

Springer Undergraduate Texts
in Mathematics and Technology

SUMAT



Makoto Tsukada · Yuji Kobayashi ·
Hiroshi Kaneko · Sin-Ei Takahasi ·
Kiyoshi Shirayanagi · Masato Noguchi

Linear Algebra with Python

Theory and Applications

Springer Undergraduate Texts in Mathematics and Technology

Series Editors

Helge Holden, Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway

Keri A. Kornelson, Department of Mathematics, University of Oklahoma, Norman, OK, USA

Editorial Board

Lisa Goldberg, Department of Statistics, University of California, Berkeley, Berkeley, CA, USA

Armin Iske, Department of Mathematics, University of Hamburg, Hamburg, Germany

Palle E. T. Jorgensen, Department of Mathematics, University of Iowa, Iowa City, IA, USA

Springer Undergraduate Texts in Mathematics and Technology (SUMAT) publishes textbooks aimed primarily at the undergraduate. Each text is designed principally for students who are considering careers either in the mathematical sciences or in technology-based areas such as engineering, finance, information technology and computer science, bioscience and medicine, optimization or industry. Texts aim to be accessible introductions to a wide range of core mathematical disciplines and their practical, real-world applications; and are fashioned both for course use and for independent study.

Makoto Tsukada · Yuji Kobayashi ·
Hiroshi Kaneko · Sin-Ei Takahasi ·
Kiyoshi Shirayanagi · Masato Noguchi

Linear Algebra with Python

Theory and Applications



Springer

Makoto Tsukada
Department of Information Science
Toho University
Funabashi, Chiba, Japan

Hiroshi Kaneko
Laboratory of Mathematics and Games
Funabashi, Chiba, Japan

Kiyoshi Shirayanagi
Department of Information Science
Toho University
Funabashi, Chiba, Japan

Yuji Kobayashi
Laboratory of Mathematics and Games
Funabashi, Chiba, Japan

Sin-Ei Takahasi
Laboratory of Mathematics and Games
Funabashi, Chiba, Japan

Masato Noguchi
Department of Information Science
Toho University
Funabashi, Chiba, Japan

ISSN 1867-5506 ISSN 1867-5514 (electronic)
Springer Undergraduate Texts in Mathematics and Technology
ISBN 978-981-99-2950-4 ISBN 978-981-99-2951-1 (eBook)
<https://doi.org/10.1007/978-981-99-2951-1>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use. The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd. The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore



The Scholar Cat (by Ai Takahashi)

Preface

Welcome to the wonderland of linear algebra! The world of linear algebra is full of many elegant theories that lead to powerful applications. Let us explore this exciting realm using the programming language Python as a tool.

About This Book

This book is a textbook for learning the basic theory of linear algebra with awareness of its potential applications. We can better grasp the specific uses and applications of linear algebra by understanding the mathematical theory that underpins it.

Linear algebra begins with the theory of vectors and matrices. In high school mathematics, vectors appear in plane or space geometry and apply to areas of physics such as mechanics. However, this is just one aspect of vectors. Functions such as polynomials and trigonometric functions can also be regarded as vectors. In this book, the reader will learn how to treat even sounds and images as vectors. Calculations related to vectors and matrices are collectively called linear calculations, and the differentiation and integration of functions can be regarded as types of linear calculation. In physics, the state of a system is formulated by a differential equation, and in order to solve it, the eigenvalue problem in linear algebra must be addressed. In engineering, a technique called Fourier analysis is used in fields such as speech processing, image processing, communication theory, and control theory. It is closely related to the notion of orthogonality defined through the inner product of vectors. The calculations used in probability theory and statistics are mainly integral calculations and are associated with simultaneous linear equations and eigenvalue problems.

Today, we often encounter terms such as AI (artificial intelligence) and big data. In order to properly understand these new fields, a solid grasp of linear algebra and its uses is becoming more and more important. In this book, we will build the theory of linear algebra from the rudiments to the essential concepts, occasionally referring to the applications mentioned above.

For undergraduate students in science or engineering departments, calculus (analysis) and linear algebra are required subjects. It might be easy to imagine the utility of concrete examples such as velocity and acceleration for differentiation and area and volume for integration. However, it is harder to understand the usefulness of linear algebra concepts such as matrix multiplication, determinants, inverse matrices, and eigenvalues. Students learn the manual computational techniques in their linear algebra classes. However, as you can easily imagine from the above application examples, we need large-scale calculations, most of which are beyond the scale of manual calculations. This is why computers are both effective and indispensable in linear algebra.

We may devise an effective computer program to solve a problem if we know how to solve it by hand. Today, however, there are various linear computing tools that are already coded. In many cases, we do not even have to write our own code. The ability to identify the most appropriate tools and use them effectively is more important. We need to know various cases in which linear algebra is applied, and more importantly, we need to understand the principles underlying the utility of the tool in each individual case.

Chapters 9 and 10 cover several such cases. Before we are ready for these chapters, however, we will explain, in Chaps. 1–8, most of the theories of linear algebra in a step-by-step manner. Along the way, each mathematical fact will be given as much original proof as possible, without cutting corners. In concrete numerical calculations, we will explain the use of the programming language Python as a calculator.

Why Python?

Python is a free and open-source programming language that provides many useful tools for doing various calculations. It can treat fractions as fractions, calculate expressions containing character constants and variables, and display the results as character expressions. Moreover, Python has many external libraries for special purposes, such as image processing, numerical computation, symbolic computation, graph drawing, and so on. For example, SymPy is the library for performing symbolic computations or computer algebra, such as solving linear or algebraic equations and factorization of polynomials. In addition to SymPy, there are many free computer algebra systems such as Maxima, Reduce, and SageMath, as well as commercial computer algebra systems such as Maple, Mathematica, and Magma. However, most of these are huge systems that aspire to be full-featured mathematics systems. On the other hand, SymPy is compact and lightweight, but sufficient for performing linear algebra.

Python has an interactive mode, in which we can proceed with formula transformation while checking each calculation step as if we were using a calculator. We can also see two- or three-dimensional vectors displayed during the calculation. Computational problems specific to linear algebra, such as finding determinants, inverse matrices, and eigenvalues, can also be

solved in one-line code. Indeed, many of the exercises presented in college-level linear algebra textbooks can be immediately solved using Python. We can also create new exercises, and with a little ingenuity, even exercises suitable for manual calculations with paper and pencil. In fact, one of the authors, who has been teaching linear algebra for many years, has routinely assigned his class problems that were randomly generated in Python, so that his students will learn to solve different numerical problems of similar difficulty.

However, we cannot understand the true benefits of linear algebra using such exercise-level calculations. For that purpose, it is better to handle large-scale (high-dimensional) data such as audio and images. Readers will find that linear algebra is more interesting if they use the techniques described in this book to process their own images and sound recordings. It would not be an impractical goal to implement what we learned on a small microcomputer board such as Raspberry Pi to make an AI robot. It is amazing that Python code as it is described in this book almost always runs at practical speed.

About the Structure and Reading of This Book

The relationship between the chapters in this book is shown in Fig. 1. Chapters 3–5 and 7 contain many theories peculiar to finite-dimensional linear spaces, which are the essence of linear algebra. Some of the applications covered in this book require that we build a theory in an infinite-dimensional linear space. This theory will lead us to the field of functional analysis; Chap. 6 contains some topics related to this field. The reader can skip the more difficult calculations of determinants and inverse matrices in Chap. 5, which might risk engendering an aversion to linear algebra. Then, after Chaps. 6 and 7, it may be a good idea to read Chap. 10 on the applications of linear algebra first, bypassing the more challenging Chaps. 8 and 9. The readers unfamiliar with Python should read Appendix first, where it is explained how to set up the environment for Python and how to use it. Here is an overview of each chapter.

- **Chapter 1 “Mathematics and Python”**

We review the minimum requirements of mathematics, such as propositions, numbers, sets, mappings, etc., to learn the concepts underpinning linear algebra. We also learn how these concepts are expressed in Python.

- **Chapter 2 “Linear Spaces and Linear Mappings”**

We introduce the concepts of linear space, which is the stage set for linear algebra, and linear mapping, which plays a leading role in linear algebra. We explain linear algebra from an abstract linear space defined with axioms.

- **Chapter 3 “Basis and Dimension”**

We learn the concept of basis. If there is a basis of size n in an abstract linear space, we can regard it as a concrete n -dimensional space consisting of vectors with n components. A basis can be thought of as a coordinate system of a linear space, and one of the main themes in linear algebra is how to choose a convenient coordinate system.

- **Chapter 4 “Matrices”**

We learn that a linear mapping can be represented by a matrix. The meaning of operations such as matrix multiplication becomes clear here.

- **Chapter 5 “Elementary Operations and Matrix Invariants”**

We learn about the basic transformation of matrices, and we use this transformation to calculate determinants and inverse matrices, or to solve simultaneous linear equations. A matrix is a representation of a linear mapping through a basis, and when the basis is changed, the expression of the matrix changes as well. However, there are original features (invariants) associated with a linear mapping that remain unchanged under any representation. We learn how to find the features of such a mapping using elementary operations.

- **Chapter 6 “Inner Product and Fourier Expansion”**

We learn about the inner product, where the product of vectors is a scalar. We introduce the notion of orthogonality between vectors through the inner product and show how this concept develops into a method called Fourier analysis. Fourier analysis plays a major role in mathematics, engineering, physics, and more, and some of its applications are explained here.

- **Chapter 7 “Eigenvalues and Eigenvectors”**

We learn about the eigenvalues and eigenvectors of matrices, which are especially important notions in linear algebra. The eigenvalues are features that cannot be obtained merely by the elementary operations of a matrix, but they appear clearly when the matrix is diagonalized.

- **Chapter 8 “Jordan Normal Form and Spectrum”**

We learn to transform and analyze a not-necessarily diagonalizable matrix into what is called the Jordan normal form. The Jordan normal form is one of the final destinations of matrix theory. As an application of this, we prove the Perron-Frobenius theorem, which often appears in applications.

- **Chapter 9 “Dynamical Systems”**

We introduce the analysis of the behavior of the solution of a linear differential equation as an application of the Jordan normal form, and the ergodic theorem of a Markov chain as an application of the Perron-Frobenius theorem. The former is a system that changes deterministically over time, and the latter is a system that changes stochastically.

- **Chapter 10 “Applications and Development of Linear Algebra”**

We discuss singular value decomposition and the generalized inverse matrix with some concrete applications. We rigorously explain the essential parts of these application examples so that they can be completed only here. We hope this will be a stepping stone that inspires readers to consult more specialized books to examine these themes individually.

- **Appendix**

We show how to prepare the computer, including the installation of Python, in order to read this book. Those who already have a computer environment set up can proceed to the first chapter.

Traditionally, first-year college linear algebra textbooks set the Jordan normal form as the final goal, but do not touch on singular value decompositions or generalized inverse matrices. However, in order to understand the

Jordan normal form, there are hurdles that must be overcome, such as the analysis of the generalized eigenspace. On the other hand, the singular value decomposition and the generalized inverse matrix are the consequences of both the simultaneous equation theory and the eigenvalue problem, which are the two major themes of linear algebra. Moreover, they are adaptable to any matrices including non-square matrices, and are presently used in applications more than the Jordan normal form. We think that these topics should be taken up not only due to their importance but also from the aspect of motivation for learning linear algebra. We hope that this book will “throw a stone”¹ at the current thinking about the curriculum of linear algebra.

The reader can download all of the code for the programs described in this book from the website below. In addition, they can also access the code that created the inset and other diagrams, as well as the solutions to the exercises from the same site. The website also provides the latest information about Python: <https://www.math-game-labo.com/>.

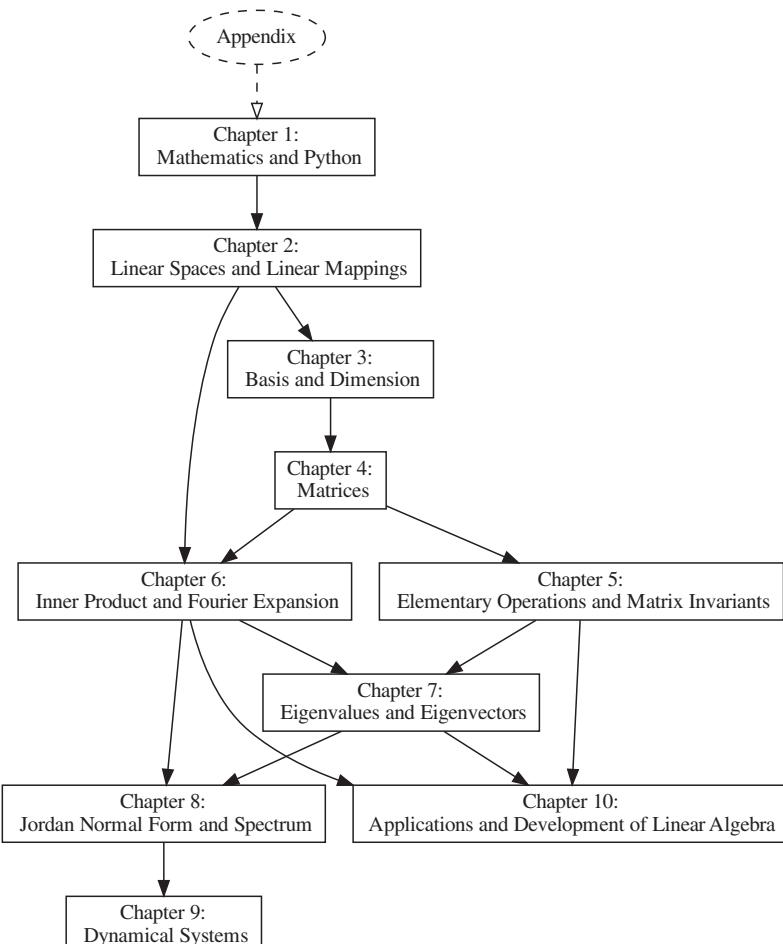


Fig. 1 Structure of this book

¹This is a Japanese idiom meaning “introduce a new topic for discussion.”

We would like to thank Mr. Masayuki Nakamura from Springer Japan for helping us share this book with the world. We would also like to thank Mr. Hiroshi Yahagi for providing a laboratory as a platform for discussions among the co-authors. Lastly, we express our sincere appreciation to the families of all our co-authors for allowing us to concentrate on writing this book.

Kashiwa, Japan
Funabashi, Japan
Saitama, Japan
Funabashi, Japan
Yokohama, Japan
Funabashi, Japan

Makoto Tsukada
Yuji Kobayashi
Hiroshi Kaneko
Sin-Ei Takahasi
Kiyoshi Shirayanagi
Masato Noguchi

Contents

1	Mathematics and Python	1
1.1	Propositional Logic	1
1.2	Numbers	3
1.3	Sets	5
1.4	Ordered Pairs and Tuples	10
1.5	Mappings and Functions	12
1.6	Classes and Objects in Python	14
1.7	Lists, Arrays and Matrices	17
1.8	Preparation of Image Data	19
1.8.1	Binarization of Image Data with PIL and NumPy	19
1.8.2	GUI for Creating Complex-Valued Data of Handwritten Characters	23
1.8.3	Data of Handwritten Letters with Grayscale	25
2	Linear Spaces and Linear Mappings	29
2.1	Linear Spaces	29
2.2	Subspaces	37
2.3	Linear Mappings	39
2.4	Application: Visualizing Sounds	42
3	Basis and Dimension	47
3.1	Finite-Dimensional Linear Spaces	47
3.2	Linear Dependence and Linear Independence	52
3.3	Basis and Representation	55
3.4	Dimension and Rank	58
3.5	Direct Sums	63
3.6	Remarks on Dimension	65
4	Matrices	69
4.1	Matrix Operations	69
4.2	Matrices and Linear Mappings	73
4.3	Composition of Linear Mappings and Product of Matrices	78

4.4	Inverse Matrix, Basis Change, and Similarity of Matrices	86
4.5	Adjoint Matrix	89
4.6	Measuring Matrix Computation Time	90
5	Elementary Operations and Matrix Invariants	93
5.1	Elementary Matrices and Operations	93
5.2	Rank	99
5.3	Determinant.	101
5.4	Trace	110
5.5	Systems of Linear Equations	111
5.6	Inverse Matrix	116
6	Inner Product and Fourier Expansion	121
6.1	Norm and Inner Product	121
6.2	Orthonormal Systems and Fourier Transform	125
6.3	Function Spaces	133
6.4	Least Squares, Trigonometric Series, and Fourier Series.	135
6.5	Orthogonal Function Systems.	140
6.6	Convergence of Vector Sequences	144
6.7	Fourier Analysis	146
7	Eigenvalues and Eigenvectors	153
7.1	Unitary Matrices and Hermitian Matrices	153
7.2	Eigenvalues	156
7.3	Diagonalization.	164
7.4	Matrix Norm and Matrix Functions	171
8	Jordan Normal Form and Spectrum	179
8.1	Direct Sum Decomposition.	179
8.2	Jordan Normal Form.	181
8.3	Jordan Decomposition and Matrix Power.	190
8.4	Spectrum of a Matrix	193
8.5	Perron–Frobenius Theorem.	198
9	Dynamical Systems	203
9.1	Differentiation of Vector-(Matrix-) Valued Functions	203
9.2	Newton’s Equation of Motion.	205
9.3	Linear Differential Equations	209
9.4	Stationary States of Markov Chain.	215
9.5	Markov Random Fields.	219
9.6	One-Parameter Semigroups	226
10	Applications and Development of Linear Algebra	231
10.1	Linear Equations and Least Squares.	231
10.2	Generalized Inverse and Singular Value Decomposition	237
10.3	Tensor Products.	242

10.4	Tensor Product Representation of Vector-Valued Random Variables	250
10.5	Principal Component Analysis and KL Expansion.	254
10.6	Estimation of Random Variables by Linear Regression Models.	262
10.7	Kalman Filter	267
Appendix	271
Afterword and Bibliography	293
Symbol Index	297
Python Index	299
Index	303



In this chapter, we survey the notions of propositions, real and complex numbers, sets, and mappings (functions) that are the basics of mathematics needed to study linear algebra. We will also learn how these concepts are expressed in Python. Let us learn mathematics in a practical way by using Python.

