt.text(-1,12,r'$\cal A$',fontsize=16)
17 plt.plot([0,0],[mint-3,maxt],color='black',ls='-') # t-axis
18 plt.yticks(np.arange(-2,maxt,2)); plt.tight_layout()
19
20 #--- Two Supporting hyperplanes
21 #-----
22 r = np.linspace(-2.5,2,2)
23 plt.plot(r,g(r,2),color='blue',ls='-')
24 plt.plot(r,g(r,0),color='blue',ls='--')
25
26 #--- Add Texts
27 #-----
28 p=2.1
29 plt.text(p,g(p,0),r'$\alpha=0$',fontsize=14)
30 plt.text(p,g(p,2),r'$\alpha=2$',fontsize=14) # the optimal
31 plt.plot(0,2,'r*',markersize=8)
32 plt.text(0.1,1.9,r'$f^*=g^*=2$',fontsize=14)
33
34 plt.savefig('png-duality-example.png',bbox_inches='tight')
35 plt.show()

```

**Example A.23.** Solve the following **nonconvex problem**

$$\begin{aligned} \min_x \quad & x^4 - 50x^2 + 25x \\ \text{subj.to} \quad & x \geq -2. \end{aligned} \tag{A.3.21}$$

**Solution.** For the nonconvex problem, a code is implemented similar to `duality_convex.py` on p.347.

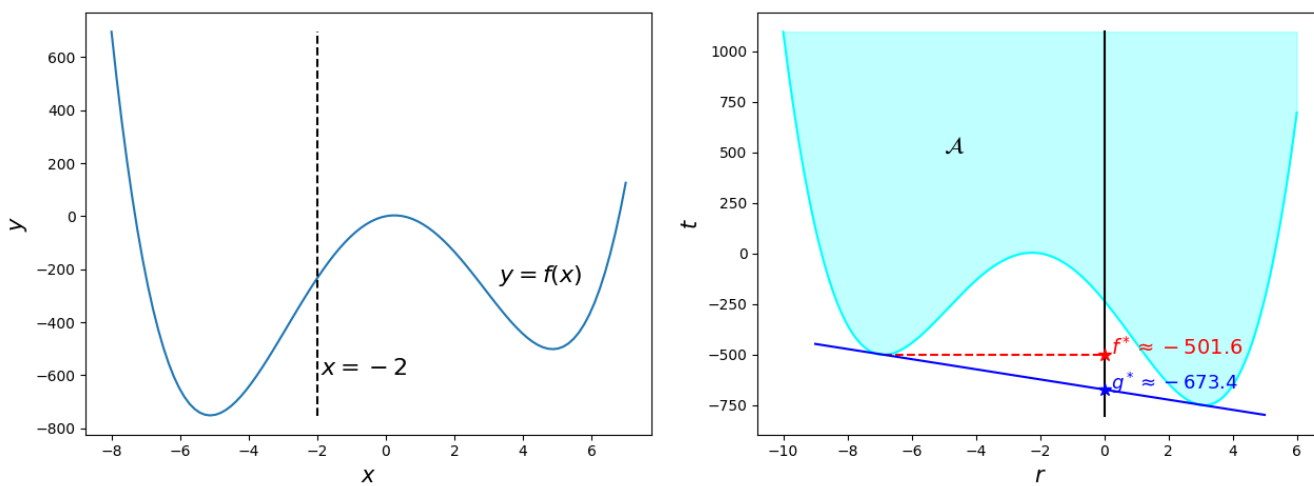


Figure A.2: The nonconvex problem: (left) The graph of  $y = f(x)$  and (right) the epigraph and weak duality.

- The **Lagrangian** of the problem (A.3.21) reads

$$\mathcal{L}(x, \alpha) = x^4 - 50x^2 + 25x + \alpha(-x - 2); \tag{A.3.22}$$

its **epigraph** is defined as

$$\mathcal{A} = \{(r, t) \mid -x - 2 \leq r, \ x^4 - 50x^2 + 25x \leq t, \ \text{for some } x\}, \tag{A.3.23}$$

which is shown as the cyan-colored region in Figure A.2.

- The **primal optimal**  $f^*$  is obtained by projecting the negative side of the epigraph ( $r \leq 0$ ) to the  $t$ -axis and taking the minimum,  $f^* \approx -501.6$ .
- The **dual optimal**  $g^*$  is computed as the highest intersection of the  $t$ -axis and all nonvertical supporting hyperplanes of  $\mathcal{A}$ ,  $g^* \approx -673.4$ .
- For the nonconvex problem, there does not exist a supporting hyperplane of  $\mathcal{A}$  passing through  $(0, f^*)$ , thus **strong duality does not hold**.

## A.4. Rank-One Matrices and Structure Tensors

### Rank-one Matrices

**Definition A.24.** A **rank-one matrix** is a matrix with rank equal to one.

**Theorem A.25.** Every **rank-1 matrix**  $A \in \mathbb{R}^{m \times n}$  can be written as an **outer product**

$$A = \mathbf{u}\mathbf{v}^T, \quad \mathbf{u} \in \mathbb{R}^m, \quad \mathbf{v} \in \mathbb{R}^n. \quad (\text{A.4.1})$$

**Theorem A.26.** Let  $A = \mathbf{u}\mathbf{v}^T$  be a rank-1 matrix. Then

$$\|A\|_2 = \|A\|_F = \|\mathbf{u}\|_2 \|\mathbf{v}\|_2. \quad (\text{A.4.2})$$

**Proof.** Using the definitions of the norms, we have

$$\begin{aligned} \|A\|_2 &\equiv \max_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{u}\mathbf{v}^T \mathbf{x}\|_2 \\ &= \max_{\|\mathbf{x}\|_2=1} \|\mathbf{u}\|_2 \cdot |\mathbf{v}^T \mathbf{x}| = \|\mathbf{u}\|_2 \|\mathbf{v}\|_2, \\ \|A\|_F &= \sqrt{\text{tr}(AA^T)} = \sqrt{\text{tr}(\mathbf{u}\mathbf{v}^T \mathbf{v}\mathbf{u}^T)} \\ &= \sqrt{\|\mathbf{u}\|_2^2 \|\mathbf{v}\|_2^2} = \|\mathbf{u}\|_2 \|\mathbf{v}\|_2, \end{aligned} \quad (\text{A.4.3})$$

which completes the proof.  $\square$

**Example A.27.** Let  $\mathbf{u} = [6, 3, -6]^T$  and  $\mathbf{v} = [1, -2, 2]^T$ .

(a) Form  $A = \mathbf{u}\mathbf{v}^T$ .

(b) Find  $\|A\|_2$ .

### Structure Tensor

**Definition A.28.** The **structure tensor** is a matrix derived from the gradient of a function  $f(\mathbf{x})$ ,  $\mathbf{x} \in \mathbb{R}^n$ : it is defined as

$$S(\mathbf{x}) = (\nabla f)(\nabla f)^T(\mathbf{x}), \quad (\text{A.4.4})$$

which describes the distribution of the gradient of  $f$  in a neighborhood of the point  $\mathbf{x}$ . The structure tensor is often used in **image processing** and **computer vision**.

**Claim A.29. Structure Tensor in Two Variables**  $\mathbf{x} = (x, y) \in \mathbb{R}^2$   
When  $\mathbf{x} = (x, y) \in \mathbb{R}^2$ , the structure tensor in (A.4.4) reads

$$S(\mathbf{x}) = (\nabla f)(\nabla f)^T(\mathbf{x}) = \begin{bmatrix} f_x^2 & f_x f_y \\ f_x f_y & f_y^2 \end{bmatrix}(\mathbf{x}). \quad (\text{A.4.5})$$

- (a) The matrix  $S$  is symmetric and **positive semidefinite**.
- (b)  $\|S\|_2 = \|\nabla f\|_2^2$ , which is the maximum eigenvalue of  $S$ .
- (c)  $\det S = 0$ , which implies that the number 0 is an eigenvalue of  $S$ .

The two eigenvalue-eigenvector pairs  $(\lambda_i, \mathbf{v}_i)$ ,  $i = 1, 2$ , are

$$\begin{aligned} \lambda_1 &= \|\nabla f\|_2^2, & \mathbf{v}_1 &= \nabla f \\ \lambda_2 &= 0, & \mathbf{v}_2 &= [-f_y, f_x]^T \end{aligned} \quad (\text{A.4.6})$$

### Proof.

- (a) The matrix  $S$  is clearly symmetric. Let  $\mathbf{v} \in \mathbb{R}^2$ . Then

$$\mathbf{v}^T S \mathbf{v} = \mathbf{v}^T (\nabla f)(\nabla f)^T \mathbf{v} = |(\nabla f)^T \mathbf{v}|^2 \geq 0, \quad (\text{A.4.7})$$

which proves that  $S$  is positive semidefinite.

- (b) It follows from (A.4.2). Since  $S$  is symmetric,  $\|S\|_2$  must be the maximum eigenvalue of  $S$ . (See Theorem 5.44 (f).)
- (c)  $\det S = f_x^2 f_y^2 - (f_x f_y)^2 = 0 \Rightarrow S$  is not invertible  $\Rightarrow$  An eigenvalue of  $S$  must be 0.
- (d) It is not difficult to check that  $S \mathbf{v}_i = \lambda_i \mathbf{v}_i$ ,  $i = 1, 2$ .  $\square$



### Structure Tensor, Applied to Color Image Processing

- Let  $I(\mathbf{x}) = (r, g, b)(\mathbf{x})$  be a **color image** defined on a rectangular region  $\Omega$ ,  $\mathbf{x} = (x, y) \in \Omega \subset \mathbb{R}^2$ .

- Then

$$\nabla I(\mathbf{x}) = \begin{bmatrix} (r, g, b)_x \\ (r, g, b)_y \end{bmatrix}(\mathbf{x}), \quad (\text{A.4.8})$$

and therefore the structure tensor of  $I$  reads

$$\begin{aligned} S_I(\mathbf{x}) &= (\nabla I)(\nabla I)^T(\mathbf{x}) = \begin{bmatrix} (r, g, b)_x \\ (r, g, b)_y \end{bmatrix} \begin{bmatrix} (r, g, b)_x^T & (r, g, b)_y^T \end{bmatrix}(\mathbf{x}) \\ &= \begin{bmatrix} r_x^2 + g_x^2 + b_x^2 & r_x r_y + g_x g_y + b_x b_y \\ r_x r_y + g_x g_y + b_x b_y & r_y^2 + g_y^2 + b_y^2 \end{bmatrix}(\mathbf{x}). \end{aligned} \quad (\text{A.4.9})$$

**Claim A.30.** For the structure tensor  $S_I$  in (A.4.9):

- (a) We have

$$\|S_I\|_2 = \|\nabla I\|_2^2 = \lambda_{\max}(S_I). \quad (\text{A.4.10})$$

- (b) Rewrite  $S_I$  as

$$S_I = \begin{bmatrix} J_{xx} & J_{xy} \\ J_{xy} & J_{yy} \end{bmatrix}. \quad (\text{A.4.11})$$

Then

$$\lambda_{\max}(S_I) = \lambda_1 = \frac{J_{xx} + J_{yy} + \sqrt{(J_{xx} - J_{yy})^2 + 4J_{xy}^2}}{2}, \quad (\text{A.4.12})$$

and its corresponding eigenvector reads

$$\mathbf{v}_1 = \begin{bmatrix} J_{yy} - \lambda_1 \\ -J_{xy} \end{bmatrix}, \quad (\text{A.4.13})$$

which is the **edge normal direction**.

**Proof.** Here are hints.

- (a) See Theorem 5.44 (f) and Theorem 5.49.

- (b) Use the **quadratic formula** to solve  $\det(S_I - \lambda I) = 0$  for  $\lambda$ .

**Matlab-code** A.31. Structure Tensor