1.1 Propositional Logic

In mathematics, a statement that describes a “matter”,¹ which is subject to judgment as to whether it is *true* or *false*, is called a *proposition*. True/false is called the *truth value* of a proposition. The truth value of the proposition “2 is a prime number” is true, and the value of “6 is a prime number” is false.

Suppose that P and Q are propositions. Then, $\neg P$ (not P), $P \wedge Q$ (P and Q), $P \vee Q$ (P or Q), $P \rightarrow Q$ (P implies Q) and $P \leftrightarrow Q$ (P if and only if Q) are also propositions whose truth values are determined by tables shown in Table 1.1 called *truth tables*.

In the context of Python, a truth value is called a *Boolean value*, and the values true and false are expressed by literals² True and False, respectively. In Python $\neg P$, $P \wedge Q$ and $P \vee Q$ are expressed by not P , P and Q and P or Q , respectively. Python does not provide logical operators for \rightarrow and \leftrightarrow , but since True/False is represented by an integer 1/0 in Python, the inequality sign $<=$ and equality sign $==$ can be used for \rightarrow and \leftrightarrow , respectively. Python treats expressions in Boolean values (also Boolean values themselves) as objects.

We can confirm the truth value tables in Table 1.1 by the following simple Python codes:

Program: table1.py

```
In [1]: 1 | for P in [True, False]:  
2 |     print(P, not P)
```

```
True False  
False True
```

In this book, the icon represents the output by the print function in interactive mode.

¹ A matter is distinguished from a “thing” that is an element of a set stated in Sect. 1.3.

² A literal is a constant value written in Python code. In addition to True and False, the real number 3.14 and string ‘Python’ are literals, for example.

Table 1.1 The truth value tables

P	$\neg P$
true	false
false	true

P	Q	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
true	true	true	true	true	true
true	false	false	true	false	false
false	true	false	true	true	false
false	false	false	false	true	true

Program: table2.py

```
In [1]: 1 | for P in [True, False]:
2 |     for Q in [True, False]:
3 |         print(P, Q, P and Q, P or Q, P <= Q, P == Q)
```



```
True True True True True True
True False False True False False
False True False True True False
False False False False True True
```

If the truth value tables of P and Q coincide, we write $P \Leftrightarrow Q$. In this case, we say P is *equivalent* to Q or P holds *if and only if* Q holds. For example, by looking into the table above, we see $P \wedge Q \Leftrightarrow Q \wedge P$, $P \vee Q \Leftrightarrow Q \vee P$, and $P \leftrightarrow Q \Leftrightarrow Q \leftrightarrow P$.

Exercise 1.1 Show the equivalences

$$P \rightarrow Q \Leftrightarrow \neg P \vee Q \quad \text{and} \quad P \leftrightarrow Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$$

by making the truth value tables for them. Moreover, confirm the equivalences by Python.

Exercise 1.2 Prove the associative laws

$$(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R), \quad (P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$$

and the distributive laws

$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R), \quad P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R).$$

A proposition including a variable x , in which the truth value changes depending on the value of x such as “ x is an integer greater than 2” or “ x is a prime number”, is called a *propositional function*. Let $P(x)$ and $Q(x)$ be propositional functions. The expression

$$P(x) \Rightarrow Q(x)$$

means that the truth value of $Q(x)$ is true whenever the truth value of $P(x)$ is true regardless of the value of variable x , and we read it as $P(x)$ implies $Q(x)$. This also means that the propositional function $P(x) \rightarrow Q(x)$ is always true regardless of the value of x . For example, we use it to say

$$x \text{ is a multiple of } 6 \Rightarrow x \text{ is even.}$$

On the other hand, $P(x) \Leftrightarrow Q(x)$ means that the truth values of $P(x)$ and $Q(x)$ always match for any value of the variable x . This means that $P(x) \Leftrightarrow Q(x)$ is true for any value of x , or equivalently both $P(x) \Rightarrow Q(x)$ and $Q(x) \Rightarrow P(x)$ hold. For example,

$$x \text{ is even and a multiple of 3} \Leftrightarrow x \text{ is a multiple of 6.}$$

Let $P(n)$ be a propositional function in natural number n . If $P(1)$ is true and $P(n) \Rightarrow P(n+1)$ holds, then we conclude that $P(n)$ is true for every natural number n . This reasoning is called a *mathematical induction*.

1.2 Numbers

The following algebraic properties for real numbers are well known and hold even for complex numbers³:

$$\begin{array}{lll} \text{R1. } x + y = y + x, & \text{R2. } (x + y) + z = x + (y + z), & \text{R3. } x + 0 = x, \\ \text{R4. } x + (-x) = 0, & \text{R5. } xy = yx, & \text{R6. } (xy)z = x(yz), \\ \text{R7. } 1x = x, & \text{R8. } x \neq 0 \Rightarrow \frac{1}{x} \cdot x = 1, & \text{R9. } x(y + z) = xy + xz. \end{array}$$

Exercise 1.3 Deduce the equalities $0x = x0 = 0$ from properties R1–R9.

For a complex number $z = x + iy$ where i is the imaginary unit $\sqrt{-1}$ and x and y are real numbers, we call x and y the *real part* and the *imaginary part* of z and denote them by $x = \operatorname{Re} z$ and $y = \operatorname{Im} z$, respectively. The *absolute value* $|z|$ and the *complex conjugate* \bar{z} of z are defined by

$$|z| \stackrel{\text{def}}{=} \sqrt{x^2 + y^2} \quad \text{and} \quad \bar{z} \stackrel{\text{def}}{=} x - iy,$$

where the notation $\stackrel{\text{def}}{=}$ is used to mean that the left-hand side is defined by the expression of the right-hand side. With these notations the following properties are satisfied:

$$\begin{array}{lll} \text{Z1. } |z|^2 = z\bar{z}, & \text{Z2. } |z_1 z_2| = |z_1| |z_2|, & \text{Z3. } z = \bar{z} \Leftrightarrow z \text{ is real,} \\ \text{Z4. } \overline{z_1 + z_2} = \overline{z_1} + \overline{z_2}, & \text{Z5. } \overline{z_1 \cdot z_2} = \overline{z_1} \cdot \overline{z_2}, & \text{Z6. } |z_1 + z_2| \leq |z_1| + |z_2|. \end{array}$$

Exercise 1.4 Prove properties Z1–Z6 above from properties R1–R9 of real numbers.

Every complex number z has an expression

$$z = |z|(\cos \theta + i \sin \theta)$$

called the *polar representation*, where θ is a real number called the *argument* of z . When $z = 0$, we do not define its argument. We can find in a textbook of elementary analysis the Maclaurin expansions

³ We can deduce these properties from the Peano axioms of natural numbers. It is a long journey, starting with proving the properties of natural numbers and extending them to the cases of integers, rational numbers, real numbers, and complex numbers. We omit it here.

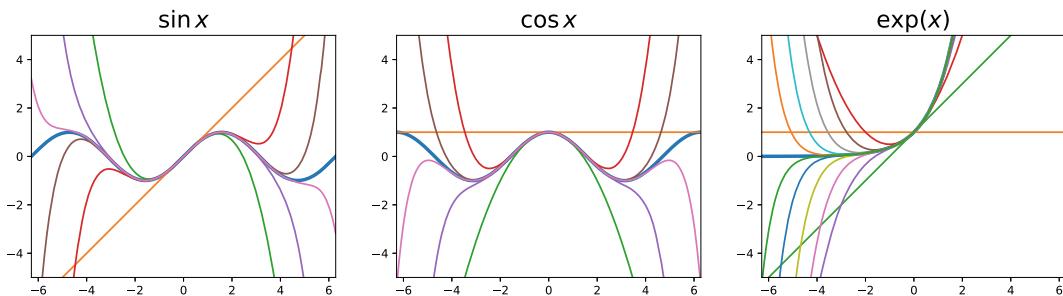


Fig. 1.1 Behaviors of convergence for Maclaurin's expansions of $\sin x$, $\cos x$, e^x

$$\begin{aligned}\sin x &= \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, \\ e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots\end{aligned}$$

of the functions $\sin x$, $\cos x$ and e^x (Fig. 1.1).⁴ Substitute x by $i\theta$, replace i^2 with -1 , rearrange the order of additions in the last expression, and compare to the first two expressions; then we obtain

$$e^{i\theta} = \cos \theta + i \sin \theta.$$

This is the well-known *Euler's formula*, and from this, the polar representation of z is expressed by

$$z = |z| e^{i\theta}.$$

Exercise 1.5 Prove the following formulas:

- (1) $e^{i(\theta_1+\theta_2)} = e^{i\theta_1} e^{i\theta_2}$ for any $\theta_1, \theta_2 \in \mathbb{R}$. (Hint: use the trigonometric addition theorem.)
- (2) $(e^{i\theta})^n = e^{in\theta}$ (**de Moivre's theorem**) for any natural number n . (Hint: use mathematical induction.)

Exercise 1.6 Using Python and its library Matplotlib, draw the graphs in Fig. 1.1.

Let $z = ae^{i\theta}$ with $a > 0$ and $2\pi > \theta \geq 0$. For a natural number n , let

$$w = \sqrt[n]{a} e^{\frac{i\theta}{n}},$$

then by de Moivre's theorem, we have $w^n = z$. We call this w the n -root of z and denote it by $\sqrt[n]{z}$.

Python can handle integers and real numbers. They are stored differently in the computer memory and are distinguished as an `int` class object and as a `float` class object. The ranges of numbers that can be handled are also limited. For now, we do not need to be so careful about these differences and limitations. We will explain these when there are cases where we should be careful.

⁴The reader is advised to see a textbook on analysis for a discussion of convergence of these series.

Complex numbers are also available in Python by default. Using `j`, the imaginary unit is expressed by `1j` or `1.0j`.⁵ The following is a dialogue to find the real part, the imaginary part, the absolute value, and the complex conjugate of $x = 1 + 2i$.

```
In [1]: x = 1 + 2j
x.real, x.imag, abs(x), x.conjugate()
```

```
Out[1]: (1.0, 2.0, 2.23606797749979, (1-2j))
```

Moreover, putting $y = 3 + 4i$, let us calculate the sum $x + y$, the product xy , and the quotient x/y .

```
In [2]: y = 3 + 4j
x + y, x * y, x / y
```

```
Out[2]: ((4+6j), (-5+10j), (0.44+0.08j))
```

Using $e = 2.718281828459045$ and $\pi = 3.141592653589793$ as approximations of the logarithmic constant and the circular constant respectively, let us calculate $e^{\pi i}$. We use the power operation `**` in Python.

```
In [3]: 2.718281828459045 ** 3.141592653589793j
```

```
Out[3]: (-1+1.2246467991473532e-16j)
```

Comparing to the real part, the imaginary part has a very small numerical error $1.2246467991473532 \times 10^{-16}$ that is almost 0. In the library NumPy of Python, names of the circular constant `pi` and the logarithmic constant `e` are defined, but these constants contain numerical errors and cause the same error.

```
In [4]: from numpy import pi, e
e**(pi * 1j)
```

```
Out[4]: (-1+1.2246467991473532e-16j)
```

1.3 Sets

An object is a “thing” that appears in mathematics, such as numbers, points, etc. A collection of objects is also an object called a *set*. Each collected object is called an *element* of the set. A set must have a clear range of its elements. There are two standard ways to express the range. One way is to list all elements of a set. This is called the *extensional definition*. For example, the set of all prime numbers less than 10 is denoted by

$$\{2, 3, 5, 7\},$$

and the set of all planets of the solar system is

$$\{\text{Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune}\}.$$

⁵ The use of the letter `j` for the imaginary unit comes from the practice of electrical engineers who use the letter `i` to represent electric current. Placing a literal of an `int` class object or a `float` class object just before the symbol `j`, it becomes a literal of a `complex` class object expressing a purely imaginary number.

The other way is called the *intensional definition* which uses a property that exactly characterizes the elements of a set. The two examples above are expressed in the intensional way as

$$\{x \mid x \text{ is a prime number less than } 10\}$$

and

$$\{x \mid x \text{ is a planet of the solar system}\},$$

respectively. As another example of an intensional definition,

$$\left\{ \frac{m}{n} \mid m \text{ is an integer and } n \text{ is a positive integer} \right\}$$

expresses the set of all fractions (rational numbers).

If x is an element of a set A , we denote this by $x \in A$ and say that x *belongs to* A . The negation of $x \in A$ is denoted by $x \notin A$. A set that has no element is also considered to be a set called the *empty set*, and it is represented by \emptyset or $\{\}$. A set consisting of at most a finite number of elements, including the empty set, is called a *finite set*, and a set that is not finite is said to be *infinite*. The first two examples above are both finite sets. The last example (the set of all rational numbers) is an infinite set.

Some sets which frequently appear in mathematics are marked with special symbols. An integer greater than or equal to 1 is called a *natural number*, and the set of all natural numbers is expressed by \mathbb{N} . A common but somewhat ambiguous expression⁶

$$\mathbb{N} = \{1, 2, 3, \dots, n, \dots\}.$$

is used. Whereas, \mathbb{R} and \mathbb{C} will stand for the set of all real numbers and the set of all complex numbers, respectively. It is not easy to give a rigorous definition of \mathbb{R} .⁷ As far as \mathbb{R} is defined, \mathbb{C} can be expressed by intentional definition as

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

where i is the imaginary unit. All \mathbb{N} , \mathbb{R} and \mathbb{C} are infinite sets.

We can visualize \mathbb{R} and \mathbb{C} in Fig. 1.2 as the *real line* and the *complex plane*, respectively. The parts $\{z \mid |z| = 1\}$ and $\{z \mid |z| \leq 1\}$ of \mathbb{C} are called the *unit circle* and the *unit disk* of the complex plane, respectively.

Let A and B be arbitrary sets. When $x \in A \Rightarrow x \in B$,⁸ that is, all the elements of A are also elements of B , A is said to be a *subset* of B . We also say that B *contains* A and this situation is expressed as $A \subseteq B$. For example, the unit circle and disk above are subsets of \mathbb{C} . Always, $\emptyset \subseteq A$ holds for any set A . When $x \in A \Leftrightarrow x \in B$ holds, that is, both $A \subseteq B$ and $B \subseteq A$ hold, we write $A = B$ and say that A and B are equal. If $A \subseteq B$ and $A \neq B$, then A is called a *proper subset* of B and we denote this situation by $A \subsetneq B$.

The set $\{X \mid X \subseteq A\}$ of all subsets of A is called the *power set* of A and is denoted by 2^A . For example,

$$2^\emptyset = \{\emptyset\}, \quad 2^{\{1\}} = \{\emptyset, \{1\}\}, \quad 2^{\{1, 2\}} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}.$$

⁶ Undefined symbols “...” and variable n are used.

⁷ The reader should consult a book on real number theory.

⁸ The variable x without any mention before is called a *free variable* and includes the meaning of “any x ”.

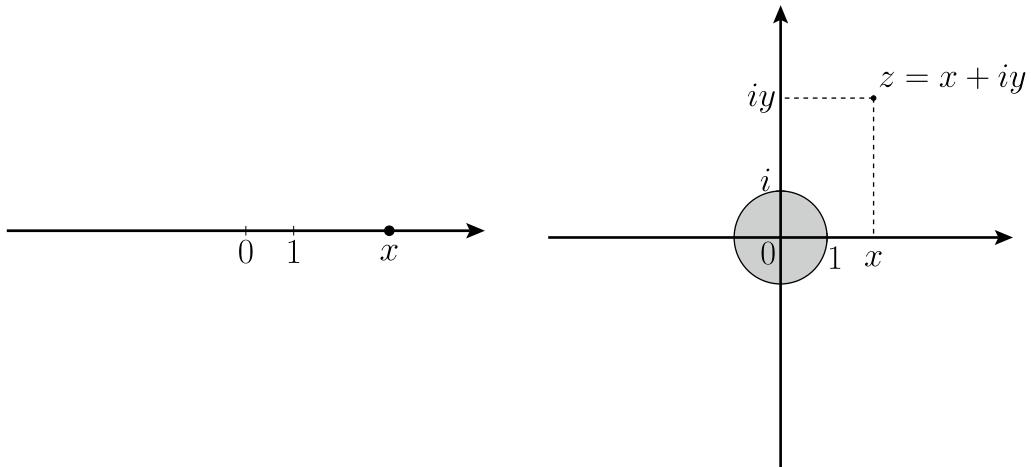


Fig. 1.2 The real line (left) and the complex plane (right) with the unit circle and disk

Exercise 1.7 Write down the extensional description of the power set $2^{\{1, 2, 3\}}$. Also, prove that the number of elements of the power set of a set with n elements is 2^n .

We define the following *set operations* for sets A and B :

$$\begin{aligned} A \cup B &\stackrel{\text{def}}{=} \{x \mid (x \in A) \vee (x \in B)\}, \\ A \cap B &\stackrel{\text{def}}{=} \{x \mid (x \in A) \wedge (x \in B)\}, \\ A \setminus B &\stackrel{\text{def}}{=} \{x \mid (x \in A) \wedge \neg(x \in B)\}. \end{aligned}$$

We call them the *union*, *intersection*, and *difference set*, respectively. These set operations are closely related to the operations of propositional logic. In fact,

$$\begin{aligned} x \in A \cup B &\Leftrightarrow (x \in A) \vee (x \in B), \\ x \in A \cap B &\Leftrightarrow (x \in A) \wedge (x \in B), \\ x \in A \setminus B &\Leftrightarrow (x \in A) \wedge \neg(x \in B). \end{aligned}$$

There are many elementary properties of set operations. For example, the equality

$$A \cup B = B \cup A$$

is shown as

$$x \in A \cup B \Leftrightarrow (x \in A) \vee (x \in B) \Leftrightarrow (x \in B) \vee (x \in A) \Leftrightarrow x \in B \cup A,$$

and the equality

$$(A \cup B) \cup C = A \cup (B \cup C)$$

follows from the associative law of \vee in propositional logic:

$$\begin{aligned} x \in (A \cup B) \cup C &\Leftrightarrow ((x \in A) \vee (x \in B)) \vee (x \in C) \\ &\Leftrightarrow (x \in A) \vee ((x \in B) \vee (x \in C)) \Leftrightarrow x \in A \cup (B \cup C). \end{aligned}$$

Exercise 1.8 Prove the equalities $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ and $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. (Hint: use the distributive laws in Exercise 1.2).

When we only consider elements and subsets of some (large) set U in our mathematical discussions, U is called the *universal set*. For instance, \mathbb{N} can be the universal set when we discuss something about prime numbers, and so can be \mathbb{R} when we discuss rational and irrational numbers. If U is the universal set in some context and $P(x)$ is a propositional function, the set of $x \in U$ such that $P(x)$ is true is expressed by

$$\{x \in U \mid P(x)\}.$$

For a subset A of U we denote $U \setminus A$ by A^C and call it the *complement* of A . With a free variable x ranging over U , we have

$$x \in A^C \Leftrightarrow \neg(x \in A).$$

In Python, we can use the extensional definition of a set as in mathematics. Let us use it in interactive mode.

```
In [1]: A = {2, 3, 5, 7}; A
Out[1]: {2, 3, 5, 7}

In [2]: B = set([6, 9, 3]); B
Out[2]: {9, 3, 6}

In [3]: C = {3, 6, 9, 6, 3}; C
Out[3]: {9, 3, 6}

In [4]: set()
Out[4]: set()
```

The set $\{3, 6, 9\}$ is also expressed as `set([3, 6, 9])`. The duplicated elements in set C are removed. The empty set is expressed by `set()`. Note that `{}` has a different meaning in Python.⁹ In this book we use a semicolon ; only for referring to a variable immediately after its definition as above in interactive mode. Sets are considered to be the same even if the orders of their elements are different.¹⁰

```
In [5]: C == {3, 6, 9}
Out[5]: True
```

⁹For historical reasons, `{}` means the dictionary with empty entries. Dictionaries will be explained in Sect. 1.6.

¹⁰Data are assigned to computer memory through hash functions, and so the order of assignment of elements in a set depends on them.

Use `in` for the symbol \in of belonging.

```
In [6]: 2 in A
```

```
Out[6]: True
```

```
In [7]: 2 in B
```

```
Out[7]: False
```

Use `|` and `&` for the union \cup and the intersection \cap , respectively.

```
In [8]: A | B
```

```
Out[8]: {2, 3, 5, 6, 7, 9}
```

```
In [9]: A & B
```

```
Out[9]: {3}
```

The set inclusion between sets X and Y is expressed as `X.issubset(Y)`.

```
In [10]: (A & B).issubset(A)
```

```
Out[10]: True
```

```
In [11]: A.issubset(A | B)
```

```
Out[11]: True
```

Here, `issubset` is said to be a *method*¹¹ of `set` class. Both union and intersection have also method expressions of `set` class.

```
In [12]: A | B == A.union(B)
```

```
Out[12]: True
```

```
In [13]: A & B == A.intersection(B)
```

```
Out[13]: True
```

A set is said to be an object of `set` class in Python. Objects of `set` class are limited to those with a finite number of elements. In mathematics, we can think of the set of all prime numbers, but it is impossible to express it as a `set` class object. As long as it is a finite set, Python has an expression of a `set` class object similar to the intensional definition of a set in mathematics.

```
In [14]: {x for x in range(2, 10) if all([x%n for n in range(2, x)])}
```

```
Out[14]: {2, 3, 5, 7}
```

¹¹ We will explain what methods are in Sect. 1.6.

This means the set of all integers x with $2 \leq x < 10$ not divisible by any integer n with $2 \leq n < x$ (that is, prime numbers). In Python, such an expression is called the *inline for sentence*.

There are subsets of \mathbb{R} called *intervals*:

$$\begin{aligned} [a, b] &\stackrel{\text{def}}{=} \{x \mid a \leq x \leq b\}, & (a, b) &\stackrel{\text{def}}{=} \{x \mid a < x < b\}, \\ (a, b] &\stackrel{\text{def}}{=} \{x \mid a < x \leq b\}, & [a, b) &\stackrel{\text{def}}{=} \{x \mid a \leq x < b\}, \end{aligned}$$

for $a, b \in \mathbb{R}$ with $a < b$. They are called a *closed*, *open*, *open-closed*, and *closed-open* interval, respectively. All of them are called *finite intervals*. *Infinite intervals* are

$$\begin{aligned} (-\infty, a) &\stackrel{\text{def}}{=} \{x \mid x < a\}, & [a, \infty) &\stackrel{\text{def}}{=} \{x \mid a \leq x\}, \\ (-\infty, a] &\stackrel{\text{def}}{=} \{x \mid x \leq a\}, & (a, \infty) &\stackrel{\text{def}}{=} \{x \mid a < x\}, \\ (-\infty, \infty) &\stackrel{\text{def}}{=} \mathbb{R} \end{aligned}$$

for $a \in \mathbb{R}$, here $-\infty$ and ∞ are symbols not representing any number.

1.4 Ordered Pairs and Tuples

A pair (x, y) coupling two objects x and y is called an *ordered pair*. An expression (x_1, x_2, \dots, x_n) with n items x_1, x_2, \dots, x_n aligned is called an *n -fold tuple*. An ordered pair is a two-fold tuple. Each of items x_1, x_2, \dots, x_n in an n -fold tuple (x_1, x_2, \dots, x_n) is called a *component*. The order and duplication of components are meaningful, unlike the extensional definition of a set. For example, all of $\{1, 2\}$, $\{2, 1\}$ and $\{1, 2, 2, 1\}$ represent the same set, but we consider $(1, 2)$, $(2, 1)$ and $(1, 2, 2, 1)$ are different objects as two ordered pairs and one four-fold tuple.

Now let X and Y be nonempty sets. The set of all ordered pairs (x, y) for $x \in X$ and $y \in Y$ is called the *direct product* or the *Cartesian product* of X and Y , and written as $X \times Y$. Similarly, for nonempty sets X_1, X_2, \dots, X_{n-1} and X_n we denote by $X_1 \times X_2 \times \dots \times X_n$ the set of all n -fold tuples (x_1, x_2, \dots, x_n) with $x_1 \in X_1, x_2 \in X_2, \dots, x_n \in X_n$. When $X_1 = X_2 = \dots = X_n = X$, X^n denotes the direct product $X_1 \times X_2 \times \dots \times X_n$. If X is a finite set and consisting of k elements, X^n has k^n elements.

The direct product $\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ is the xy -coordinate plane, which is the set of all points (x, y) with $x, y \in \mathbb{R}$, and $\mathbb{R} \times \mathbb{R} \times \mathbb{R} = \mathbb{R}^3$ is the xyz -coordinate space. Identifying (x) with x itself, \mathbb{R}^1 is the same as \mathbb{R} .

In Python, an n -fold tuple is a `tuple` class object which is expressed by components lined up in a row and enclosed in parentheses. In mathematics we distinguish between the terms “element” and “component”, but in Python, we are used to calling components of tuples their *elements* as in sets.

Let $x = (1, 2)$, $y = (2, 1)$, and $z = (1, 2, 1)$. We check whether $x = y$, $x = z$, and $y = z$ hold or not by Python.

```
In [1]: x = (1, 2); x
```

```
Out[1]: (1, 2)
```

```
In [2]: y = (2, 1); y
```

```
Out[2]: (2, 1)

In [3]: z = (1, 2, 1); z

Out[3]: (1, 2, 1)

In [4]: x == y, x == z, y == z

Out[4]: (False, False, False)
```

In [4] refers to the Boolean values of `x == y`, `x == z`, and `y == z`. In Out [4], a tuple of the Boolean values is returned.

In Python, we say that an object has some *data type*, which means the object is an instance of some *class*. A name of a data type plays the role of a function which converts a value of a given object equipped with some data type into a value of another specified data type. This mechanism is called the *type casting*. Let us cast tuples `x`, `y`, and `z` into `set` class objects.

```
In [5]: set(x) == set(y), set(x) == set(z), set(y) == set(z)

Out[5]: (True, True, True)

In [6]: set(x), set(y), set(z)

Out[6]: ({1, 2}, {1, 2}, {1, 2})
```

All of `set(x)`, `set(y)`, and `set(z)` express the same set $\{1, 2\}$. Sometimes, the casting function `set` is used to delete duplication of elements in a tuple `t` by writing `tuple(set(t))`.

We can access the elements of a tuple by indices (subscripts). Indices are integers, and the first (left-most) element has index 0, the second 1, the third 2, and so on.

```
In [1]: a = (2, 3, 5, 7); a

Out[1]: (2, 3, 5, 7)

In [2]: a[0], a[1], a[2], a[3]

Out[2]: (2, 3, 5, 7)
```

Negative integers $-1, -2, -3, \dots$ can be allowed as indices which start from the last (rightmost) element backward to the previous one.

```
In [3]: a[-1], a[-2], a[-3], a[-4]

Out[3]: (7, 5, 3, 2)
```

If a tuple has n elements, the maximum index is $n - 1$ and the minimum index is $-n$. Integers out of this range cause an indexing error.

We sometimes need a tuple with only one element. The expression `(1)` means just the same as integer 1 as stated above, and we must express such a tuple as `(1,)` in Python.

```
In [1]: x = (1,); x
Out[1]: (1,)
In [2]: x[0]
Out[2]: 1
```

1.5 Mappings and Functions

Let X and Y be nonempty sets. A *mapping* from X to Y assigns to each element of X an element of Y . A mapping is also an object in mathematics, and if necessary, we can name it as f, g, h , etc. We denote a mapping f from X to Y by $f : X \rightarrow Y$. The unique element $y \in Y$ that f assigns to $x \in X$ is denoted by $f(x)$.

The set of all mappings from X to Y is denoted by Y^X , that is,

$$Y^X \stackrel{\text{def}}{=} \{f \mid f : X \rightarrow Y\}.$$

This notation¹² may look awkward but it comes from the fact that the number of elements of Y^X just equals y^x for X with x elements and Y with y elements.

Exercise 1.9 Prove the above fact on the size of the power set Y^X .

In mathematics, the word *function* is a synonym for mapping, but it is usually used in cases where the target set Y is \mathbb{R} or \mathbb{C} . In Python, a mapping in mathematics is always called a function (an object of the function class). There are some definite functions in mathematics which cannot be defined by Python. One example is the function that assigns 1 to rational numbers and 0 to irrational numbers. On the other hand, Python's *function* contains a *procedure* (*subroutine*, or *subprogram*), which is a block of tasks and runs only when it is called. For example, the *print* function, if we call it in the format *print(x)*, prints out the value of argument *x* readable enough for us in the shell window. This function is a type of built-in function which can be used everywhere without importing its name. We do not need to know how it is implemented, and we only need to know the rule determining the relation between its argument (input) and its result (output). Such a rule is called the *specification* of the function. The casting functions *tuple* and *set* are also examples of built-in functions.

Let us compare the function definitions in mathematics and in Python, with some examples.

1. Definition by a simple expression: $f(x) \stackrel{\text{def}}{=} x^2 - 1$.

```
def f(x):
    return x**2 - 1
```

2. Definition by divided cases: $g(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{otherwise.} \end{cases}$

```
def g(x):
    if x < 0:
        return 0
```

¹² This is a comprehensive notation in the sense that the power set 2^A is considered to be the set of all mappings from A to $\{0, 1\}$, and the direct product X^n is considered to be the set of all mappings from $\{1, 2, \dots, n\}$ to X .

```
    else:
        return 1
```

This is equivalent to the following description with the *ternary operator*.

```
def g(x):
    return 0 if x < 0 else 1
```

3. An anonymous function: $x \mapsto x^2 - 1$.

```
lambda x: x**2 - 1
```

Python's anonymous function is called the *lambda expression* whose idea and name come from λ -calculus in the theory of mathematical logic. Situations using a lambda expression will occur when the act of naming for reuse of the function is not necessary. In mathematics the function $f : \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x) \stackrel{\text{def}}{=} x^2 - 1$, for example, is sometimes written as $f : x \mapsto x^2 - 1$, which is more descriptive to express the relationship between x and $f(x)$. Python allows lambda expressions for such use. The function f defined by `def f(x): return x**2 - 1` can be written as `f = lambda x: x**2 - 1`.

The factorial function $f(n) = n!$ is defined using mathematical induction by

$$f(n) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } n = 0, \\ n \cdot f(n-1) & \text{otherwise.} \end{cases}$$

In Python, such a function is called a *recursive* function. It calls itself during its execution and is defined as follows:

```
def f(x):
    if n == 0:
        return 1
    else:
        return n * f(n-1)
```

For a mapping (function) $f : X \rightarrow Y$, the subset

$$\{(x, y) \mid x \in X \text{ and } y = f(x)\}$$

of the direct product $X \times Y$ is called the *graph* of f . For two mappings f and g , if their graphs are equal as sets, they are the same mapping, and written $f = g$.

Consider a mapping $f : X \rightarrow Y$, then X is called the *domain* of f . For a subset A of X , the subset $f(A)$ of Y defined by

$$f(A) \stackrel{\text{def}}{=} \{f(x) \mid x \in A\} = \{y \mid \text{there exists } x \in A \text{ such that } y = f(x)\}$$

is called the *image* of A with respect to f . On the other hand, for a subset B of Y , the subset $f^{-1}(B)$ of X defined by

$$f^{-1}(B) \stackrel{\text{def}}{=} \{x \mid f(x) \in B\}$$

is called the *inverse image* of B with respect to f .

The image $f(X)$ of the domain X is called the *range* of f and is denoted by $\text{range}(f)$. If $\text{range}(f) = Y$, then f is said to be *surjective* or *onto*. If $f(x) \neq f(x')$ whenever $x \neq x'$ for $x, x' \in X$, then f is said to be *injective* or *one-to-one*. An injective and surjective (one-to-one onto) mapping is called a

bijection mapping or simply a *bijection*. If f is a bijection, for each $y \in Y$, there exists unique $x \in X$ such that $f(x) = y$, so we have the mapping $h : Y \rightarrow X$ which sends $y \in Y$ with $y = f(x)$ to $x \in X$. This h is called the *inverse mapping* of f and denoted by f^{-1} . We have $f^{-1}(f(x)) = x$.

Let $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ be mappings. The mapping $h : X \rightarrow Z$ defined by $h : x \mapsto g(f(x))$ is called the *composition* of f and g . This h is denoted by $g \circ f$. We have $(g \circ f)(x) = g(f(x))$.

Suppose both $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ are bijections. Putting $z = (g \circ f)(x)$, we have

$$(f^{-1} \circ g^{-1})(z) = f^{-1}\left(g^{-1}(g(f(x)))\right) = f^{-1}(f(x)) = x,$$

Hence, $f^{-1} \circ g^{-1}$ is the inverse mapping of $g \circ f$, that is,

$$(g \circ f)^{-1} = f^{-1} \circ g^{-1}.$$

The mapping $\text{id}_X : X \rightarrow X$ that sends $x \in X$ to x is called the *identity mapping* on X . It is bijective and has the inverse, which is id_X itself. If $f : X \rightarrow Y$ is bijective, then $f^{-1} \circ f = \text{id}_X$, $f \circ f^{-1} = \text{id}_Y$ and $(f^{-1})^{-1} = f$ hold.

1.6 Classes and Objects in Python

Data that Python can handle are called *objects*. A *class* is an abstract concept that puts together all the objects with the same data structure. For example, the literal `3.14` is an object of `float` class representing a real number. The actual object is stored in computer memory as binary data in the form of floating-point data. An `int` class object has a different form in memory which represents an integer.

In Python, there are several types of objects which possess elements. Objects possessing elements are divided into two kinds, those in which elements are arranged in order such as tuples, and those in which the order of elements does not matter such as sets. *Strings* (`str` class objects) and *lists* (`list` class objects) are other examples of the former type. The elements of these objects are indexed by integers as so are the elements of a tuple.

A string and its use:

```
In [1]: A = 'Hello Python!'; A
Out[1]: 'Hello Python!'
In [2]: print(A)
 Hello Python!
In [3]: print(A[0], A[1], A[2], A[3])
 H e l l o
In [4]: print(A[-1], A[-2], A[-3], A[-4])
 ! n o h
```

A literal that represents a string encloses the entire string with quotation marks ' or double quotation marks ". The elements of a string can be referred to by subscripting them in the same way as a tuple.