```

----- structure_tensor.m -----
1  function [grad,theta] = structure_tensor(u)
2  % [grad,theta] = structure_tensor(u)
3  % This program uses the Structure Tensor (ST) method to compute
4  %   the Sobel gradient magnitude, |grad(u)|, and
5  %   the edge normal angle, theta.
6
7  %%--- Sobel Derivatives -----
8  [ux,uy] = sobel_derivatives(u);
9
10 %%--- Jacobian: J = (grad u)*(grad u)' -----
11 Jxx = dot(ux,ux,3); Jxy = dot(ux,uy,3); Jyy = dot(uy,uy,3);
12
13 %%--- The first Eigenvalue of J and direction (e1,v1) ---
14 D = sqrt((Jxx-Jyy).^2 + 4*Jxy.^2);
15 e1 = (Jxx+Jyy+D)/2;
16 grad = sqrt(e1);           % sqrt(e1) = magnitude
17 theta = atan2(-Jxy,Jyy-e1); % v1 = (Jyy-e1,-Jxy)

```

```

----- sobel_derivatives.m -----
1  function [ux,uy] = sobel_derivatives(u)
2  % Usage: [ux,uy] = sobel_derivatives(u);
3  % It produces Sobel derivatives, using conv2(u,C,'valid').
4
5  %%--- initialization -----
6  if isa(u,'uint8'), u = im2double(u); end
7  [m,n,d] = size(u);
8  ux = zeros(m,n,d); uy = zeros(m,n,d);
9  C = [1 2 1; 0 0 0; -1 -2 -1];
10
11 %%--- conv2, with 'valid' -----
12 for k = 1:d
13     ux(2:end-1,2:end-1,k) = conv2(u(:,:,k),C, 'valid');
14     uy(2:end-1,2:end-1,k) = conv2(u(:,:,k),C, 'valid');
15 end
16
17 %%--- expand, up to the boundary -----
18 ux(1, :, :) = ux(2, :, :); ux(end, :, :) = ux(end-1, :, :);
19 ux(:, 1, :) = ux(:, 2, :); ux(:, end, :) = ux(:, end-1, :);
20 uy(1, :, :) = uy(2, :, :); uy(end, :, :) = uy(end-1, :, :);
21 uy(:, 1, :) = uy(:, 2, :); uy(:, end, :) = uy(:, end-1, :);

```

## A.5. Boundary-Effects in Convolution Functions in Matlab and Python SciPy

**Note:** You should first understand that there are **3 different modes** for **the computation of convolution** in **Matlab** and **Python**.

- **full:** At points where the given two arrays **overlap partially**, the arrays are **padded with zeros** to get the convolution there.
- **valid:** The output is calculated only at positions where the arrays **overlap completely**. This mode does not use zero padding at all.
- **same:** It crops the middle part out of the ‘full’ mode so that its size is the same as the size of the first array (Matlab) or the larger array (SciPy).

### **Observation** A.32. **Boundary-Effects in Convolutions**

A **post-processing** is required, if the user wants to **suppress boundary effects**, **get the convolution result having the same size as the input**, or both.

**Example** A.33. Let  $f(x, y) = \sin(\pi x) \cos(\pi y)$ .

Find the **Sobel derivative**  $\partial f / \partial x$  by convolving the filter

$$C = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}.$$

Observe **boundary effects** and try to suppress them.

**Solution.**

- Codes are implemented in
  - **Matlab:** `conv2`
  - **SciPy:** `scipy.signal.convolve2d`

for the 3 different modes.

- Around the boundary, values are expanded from the “valid”.
- The convolution results are plotted to compare.

```

----- matlab_conv2_boundary.m -----
1  %%--- Initial setting -----
2  n = 21; h = 1/(n-1);
3  x = linspace(0,1,n); [X,Y] = meshgrid(x);
4
5  F = sin(pi*X).*cos(pi*Y);    % the function
6  Fx = pi*cos(pi*X).*cos(pi*Y); % the true derivative
7
8  %%--- conv2: Sobel derivative -----
9  C = [1 0 -1; 2 0 -2; 1 0 -1] / (8*h);
10 Fx_full = conv2(F,C,'full');
11 Fx_same = conv2(F,C,'same');
12 Fx_valid = conv2(F,C,'valid');
13
14 %%--- Expansion of conv2(valid) -----
15 Fx_expand = zeros(size(F));
16 Fx_expand(2:end-1,2:end-1) = Fx_valid;
17 Fx_expand(1,:) = Fx_expand(2,:); Fx_expand(end,:) = Fx_expand(end-1,:);
18 Fx_expand(:,1) = Fx_expand(:,2); Fx_expand(:,end) = Fx_expand(:,end-1);

```

```

----- scipy_convolve_boundary.py -----
1  import numpy as np; import scipy
2  import matplotlib.pyplot as plt
3  from matplotlib import cm; from math import pi
4
5  ##--- Initial setting -----
6  n = 21; h = 1/(n-1);
7  x = np.linspace(0,1,n); X,Y = np.meshgrid(x,x);
8
9  F = np.sin(pi*X)*np.cos(pi*Y);    # the function
10 Fx = pi*np.cos(pi*X)*np.cos(pi*Y); # the true derivative
11
12 ##--- conv2: Sobel derivative -----
13 C = np.array([[1,0,-1], [2,0,-2], [1,0,-1]]) / (8*h);
14 Fx_full = scipy.signal.convolve2d(F,C,mode='full');
15 Fx_same = scipy.signal.convolve2d(F,C,mode='same');
16 Fx_valid = scipy.signal.convolve2d(F,C,mode='valid');
17
18 ##--- Expansion of convolve2d(valid) -----
19 Fx_expand = np.zeros(F.shape);
20 Fx_expand[1:-1,1:-1] = Fx_valid;
21 Fx_expand[0,:] = Fx_expand[1,:]; Fx_expand[-1,:] = Fx_expand[-2,:];
22 Fx_expand[:,0] = Fx_expand[:,1]; Fx_expand[:, -1] = Fx_expand[:, -2];

```

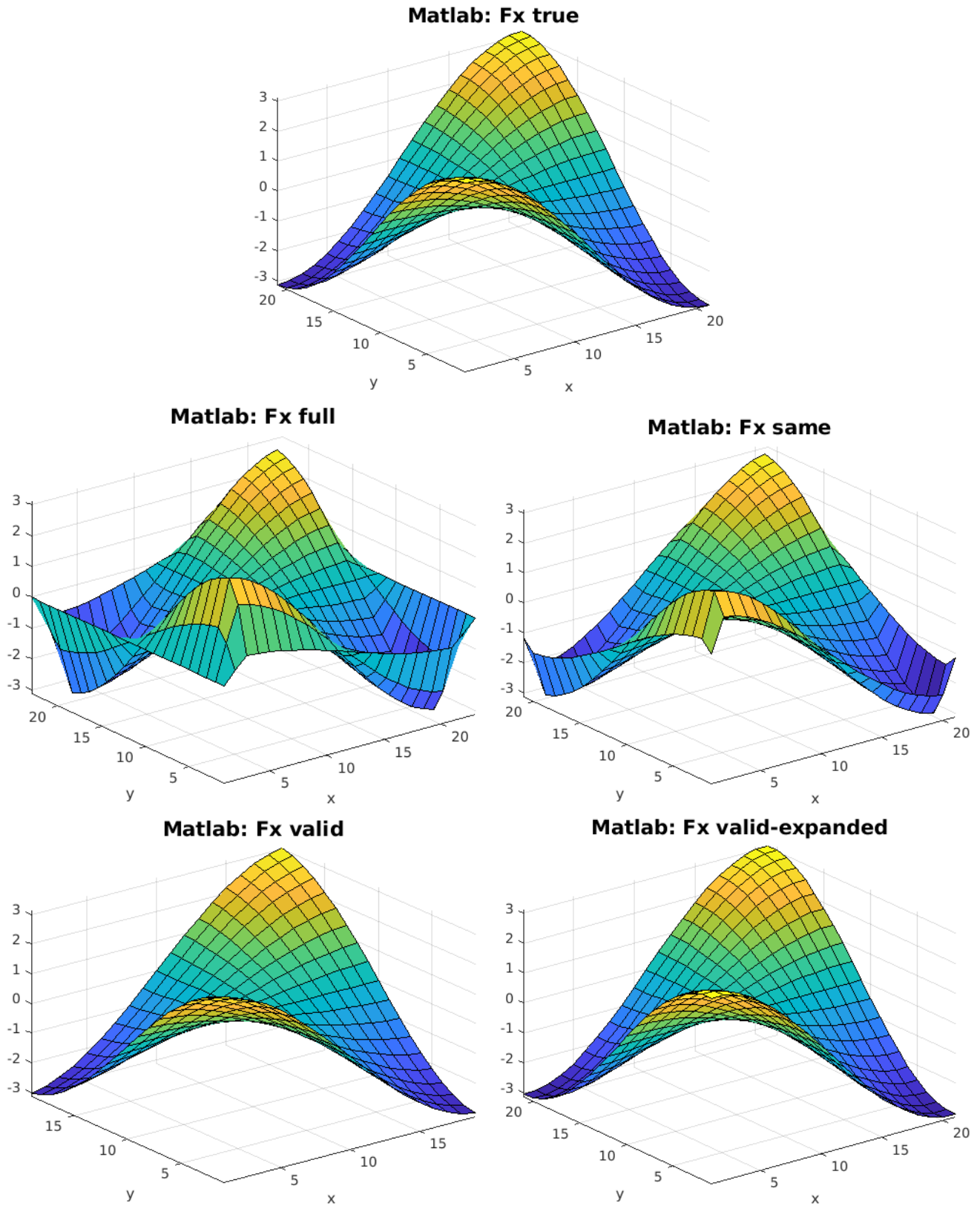


Figure A.3: Matlab convolutions.

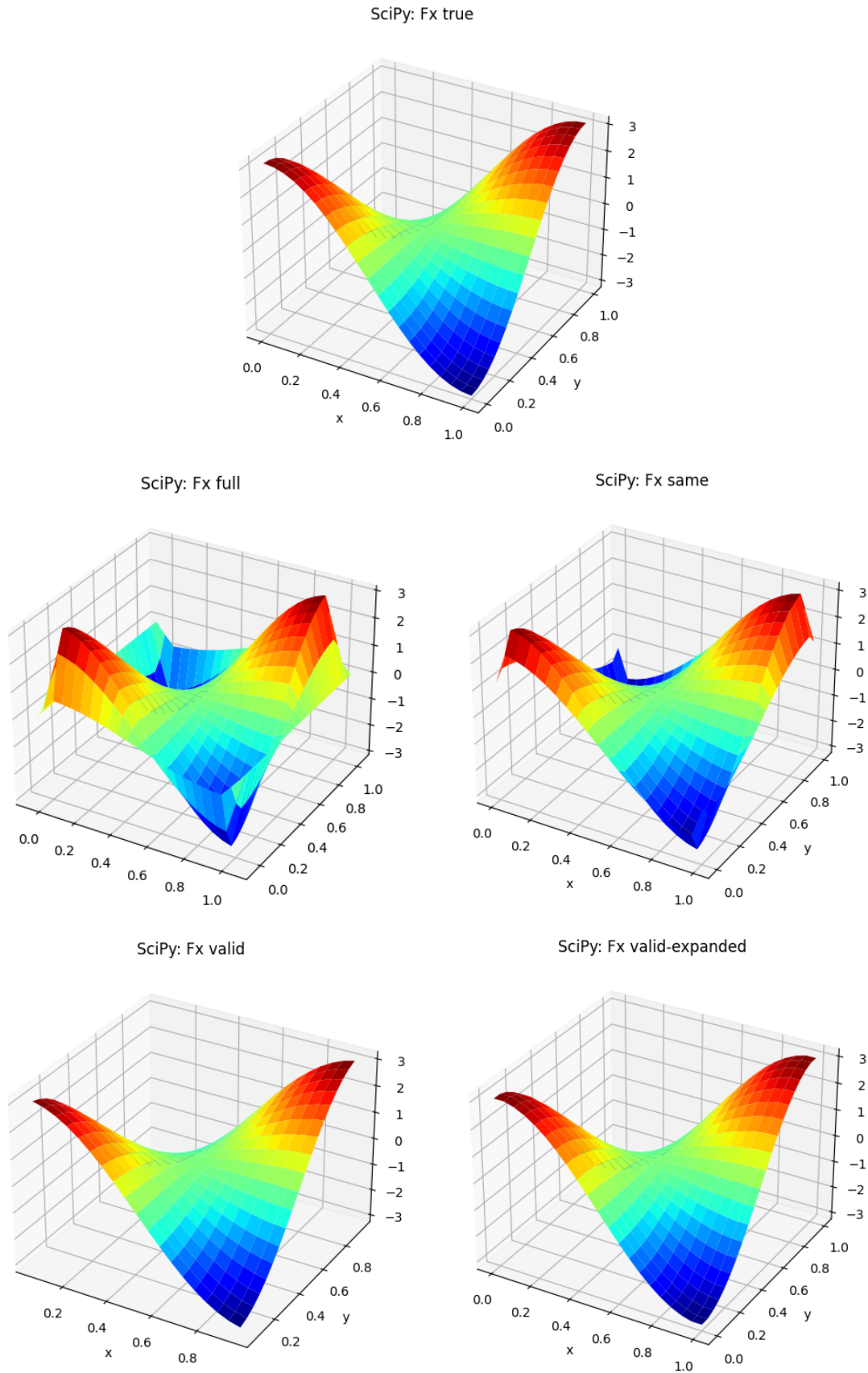


Figure A.4: SciPy convolutions.

## A.6. From Python, Call C, C++, and Fortran

**Note:** A good programming language must be **easy to learn and use** and **flexible and reliable**.

### Python

- **Advantages**
  - Easy to learn and use
  - Flexible and reliable
  - Extensively used in **Data Science**
  - Handy for **Web Development** purposes
  - Having **Vast Libraries** support
  - Among the **fastest-growing** programming languages in the tech industry, machine learning, and AI
- **Disadvantage**
  - **Slow!!**

### Strategy A.34. Speed up Python Programs

- Use **numpy** and **scipy** for all mathematical operations.
  - Always use **built-in functions** wherever possible.
- 
- **Cython**: It is designed as a **C-extension for Python**, which is developed for users not familiar with C. **A Good Choice!**
- 
- **Create and import modules/functions in C, C++, or Fortran**
    - Easily more than **100× faster** than Python scripts
    - **The Best Choice!!**

## Python Extension

**Example A.35.** Functions are implemented in **(Python, F90, C, C++)** and **called from Python**. Let  $x, y \in \mathbb{R}^n$ .

```

for j=1:m
    dotp = dot(x,y);  (V)
end

```

```

for j=1:m, dotp = 0;
    for i=1:n
        dotp = dotp+x(i)*y(i);  (S)
    end
end

```

## f90

```

_____ test_f90.f90 _____
1  subroutine test_f90_v(x,y,m,dotp)
2      implicit none
3      real(kind=8), intent(in) :: x(:), y(:)
4      real(kind=8), intent(out) :: dotp
5      integer :: m,j
6
7      do j=1,m
8          dotp = dot_product(x,y)
9      enddo
10     end
11
12     subroutine test_f90_s(x,y,m,dotp)
13         implicit none
14         real(kind=8), intent(in) :: x(:), y(:)
15         real(kind=8), intent(out) :: dotp
16         integer :: n,m,i,j
17
18         n =size(x)
19         do j=1,m
20             dotp=0
21             do i=1,n
22                 dotp = dotp+x(i)*y(i)
23             enddo
24         enddo
25     end

```



**C++**

The **numeric library** is included for vector operations.

```
test_gpp.cpp
1  #include <iostream>
2  #include <vector>
3  #include <numeric>
4  using namespace std;
5  typedef double VTYPE;
6
7  extern "C" // required when using C++ compiler
8  VTYPE test_gpp_v(VTYPE*x,VTYPE*y, int n, int m)
9  {
10     int i,j;
11     VTYPE dotp;
12
13     for(j=0;j<m;j++){
14         dotp = inner_product(x, x+n, y, 0.0);
15     }
16     return dotp;
17 }
18
19 extern "C" // required when using C++ compiler
20 VTYPE test_gpp_s(VTYPE*x,VTYPE*y, int n, int m)
21 {
22     int i,j;
23     VTYPE dotp;
24
25     for(j=0;j<m;j++){dotp=0.;
26         for(i=0;i<n;i++){
27             dotp += x[i]*y[i];
28         }
29     }
30     return dotp;
31 }
```

**Python**

```
test_py3.py
1  import numpy as np
2
3  def test_py3_v(x,y,m):
4      for j in range(m):
5          dotp = np.dot(x,y)
6          return dotp
7
8  def test_py3_s(x,y,m):
9      n = len(x)
10     for j in range(m):
11         dotp = 0;
12         for i in range(n):
13             dotp +=x[i]*y[i]
14     return dotp
```

**Compiling**

Modules in f90, C, and C++ are compiled by executing the shell script.

```
Compile-f90-c-cpp
1  #!/usr/bin/bash
2
3  LIB_F90='lib_f90'
4  LIB_GCC='lib_gcc'
5  LIB_GPP='lib_gpp'
6
7  ### Compiling: f90
8  f2py3 -c --f90flags='-O3' -m $LIB_F90 *.f90
9
10 ### Compiling: C (PIC: position-independent code)
11 gcc -fPIC -O3 -shared -o $LIB_GCC.so *.c
12
13 ### Compiling: C++
14 g++ -fPIC -O3 -shared -o $LIB_GPP.so *.cpp
```

**Python Wrap-up**

An executable Python wrap-up is implemented as follows.

```

Python_calls_F90_GCC.py
1  #!/usr/bin/python3
2
3  import numpy as np
4  import ctypes, time
5  from test_py3 import *
6  from lib_f90 import *
7  lib_gcc = ctypes.CDLL("./lib_gcc.so")
8  lib_gpp = ctypes.CDLL("./lib_gpp.so")
9
10 n=100; m=1000000
11 #n=1000; m=1000000
12
13 x = np.arange(0.,n,1); y = x+0.1;
14
15 print('-----')
16 print('Speed test: (dot-product: n=%d), m=%d times' %(n,m))
17 print('-----')
18
19 ### Python #####
20 t0 = time.time(); result = test_py3_v(x,y,m)
21 print('test_py3_v: e-time = %.4f; result = %.2f' %(time.time()-t0,result))
22
23 t0 = time.time(); result = test_py3_s(x,y,m)
24 print('test_py3_s: e-time = %.4f; result = %.2f\n' %(time.time()-t0,result))
25
26 ### Fortran #####
27 t0 = time.time(); result = test_f90_v(x,y,m)
28 print('test_f90_v: e-time = %.4f; result = %.2f' %(time.time()-t0,result))
29
30 t0 = time.time(); result = test_f90_s(x,y,m)
31 print('test_f90_s: e-time = %.4f; result = %.2f\n' %(time.time()-t0,result))
32
33 ### C #####
34 lib_gcc.test_gcc_s.argtypes = [np.ctypeslib.ndpointer(dtype=np.double),
35     np.ctypeslib.ndpointer(dtype=np.double),
36     ctypes.c_int,ctypes.c_int] #input type
37 lib_gcc.test_gcc_s.restype = ctypes.c_double #output type
38
39 t0 = time.time(); result = lib_gcc.test_gcc_s(x,y,n,m)
40 print('test_gcc_s: e-time = %.4f; result = %.2f\n' %(time.time()-t0,result))
41

```

```

42  ### C++ #####
43  lib_gpp.test_gpp_v.argtypes = [np.ctypeslib.ndpointer(dtype=np.double),
44      np.ctypeslib.ndpointer(dtype=np.double),
45      ctypes.c_int,ctypes.c_int] #input type
46  lib_gpp.test_gpp_v.restype = ctypes.c_double #output type
47
48  t0 = time.time(); result = lib_gpp.test_gpp_v(x,y,n,m)
49  print('test_gpp_v: e-time = %.4f; result = %.2f' %(time.time()-t0,result))
50
51  lib_gpp.test_gpp_s.argtypes = [np.ctypeslib.ndpointer(dtype=np.double),
52      np.ctypeslib.ndpointer(dtype=np.double),
53      ctypes.c_int,ctypes.c_int] #input type
54  lib_gpp.test_gpp_s.restype = ctypes.c_double #output type
55
56  t0 = time.time(); result = lib_gpp.test_gpp_s(x,y,n,m)
57  print('test_gpp_s: e-time = %.4f; result = %.2f\n' %(time.time()-t0,result))

```

### Performance Comparison

A Linux OS is used with an **Intel Core i7-10750H CPU @ 2.60GHz**.

n=100, m=1000000 ⇒ **200M flops**

```

1  -----
2  Speed test: (dot-product: n=100), m=1000000 times
3  -----
4  test_py3_v: e-time = 0.7672; result = 328845.00
5  test_py3_s: e-time = 18.2175; result = 328845.00
6
7  test_f90_v: e-time = 0.0543; result = 328845.00
8  test_f90_s: e-time = 0.0530; result = 328845.00
9
10 test_gcc_s: e-time = 0.0603; result = 328845.00
11
12 test_gpp_v: e-time = 0.0600; result = 328845.00
13 test_gpp_s: e-time = 0.0612; result = 328845.00

```

n=1000, m=1000000 ⇒ **2B flops**