A list and its use:

```
In [1]: B = ['Earth', 'Mars', 'Jupiter']; B
```

```
Out[1]: ['Earth', 'Mars', 'Jupiter']
```

```
In [2]: print(B[0], B[1], B[2])
```

 Earth Mars Jupiter

```
In [3]: print(B[-1], B[-2], B[-3])
```

 Jupiter Mars Earth

```
In [4]: B.append('Saturn'); B
```

```
Out[4]: ['Earth', 'Mars', 'Jupiter', 'Saturn']
```

A list encloses its elements in [and]. We can refer to an element of it by subscripts. Method `append` of `list` class adds a new element at the end of an object of `list` class, and the contents of the list are partially changed.

Dictionary (`dict` class object) consists of pairs of a key and a corresponding value. In the literal of a dictionary, each pair consists of a key and its value sandwiching a colon, and all pairs are separated by commas and enclosed in braces.

A dictionary and its use:

```
In [1]: C = {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th'}; C
```

```
Out[1]: {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th'}
```

```
In [2]: C['Earth']
```

```
Out[2]: '3rd'
```

```
In [3]: C['Saturn'] = '6th'; C
```

```
Out[3]: {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th', 'Saturn': '6th'}
```

Lists and tuples use an integer as an index to refer to the value of their element, while a dictionary uses a key as an index to refer to the value paired with that key. By adding a new key and its value, the contents of a dictionary can be partially changed.

An object possessing elements can be checked by using `in` as to whether given data are contained in it. Also, using `in`, the elements of object can be values for a loop counter of a `for` statement.

Program: planet.py

```
In [1]: 1 | C = {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th'}
2 | for x in C:
3 |     print(f'{x} is the {C[x]} planet in the solar system.')
4 | print()
5 | for x in sorted(C):
6 |     print(f'{x} is the {C[x]} planet in the solar system.')
```

Line 2: Loop counter `x` walks around the keys of dictionary `C`.

Line 3: `f'{x}` is the `{C[x]}` planet in the solar system.' is called an *f-string*, and it is the string obtained by substituting the literal of the value `x` and `C[x]`, respectively, into the position enclosed by `{ }`.

Line 4: The `print` function without no arguments sends only a newline code.

Line 5: The order of keys in a dictionary is decided by the convenience of Python (see footnote 10). If one wants to sort with a certain intention, use the `sorted` function. In this example, the lexicographical order is expected. We can choose another order by changing the argument of `sorted`.

```
Earth is the 3rd planet in the solar system.
Mars is the 4th planet in the solar system.
Jupiter is the 5th planet in the solar system.

Earth is the 3rd planet in the solar system.
Jupiter is the 5th planet in the solar system.
Mars is the 4th planet in the solar system.
```

Note that the f-string is a relatively recent notation added since Python 3.6. It is better also to know other equivalent notations used before.

```
In [1]: x, y, z = 1, 2, 3; x, y, z
```

```
Out[1]: (1, 2, 3)
```

```
In [2]: f'{x}{y}{z}'
```

```
Out[2]: '123'
```

```
In [3]: '{x}{y}{z}'.format(x, y, z)
```

```
Out[3]: '123'
```

```
In [4]: '%s%s%s' % (x, y, z)
```

```
Out[4]: '123'
```

In [2] uses the f-string, In [3] uses the `format` method of `string` class, and In [4] uses the `format` operator `%`.

The `list` class has the `sort` method which is similar to but not the same as the `sorted` function.

```
In [5]: A = [z, y, x]; A
```

```
Out[5]: [3, 2, 1]
```

```
In [6]: sorted(A)
```

```
Out[6]: [1, 2, 3]
```

```
In [7]: A
```

```
Out[7]: [3, 2, 1]
```

```
In [8]: A.sort()  
A
```

```
Out[8]: [1, 2, 3]
```

A function specialized for manipulating an object of a certain class is called a *method* of that class. The `sort` method is said to be *destructive*, for it rewrites the object to which the method subjects. The `append` method of `list` class is also destructive, because it modifies the contents of the list. Also, `C['Saturn'] = '6th'` rewrites the contents of dictionary `C`. Lists, sets, and dictionaries are called *mutable* objects in the sense that we can change some or all of them after they are created. Tuples and strings are *immutable* objects and their contents cannot be changed after the object is created. Objects of the classes `int`, `float`, and `complex` are also immutable. Immutable objects can be keys of a dictionary.

1.7 Lists, Arrays and Matrices

A sequence of several kinds of items is called a list in general. The number of the items is called the *length* or *size* of the list.

In mathematics, the use of the terms tuples and lists may be roughly described as follows. In the context where a tuple is used, its length is constant and each of their components is an element of a fixed particular set, that is, a tuple is an element of some Cartesian product set. A list, on the other hand, with variable length is used to simply enumerate objects and is not aware that it is an element of any Cartesian product set.

The difference between tuples and lists in Python is simple. Tuples are immutable and lists are mutable. As we have seen, tuples enclose their elements with (and), whereas lists enclose them with [and].

```
In [1]: A = (1, 2, 3); A
```

```
Out[1]: (1, 2, 3)
```

```
In [2]: B = [1, 2, 3]; B
```

```
Out[2]: [1, 2, 3]
```

```
In [3]: A[0], B[0]
```

```
Out[3]: (1, 1)
```

```
In [4]: B[0] = 0; B
```

```
Out[4]: [0, 2, 3]
```

```
In [5]: A[0] = 0
```

```
Out[5]: Traceback (most recent call last):  
      File "<pyshell#13>", line 1, in <module>  
        A[0] = 0  
TypeError: 'tuple' object does not support item assignment
```

To refer to the i -th element of a tuple or list, use `A[i - 1]` or `B[i - 1]`, because the subscript starts from 0 not from 1. An attempt to change the element of `tuple` in `In[5]` raises an error.

The `tuple` class object does not have any method such as `append` or `sort`. As we consider the empty set with no element, both tuples and lists may have no element. They are of length 0 and are expressed by `()` and `[]` respectively.

Let $k_1, k_2, \dots, k_n \in \mathbb{N}$ and

$$I = \{1, 2, \dots, k_1\} \times \{1, 2, \dots, k_2\} \times \cdots \times \{1, 2, \dots, k_n\}.$$

For any nonempty set X , we call a mapping $A : I \rightarrow X$ an *n-dimensional array* on X and denote it by

$$[a_{i_1 i_2 \dots i_n}]_{i_1=1}^{k_1}, \underset{i_2=1}{i_2}, \dots, \underset{i_n=1}{i_n},$$

where for $(i_1, i_2, \dots, i_n) \in I$, $a_{i_1 i_2 \dots i_n} = A((i_1, i_2, \dots, i_n))$ is an element of X called the $i_1 i_2 \dots i_n$ -th *component* or *element* of A , and $i_1 i_2 \dots i_n$ is its *subscript* or *index*. A one-dimensional array $[a_i]_{i=1}^k$ is considered to be a k -fold tuple (a_1, a_2, \dots, a_k) . A two-dimensional array $[a_{ij}]_{i=1}^k, \underset{j=1}{j}$ written as

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1l} \\ a_{21} & a_{22} & \cdots & a_{2l} \\ \vdots & \vdots & & \vdots \\ a_{k1} & a_{k2} & & a_{kl} \end{bmatrix}$$

is called a *matrix*¹³ of the *shape* (k, l) or a $k \times l$ *matrix*. The horizontal arrangements of a matrix are called *rows*, and they are counted from the top as the first, second, third, ... and k -th row. The vertical arrangements of a matrix are called *columns*, and they are counted from the left as the first, second, ..., and l -th column.

Many programming languages have a data structure called an *array*, with which we can represent a matrix. There is no special class called an array in the default Python. This is because we can usually represent a matrix by nesting lists.

Program: matrix.py

```
In [1]: 1 | A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2 | for i in range(3):
3 |     for j in range(3):
4 |         print(f'A[{i}][{j}]={A[i][j]}', end=', ')
5 |     print()
```

Line 1: Object `A` is a list with three elements all of which are lists.

Line 4: The `print` function normally sends a newline code at the `end`, but if some string is assigned in keyword argument `end`, the newline code will be replaced by the string.



```
A[0][0]==1, A[0][1]==2, A[0][2]==3,
A[1][0]==4, A[1][1]==5, A[1][2]==6,
A[2][0]==7, A[2][1]==8, A[2][2]==9,
```

A typical programming language implementation requires that array elements have the same data type. However, in Python, arrays are substituted by nested lists, so array elements can have different data types. For example, we can use an array where the first column is a string that is the name of

¹³ Since a_{11} can be read as the 11th component of a one-dimensional array, a_{ij} should be originally written as $a_{i,j}$, but unless misunderstood, we write like this according to convention.

a student, the second column is an integer that is the student ID number, and so on. In spite of this flexibility, using a list for a matrix, the calculation slows down as the size of the matrix increases, with the additional reason that Python's implementation is an interpreter.¹⁴

On the other hand, NumPy allows us to perform fast vector and matrix calculations by using the *ndarray* class (we will call it simply *array*¹⁵ in subsequent descriptions). This class provides convenient functions and methods for those calculations, and it makes our code easier to write and easier to read.

1.8 Preparation of Image Data

1.8.1 Binarization of Image Data with PIL and NumPy

Prepare a grayscale image file. If necessary, refer to Appendix A.6 and create a grayscale image file from your favorite photos or images. The image is expressed by a matrix whose element is the graylevel of each pixel. Here, it is assumed that the prepared grayscale image file is *mypict1.jpg* of Appendix A.4.

Program: *graph1.py*

```
In [1]: 1 import PIL.Image as Img
2 from numpy import array, zeros
3 import matplotlib.pyplot as plt
4
5 im = Img.open('mypict1.jpg')
6 A = array(im)
7 m, n = A.shape
8 avr = A.sum() / m / n
9 print((m, n), avr)
10
11 L = zeros(256)
12 for i in range(m):
13     for j in range(n):
14         L[A[i, j]] += 1
15
16 plt.plot(range(256), L)
17 plt.plot([avr, avr], [0, L.max()])
18 plt.show()
```

Line 1: Import module *PIL.Image* from *PIL* and name it *Img*.

Line 2: Import names *array* and *zeros* from *NumPy*.

Line 3: Import module *matplotlib.pyplot* and name it *plt*.

Line 5: Read the data from *mypict1.jpg*.

Line 6: Generate an array *A* from *im* which is an object of *PIL*'s image data structure. The function *array* defined in *NumPy* is a data structure which is more convenient than a matrix expressed by a nested list.

¹⁴ A system that interprets code and executes it on a computer is called a language processing system. An *interpreter* is one in which a language processor is resident in memory and interprets and executes code line by line. On the other hand, a *compiler* converts the whole code into machine language that can be executed without the hand of the language processing system. By thinking of the former as an interpreter and the latter as a translator, we can understand the advantages and disadvantages of each.

¹⁵ Since the name *array* is already used in the built-in library, *nd* is added to distinguish it. The prefix *nd* means *n*-dimensional. We will not use built-in library *array* in this book.

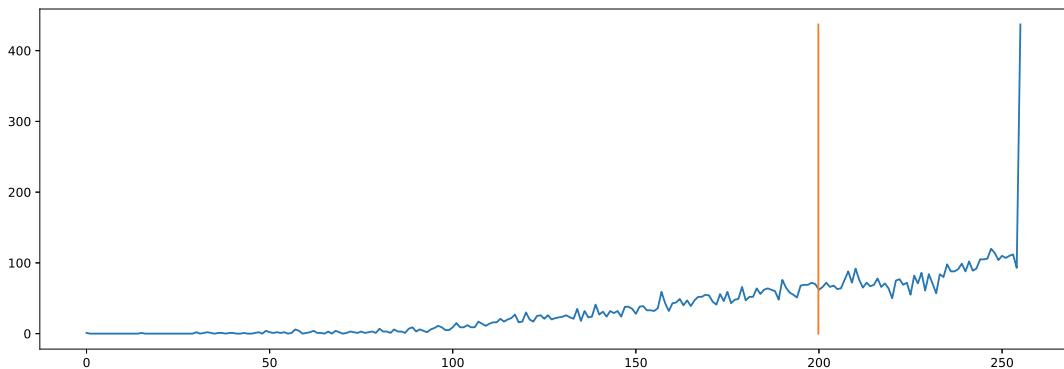


Fig. 1.3 Graph of graylevel distribution of pixels

Lines 5–9: A consists of $m \times n$ elements. The average `avr` of graylevels of the image is the sum `A.sum()` of all elements of A divided by the number of pixels $m \times n$.

Lines 11–14: Make up the array L whose k-th element is the number of all pixels with graylevel k for each integer k from 0 to 255.

Line 16: Draw the graph (Fig. 1.3) of L, which displays the distribution of graylevels of the image.

Line 17: Draw a vertical line at the value of `avr`.



```
(100, 88) 199.83977272727273
```

Let us look at the contents of A.

In [2]: A

```
Out[2]: array([[255, 253, 252, ..., 221, 197, 230],
   [245, 248, 250, ..., 149, 155, 212],
   [255, 255, 255, ..., 107, 168, 222],
   ...,
   [215, 222, 215, ..., 242, 255, 248],
   [243, 246, 243, ..., 244, 253, 252],
   [246, 235, 224, ..., 245, 249, 248]], dtype=uint8)
```

In [3]: A[0, 0]

```
Out[3]: 255
```

Because it is too large to display all the contents of A in the shell window, they are truncated. The expression `dtype=uint8` means that the data type of each element is an 8-bit unsigned integer (an integer from 0 to 255) which represents the graylevel; 0 is pure black and 255 is pure white. In an array, its elements must be uniformly of the same type. The elements of array A are referenced with `A[i, j]`. Like lists and tuples, subscripts start with 0.

Program: mypict1.py

```
In [1]: 1 import PIL.Image as Img
2 from numpy import array
3
4 A = array(Img.open('mypict1.jpg'))
5 B = A < 200
6 m, n = B.shape
7 h = max(m, n)
8 x0, y0 = m/h, n/h
9
10 def f(i, j):
11     return (y0*(-1 + 2*j/(n - 1)), x0*(1 - 2*i/(m - 1)))
12
13 P = [f(i, j) for i in range(m) for j in range(n) if B[i, j]]
14 with open('mypict1.txt', 'w') as fd:
15     fd.write(repr(P))
```

Line 5: The expression $A < 200$ returns the array of Boolean class objects with the same shape as A , whose element is True if the corresponding element of A is less than 200, and False otherwise. Thus, 200 is the threshold of the graylevel. This array is assigned to B . You should choose another value for this threshold with reference to the average and distribution of the graylevels of your image. If this value is less (resp. greater) than the average, the number of pixels with the value True will be larger (resp. smaller) than the one with False in array B .

In [2]: B

```
Out[2]: array([[False, False, False, ..., False,  True, False],
   [False, False, False, ...,  True,  True, False],
   [False, False, False, ...,  True,  True, False],
   ...,
   [False, False, False, ..., False, False, False],
   [False, False, False, ..., False, False, False],
   [False, False, False, ..., False, False, False]])
```

Lines 7,8: Make x_0 or y_0 equal to 1 for the longer side of the image.

Lines 10,11: In order to make the pixel at the i -th row and the j -th column of the image correspond to the point (x, y) on the xy -coordinate plane, define the function f by

$$f : (i, j) \mapsto \left(x_0 \left(-1 + \frac{2j}{n-1} \right), y_0 \left(1 - \frac{2i}{m-1} \right) \right).$$

This f sends the four corners of the image to the corresponding corners of a rectangle in the xy -coordinate plane, precisely

$$\begin{aligned}(0, 0) &\mapsto (-x_0, y_0), \\ (0, n-1) &\mapsto (-x_0, y_0), \\ (m-1, 0) &\mapsto (-x_0, -y_0), \\ (m-1, n-1) &\mapsto (-x_0, -y_0).\end{aligned}$$

If the image is square, i.e. $m = n$, then $x_0 = y_0 = 1$, but otherwise the longer side just fits between -1 and 1 .

Line 13: Scan B to create the list P of the coordinates (x, y) such that $B[i, j]$ equals True.

Lines 14,15: Convert P to a string with the `repr` function and write it in the text file `mypict1.txt`. '`w`' means to open the file for writing. There is also the `str` function to convert the object to a

string. It gives a human-readable string, but that string may not be evaluated as Python code. The string converted by the `repr` function can be restored by the `eval` function.

```
In [3]: A = array([1, 2, 3]); A
```

```
Out[3]: array([1, 2, 3])
```

```
In [4]: print(A)
```

```
Out[4]: [1 2 3]
```

```
In [5]: str(A)
```

```
Out[5]: '[1 2 3]'
```

```
In [6]: repr(A)
```

```
Out[6]: 'array([1, 2, 3])'
```

```
In [7]: eval('array([1, 2, 3])')
```

```
Out[7]: array([1, 2, 3])
```

The following program reads the text file `mypict1.txt` and displays the black-and-white image.

Program: mypict2.py

```
1 | import matplotlib.pyplot as plt
2 |
3 | with open('mypict1.txt', 'r') as fd:
4 |     P = eval(fd.read())
5 | x, y = zip(*P)
6 | plt.scatter(x, y, s=3)
7 | plt.axis('scaled'), plt.xlim(-1, 1), plt.ylim(-1, 1), plt.show()
```

Lines 3,4: Open file `mypict1.txt` for reading, and `eval` converts the string data of the file created by `mypict1.py` into a list of data.