```

1 -----
2 Speed test: (dot-product: n=1000), m=1000000 times
3 -----
4 test_py3_v: e-time = 0.8984; result = 332883450.00
5 test_py3_s: e-time = 201.4086; result = 332883450.00
6
7 test_f90_v: e-time = 0.8331; result = 332883450.00
8 test_f90_s: e-time = 0.8318; result = 332883450.00
9
10 test_gcc_s: e-time = 0.8575; result = 332883450.00
11
12 test_gpp_v: e-time = 0.8638; result = 332883450.00
13 test_gpp_s: e-time = 0.8623; result = 332883450.00

```

n=100, m=10000000 ⇒ **2B flops**

```

1 -----
2 Speed test: (dot-product: n=100), m=10000000 times
3 -----
4 test_py3_v: e-time = 7.8289; result = 328845.00
5 test_py3_s: e-time = 195.0932; result = 328845.00
6
7 test_f90_v: e-time = 0.5656; result = 328845.00
8 test_f90_s: e-time = 0.5456; result = 328845.00
9
10 test_gcc_s: e-time = 0.6090; result = 328845.00
11
12 test_gpp_v: e-time = 0.6055; result = 328845.00
13 test_gpp_s: e-time = 0.6089; result = 328845.00

```

### Summary A.36. Python Calls C, C++, and Fortran

- **Compiled modules** are **200+ times faster** than **Python Scripts**.
- **Compiled modules** are yet faster than **Python Built-in's**.
- **Fortran** is about 5 ~ 10 % faster than **C/C++**.

### Innovative projects often require a **completely new code**:

- You may search-&-download public-domain functions, for which you do not have to re-implement in Python.
- If a function needs a long script, you *should* try C, C++, or Fortran.



# APPENDIX **P**

## **Projects**

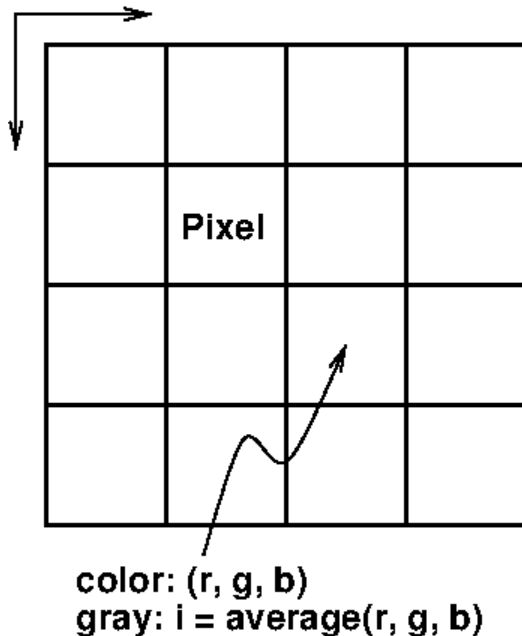
### **Contents of Chapter P**

P.1. Project: Canny Edge Detection Algorithm for Color Images . . . . .	366
P.2. Project: Text Extraction from Images, PDF Files, and Speech Data . . . . .	380

## P.1. Project: Canny Edge Detection Algorithm for Color Images

Through the project, we will build an **edge detection (ED) algorithm** which can sketch the edges of objects on **color images**.

### What are Images?



- A rectangular array of pixels
- In the **RGB** representation:  
**An image is a function**  
$$u : \Omega \rightarrow \mathbb{R}_+^3 := \{(r, g, b) : r, g, b \geq 0\}$$
- In practice,  
$$u : \Omega \subset \mathbb{N}^2 \rightarrow \mathbb{N}_{[0,255]}^3$$
  
due to **sampling & quantization**
- $u = u(m, n, d)$ , for  $d = 1$  or  $3$
- **rgb2gray formula:**  
$$0.299 * R + 0.587 * G + 0.114 * B$$

- Most of ED algorithms are developed for **grayscale images**. (Color images must be transformed to a grayscale image.)
- Edge detection algorithms consist of a few steps. For example, the **Canny edge detection algorithm** has five steps [2] (Canny, 1986):
  1. Noise reduction (image blur)
  2. Gradient calculation
  3. Edge thinning (non-maximum suppression)
  4. Double threshold
  5. Edge tracking by **hysteresis**

We will learn Canny's algorithm, implementing every step in detail.



**“edge”: A Built-in Function**

```

Edge_Detection.m
1  close all; clear all
2
3  global beta delta Dt nmax level
4  beta = 0.05; delta = 0.2; Dt = 0.25; nmax = 10; %TV denoising
5  level = 1;
6
7  if exist('OCTAVE_VERSION','builtin'), pkg load image; end
8  %-----
9  TheImage = 'DATA/Lena256.png';
10 %TheImage = 'DATA/synthetic-Checkerboard.png';
11 [Filepath,Name,Ext] = fileparts(TheImage);
12 v0 = im2double(imread(TheImage));
13 [m,n,d] = size(v0);
14     if level>=1, figure, imshow(v0); end
15     fprintf('%s: size=(%d,%d,%d)\n',TheImage,m,n,d)
16
17 %--- Use built-in: "edge" -----
18 vg = rgb2gray(v0);
19 E = edge(vg,'Canny');
20     if level>=1, figure,imshow(E); end
21     if level>=2, imwrite(vg, strcat(Name, '_Gray.png'));
22                 imwrite(E, strcat(Name, '_Gray-Builtin-Edge.png')); end
23
24 %-----
25 %--- New Trial: Color Edge Detection -----
26 %-----
27 ES = color_edge(v0,Name);

```



Figure P.1: Edge detection, using the built-in function “edge”, which is **not perfect but somewhat acceptable**.

## A Critical Issue on Edge Detection Algorithms

**Observation P.1.** When color images are transformed to grayscale, it is occasionally the case that **edges**

- either lose their strength
- or even disappear.

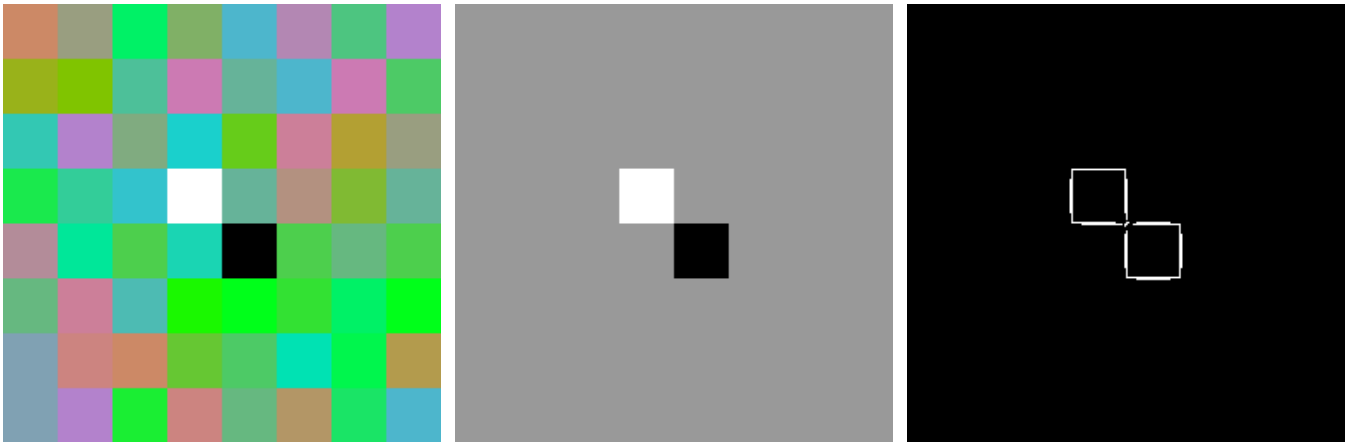


Figure P.2: **Canny edge detection for color images:** A synthetic image produced by `transform_to_gray.m`, its grayscale, and the result of the built-in function “edge”.

**Project Objectives:** The project will develop an edge detection algorithm, which is less problem (so, more effective) for color images.

- We will design an effective algorithm as good as (or better than) the well-tuned built-in function (edge).
- For your convenience and success, you will be provided with a **modelcode**, saved in **Edge-Detection.MM.tar**.

**A Modelcode, for Color Edge Detection**

```

_____ color_edge.m _____
1  function ES = color_edge(v0,Name)
2  % function ES = color_edge(v0,Name)
3  % Edge detection for color images,
4  % by dealing with gradients separately for RGB
5
6  global beta delta Dt nmax level
7
8  [m,n,d] = size(v0);
9  vs = zeros(m,n,d); grad = zeros(m,n,d); theta = zeros(m,n,d);
10 TH = zeros(m,n); ES = zeros(m,n);
11
12 %--- Steps 1 & 2: Channel-by-Channel -----
13 for k=1:d
14     %u = imgaussfilt(v0(:,:,k),2);      % Step 1
15     u = tv_denoising(v0(:,:,k));      % Step 1: Image Denoising/Blur
16     vs(:,:,k) = u;
17     [grad(:,:,k),theta(:,:,k)] = sobel_grad(u); % Step 2: Grad Calculation
18 end
19
20 %--- Combine for (1D) Gradient Intensity -----
21 [E0,I] = max(grad,[],3); E0 = E0/max(E0(:));
22 for i=1:m, for j=1:n, TH(i,j) = theta(i,j,I(i,j)); end,end
23     if level>=2,
24         imwrite(vs,strcat(Name,'_TV_denoised.png'));
25         imwrite(E0,strcat(Name,'_New-Sobel-Grad.png'));
26     end
27
28 %--- Step 3: Edge Thinning -----
29 E1 = non_max_suppression(E0,TH);
30     if level>=2,
31         imwrite(E1,strcat(Name,'_Sobel-Grad-Supressed.png'));
32     end
33
34 %--- Step 4: Double Threshold -----
35 highRatio = 0.09; lowRatio = 0.3; % Set them appropriately!!
36 [strong,weak] = double_threshold(E1,highRatio,lowRatio);
37
38 %--- Step 5: Edge Tracking by Hysteresis -----
39 % YOU WILL DO THIS
40
41 %--- Step 6: (Extra Step) Edge-Trim -----
42 % YOU WILL DO THIS

```

### P.1.1. Noise Reduction: Image Blur

This step can be done by applying the Gaussian filter `imgaussfilt`.

- It is a simple averaging algorithm and blurs the whole image.  
 $\Rightarrow$  It can make edge strength weaker; we will try another method.

Let  $v_0$  be an observed (noisy) image defined on  $\Omega \subset \mathbb{R}^2$ . Consider the evolutionary **total variation (TV)** model [11]:

$$u_t - \nabla \cdot \left( \frac{\nabla u}{|\nabla u|} \right) = \beta(v_0 - u), \quad (\text{TV}) \quad (\text{P.1.1})$$

where the left-side is the negation of the **mean curvature** and  $\beta$  denotes a constraint parameter, a **Lagrange multiplier**.

- The TV model tends to converge to a piecewise constant image. Such a phenomenon is called the **staircasing effect**.  $\Rightarrow$  The TV model can be used for both **noise reduction** and **edge sharpening**.

#### Numerical Discretization

For the **time-stepping procedure**, we simply employ the **explicit method**, the **forward Euler method**:

$$\frac{u^{n+1} - u^n}{\Delta t} - \nabla \cdot \left( \frac{\nabla u^n}{|\nabla u^n|} \right) = \beta(v_0 - u^n), \quad u^0 = v_0, \quad (\text{P.1.2})$$

which equivalently reads

$$u^{n+1} = u^n + \Delta t(\beta(v_0 - u^n) - Au^n), \quad u^0 = v_0, \quad (\text{P.1.3})$$

where

$$Au^n \approx -\nabla \cdot \left( \frac{\nabla u^n}{|\nabla u^n|} \right) = -\left( \frac{u_x^n}{|\nabla u^n|} \right)_x - \left( \frac{u_y^n}{|\nabla u^n|} \right)_y.$$

#### Remark P.2. The TV Denoising

- The TV model, (P.1.3), requires to set  $\Delta t$  small enough for **stability**.
- It can reduce noise effectively in a few iterations.
- Implementation details can be found from `tv_denoising.m` and `curvaturePG.m`.