Line 5: `P` is the list $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$ of tuples (x_i, y_j) . Divide this into the list $[x_1, x_2, \dots, x_n]$ of the x -coordinates and the list $[y_1, y_2, \dots, y_n]$ of y -coordinates, and express them by `x` and `y` respectively.

Line 6: Create a scatter plot with Matplotlib. Keyword argument `s=3` of method `scatter` determines the size of the point plotted at the coordinates (x_i, y_i) .

Line 7: Argument '`scaled`' in `plt.axis` matches the aspect ratio of the graph to the ratio of the x -coordinate range to the y -coordinate range. In this way, the image is displayed on the left of Fig. 1.4. The quality of a black-and-white picture depends on the threshold you choose in `mypict1.py`. The right of Fig. 1.4 is a magnified view of the part specified by `plt.xlim(-0.25, 0.25), plt.ylim(-0.25, 0.25)` in Line 7.

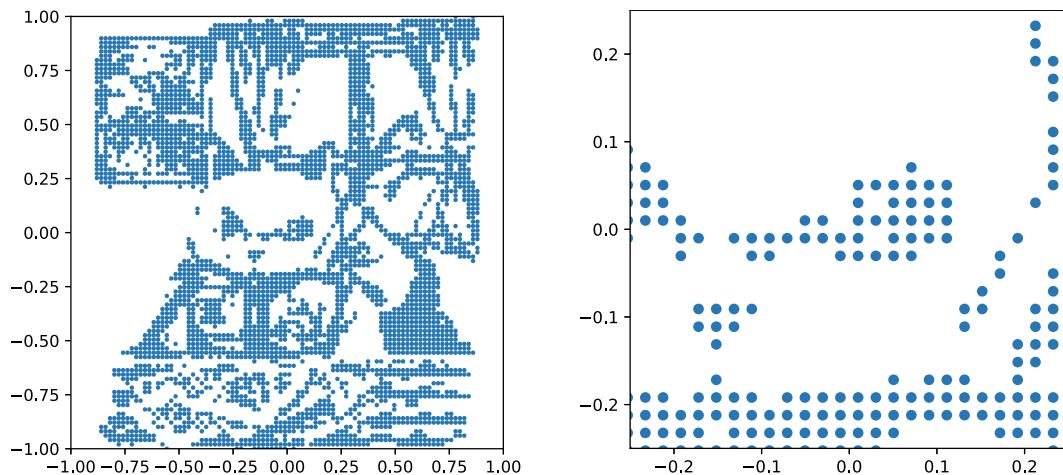


Fig. 1.4 Black-and-white image (left) and an enlarged view of part of it (right)

1.8.2 GUI for Creating Complex-Valued Data of Handwritten Characters

In Chaps. 6 and 10 we will deal with complex-valued data. In this chapter we make a GUI¹⁶ tool that creates line figures such as handwritten Chinese characters¹⁷ on a two-dimensional plane (Fig. 1.5), converts them into complex-valued data, and saves them into a file. Here, we use the built-in library *Tkinter*.

We think of a handwritten character as a sequence of points arranged in the order in which they are written on the square of $-1 \leq \operatorname{Re} z \leq 1$ and $-1 \leq \operatorname{Im} z \leq 1$ ($z \in \mathbb{C}$) in the complex plane. The sequence is expressed as a list of complex numbers and its representation by a string is stored in a text file.

Program: `tablet.py`

```
In [1]: 1  from tkinter import Tk, Button, Canvas
2
3  def point(x, y):
4      return C.create_oval(x - 1, y - 1, x + 1, y + 1, fill='black')
5
6  def PushButton(event):
7      x, y = event.x, event.y
8      Segs.append([(x, y), point(x, y)])
9
10 def DragMouse(event):
11     x, y = event.x, event.y
12     Segs[-1].append((x, y), point(x, y)))
13
14 def Erase():
15     if Segs != []:
16         seg = Segs.pop()
17         for p in seg:
18             C.delete(p[1])
19
20 def Save():
21     if Segs != []:
22         L = []
```

¹⁶ An abbreviation for Graphical User Interface, a program that can interact with a computer by operating the screen with the mouse.

¹⁷ Kanji “ku-kan” meaning space.

In [1]:

```

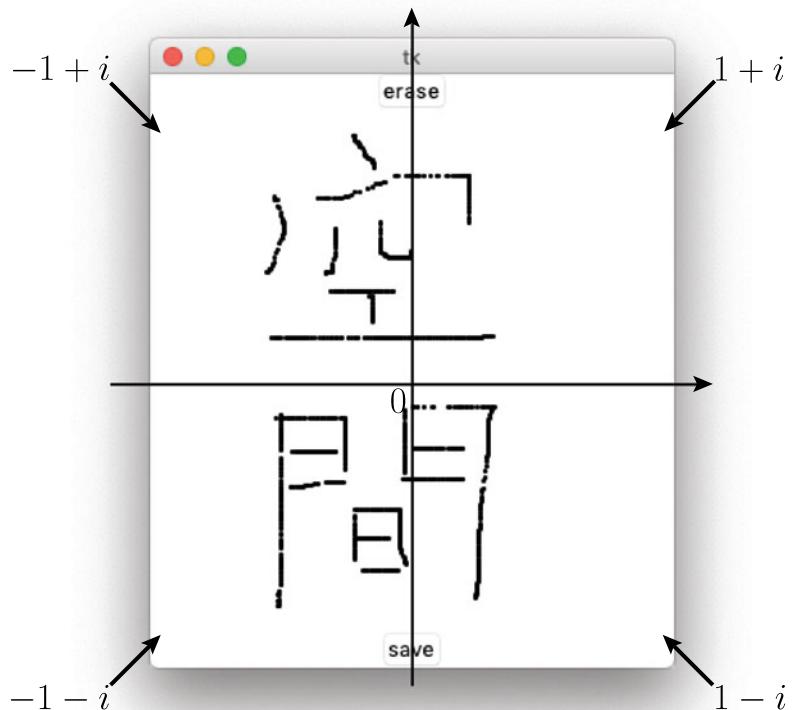
23     for seg in Segs:
24         for (x, y), _ in seg:
25             L.append((x - 160) / 160 + 1j * (160 - y) / 160)
26     with open(filename, 'w') as fd:
27         fd.write(repr(L))
28     print('saved!')
29
30 filename = 'tablet.txt'
31 Segs = []
32 tk = Tk()
33 Button(tk, text="erase", command=Erase).pack()
34 C = Canvas(tk, width=320, height=320)
35 C.pack()
36 C.bind("<Button-1>", PushButton)
37 C.bind("<B1-Motion>", DragMouse)
38 Button(tk, text="save", command=Save).pack()
39 tk.mainloop()

```

Line 1: Use the library `tkinter` for GUI tools. On a `Tk` class object of `tkinter`, place a `Button` class object that can be operated with mouse clicks and a `Canvas` class object on which we can draw line shapes with mouse drags.

Lines 3,4: Create an object that represents a point on the canvas. Since this function has only one line, it is possible to write the expression of the return value in Line 4 directly in the `PushButton` function in Line 6 and the `DragMouse` function in Line 10, but since one line will become long and hard to read, this is defined as a supplementary function. In programming, it is also important to write a code easier to read.

Fig. 1.5 GUI to create a line figure on the complex plane



Lines 6–28: Define functions that will be executed when mouse operations are detected. When the mouse button is clicked, a new line segment is drawn, and when the mouse is dragged, its trajectory becomes a line segment. Line segments are stored as two lists. One is a list of tuples that represent the position coordinates of points. The other is a list of point objects (small black circles) on the canvas. We use the former when saving the figure and the latter when erasing a line segment. We consider a list, which records changes in the pair of coordinates of the mouse pointer position and the point object generated on the canvas while dragging the mouse, as a line segment.

Line 30: `filename` is the name of file to save data in when the save button is clicked.

Line 31: `Segs` is a list of list objects expressing line segments.

Lines 32–38: Create a window on the screen, embed a canvas and buttons on it, and decide which function to call by operating the mouse. When the erase button is pressed, the line segment drawn last is removed. When the save button is clicked, the position coordinate data of `Segs` are converted into complex numbers and saved as a list in the file.

Line 39: Enter an infinite loop waiting for mouse activity. The program will not exit unless the created window is closed. It is called *event-driven programming* in the sense that we describe in a program what action to take when a certain event occurs and wait till an event occurs.

1.8.3 Data of Handwritten Letters with Grayscale

Character data may be treated as a grayscale plane figure instead of a line figure. Figure 1.6 shows some MNIST data of handwritten digits which are often used as learning and test data of machine learning systems.

Visit the home page <http://yann.lecun.com/exdb/mnist/> in which we will find the following tags:

```
train-images-idx3-ubyte.gz: training set images (9912422 bytes)
train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes).
```

Download these files, unzip¹⁸ and rename these files as follows respectively:

```
train-images.bin
train-labels.bin
test-images.bin
test-labels.bin.
```

Put them in the same folder where the program `mnist.py` below is located. The file `train-images.bin` (resp. `test-images.bin`) has a total of 60,000 (resp. 10,000) images. The label (the digit expressing the image) for each image can be found in `train-labels.bin` and `test-labels.bin`. Usually, `train-*` files are used as learning data for machine learning and `test-*` files are used for test data. All of these are binary files and their data formats are written at the end of the above home page.¹⁹

Let us look inside the file. We check the number of image data for each character and calculate the average of them.

¹⁸ Compressed files with `.gz` extension can be decompressed by double-clicking on macOS and Raspberry Pi. For Windows, we need to install free software such as 7-Zip or Lhaplus.

¹⁹ MNIST handwriting data is also available through scikit-learn and TensorFlow, Python libraries related to machine learning.

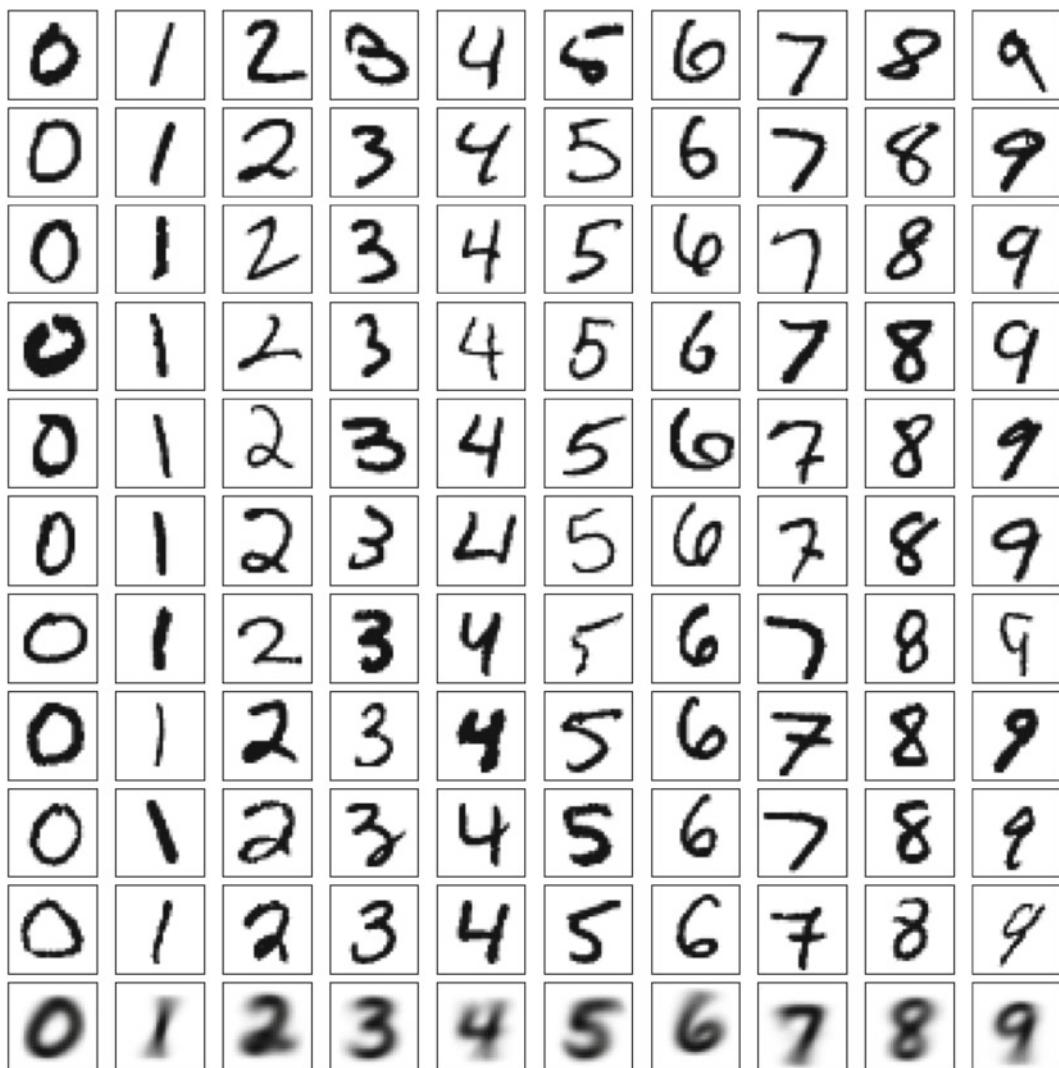


Fig. 1.6 Selected MNIST data

Program: mnist.py

```
In [1]: 1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 N = 10000
5 with open('test-images.bin', 'rb') as f1:
6     X = np.fromfile(f1, 'uint8', -1)[16:]
7 X = X.reshape((N, 28, 28))
8 with open('test-labels.bin', 'rb') as f2:
9     Y = np.fromfile(f2, 'uint8', -1)[8:]
10 D = {y: [] for y in set(Y)}
11 for x, y in zip(X, Y):
12     D[y].append(x)
13 print([len(D[y]) for y in sorted(D)])
```

```
In [1]: 15 fig, ax = plt.subplots(11, 10, figsize=(10, 10))
16 plt.subplots_adjust(wspace=0.1, hspace=0.1,
17                     left=0.01, right=0.99, bottom=0.01, top=0.99)
18 for y in D:
19     for k in range(10):
20         A = 255 - D[y][k]
21         ax[k][y].imshow(A, 'gray')
22         ax[k][y].tick_params(labelbottom=False, labelleft=False,
23                               color='white')
24         A = 255 - sum([x.astype('float') for x in D[y]]) / len(D[y])
25         ax[10][y].imshow(A, 'gray')
26         ax[10][y].tick_params(labelbottom=False, labelleft=False,
27                               color='white')
28 plt.show()
```

Line 4: N is the number of data.

Lines 5,6: Open the binary file of the character images and read it until the last as unsigned 8-bit integers. We can change the last argument -1 of the np.fromfile function to a positive integer, which means the number of the character images to read. The first 15 bytes consist of the header, so read only the data from the 16th byte and set it to array X.

Line 7: X is a one-dimensional array with $N \times 28 \times 28$ elements, so it is converted into a three-dimensional array. We consider it as N matrices of shape (28, 28) whose components are unsigned 8-bit integers revealing the values of the grayscale.

Lines 8,9: Read the binary file of labels. The first n -th byte counted from the 8th byte is an integer value from 0 to 9 which is the digit (label) for the n -th matrix in X. Put this list of labels in array Y.

Line 10: Create a dictionary whose key is the label k and value is a list of matrices each of which is a pattern handwritten as the digit k . The list is initially empty. set(Y) is the set of all elements in array Y and it should actually be {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Lines 11–13: zip(X, Y) is the list of tuples (x, y) that are created by sequentially extracting x from X and y from Y. Matrix x has label y, so add x to the list of items with label y in the dictionary. In this way, the dictionary D in which images are classified by labels is completed. Line 13 checks how many images there are in each label. The keys in the dictionary are sorted by the sorted function, and the number of the images is displayed in the form of a list in the order 0, 1, 2, ..., 9.

Lines 15–28: Display the first 10 patterns and the total average of images for each label. Line 15 prepares to embed multiple graphs in one diagram. Arrange the charts in 11 rows by 10 columns. The bottom row is for the total average of the images. Lines 16 and 17 adjust the margins of the graph. On line 20, the black-and-white inverted image is set as A. Line 21 draws each A in grayscale. Lines 22 and 23 set not to draw the scale of each axis of the graph. Lines 24–27 calculate the average by taking a simple average of the grayscales in pixels, and set it to A. Line 25 draws the average A. Line 28 actually displays Fig. 1.6.

 [980, 1135, 1032, 1010, 982, 892, 958, 1028, 974, 1009]

The test data have 980, 1135, 1032, 1010, 982, 892, 958, 1028, 974, and 1009 images for each label.



Linear Spaces and Linear Mappings

2

Based on the mathematical preparations in the previous chapter, we will learn about a linear space, the stage on which linear algebra is played, and a linear mapping which plays a leading part in linear algebra.

A linear space is just a set of vectors (with some properties). Plane vectors and space vectors, which you may be familiar with from highschool mathematics, are examples of vectors. Here we will learn that functions like polynomials and trigonometric functions are also vectors. Extending this consideration, sounds can be regarded as vectors. Utilizing Python, we can see or hear these various vectors.

In this chapter, we will be using abstract arguments (abstract linear spaces). You might think that an abstract argument is tedious and boring, but it is the most important method in science for treating seemingly different objects in a unified way and enables to gain a good understanding. Progress will seem much easier if you can become comfortable with this process of getting results through rigorous step-by-step arguments starting with a few axioms.

In the subsequent chapters, these abstract arguments will be realized as concrete computational methods using our own hands or computers.

2.1 Linear Spaces

Throughout this book, \mathbb{K} denotes either the set \mathbb{R} of all real numbers or the set \mathbb{C} of all complex numbers. \mathbb{K} is called a *scalar field* and its element is called a *scalar*.

A set V is called a *linear space* or a *vector space* over \mathbb{K} (an element of V is called a *vector*), if it satisfies the following five conditions.¹

- L1. For any vectors x and y , their *vector sum* $x + y \in V$ is determined (**closed under vector summation**).²
- L2. For any vector x and any scalar a , the *scalar multiple* $ax \in V$ is determined (**closed under scalar multiplication**).
- L3. A special vector $\mathbf{0}$ called the *zero vector* is determined in V (**existence of zero**).
- L4. For any vector x , a vector $-x \in V$ called its *inverse vector* is determined (**existence of inverse**).

¹ To avoid confusion between scalars and vectors, a bold font is used for vectors.

² $+$ is a binary operation, a mapping from $V \times V$ to V . It is important that $x + y$ is determined within V , and this situation is expressed as “closed”.

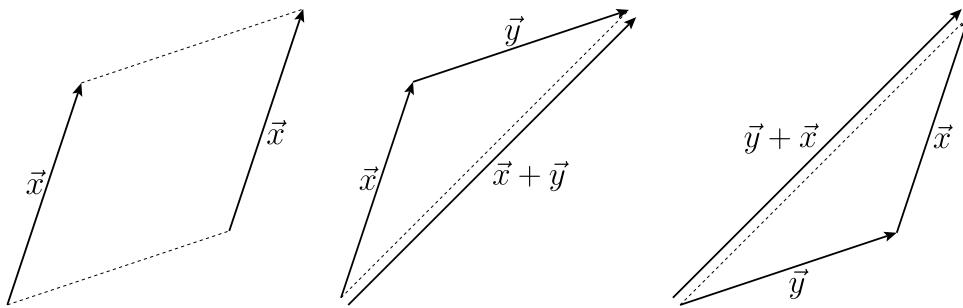


Fig. 2.1 Vector sum

L5. On the vector sum, scalar multiple, zero vector and inverse vector determined above, the following set of conditions called **axioms of linear space**³ is satisfied:

- | | |
|--------------------------|---------------------------------|
| (a) $x + y = y + x$ | (b) $(x + y) + z = x + (y + z)$ |
| (c) $x + \mathbf{0} = x$ | (d) $x + (-x) = \mathbf{0}$ |
| (e) $a(x + y) = ax + ay$ | (f) $(a + b)x = ax + bx$ |
| (g) $(ab)x = a(bx)$ | (h) $1x = x.$ |

Here, we distinguish scalar $0 \in \mathbb{K}$ from vector $\mathbf{0} \in V$, while $1 \in \mathbb{K}$ is a scalar.

A linear space over \mathbb{R} is sometimes called a *real linear space*, where a vector can be multiplied by only a real number. A linear space over \mathbb{C} is called a *complex linear space*, and the multiplication by a complex number is allowed. In particular, \mathbb{R} itself is a real linear space and \mathbb{C} is a complex linear space. \mathbb{C} is also a real linear space by considering a complex number as a vector, a real number as a scalar and the multiple of a complex number by a real number as a scalar multiple. Here, scalar 0 and vector $\mathbf{0}$ coincide. On the other hand, \mathbb{R} is not a complex linear space, even if we regard a complex number as a scalar and the multiple of a real number by a complex number as a scalar multiple. In fact, this scalar multiplication is not closed in \mathbb{R} .

A directed line segment on a plane (or in space) is identified with the one obtained from it by parallel translation. They have the same size and the same direction, and we express this quantity as \vec{x} . Figure 2.1 shows graphically how the vector sum $\vec{x} + \vec{y}$ is determined, and the reason why it is equal to $\vec{y} + \vec{x}$ (Axiom (a) of linear space).

We define the scalar multiple $a\vec{x}$ as follows. When $a > 0$, we multiply the size of \vec{x} by a without changing direction. When $a < 0$, we multiply the size by $|a|$ and change the direction backward. When $a = 0$, $a\vec{x}$ has size 0 and has no special direction and is the zero vector $\vec{0}$. In addition, $a\vec{x}$ for $a = -1$ is the inverse vector $-\vec{x}$. In this way, vector sum, scalar multiple, zero vector and inverse vector are determined. Considering only real numbers for scalar multiplication,⁴ the set of all these vectors becomes a real linear space.

³This set of properties is a starting point of the theory of linear algebra and so we call them axioms. To show that a certain space is a linear space, you need to determine vector sum, scalar multiple, zero vector and inverse vector and then to prove these axioms.

⁴It might be difficult to image the multiplication of a plane (or space) vector by a complex number. But later, in Chaps. 7 and 8, we will find merit in considering a real linear space as a complex linear space by extending the scalar multiplication.

Exercise 2.1 Explain the geometrical meaning of Axioms (b)–(h) in the linear space as shown in Fig. 2.1.

In Python, we can express vectors with tuples or lists, but it does not support vector operations. Using NumPy, we can manipulate vector sums and scalar multiples as common mathematical formulas. Together with the library Matplotlib let us check $\vec{x} + \vec{y} = \vec{y} + \vec{x}$ visually on a plane.

Program: vec2d.py

```
In [1]: 1 from numpy import array
2 import matplotlib.pyplot as plt
3
4 o, x, y = array([0, 0]), array([3, 2]), array([1, 2])
5 arrows = [(o, x + y, 'b'), (o, x, 'r'), (x, y, 'g'),
6           (o, y, 'g'), (y, x, 'r')]
7 for p, v, c in arrows:
8     plt.quiver(p[0], p[1], v[0], v[1],
9                 color=c, units='xy', scale=1)
10 plt.axis('scaled'), plt.xlim(0, 5), plt.ylim(0, 5), plt.show()
```

Line 4: The vectors $\vec{0} = (0, 0)$, $\vec{x} = (3, 2)$ and $\vec{y} = (1, 2)$ are expressed by arrays o , x and y , respectively. Using arrays for vectors, the vector sum $\vec{x} + \vec{y}$ is expressed as $x + y$ and the scalar multiple $2\vec{x}$ as $2 * x$.

Lines 5,6: The list of vectors which are drawn as arrows. A triple of starting point, vector and color is the parameter that determines the arrow.

Lines 7–9: Draw an arrow for each triple (p, v, c) in the list `arrows` using the function `quiver` in Matplotlib. The first four actual arguments are called *position arguments*. The latter three arguments `color`, `units`, `scale` are called *name arguments*, and the right-hand sides of `=` are actual augments. The order of position arguments is important, but the order of name arguments does not matter. You can change the order because the names convey the meanings. `units='xy'` implies how to give the size of an arrow and specifies it with x - and y -coordinates. `scale=1` is the scale of an arrow and the larger value reduces the size.

Line 10: Drawing the frames we get the left-hand plot in Fig. 2.2.

Program: vec3d.py

```
In [1]: 1 from vpython import vec, arrow, mag
2
3 o = vec(0, 0, 0)
4 x, y, z = vec(1, 0, 0), vec(0, 1, 0), vec(0, 0, 1)
5 arrows = [(o, x + y), (x, y + z), (o, x + y + z), (o, x), (y, x),
6           (z, x), (y + z, x), (o, y), (x, y), (z, y), (x + z, y),
7           (o, z), (x, z), (y, z), (x + y, z)]
8 for p, v in arrows:
9     arrow(pos=p, axis=v, color=v, shaftwidth=mag(v) / 50)
```

Lines 3,4: Using the class `vector` of VPython, represent the vectors $\vec{0} = (0, 0, 0)$, $\vec{x} = (1, 0, 0)$, $\vec{y} = (0, 1, 0)$ and $\vec{z} = (0, 0, 1)$ by o , x , y and z , respectively.

Lines 5–7: The list of vectors drawn as arrows. A couple of a starting point and a vector is the parameter determining an arrow.

Lines 8,9: Draw an arrow for each couple (p, v) in `arrows`. The color of the vector and the thickness of the axis are determined by the components and the size of the vector. We can change the viewpoint of the right-hand plot in Fig. 2.2 with a mouse.

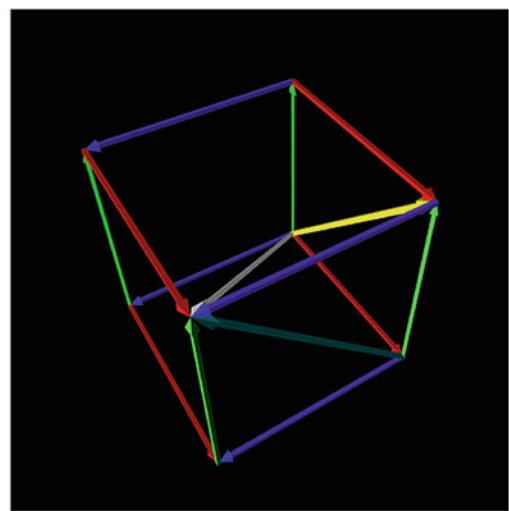
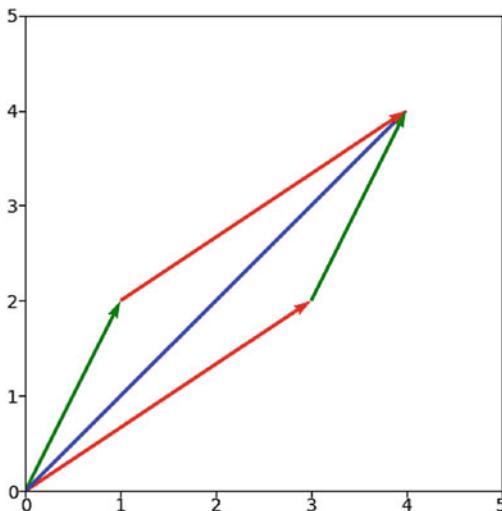


Fig. 2.2 Plane vector (left) and space vector (right)

Exercise 2.2 Using `vec3d.py`, check the other axioms of linear space.

From the axioms of linear space, the following assertions are immediately derived.⁵

1. $x + y = z \Rightarrow y = z - x$ (remark: $z - x$ means $z + (-x)$).

\therefore Assume $x + y = z$, then

$$\begin{aligned}
 y &= y + \mathbf{0} && \text{(Axiom (c))} \\
 &= y + (x + (-x)) && \text{(Axiom (d))} \\
 &= (y + x) + (-x) && \text{(Axiom (b))} \\
 &= (x + y) + (-x) && \text{(Axiom (a))} \\
 &= z + (-x). && \text{(assumption)}
 \end{aligned}$$

2. $a\mathbf{x} = \mathbf{y} \Rightarrow \mathbf{x} = \frac{\mathbf{y}}{a}$, if $a \neq 0$ (remark: $\frac{\mathbf{y}}{a}$ means $\frac{1}{a}\mathbf{y}$).

\therefore Assume $a \neq 0$ and $a\mathbf{x} = \mathbf{y}$, then

⁵ In the claim “ Q implies R when P ”, P , Q and R are called a precondition, an assumption and a conclusion, respectively. This form of assertion means that R is derived by proof assuming that P and Q are true. Sometimes a precondition does not appear or is omitted. In each claim here the precondition that \mathbf{x} , \mathbf{y} and \mathbf{z} are vectors in a linear space and a is a scalar is omitted.

$$\begin{aligned}
 x &= 1x && \text{(Axiom (h))} \\
 &= \left(\frac{1}{a} \cdot a\right)x && \text{(property of numbers)} \\
 &= \frac{1}{a}(ax) && \text{(Axiom (g))} \\
 &= \frac{1}{a}y. && \text{(assumption)}
 \end{aligned}$$

3. $x + y = x \Rightarrow y = \mathbf{0}$ (**uniqueness of zero vector**).

\because If $x + y = x$, then $y = x + (-x) = \mathbf{0}$ by 1.

4. $0x = \mathbf{0}$.

\because Because $x + 0x = 1x + 0x = (1 + 0)x = 1x = x$, we have $0x = \mathbf{0}$ by 3.

5. $a\mathbf{0} = \mathbf{0}$.

\because By 4, $a\mathbf{0} = a(0\mathbf{0}) = (a0)\mathbf{0} = 0\mathbf{0} = \mathbf{0}$.

6. $x + y = \mathbf{0} \Rightarrow y = -x$ (**uniqueness of inverse vector**).

\because From $x + y = \mathbf{0}$, we have $y = \mathbf{0} + (-x) = (-x) + \mathbf{0} = -x$ by 1.

7. $(-1)x = -x$.

\because Because $x + (-1)x = 1x + (-1)x = (1 + (-1))x = 0x = \mathbf{0}$, we have $-x = (-1)x$ by 6.

8. $-(-x) = x$.

\because Because $(-x) + x = x + (-x) = \mathbf{0}$, we have $-(-x) = x$ by 6.

9. $ax = \mathbf{0} \Rightarrow a = 0$ or $x = \mathbf{0}$.

\because Suppose $ax = \mathbf{0}$. If $a \neq 0$, then $x = \frac{1}{a}\mathbf{0} = \mathbf{0}$ by 2 and 5.

Exercise 2.3 Prove the following:

- (1) $\overbrace{x + x + \cdots + x}^n = nx$, (Hint: use Axioms (f), (h) and mathematical induction)
- (2) $ab \neq 0 \Rightarrow \frac{x}{a} + \frac{y}{b} = \frac{bx + ay}{ab}$,
- (3) If $x \neq \mathbf{0}$, then $ax = x \Rightarrow a = 1$,
- (4) If $x \neq \mathbf{0}$, then $ax = bx \Rightarrow a = b$.

The direct product \mathbb{K}^n of n copies of \mathbb{K} is a linear space over \mathbb{K} as shown below. In this book, we employ the following notation. When we consider an element $\vec{x} = (x_1, x_2, \dots, x_n)$ of \mathbb{K}^n as a vector, we sometimes write it in a column form

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

by arranging the components vertically. To save space the row notation (x_1, x_2, \dots, x_n) is also used. We will use them appropriately depending on where they are used. The vector sum is determined by

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \stackrel{\text{def}}{=} \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}.$$

In this case, the column notation is easier to see. We determine the scalar multiple, the zero vector and the inverse vector by

$$a(x_1, x_2, \dots, x_n) \stackrel{\text{def}}{=} (ax_1, ax_2, \dots, ax_n),$$

$$\mathbf{0} \stackrel{\text{def}}{=} (0, 0, \dots, 0)$$

and

$$-(x_1, x_2, \dots, x_n) \stackrel{\text{def}}{=} (-x_1, -x_2, \dots, -x_n),$$

respectively. Here we use the row notation to save space.

Now, Axiom (a) can be confirmed as follows:

$$\begin{aligned} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} &= \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix} && \text{(definition of vector sum)} \\ &= \begin{bmatrix} y_1 + x_1 \\ y_2 + x_2 \\ \vdots \\ y_n + x_n \end{bmatrix} && \text{(property of numbers)} \\ &= \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} + \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. && \text{(definition of vector sum)} \end{aligned}$$

Exercise 2.4 Prove that \mathbb{K}^n satisfies Axioms (b)–(h).

A point (x, y) in an xy -coordinate plane corresponds to a vector in \mathbb{R}^2 , and a point (x, y, z) in an xyz -coordinate space corresponds to a vector in \mathbb{R}^3 one-to-one (Fig. 2.3, the left and the center). When $n \geq 4$, we cannot draw an element (x_1, x_2, \dots, x_n) of \mathbb{R}^n as a point or an arrow, but we can present it as the graph of the function from $\{1, \dots, n\}$ to \mathbb{K} corresponding i to x_i (Fig. 2.3, right).

For a nonempty set X , the set \mathbb{K}^X of all functions defined on X with values in \mathbb{K} becomes a linear space over \mathbb{K} . For functions $f, g \in \mathbb{K}^X$ define the vector sum by

$$f + g : x \mapsto f(x) + g(x).$$

For a scalar $a \in \mathbb{K}$ define the scalar multiple by

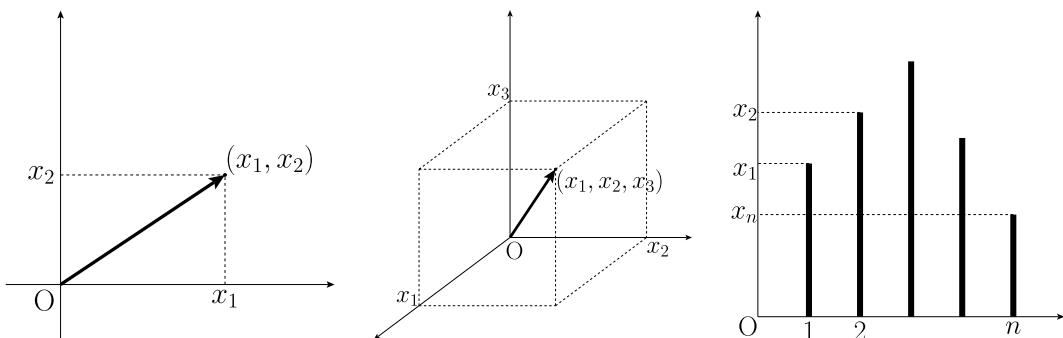


Fig. 2.3 Two-dimensional, three-dimensional and n -dimensional vectors (from left to right)

$$f : x \mapsto af(x).$$

The constant function

$$0 : x \mapsto 0$$

with value 0 is the zero vector, and the function

$$-f : x \mapsto -f(x)$$

is the inverse vector of f . Then, we can check Axiom (a) as follows. For any $x \in X$,

$$\begin{aligned} (f + g)(x) &= f(x) + g(x) \text{ (definition of vector sum)} \\ &= g(x) + f(x) \text{ (property of numbers)} \\ &= (g + f)(x). \text{ (definition of vector sum)} \end{aligned}$$

Exercise 2.5 Check that \mathbb{K}^X satisfies Axioms (b)–(h).