```

                                test_denoising.m
1  close all; clear all
2
3  global beta delta Dt nmax level
4  beta = 0.05; delta = 0.2; Dt = 0.25; nmax = 10; %TV denoising
5  level = 2;
6
7  if exist('OCTAVE_VERSION','builtin'), pkg load image; end
8  %-----
9  TheImage = 'DATA/Lena256.png';
10 v0 = im2double(imread(TheImage));
11 [m,n,d] = size(v0);
12
13 ug = zeros(m,n,d); ut = zeros(m,n,d);
14 for k=1:d
15     ut(:,:,k) = tv_denoising(v0(:,:,k));
16     ug(:,:,k) = imgaussfilt(v0(:,:,k),2); % sigma=2
17 end
18
19 imwrite(ut,'Lena256_test-TV_denoised.png')
20 imwrite(ug,'Lena256_test-Gaussian-filter.png')

```

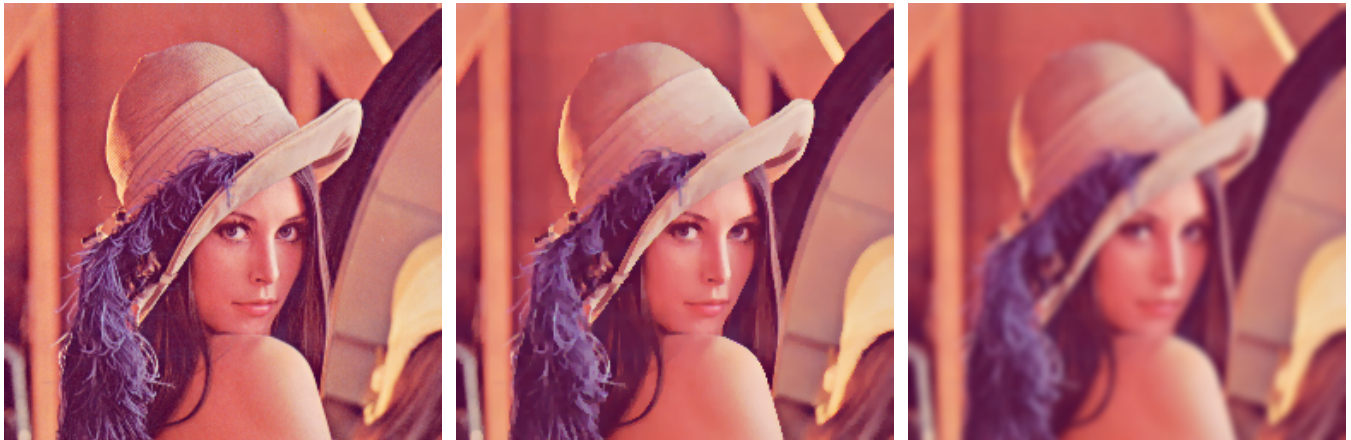


Figure P.3: Step 1: **Image denoising** or **image blur**. The original Lena (left), the TV-denoised image (middle), and the Gaussian-filtered image (right).

## P.1.2. Gradient Calculation: Sobel Gradient

**Note:** **Edges** correspond to a **change of pixels' intensity**. To detect it, the easiest way is **to apply filters** that highlight the intensity change in both directions: horizontal ( $x$ ) and vertical ( $y$ ).

### Algorithm P.3. Sobel gradient

- The image gradient,  $\nabla u = (u_x, u_y)$ , is calculated as the **convolution** of the image ( $u$ ) and the **Sobel kernels** ( $K_x, K_y$ ).

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \quad (\text{P.1.4})$$

That is,

$$u_x = \text{conv}(u, K_x), \quad u_y = \text{conv}(u, K_y). \quad (\text{P.1.5})$$

- Then the magnitude  $G$  and the slope  $\theta$  of the gradient are calculated as follow:

$$G = \sqrt{u_x^2 + u_y^2}, \quad \theta = \arctan\left(\frac{u_y}{u_x}\right) = \text{atan2}(u_y, u_x). \quad (\text{P.1.6})$$

See `sobel_grad.m` on the following page.

### Note: The Gradient Magnitude & `atan2`

- The **gradient magnitude** is saved in `E0`.  
(See Line 21 of `color_edge.m` on p.369.)
  - It is normalized for its maximum value to be 1, **for the purpose of figuring**.

- `>> help atan2`

`atan2(Y,X)` is the four quadrant arctangent of the elements of `X` and `Y` such that  $-\pi \leq \text{atan2}(Y,X) \leq \pi$ .

...

```

----- sobel_grad.m -----
1 function [grad,theta] = sobel_grad(u)
2 % [grad,theta] = sobel_grad(u)
3 % It computes the Sobel gradient magnitude, |grad(u)|,
4 % and edge normal angle, theta.
5
6 [m,n,d]=size(u);
7 grad = zeros(m,n); theta = zeros(m,n);
8
9 %%-----
10 for q=1:n
11     qm=max(q-1,1); qp=min(q+1,n);
12     for p=1:m
13         pm=max(p-1,1); pp=min(p+1,m);
14         ux = u(pp,qm)-u(pm,qm) +2.*(u(pp,q)-u(pm,q)) +u(pp,qp)-u(pm,qp);
15         uy = u(pm,qp)-u(pm,qm) +2.*(u(p,qp)-u(p,qm)) +u(pp,qp)-u(pp,qm);
16         grad(p,q) = sqrt(ux^2 + uy^2);
17         theta(p,q) = atan2(uy,ux);
18     end
19 end

```



Figure P.4: The gradient magnitudes of (R,G,B) components of Lena.

**Note:** Through the project, we may get the **single-component gradient** by simply **taking the maximum of (R,G,B)-gradients**. See Line 21 of `color_edge.m`, p.369.

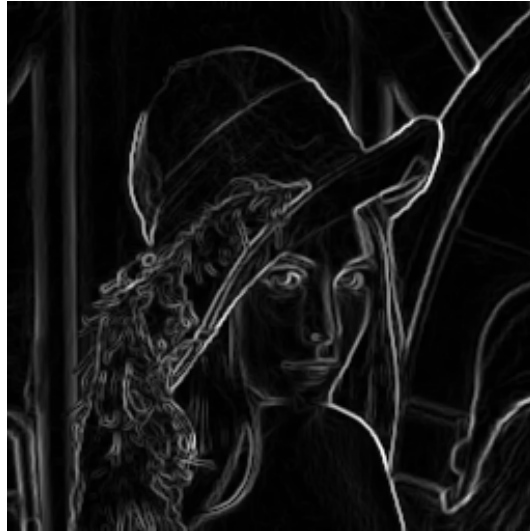


Figure P.5: Step 2: The maximum of (R,G,B) gradients, for the Lena image.

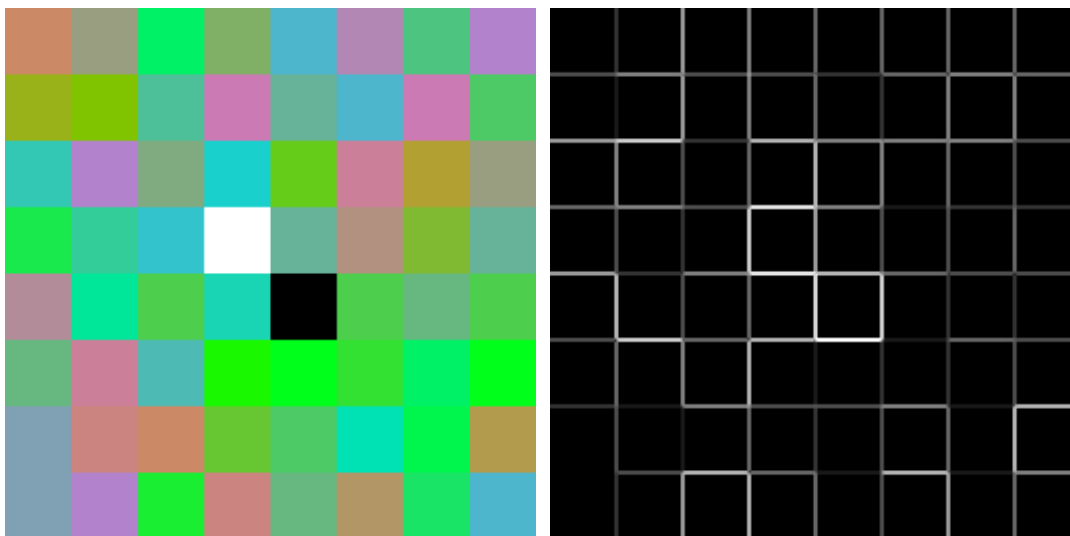


Figure P.6: The color checkerboard image in Figure P.2 (left) and the maximum of its (R,G,B)-gradients (right).

Now, **more reliable edges** can be detected!



### P.1.3. Edge Thinning: Non-maximum Suppression

As one can see from above figures, some of the edges are thick and others are thin.

- The goal of **edge thinning** is to mitigate the thick edges.

#### Algorithm P.4. Non-maximum Suppression

- The algorithm goes through the gradient intensity matrix and
- finds the pixels with the maximum value in the **edge normal directions**.

*The principle is simple!*

```

1  function Z = non_max_suppression(E0,TH)
2  % function Z = non_max_suppression(E0,TH)
3
4  [m,n] = size(E0);
5  Z = zeros(m,n);
6
7  TH(TH<0) = TH(TH<0)+pi;
8  R = mod(floor((TH+pi/8)/(pi/4)),4); % region=0,1,2,3
9
10 for i=2:m-1, for j=2:n-1
11     if R(i,j)==0, % around 0 degrees
12         a=E0(i-1,j); b=E0(i+1,j);
13     elseif R(i,j)==1, % around 45 degrees
14         a=E0(i-1,j-1); b=E0(i+1,j+1);
15     elseif R(i,j)==2, % around 90 degrees
16         a=E0(i,j-1); b=E0(i,j+1);
17     else, % around 135 degrees
18         a=E0(i-1,j+1); b=E0(i+1,j-1);
19     end
20     if E0(i,j)>=max(a,b), Z(i,j) = E0(i,j); end
21 end,end

```



Figure P.7: Step 3: Non-maximum suppression. The Sobel gradient in Figure P.5 (left) and the non-maximum suppressed (right).

## P.1.4. Double Threshold

The **double threshold** step aims at identifying 3 kinds of pixels: strong, weak, and non-relevant:

- **Strong pixels**: pixels of high intensities  
⇒ They *surely* contribute to the final edge.
- **Weak pixels**: pixels of mid-range intensities  
⇒ Not to be considered as non-relevant for the edge detection.
- **Other pixels** are considered as non-relevant for the edge.

### Note: Implementation of Double Threshold

0. Set high and low thresholds.
1. **High threshold** is used to identify the strong pixels.
2. **Low threshold** is used to identify the non-relevant pixels.
3. All pixels having intensity between both thresholds are flagged as weak.
4. The Hysteresis mechanism (Step 5) will help us identify the ones that could be considered as strong and the ones that are considered as non-relevant.

### **In-Reality** P.5. **highThreshold & lowThreshold**

It is often the case that you should assign two ratios to set the highThreshold and lowThreshold.

- For example:

```
highRatio = 0.09; lowRatio = 0.3;
highThreshold = highRatio*max(E1(:));
lowThreshold = lowRatio *highThreshold;
```

where E1 is the non-maximum suppressed gradient intensity.

**Note:** In order to detect weak edges more effectively, one can employ **dynamic (variable) thresholds**. (⇒ It requires some more research.)

```

double_threshold.m
1  unctio[n] [strong,weak] = double_threshold(E1,highRatio,lowRatio)
2
3  highThreshold = highRatio*max(E1(:));
4  lowThreshold  = lowRatio *highThreshold;
5
6  strong = (E1>=highThreshold);
7  weak   = (E1<highThreshold).*(E1>=lowThreshold);

```



Figure P.8: Step 4: Double threshold. The strong pixels (left), the weak pixels (middle), and the combined (right).

#### Remark P.6. Step 4: Double Threshold

- You should select `highRatio` and `lowRatio` appropriately.
- You may have to eliminate isolated pixels from strong pixels, which can be carried out as a post-processing.

### P.1.5. Edge Tracking by Hysteresis

#### Algorithm P.7. The Edge Tracking Rule

The **hysteresis** consists of transforming **weak pixels** into strong ones  
 $\iff$  **at least one of 8 surrounding pixels is a strong one.**

You will implement a function for edge tracking.

## Here are What to Do

1. Download the modelcode: [Edge-Detection.MM.tar](#).
  2. **Complete Steps 4 & 5** with `color_edge.m`.
    - For Step 4, the major work must be to set appropriately two parameters: `highRatio` and `lowRatio`.
    - For Step 5, you have to **implement a function** named `hysteresis` which realizes the **edge tracking rule** in Algorithm P.7.
  3. **Post-processing (Step 6)**: Implement a trimming function in order to eliminate isolated edge pixels (or, of length e.g.  $\leq 3$ ).
- 
4. Run the resulting code of yours to get the edges.
  5. Download **another image**, run your code, and tune it to get more reliable edges. For tuning:
    - You may try to use `imgaussfilt` with various  $\sigma$ , rather than the TV-denoising. See Line 15 of `color_edge.m`, p.369.
    - Try various combinations of `highRatio` and `lowRatio`.
- 
6. **Extra Credit**:
    - (a) **Analysis**: For **noise reduction**, you can employ either the TV-denoising model or the builtin function `imgaussfilt(Image,  $\sigma$ )` with various choices of  $\sigma$ . Analyze effects of different choices of parameters and functions on edge detection.
    - (b) **New Idea for the Gradient Intensity**. We chose the maximum of (R,G,B)-gradients; see Line 21 of `color_edge.m`. Do you have any idea better than that?

**Report what you have done, including newly-implemented M-files, choices of parameters and functions, and images and their edges.**

## P.2. Project: Text Extraction from Images, PDF Files, and Speech Data

In text extraction applications, the core technology is the **Optical Character Recognition (OCR)**.

- Its primary function is to extract texts from images.
- In modern days, using advanced **machine learning** algorithms, the OCR can identify and convert **image texts** into **audio files**, for easy listening.

There are some powerful text extraction software (having accuracy 98+%).

- However most of them are not freely/conveniently available.
- We will develop two text extraction algorithms, one for image data and the other for speech data.

**Project Objectives:** To develop two separate Python programs:

pdfim2text & speech2text.

### 1. **PDF-Image to Text** (pdfim2text)

- **Input:** (an image) or (a pdf file)
  - A PDF may include images.
  - When a PDF is generated by scanning, each page is an image.
- **Core Task:** Extract all texts; convert texts to audio data.

### 2. **Speech to Text+Speech** (speech2text)

- **Input:** speech data from (**microphone**) or (a wave file)
- **Core Task:** Extract texts; play the extracted texts.

**An Example**