Consider two functions $f, g \in \mathbb{R}^{[-\pi, \pi]}$ given by

$$f(x) \stackrel{\text{def}}{=} x^2 - 1, \quad g(x) \stackrel{\text{def}}{=} 2 \sin 2x.$$

By drawing the graphs using Matplotlib, let us see the forms of their vector sums, scalar multiples and inverse vectors (Fig. 2.4).

Program: func.py

```
In [1]: 1 from numpy import pi, sin, cos, linspace
2 import matplotlib.pyplot as plt
3
4 zero = lambda x: 0*x
5 f = lambda x: x**2 - 1
6 g = lambda x: 2*sin(2*x)
7 fig = plt.figure(figsize=(20, 5))
8 x = linspace(-pi, pi, 101)
9 ax1 = fig.add_subplot(131)
10 for y in [zero(x), f(x), g(x), f(x) + g(x)]:
```

In [1]:

```

11     ax1.plot(x, y)
12     ax2 = fig.add_subplot(132)
13     for y in [zero(x), f(x), -f(x), f(x)/2]:
14         ax2.plot(x, y)
15     ax3 = fig.add_subplot(133)
16     for y in [zero(x), g(x), -g(x), 3*g(x)]:
17         ax3.plot(x, y)
18 plt.show()

```

Lines 4–6: Define the functions $\text{zero} : x \mapsto 0x$, $f : x \mapsto x^2 - 1$ and $g : x \mapsto 2 \sin 2x$.⁶

Line 7: Create a figure to draw graphs with size `figsize=(20, 5)`.

Line 8: `x` is an array of 101 points that divide the interval $[-\pi, \pi]$ equally.

Lines 9–11: Draw the graphs of the functions zero , f , g and $f + g$. The first argument of `add_subplot` on Line 9 specifies the matrix of cells where the graphs are drawn when the figure is divided into $m \times n$ cells. The 3-digit number mni means the i -th cell of $m \times n$ cells.⁷ Here the number 131 means the graphs are drawn on the first cell of 1×3 cells (Fig. 2.4, left). The loop counter `y` runs on `zero(x)`, `f(x)`, `g(x)` and `f(x) + g(x)`, which are, respectively, the arrays obtained by applying the functions zero , f , g and $f + g$ on each element in `x` simultaneously.⁸

Lines 12–14: Draw the graphs of the functions zero , f , $-f$ and $\frac{f}{2}$ in the second cell (Fig. 2.4, center).

Lines 15–17: Draw the graphs of the functions zero , g , $-g$ and $3g$ in the third cell (Fig. 2.4, right).

The set $\mathbb{C}^{[0, 2\pi]}$ of all functions defined on $[0, 2\pi]$ valued in \mathbb{C} is a complex linear space. Let $f_n \in \mathbb{C}^{[0, 2\pi]}$ ($n = 0, \pm 1, \pm 2, \dots$) be the functions sending t to e^{int} . Recall that $\operatorname{Re} f_n(t) = \cos nt$ and $\operatorname{Im} f_n(t) = \sin nt$ hold by Euler's formula given in Sect. 1.2. We draw f_n as functions

$$t \mapsto (t, \operatorname{Re} f_n(t), \operatorname{Im} f_n(t))$$

of $t \in [0, 2\pi]$ valued in three-dimensional space.

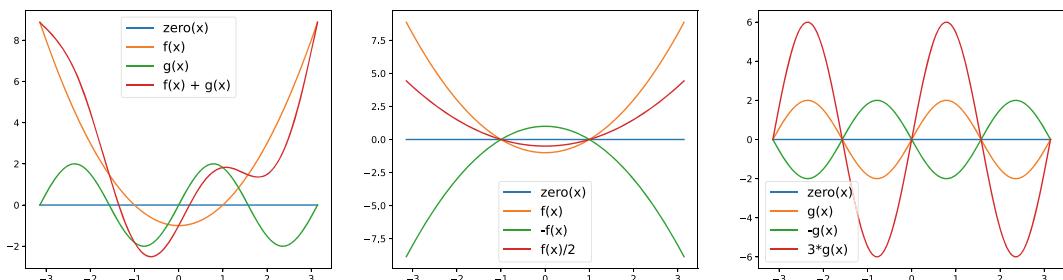
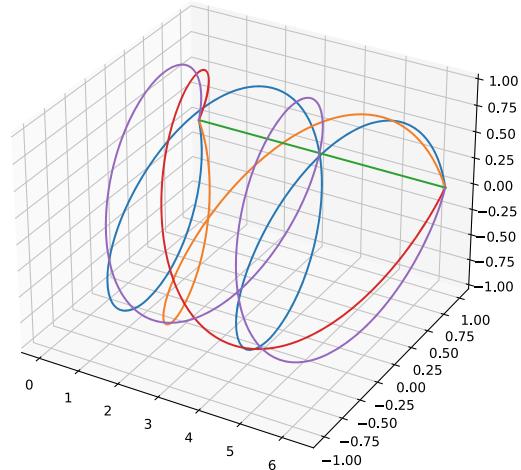


Fig. 2.4 Vector sum, scalar multiple, zero vector and inverse vector in $\mathbb{R}^{[-\pi, \pi]}$

⁶ We frequently use these definitions of functions using the lambda formula, though it is not recommended in the documents PEP8 in which a standard coding style of Python is given.

⁷ m , n and i must be natural numbers less than 10, but if any of them is greater than or equal to 10, we can write them separated with commas, which become the first three arguments of `fig.add_subplot`.

⁸ Applying a function to an array, the function is applied to each element of the array all at once. This is called a *broadcast*. To apply a function to each element of a list, we need to use the intentional definition or built-in function `map`. By using arrays and broadcasting, the code becomes easier to read and the run time is faster when the number of elements is large. It can be applied also to multi-dimensional arrays.

Fig. 2.5 Vectors in $\mathbb{C}^{[0,2\pi]}$ **Program:** cfunc.py

```
In [1]: 1 | from numpy import exp, pi, linspace
2 | import matplotlib.pyplot as plt
3 |
4 | f = lambda n, t: exp(1j * n * t)
5 | t = linspace(0, 2 * pi, 1001)
6 |
7 | fig = plt.figure(figsize=(7, 7))
8 | ax = fig.add_subplot(111, projection='3d')
9 | for n in range(-2, 3):
10 |     z = f(n, t)
11 |     ax.plot(t, z.real, z.imag)
12 | plt.show()
```

Line 4: Define the complex function $f_n : t \mapsto e^{int}$.

Line 5: t is the array of 1001 points that divide the interval $[0, 2\pi]$ equally.

Line 7: `figsize=(7, 7)` specifies the size of the figure to be displayed.

Line 8: The argument 111 means that only one cell is set for display. `projection='3d'` means the 3d coordinate system is projected on the cell.

Lines 9–12: Draw the graphs of $t \mapsto f_n(t)$ for $n = -2, -1, 0, 1, 2$. They are actually polygonal lines tracing the 1001 points. Each plane orthogonal to the t -axis ($0 \leq t \leq 2\pi$) is regarded as the complex plane (Fig. 2.5).

Exercise 2.6 Using the above program, observe the shapes of graphs of the functions for different n . In particular, choose n around 500 and around 1000.

2.2 Subspaces

Let V be a linear space over \mathbb{K} . A subset W of V is a *subspace* of V , if W itself is a linear space over \mathbb{K} with the vector sum, scalar multiple, zero vector and inverse vector defined in V , that is,

- S1. $\mathbf{x} + \mathbf{y} \in W$ for any $\mathbf{x}, \mathbf{y} \in W$ (**closed under vector summation**),
- S2. $a\mathbf{x} \in W$ for any $a \in \mathbb{K}$ and $\mathbf{x} \in W$ (**closed under scalar multiplication**),
- S3. $\mathbf{0} \in W$ (**W has the zero vector**),
- S4. $-\mathbf{x} \in W$ for any $\mathbf{x} \in W$ (**closed under vector inversion**).

Because Axioms (a)–(h) of linear space are already satisfied in V , it is not necessary to confirm that they are satisfied in W . Condition S3 follows from S2 if W is not empty, by putting $a = 0$ and choosing any $\mathbf{x} \in W$ in S2. Condition S4 also follows from S2, putting $a = -1$. By the above observation and Exercise 2.7 below, Conditions S1–S4 can be combined into one condition (S5 below) when $W \neq \emptyset$.

- S5. $a\mathbf{x} + b\mathbf{y} \in W$ for any $a, b \in \mathbb{K}$ and for any $\mathbf{x}, \mathbf{y} \in W$.

Exercise 2.7 Prove that S1 and S2 are equivalent to S5. Moreover, using mathematical induction, prove that a subspace W satisfies

$$a_1\mathbf{x}_1 + a_2\mathbf{x}_2 + \cdots + a_n\mathbf{x}_n \in W$$

for any $a_1, a_2, \dots, a_n \in \mathbb{K}$ and $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n \in W$.

Obviously, V itself is a subspace of V . On the other hand, the singleton set $\{\mathbf{0}\}$ is also a subspace. In fact, $\mathbf{0} + \mathbf{0} = \mathbf{0}$ and $a\mathbf{0} = \mathbf{0}$ for all $a \in \mathbb{K}$, so it is closed under a vector sum and scalar multiple. These two subspaces are called the *trivial subspaces* of V .

Exercise 2.8 Prove that $\{(x, 0) \mid x \in \mathbb{K}\}$ and $\{(x, x) \mid x \in \mathbb{K}\}$ are subspaces of \mathbb{K}^2 .

A nontrivial subspace of a plane is a line passing through the origin, and a nontrivial subspace of three-dimensional space is either a line or a plane passing through the origin.

Exercise 2.9 Explain geometrically why subspaces of a plane and space are given as above.

Let W_1 and W_2 be subspaces of V . Then, the intersection $W_1 \cap W_2$ is a subspace of V . In fact, $\mathbf{0} \in W_1 \cap W_2$ because $\mathbf{0} \in W_1$ and $\mathbf{0} \in W_2$. In addition, for any $a, b \in \mathbb{K}$ and $\mathbf{x}, \mathbf{y} \in W_1 \cap W_2$, we see $a\mathbf{x} + b\mathbf{y} \in W_1$ because $\mathbf{x}, \mathbf{y} \in W_1$ and W_1 is a subspace. Similarly, $a\mathbf{x} + b\mathbf{y} \in W_2$, and hence, $a\mathbf{x} + b\mathbf{y} \in W_1 \cap W_2$. Thus, S5 in Exercise 2.7 is satisfied, and so $W_1 \cap W_2$ is a subspace.

Next, let $\{W_i \mid i \in I\}$ be a nonempty family⁹ of subspaces of V . We define the intersection

$$\bigcap_{i \in I} W_i \stackrel{\text{def}}{=} \{\mathbf{x} \mid \mathbf{x} \in W_i \text{ for all } i \in I\}.$$

In particular, when $I = \{1, 2, \dots, n\}$, we can write

$$\bigcap_{i \in I} W_i = W_1 \cap W_2 \cap \cdots \cap W_n.$$

⁹ A set of sets is called a *family*. If you have some difficulty in understanding a family of subspaces, imagine a set of several lines passing through the origin in $V = \mathbb{R}^2$ or a set of planes in \mathbb{R}^3 .

We claim that $\bigcap_{i \in I} W_i$ is a subspace of V .

Exercise 2.10 Prove that $\bigcap_{i \in I} W_i$ is a subspace of V .

Let S be a subset of V and let $\{W_i \mid i \in I\}$ be the family of all subspaces of V containing S . Then, $W_0 = \bigcap_{i \in I} W_i$ is a subspace of V containing S . Because W_0 is contained in W_i for every $i \in I$, it is the smallest subspace containing S . Thus, we see that for any subset S of V there exists the smallest subspace containing S with respect to the inclusion relation of sets.

Exercise 2.11 Prove that for a subspace W of V , the complement W^C can never be a subspace of V . Moreover, for subspaces W_1 and W_2 of V , does $W_1 \cup W_2$ or $W_1 \setminus W_2$ become a subspace?

2.3 Linear Mappings

Let V and W be linear spaces over \mathbb{K} . A mapping $f : V \rightarrow W$ is called a *linear mapping*¹⁰ if it reflects upon W the structure of V consisting of the vector sum, scalar multiple, zero vector and inverse vector,¹¹ that is, it satisfies

- M1. $f(x + y) = f(x) + f(y)$ for any $x, y \in V$ (**preserve vector sum**),
- M2. $f(ax) = af(x)$ for any $a \in \mathbb{K}$ and $x \in V$ (**preserve scalar multiple**),
- M3. $f(\mathbf{0}_V) = \mathbf{0}_W$ (**preserve zero vector**),
- M4. $f(-x) = -f(x)$ for any $x \in V$ (**preserve inverse vector**).

Here, $\mathbf{0}_V$ and $\mathbf{0}_W$ denote the zero vectors in V and in W , respectively.

Figure 2.6 graphically shows the preservation of vector sum by a linear mapping f . Condition M3 follows from M2 by letting $a = 0$, and Condition M4 follows from M2 by letting $a = -1$ too. Conditions M1–M4 can be combined into the following condition:

- M5. $f(ax + by) = af(x) + bf(y)$ for any $a, b \in \mathbb{K}$ and $x, y \in V$.

Exercise 2.12 Show that M1 and M2 are equivalent to M5. Moreover, using mathematical induction, prove that a linear mapping f satisfies

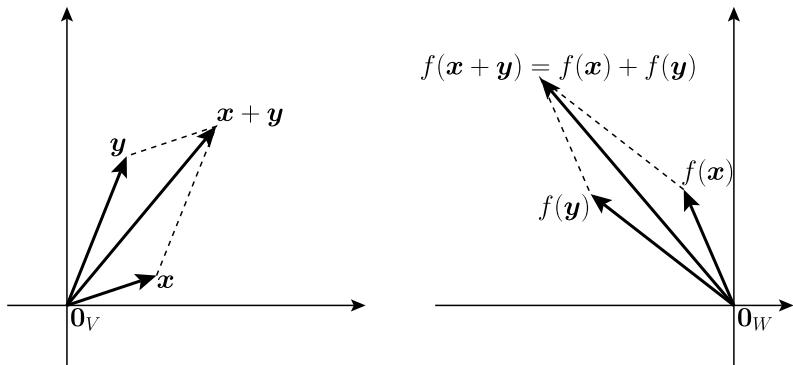
$$f(a_1x_1 + a_2x_2 + \cdots + a_nx_n) = a_1f(x_1) + a_2f(x_2) + \cdots + a_nf(x_n)$$

for any $a_1, a_2, \dots, a_n \in \mathbb{K}$ and $x_1, x_2, \dots, x_n \in V$.

¹⁰ Sometimes abbreviated to *linear map*. The term *linear transformation* also is used in some literature.

¹¹ We say that f *preserves linear structure*.

Fig. 2.6 Preservation of vector sum



For two linear mappings $f, g : V \rightarrow W$, the mapping $f + g : x \mapsto f(x) + g(x)$ is called the *sum of linear mappings*. For $a \in \mathbb{K}$ and for a linear mapping $f : V \rightarrow W$, the mapping $af : x \mapsto af(x)$ is called the *scalar multiple* of a linear mapping.

Next, let U be another linear space over \mathbb{K} . For linear mappings $f : V \rightarrow W$ and $g : U \rightarrow V$, the composition $f \circ g : x \mapsto f(g(x))$ is called the *composition* of linear mappings.

The sum, scalar multiple and composition of linear mappings are all linear mappings. We can check that the sum of linear mappings is a linear mapping as follows:

$$\begin{aligned}
 (f + g)(ax + by) &= f(ax + by) + g(ax + by) && \text{(definition of sum of mappings)} \\
 &= af(x) + bf(y) + ag(x) + bg(y) && \text{(linearity of } f \text{ and } g\text{)} \\
 &= a(f(x) + g(x)) + b(f(y) + g(y)) && \text{(axioms of linear space)} \\
 &= a(f + g)(x) + b(f + g)(y) && \text{(definition of sum of mappings)}
 \end{aligned}$$

for $a, b \in \mathbb{K}$ and $x, y \in V$.

Exercise 2.13 Prove that the scalar multiple of a linear mapping and the composition of linear mappings are linear mappings.

The set $L(V, W)$ of all linear mappings from V to W forms a linear space. In fact, the sum and the scalar multiple are defined above. The zero mapping sending every element $x \in V$ to the zero vector $\mathbf{0}$ of W is the zero vector in $L(V, W)$. For $f \in L(V, W)$, $-f$ is its inverse vector.

Exercise 2.14 Prove Axioms (a)–(h) of linear space for $L(V, W)$.