```
pdfim2text
1  #!/usr/bin/python
2
3  import pytesseract
4  from pdf2image import convert_from_path
5  from PIL import Image
6  from gtts import gTTS
7  from playsound import playsound
8  import os, pathlib, glob
9  from termcolor import colored
10
11 def takeInput():
12     pmode = 0;
13     IN = input("Enter a pdf or an image: ")
14     if os.path.isfile(IN):
15         path_stem = pathlib.Path(IN).stem
16         path_ext = pathlib.Path(IN).suffix
17         if path_ext.lower() == '.pdf': pmode=1
18     else:
19         exit()
20     return IN, path_stem, pmode
21
22 def pdf2txt(IN):
23     # you have to complete the function appropriately
24     return 'Aha, it is a pdf file.\n
25           For pdf2txt, you may save the text here without return.'
26
27 def im2txt(IN):
28     # you have to complete the function appropriately
29     return 'Now, it is an image.\n
30           For im2txt, try to return the text to play'
31
32 if __name__ == '__main__':
33     IN, path_stem, pmode = takeInput()    #pmode=0:image; pmode=1:pdf
34     if pmode:
35         txt = pdf2txt(IN)
36     else:
37         txt = im2txt(IN)
38
39     audio = gTTS(text=txt, lang="en", slow=False);
40     WAV = '0000-' + path_stem + '-text.wav';
41     audio.save(WAV); print(colored('Text: saved to <%s>' %(WAV), 'yellow'))
42     playsound(WAV); os.remove(WAV)
```

## What to Do

First download <https://skim.math.msstate.edu/LectureNotes/data/Image-Speech-Text-Processing.PY.tar>. Untar it to see the file `pdfim2text` and example codes in a subdirectory `example-code`.

1. Complete `pdfim2text` appropriately.
  - You may find clues from `example-code/pdf2txt.py`
2. Implement `speech2text` from scratch.
  - You may get hints from `speech_mic2wave.py` and `image2text.py` in the directory `example-code`.

Try to put all functions into a single file for each command, which enhances portability of the commands.

## Report

- Work in a directory, of which the name begins with your last name.
- Use the **three-page project document** as a data file for `pdfim2text`.
- zip or tar your work directory and submit via email.
- Write a report to explain what you have done, including images and wave files; upload it to Canvas.

## PDF-Image to Texts

As a part of the project, you will develop a Python program that can extract texts from PDF files and images: and generally, **from PDF files including images.**

An example PDF is the one you are reading now.

(This portion is an image, by `text2image.py`.)





# Bibliography

- [1] R. BARRETT, M. BERRY, T. CHAN, J. DEMMEL, J. DONATO, J. DONGARRA, V. EIJKHOUT, R. POZO, C. ROMINE, AND H. VAN DER VORST, *Templates for the solution of linear systems: Building blocks for iterative methods*, SIAM, Philadelphia, 1994. The postscript file is free to download from <http://www.netlib.org/templates/> along with source codes.
- [2] J. CANNY, *A computational approach to edge detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8 (1986), pp. 679–698.
- [3] O. CHUM AND J. MATAS, *Matching with prosac-progressive sample consensus*, in 2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05), vol. 1, IEEE, 2005, pp. 220–226.
- [4] C. CORTES AND V. N. VAPNIK, *Support-vector networks*, Machine Learning, 20 (1995), pp. 273–297.
- [5] M. FISCHLER AND R. BOLLES, *Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography*, Communications of the ACM, 24 (1981), pp. 381–395.
- [6] B. GROSSER AND B. LANG, *An  $\mathcal{O}(n^2)$  algorithm for the bidiagonal svd*, Lin. Alg. Appl., 358 (2003), pp. 45–70.
- [7] C. KELLY, *Iterative methods for linear and nonlinear equations*, SIAM, Philadelphia, 1995.
- [8] P. C. NIEDFELDT AND R. W. BEARD, *Recursive ransac: multiple signal estimation with outliers*, IFAC Proceedings Volumes, 46 (2013), pp. 430–435.
- [9] M. NIELSEN, *Neural networks and deep learning*. (The online book can be found at <http://neuralnetworksanddeeplearning.com>), 2013.
- [10] F. ROSENBLATT, *The Perceptron: A probabilistic model for information storage and organization in the brain*, Psychological Review, (1958), pp. 65–386.
- [11] L. RUDIN, S. OSHER, AND E. FATEMI, *Nonlinear total variation based noise removal algorithms*, Physica D, 60 (1992), pp. 259–268.
- [12] P. H. TORR AND A. ZISSERMAN, *Mlesac: A new robust estimator with application to estimating image geometry*, Computer vision and image understanding, 78 (2000), pp. 138–156.

- [13] P. R. WILLEMS, B. LANG, AND C. VÖMEL, *Computing the bidiagonal SVD using multiple relatively robust representations*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 907–926.
- [14] S. XU, *An Introduction to Scientific Computing with MATLAB and Python Tutorials*, CRC Press, Boca Raton, FL, 2022.

# Index

- $(n + 1)$ -point difference formula, 94
- `:`, Python slicing, 216
- `:`, in Matlab, 15
- `__init__()` constructor, 226
- absolute error, 47
- activation function, 266
- activation functions, popular, 275
- Adaline, 272, 276
- adaptive step size, 186
- affine function, 337
- algorithm, 47
- algorithmic design, 4
- algorithmic parameter, 273
- analog signals, 38
- anonymous function, 25
- `anonymous_function.m`, 25
- approximation, 194
- `area_closed_curve.m`, 33, 62
- artificial neurons, 265
- `atan2`, 372
- attributes, 226
- audio files, 380
- augmented matrix, 115, 119
- average slope, 109
- average speed, 66
- backbone of programming, 8
- backtracking line search, 186
- backward phase, 122
- backward-difference, 93
- basic variables, 121
- basis, 77, 236
- basis function, 76
- best approximation, 246
- bias, 267
- big Oh, 50, 51
- binary classifier, 265, 266
- boundary effects, 353
- boundary-value problem, 192
- break, 23
- `call_get_cubes.py`, 220
- Canny edge detection algorithm, 366
- cardinal functions, 87
- Cauchy-Schwarz inequality, 154
- chain rule, 74
- change of basis, 304
- change of variables, 76, 202
- change-of-base formula, 61
- characteristic equation, 142
- characteristic polynomial, 142
- `charpoly`, 143
- child class, 229
- `circle.m`, 32
- class, 225
- `Classes.py`, 229
- classification problem, 204
- closest point, 246
- clustering, 262
- CNN, 291
- code block, 215
- coding, iii, 2
- coding vs. programming, 5
- coefficient matrix, 115
- coefficients, 104
- cofactor, 137
- cofactor expansion, 137
- color image, 351
- color image processing, 351
- `color_edge.m`, 369
- column space, 235
- common logarithm, 60
- `Compile-f90-c-cpp`, 360
- complementary slackness, 341
- complex number system, 35

- computer programming, [iii](#), [2](#), [8](#)
- computer vision, [350](#)
- concave, [344](#)
- concave maximization problem, [337](#)
- condition number, [153](#), [191](#)
- conditionally stable, [47](#)
- consensus set, [206](#)
- consistent system, [115](#)
- constraint set, [181](#)
- continue, [24](#)
- contour, [36](#)
- contour, in Matlab, [18](#)
- conv2, [353](#)
- convergence of Newton's method, [101](#)
- convergence of order  $\alpha$ , [48](#)
- converges absolutely, [79](#)
- convex optimization problem, [337](#)
- convex problem, [344](#)
- convolution, [353](#), [372](#)
- correction term, [99](#)
- cost function, [266](#)
- covariance, [305](#)
- covariance matrix, [304–306](#), [308](#), [319](#)
- Covariance.py, [306](#)
- critical point, [184](#)
- critical points, [176](#)
- csvwrite, [32](#)
- curse of dimensionality, [263](#)
- curvaturePG.m, [370](#)
- cython, [212](#), [357](#)
  
- daspect, [32](#)
- data matrix, [309](#)
- debugging, [8](#)
- deepcopy, [217](#)
- default value, [228](#)
- deletion of objects, [228](#)
- dependent variable, [164](#)
- derivative, [71](#)
- derivative\_rules.m, [74](#)
- design matrix, [199](#)
- desktop calculator, [214](#)
- det, [138](#)
- determinant, [136](#), [137](#)
- determinant.m, [138](#)
  
- DFT, [39](#)
- dft.m, [40](#)
- diagonalizable, [145](#), [308](#)
- diagonalization theorem, [146](#)
- diagonalization.m, [147](#)
- difference formula,  $(n + 1)$ -points, [94](#)
- difference formula, five-point, [96](#)
- difference formula, three-point, [95](#)
- difference formula, two-point, [93](#)
- difference quotient, [66](#)
- differentiable, [165](#)
- differentiate, [72](#)
- differentiation rules, [72](#)
- digital signals, [38](#)
- directional derivative, [168](#)
- discrete Fourier transform, [38](#), [39](#)
- discrete\_Fourier.m, [40](#)
- discrete\_Fourier\_inverse.m, [40](#)
- distance, [149](#)
- diverges, [79](#)
- doc, in Matlab, [18](#)
- domain, [164](#), [181](#)
- dot product, [13](#), [148](#)
- dot product preservation, [242](#)
- dot, in Matlab, [13](#)
- double threshold, [377](#)
- double\_threshold.m, [378](#)
- dual function, [337](#), [338](#), [342](#), [344](#)
- dual problem, [179](#), [336](#)
- dual variable, [179](#)
- duality gap, [340](#)
- duality\_convex.py, [347](#)
- dyadic decomposition, [312](#), [318](#)
- dynamic (variable) thresholds, [377](#)
  
- e, [58](#)
- e, as a limit, [58](#)
- e\_limit.m, [58](#)
- echelon form, [120](#)
- edge, [367](#), [368](#)
- edge detection, [366](#)
- edge normal direction, [351](#)
- edge normal directions, [375](#)
- edge sharpening, [370](#)
- edge thinning, [375](#)

- edge tracking, 378
- edge tracking rule, 378
- Edge\_Detection.m, 367
- effective programming, 9
- eig, 143, 162
- eigenvalue, 141, 142
- eigenvalues.m, 143
- eigenvector, 141
- elementary row operations, 117, 139
- ellipsoid, 308
- ensembling, 295, 298
- epigraph, 342, 345
- equivalent system, 114
- Euclidean norm, 151
- Euler's identity, 38, 109
- Euler's number, 58
- Euler, Leonhard, 58
- eulers\_identity.m, 109
- Excel, 39
- existence, 118
- explainable ai, 285
- explicit method, 370
- exponential function, 55
- exponential growth of error, 47
- exponential regression, 56
- eye, in Matlab, 16
  
- fast Fourier transform, 39
- fastest increasing direction, 173
- feasible set, 334
- FFT, 39
- Fibonacci sequence, 27
- Fibonacci\_sequence.m, 27
- fig\_plot.m, 17
- fimplicit, 109
- finite difference method, 93
- first-order necessary conditions, 344
- five-point difference formula, 96
- fmesh, in Matlab, 18
- folding frequency, 43
- for loop, 21
- forward Euler method, 370
- forward phase, 122, 125
- forward-difference, 93
- four essential components, 12, 19
  
- Fourier transform, 38
- fplot, in Matlab, 18
- free variable, 123
- free variables, 121
- free\_fall.m, 67
- frequency increment, 39
- frequency resolution, 39
- frequently\_used\_rules.py, 218
- Frobenius norm, 152
- fsurf, in Matlab, 18
- function, 53
- function of two variables, 164
- fundamental questions, two, 118
- Fundamental Theorem of Algebra, 104
  
- Galileo's law of free-fall, 66
- general solution, 123, 125
- get\_cubes.py, 220
- get\_hypothesis\_WLS.m, 209
- ggplot, 331
- global variable, 228
- golden ratio, 26
- gradient, 171, 189
- gradient descent algorithm, 187
- gradient descent method, 181, 184, 188, 272, 286
- gradient magnitude, 372
- Gram-Schmidt process, 248, 250
- Gram-Schmidt process, normalized, 249
- Green's Theorem, 32
- Guess The Weight of the Ox Competition, 260
  
- help, in Matlab, 7, 18
- Hessian, 189
- horizontal line test, 54
- Horner's method, 105, 228
- horner, in Python, 223
- horner.m, 107, 224
- hyperparameter, 273
- hyperplane, 266, 267
- hypothesis, 206
- hypothesis space, 322
- hysteresis, 366, 378
  
- identity function, 271

- IDFT, 39
- idft.m, 40
- imag, imaginaty part, 37
- image blur, 370, 371
- image compression, 312
- image denoising, 371
- image processing, 350
- image texts, 380
- imaginary part, 37
- imaginary unit, 35
- imgaussfilt, 370, 379
- inconclusive, 79
- inconsistent system, 115
- indentation, 215
- independent variable, 164
- induced matrix 2-norm, 154
- induced matrix norm, 152
- infinity-norm, 151
- information engineering, iii
- inheritance, 229
- initialization, 19, 225
- inlier.m, 210
- inliers, 204
- inner product, 148, 150
- instance, 225
- instantaneous speed, 66
- instantiation, 225
- intercept, 267
- interp\_error.py, 92
- interpolation, 194
- Interpolation Error Theorem, 90, 94
- interpretability, 264
- interval of convergence, 83
- inverse discrete Fourier transform, 39
- inverse Fourier transform, 38
- inverse function, 53, 54
- inverse power method, 159
- inverse\_matrix.m, 130
- inverse\_power.m, 160
- inverse\_power.py, 160
- invertible matrix, 129
- invertible matrix theorem, 132, 140, 142
- Iris\_perceptron.py, 270
- iris\_sklern.py, 294
- iteration, 19
- iterative algorithm, 155
- k-nearest neighbor, 281
- k-NN, 281
- Karush-Kuhn-Tucker conditions, 344
- KD-tree, 282
- KKT conditions, 344
- Kronecker delta, 87
- Krylov subspace methods, 188
- Lagrange dual function, 179, 336
- Lagrange dual problem, 336
- Lagrange form of interpolating polynomial, 87
- Lagrange interpolating polynomial, 87
- Lagrange multiplier, 174, 370
- Lagrange multipliers, 334, 338, 342
- Lagrange polynomial, 93
- Lagrange\_interpol.py, 89
- Lagrangian, 176, 177, 334, 337, 338, 342, 344
- lazy learner, 281
- learning rate, 268, 272
- least-squares line, 198
- least-squares problem, 195, 321
- least-squares solution, 195, 252, 321
- least\_squares.m, 197
- left singular vectors, 310, 315
- left-hand limit, 70
- length, 149
- length preservation, 242
- level curve, 173
- likelihood, 277
- linear algebra basics, 113
- linear combination, 234
- linear convergence, 48
- linear dependence, 126
- linear equation, 114
- linear growth of error, 47
- linear independence, 126
- linear SVM, 280
- linear system, 114
- linear\_equations\_rref.m, 119
- linearity rule, 74
- linearization, 202

- linearly dependent, 126
- linearly independent, 126
- linspace, in Matlab, 18
- list, in Python, 216
- little oh, 50, 51
- load, 32
- localization of roots, 104
- log-likelihood function, 277
- logarithmic function, 59
- logistic cost function, 277
- Logistic Regression, 277
- logistic regression, 276
- lower bound property, 336, 337
- LS problem, 195, 321
  
- M-file, 6
- machine learning, 53, 260, 380
- machine learning algorithm, 260
- machine learning modelcode, 296
- Machine\_Learning\_Model.py, 296
- Maclaurin series, 82
- majority vote, 298
- mathematical analysis, 4
- Matlab, 12, 353
- matlab\_conv2\_boundary.m, 354
- matrix 2-norm, 154
- matrix equation, 117
- matrix norm, 152
- matrix transformation, 247
- matrix-matrix multiplication, 16
- matrix-vector multiplication, 15, 148
- maximin problem, 179, 336–338, 342
- maximum useful frequency, 43
- maximum-norm, 151
- mean curvature, 370
- Mean Value Theorem, 85
- mesh, 36
- mesh, in Matlab, 18
- method of Lagrange multipliers, 280
- method of normal equations, 197, 209, 252, 321
- method, in Python class, 226
- microphone, 380
- midpoint formula, 110
- midpoint formula for  $f''$ , 96
  
- mini-batch, 287
- minimax problem, 177, 178, 335, 337, 338, 342
- Minimization Problem, 181
- minimum point set, 206
- minimum volume enclosing ellipsoid, 308
- Minkowski distance, 282
- MLESAC, 208
- MNIST data, 284
- mod, 24
- modelcode, 296, 368
- modularization, 8, 231
- module, 8
- modulo, 24
- monomial basis, 77, 78
- multi-class classification, 274
- multi-line comments, 215
- multiple local minima problem, 264
- multiple output, 26
- MVEE, 308
- myclf.py, 297
- mysort.m, 10
- mysqrt.m, 102
  
- natural logarithm, 60
- nested loop, 22
- nested multiplication, 105
- network.py, 288
- neuron, 265
- Newton's method, 99
- Newton-Raphson method, 99
- newton\_horner, in Python, 223
- newton\_horner.m, 108, 224
- No Free Lunch Theorem, 293
- noise reduction, 370
- non-maximum suppression, 375
- non\_max\_suppression.m, 375
- nonconvex problem, 348
- nonlinear regression, 202
- nonlinear SVMs, 280
- nonsingular matrix, 129
- nonvertical supporting hyperplane, 343, 346
- norm, 149, 151
- normal, 173

- normal equations, 196, 201
- normal matrix, 152
- normal vector, 280
- np.set\_printoptions, 157
- null space, 235
- nullity, 237
- numeric library, 359
- numerical approximation, 32
- numerical differentiation, 93
- numpy, 25, 212, 222, 357
- numpy.loadtxt, 331
- numpy.savetxt, 331
- Nyquist criterion, 43
  
- object-oriented programming, 225
- objective function, 181
- observation vector, 199
- OCR, 380
- Octave, 25
- Octave, how to import symbolic package, 36
- Octave, how to know if Octave is running, 36
- one-shot learning, 264
- one-to-one function, 54
- one-versus-all, 274
- one-versus-rest, 274
- OOP, 225
- operator 2-norm, 154
- operator norm, 152
- optical character recognition, 380
- optimal step length, 190
- optimization problem, 334, 338, 342
- orthogonal, 150, 243
- orthogonal basis, 238, 248, 249
- orthogonal complement, 243
- orthogonal decomposition theorem, 244
- orthogonal matrix, 242, 308
- orthogonal projection, 239, 244, 246
- orthogonal set, 238
- orthogonal\_matrix.m, 242
- orthogonality preservation, 242
- orthonormal basis, 241, 248, 249
- orthonormal set, 241
- outer product, 349
  
- outliers, 204
- overfitting, 263
  
- p-norms, 151
- parameter estimation problem, 204
- parameter vector, 199
- parametric description, 123
- parametric vector form, 133
- parent class, 229
- partial derivative, 166
- PCA, 304
- pca\_regression.m, 325
- pdfim2text, 380–382
- peppers\_compress.m, 313
- perceptron, 267
- perceptron.py, 269
- pivot column, 121, 126
- pivot position, 121
- plot, in Matlab, 17
- polynomial approximation, 81
- polynomial interpolation, 53
- Polynomial Interpolation Error Theorem, 90, 95
- Polynomial Interpolation Theorem, 86
- polynomial of degree  $n$ , 104
- Polynomial\_01.py, 226
- Polynomial\_02.py, 227
- population of the world, 56
- population.m, 57
- positive definite, 188
- positive semidefinite, 350
- positive semidefinite matrix, 309
- post-processing, 353
- power iteration, 155
- power method, 155
- power rule, 74
- power series, 78
- power-of-2 restriction, 39
- power\_iteration.m, 157
- power\_iteration.py, 157
- principal component analysis, 304
- principal components, 311
- principal directions, 304, 306, 308
- probabilistic model, 276
- product rule, 74



- programming, [iii](#), [2](#), [8](#)
- PROSAC, [208](#)
- pseudocode, [47](#)
- pseudoinverse, [197](#), [252](#), [321](#), [323](#)
- pseudoinverse, the  $k$ -th, [323](#)
- PSNR, [314](#)
- Pythagorean theorem, [150](#), [246](#)
- Python, [212](#), [353](#)
- Python essentials, [215](#)
- Python script, [363](#)
- Python wrap-up, [213](#), [361](#)
- Python\_calls\_F90\_GCC.py, [361](#)
- python\_startup.py, [214](#)
  
- QR factorization, [250](#), [252](#)
- QR factorization algorithm, [251](#)
- QR factorization for least-squares problem, [252](#)
- QR iteration, [253](#)
- qr\_iteration.m, [254](#)
- quadratic convergence, [48](#)
- quadratic formula, [35](#), [351](#)
- quantization, [366](#)
- quotient rule, [74](#)
  
- R-RANSAC, [208](#)
- R&D, [76](#)
- radius of convergence, [80](#)
- random orthogonal matrix, [242](#)
- random sample consensus, [206](#)
- range, [164](#)
- range, in Python, [217](#)
- rank theorem, [237](#)
- rank-1 matrix, [349](#)
- rank-one matrix, [349](#)
- RANSAC, [206](#)
- ransac2.m, [210](#)
- ratio test, [79](#), [83](#)
- Rayleigh quotient, [309](#)
- readmatrix, [33](#)
- real part, [37](#)
- real, real part, [37](#)
- real-valued solution, [35](#), [37](#)
- real\_imaginary\_parts.m, [37](#)
- real\_STFT.m, [62](#)
  
- rectifier, [275](#)
- reduced echelon form, [120](#)
- reduced row echelon form, [119](#)
- REF, [120](#)
- reference semantics, in Python, [217](#)
- region, [30](#)
- regression analysis, [53](#), [198](#)
- regression coefficients, [198](#)
- regression line, [198](#)
- Regression\_Analysis.m, [326](#)
- Regression\_Analysis.py, [330](#)
- relative error, [47](#)
- Remainder Theorem, [105](#)
- remarks on Python implementation, [231](#)
- repetition, [6](#), [19](#)
- research and development, [76](#)
- retrieving elements, in Python, [216](#)
- reusability, [5](#), [6](#), [231](#)
- reverse, [53](#)
- rgb2gray formula, [366](#)
- Richardson's method, [181](#)
- right singular vectors, [310](#), [315](#)
- right-hand slope, [70](#)
- Rosenbrock function, [185](#)
- rosenbrock\_2D\_GD.py, [185](#)
- rotational symmetry, [275](#)
- row equivalent, [117](#)
- row reduced echelon form, [120](#)
- row reduction algorithm, [122](#)
- RREF, [120](#)
- rref, [119](#)
- Run\_network.py, [290](#)
  
- sampling, [366](#)
- save multiple functions in a file, [325](#)
- saveas, [32](#), [33](#)
- scalar multiplication, [234](#)
- scatter plot, [56](#)
- scene analysis, [204](#)
- Schur decomposition, [253](#)
- Scikit-learn, [292](#)
- SciPy, [353](#)
- scipy, [212](#), [357](#)
- scipy.signal.convolve2d, [353](#)
- scipy\_convolve\_boundary.py, [354](#)

- score matrix, 309
- search direction, 181, 188, 272
- secant\_lines\_abs\_x2\_minus\_1.m, 70
- second-derivative midpoint formula, 96
- self, 226
- sequence\_sqrt2.m, 49
- SGD, 187
- shared objects, 213
- sharing boundaries, 225
- short-time Fourier transform, 44
- short\_time\_DFT.m, 45
- signal\_DFT.m, 41
- similar, 144
- similarity transformation, 144
- sinc function, 83
- singular value decomposition, 310, 311, 315
- singular values, 310, 315
- sklearn.ensemble.VotingClassifier, 298
- sklearn\_classifiers.py, 298
- Slater's condition, 340
- Slater's Theorem, 340
- slicing, in Python, 216
- slope of the curve, 69
- smoothing assumption, 204
- Sobel derivative, 353
- Sobel gradient, 372
- Sobel kernels, 372
- sobel\_derivatives.m, 352
- sobel\_grad.m, 372, 373
- soft-margin classification, 280
- softplus function, 275
- solution, 114
- solution set, 114
- SortArray.m, 11
- span, 128
- spectrogram, 44
- speech2text, 380
- Speed up Python Programs, 212, 357
- sqrt, 103
- square root, 102
- squareroot\_Q.m, 3
- squaresum.m, 6
- SSE, 266
- stability, 370
- stable, 47
- staircasing effect, 370
- stand-alone functions, 231
- standard basis, 236
- standard basis for  $\mathbb{R}^n$ , 77
- standard unit vectors, 77
- Starkville, 63
- stdt.m, 45
- steepest descent direction, 189
- steepest descent method, 181
- step length, 181, 188, 272, 278
- STFT, 44
- stft2.m, 46
- stochastic gradient descent, 187, 287
- string, in Python, 216
- strong duality, 339, 346
- structure tensor, 349, 350
- structure\_tensor.m, 352
- submatrix, 137
- subordinate 2-norm, 154
- subordinate norm, 152
- subspace, 234
- sufficient decrease condition, 186
- Sum of Squared Errors, 266
- super-convergence, 102
- supergraph, 342
- superlinear convergence, 48
- supervised learning, 261
- support vector machine, 279
- surf, in Matlab, 18
- SVD, 310
- SVD theorem, 315
- SVD, algebraic interpretation, 317
- symbolic computation, 67
- symmetric, 308
- symmetric positive definite, 188
- synthetic division, 105
- system, 53
- system of linear equations, 113, 114
- tangent line, 66, 69, 100
- Taylor polynomial of order  $n$ , 83
- Taylor series, 81, 82, 97
- Taylor series, commonly used, 83
- Taylor's formula, 182

- Taylor's series formula, 97
- Taylor's Theorem, 84, 96
- Taylor's Theorem with integral remainder, 183
- Taylor's Theorem with Lagrange Remainder, 84, 96
- Taylor's Theorem, Alternative Form of, 97
- taylor, in Matlab, 83
- Term-by-Term Differentiation, 80
- term-by-term integration, 80
- test\_denoising.m, 371
- test\_f90.f90, 358
- test\_gpp.cpp, 359
- test\_py3.py, 360
- three tasks, 260
- three-point difference formula, 95
- three-point formulas, 95
- time-stepping procedure, 370
- total variation, 370
- trace, 152
- training data, 261
- transform\_to\_gray.m, 368
- transpose, 131
- truncated data matrix, 311
- truncated score matrix, 311
- tuple, in Python, 216
- TV, 370
- tv\_denoising.m, 370
- two-class classification, 265
- two-point difference formula, 93
  
- Ubuntu, 292
- unique inverse, 129
- uniqueness, 118
- unit circle, 173
  
- unit vector, 149, 241
- unstable, 47
- unsupervised learning, 262
- update direction, 272
- upper triangular matrix, 253
- util.m, 324
- util.py, 330
- util\_Covariance.py, 306
- util\_Poly.py, 230
  
- variance, 305
- vector norm, 151
- vector space, 234
- visualize\_complex\_solution.m, 36
- volume scaling factor, 136
- VotingClassifier, 298
  
- weak duality, 339
- Weierstrass Approximation Theorem, 86
- weight matrix, 204
- weighted least-squares method, 204
- weighted normal equations, 205
- while loop, 20
- why, in Matlab, 16
- win\_cos.m, 45
- Wine\_data.py, 329
- writematrix, 33
  
- x-intercept, 100
- XAI, 285
  
- zero padding, 353
- zero vector, 234
- Zeros-Polynomials-Newton-Horner.py, 223
- zeros\_of\_poly\_built\_in.py, 222