If a linear mapping $f : V \rightarrow W$ is bijective, it is called a *linear isomorphism*. The inverse mapping $f^{-1} : W \rightarrow V$ of a linear isomorphism f is also a linear isomorphism. The linearity of f^{-1} is shown as follows. Suppose that $y = f(x)$ and $w = f(v)$ for $x, v \in V$, then

$$\begin{aligned}
 f^{-1}(ay + bw) &= f^{-1}(af(x) + bf(v)) && (\text{assumption}) \\
 &= f^{-1}(f(ax + bv)) && (\text{linearity of } f) \\
 &= ax + bv && (f^{-1} \text{ is the inverse of } f) \\
 &= af^{-1}(f(x)) + bf^{-1}(f(v)) && (f^{-1} \text{ is the inverse of } f) \\
 &= af^{-1}(y) + bf^{-1}(w). && (\text{assumption})
 \end{aligned}$$

The range of a linear mapping $f : V \rightarrow W$, denoted by $\text{range}(f)$, is a subspace of W . In fact, $\text{range}(f)$ is a nonempty subset of W because $f(\mathbf{0}_V) = \mathbf{0}_W \in \text{range}(f)$. For any $a, b \in \mathbb{K}$ and $y_1, y_2 \in \text{range}(f)$ there are $x_1, x_2 \in V$ such that $y_1 = f(x_1)$, $y_2 = f(x_2)$, and we have

$$ay_1 + by_2 = af(x_1) + bf(x_2) = f(ax_1 + bx_2).$$

Therefore, $ay_1 + by_2 \in \text{range}(f)$.

On the other hand, we define

$$\text{kernel}(f) \stackrel{\text{def}}{=} \{x \mid f(x) = \mathbf{0}_W\}.$$

This is the inverse image of $\{\mathbf{0}_W\}$ by f and is called the *kernel* of f (Fig. 2.7). Again, $\text{kernel}(f)$ is a subspace of V . This can be seen as follows. First, $\text{kernel}(f)$ is a nonempty subset of V containing $\mathbf{0}_V$ because $f(\mathbf{0}_V) = \mathbf{0}_W$. Next, for any $a, b \in \mathbb{K}$ and $x_1, x_2 \in \text{kernel}(f)$, because $f(x_1) = f(x_2) = \mathbf{0}_W$, we have

$$f(ax_1 + bx_2) = af(x_1) + bf(x_2) = a\mathbf{0}_W + b\mathbf{0}_W = \mathbf{0}_W.$$

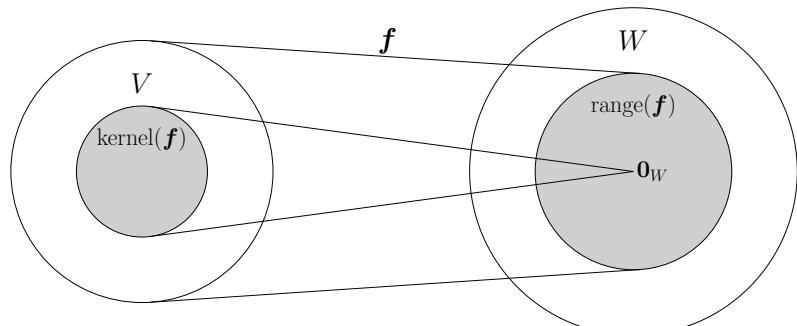
It follows that $ax_1 + bx_2 \in \text{kernel}(f)$.

The following equivalences hold about the range and the kernel of a linear mapping $f : V \rightarrow W$:

1. f is surjective $\Leftrightarrow \text{range}(f) = W$.
2. f is injective $\Leftrightarrow \text{kernel}(f) = \{\mathbf{0}_V\}$.

Equivalence in 1 is nothing but the definition of surjectivity. Let us prove 2. First, suppose that f is injective. If $x \in \text{kernel}(f)$, then $f(x) = \mathbf{0}_W$. Because $f(\mathbf{0}_V) = \mathbf{0}_W$, we see $f(x) = f(\mathbf{0}_V)$, and so $x = \mathbf{0}_V$ by the injectivity of f . Conversely, assume that $\text{kernel}(f) = \{\mathbf{0}_V\}$. Then, for $x, y \in V$ such that $f(x) = f(y)$, we have

Fig. 2.7 Kernel and range



$$f(x - y) = f(x) - f(y) = \mathbf{0}_W.$$

By assumption $x - y = \mathbf{0}_V$, that is, $x = y$.

If there is a linear isomorphism $f : V \rightarrow W$, two linear spaces V and W are called *isomorphic* and written as $V \cong W$. Isomorphic spaces are considered to have the same linear structure and we sometimes identify them. If f is injective, f induces an isomorphism from V to range (f). In this situation, we call f an *embedding* of V into W and sometimes consider V as a subspace of W .

Exercise 2.15 For given constants $a, b, c, d \in \mathbb{R}$ define the mapping $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ by sending $(x, y) \in \mathbb{R}^2$ to $(u, v) \in \mathbb{R}^2$, where (u, v) is determined by

$$\begin{cases} u = ax + by \\ v = cx + dy. \end{cases}$$

Prove that f is a linear mapping. In particular, in each of the cases $(a, b, c, d) = (1, 2, 2, 3)$ and $(a, b, c, d) = (1, 2, 2, 4)$ determine the kernel and the range of f .

Exercise 2.16 Consider the set $V = \{ax^2 + bx + c \mid a, b, c \in \mathbb{R}\}$ of all constant, linear and quadratic functions.

- (1) Prove that V is a linear space over \mathbb{R} .
- (2) Prove that $W = \{ax + b \mid a, b \in \mathbb{R}\}$ is a subspace of V .
- (3) For a function $f \in V$, $D(f)$ denotes the derivative f' of f . For example, $D(1) = 0$, $D(x) = 1$, $D(x^2) = 2x$. Prove that $D : V \rightarrow V$ is a linear mapping.
- (4) Determine range (D) and kernel (D).

2.4 Application: Visualizing Sounds

Sounds are vibrations of the air. They are considered to be functions of time valued in air pressure.

- The vector sum is making two sounds at the same time.
- The scalar multiple is turning the volume up or down.
- The zero vector is silence.
- The inverse vector is reversing the phase. We cannot hear the difference unfortunately, but by adding to a sound the sound corresponding to its inverse vector, we must get silence.¹²

In order to handle sounds by Python we use the external library SciPy, which provides tools for data science such as signal processing and statistical processing. There are several file formats for voice data, and here we use the WAV format. We can find files of sounds recorded in WAV format on the

¹² A noise canceling headphone adopts this principle, that is, it cancels the noise coming from outside by adding its inverse to it.

Web, etc.¹³ The file extension is wav. A file of a few seconds is suitable for this experiment. In the following program, replace sample.wav by the name of the file you obtained.

Program: sound2.py

```
In [1]: 1 import scipy.io.wavfile as wav
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 rate, Data = wav.read('sample.wav')
6 print(rate, Data.shape)
7 n = len(Data)
8 t = n / rate
9 dt = 1 / rate
10 print(t, dt)
11 x = np.arange(0, n*dt, dt)
12 y = Data / 32768
13
14 if len(Data.shape) == 1:
15     plt.plot(x, y)
16     plt.xlim(0, t), plt.ylim(-1, 1)
17 elif len(Data.shape) == 2:
18     fig, ax = plt.subplots(2)
19     for i in range(2):
20         ax[i].plot(x, y[:, i])
21         ax[i].set_xlim(0, t), ax[i].set_ylim(-1, 1)
22 plt.show()
```

Line 1: Import SciPy to handle WAV format files.

Line 5: Read sample.wav,¹⁴ where rate is the number of samplings per second, and Data is the sound data itself.

Lines 6–10: Check the form (Data.shape) of rate and Data. If displayed as

```
22050 (32768,)
1.486077097505669 4.5351473922902495e-05
```

the sound is recorded at the sampling rate 22050 Hz (Hertz), that is, the sound is sampled at the time interval of 1 s divided by 22050. Data is a one-dimensional array of 32768 sound samples (monaural). The recording time is 32768/22050, which is about 1.5 s. The pitch of each sound sample is recorded as a 16-bit integer value. Thus, it is quantized to integer values between –32769 and 32768. If displayed as

```
44100 (223744, 2)
5.073560090702948 2.2675736961451248e-05
```

the sound is recorded at the sampling rate 44100 Hz. Data is the two-dimensional array of 44100 pairs of left and right sound samples (stereo). The recording time is 44100/44100, which is about 0.9 s.

Line 11: x is the sequence of times from 0 to t (t is not included) at the interval of dt seconds. We obtain the same sequence by using np.linspace(0, t, n, endpoint=False).

Line 12: y is the array of the sound pressure levels represented by integers between –32769 and 32768 converted to real numbers in [–1, 1].

Lines 14–22: The processes performed for monaural and stereo are different. The block of the elif-statement is executed if the if-statement is not executed and the conditional clause following elif is satisfied.

¹³ We can record sounds in a file of WAV format by using an application called Audacity. (see Appendix A.6.)

¹⁴ When you execute this, use the name of the WAV file you prepared.

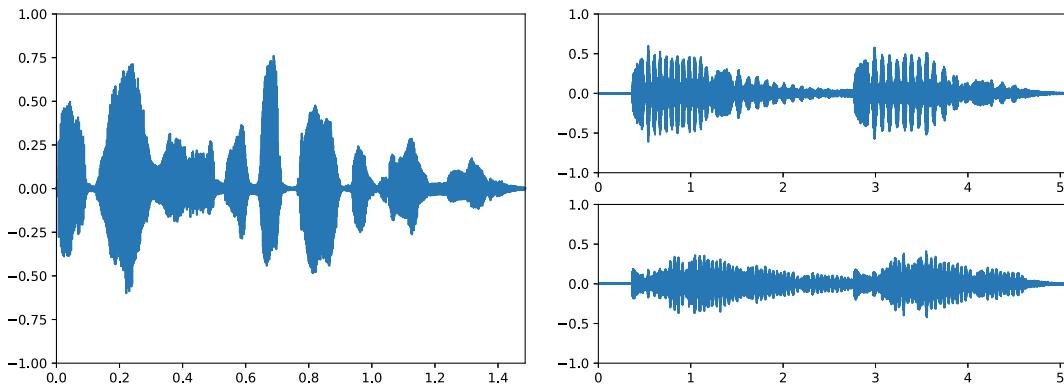


Fig. 2.8 Waveforms of the sounds (left: monaural, right: stereo)

Examples of the waveforms are displayed in Fig. 2.8. Changing `xlim` (monaural) or `set_xlim` (stereo), we can see the forms by partially enlarging the time axis.

The next program artificially creates sinewaves representing the notes do (C), mi (E) and sol (G), adds them to make a C-chord and then saves them in a wav file of 2 s. Figure 2.9 shows these waveforms from 1 s to 1.01 s.

Program: chord.py

```
In [1]: 1  from numpy import arange, pi, sin
2  import scipy.io.wavfile as wav
3  import matplotlib.pyplot as plt
4
5  xmax, rate = 2, 22050
6  x = arange(0, xmax, 1 / rate)
7
8  def f(hz):
9      octave = [2 ** n for n in [0, 1, 2, -2, -1]]
10     return [sin(2 * pi * hz * n * x) * 0.9 for n in octave]
11
12 A = f(440.000000) # la
13 B = f(493.883301) # si
14 C = f(523.251131) # do
15 D = f(587.329536) # re
16 E = f(659.255114) # mi
17 F = f(698.456463) # fa
18 G = f(783.990872) # sol
19 CEG = (C[0] + E[0] + G[0]) / 3
20 Data = (CEG * 32768).astype('int16')
21 wav.write('CEG.wav', rate, Data)
22 for y in [C[0], E[0], G[0], CEG]:
23     plt.plot(x, y)
24 plt.xlim(1, 1.01), plt.show()
```

Line 5: `xmax` is the length (in seconds) of the sound, and `rate` is the sampling frequency.

Line 6: `x` is the array of elements less than `xmax` of the arithmetic progression whose initial term is 0 and difference is the sampling period.

Lines 8–10: Make five sound sources two octaves above and below the frequency of the reference sound given in `hz`. Each is multiplied by 0.9 so that the amplitude is less than 1. The return value is a list of arrays representing the reference sound source and the sound sources one octave above, two octaves above, two octaves below and one octave below. We order the data in this way so that the sounds below the reference sound can be referenced by a negative index.

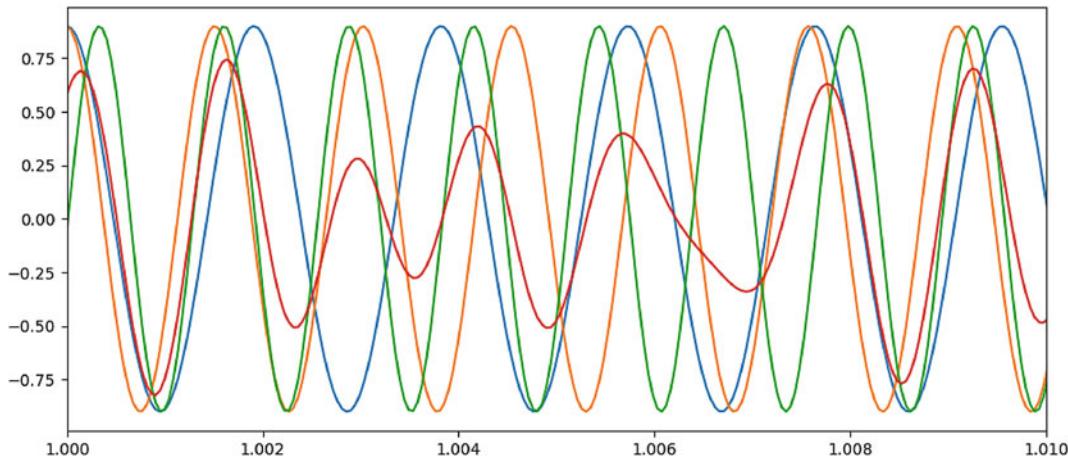


Fig. 2.9 Part of the do-mi-sol chord

Lines 12–18: Create the sound sources from A (la) to G (sol). For example, from A [0] to G [0], from A [1] to G [1] and from A [-1] to G [-1] are the reference scale, the scale one octave above and the scale one octave below, respectively.

Line 19: Make the chord. The sum is divided by 3 so that the absolute value does not exceed 1.

Line 20: Convert real values between -1 and less than 1 to integer values between -32769 and 32768 (16-bit integer). This process is called quantization.

Line 21: The resulting chord is saved as a wav file.

Lines 22–24: Starting at 1 s, draw the graphs of the waveforms of the chord and its constituent sounds for 0.01 s (Fig. 2.9).



Basis and Dimension

3

In this chapter, we learn the notions of “subspace generation” and “linear independence”, which play essential roles in linear algebra. These yield the notions of “basis” and “dimension”, and they turn out to be useful tools to analyze a linear space. Some important theorems concerning them will appear. The notion of linear independence is very important, and the readers are encouraged to understand its meaning to proceed to the next step.

Everything we learn in this chapter concerns linear equations. Using Python we can solve equations both numerically and symbolically (treating fractions and letters as such). The former is suitable for applications and the latter is suitable for mathematical thinking. This chapter will demonstrate the practical use of Python for both.

3.1 Finite-Dimensional Linear Spaces

Let V be a linear space over \mathbb{K} , and let $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ be elements of V . For $x_1, x_2, \dots, x_n \in \mathbb{K}$, the expression

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n$$

is called a *linear combination* of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$. Let W be the set of all linear combinations of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$, then W becomes the smallest subspace of V containing $A = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$.¹ This fact can be proved as follows. First, because each \mathbf{a}_i is expressed as a linear combination of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ as

$$\mathbf{a}_i = 0\mathbf{a}_1 + \cdots + 1\mathbf{a}_i + \cdots + 0\mathbf{a}_n,$$

W includes A . Next, we shall prove that W is a subspace by showing condition S5 in Exercise 2.7. W is not empty, as we have just seen above. For any $\alpha, \beta \in \mathbb{K}$ and for any two elements $x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n$ and $y_1\mathbf{a}_1 + y_2\mathbf{a}_2 + \cdots + y_n\mathbf{a}_n$ of W , we have

$$\begin{aligned} & \alpha(x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n) + \beta(y_1\mathbf{a}_1 + y_2\mathbf{a}_2 + \cdots + y_n\mathbf{a}_n) \\ &= (\alpha x_1 + \beta y_1)\mathbf{a}_1 + (\alpha x_2 + \beta y_2)\mathbf{a}_2 + \cdots + (\alpha x_n + \beta y_n)\mathbf{a}_n \in W. \end{aligned}$$

¹ The existence of such a subspace was shown at the end of Sect. 2.2.

Hence, W is a subspace of V . Furthermore, let W' be any subspace of V including A . Because W' is a subspace, $x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \dots + x_n\mathbf{a}_n$ belongs to W' for any $x_1, x_2, \dots, x_n \in \mathbb{K}$, and so $W \subseteq W'$. It follows that W is smallest with respect to the inclusion relation of sets.

We call this W the subspace *generated by* (or *spanned by*) A , and we denote it by

$$\langle A \rangle = \langle \mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \rangle.$$

As an extreme case we consider that the trivial subspace $\{\mathbf{0}\}$ is the subspace of V generated by the empty set $\{\}$, for $\{\mathbf{0}\}$ is the smallest subspace of V containing $\{\}$.

If V is generated by a finite number of vectors, it is called a *finite-dimensional linear space*. Many of the linear spaces treated in this book are finite dimensional. It is important to find a set $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ that generates a finite-dimensional linear space V , and more so to find one as small as possible. If the smallest number n (called the dimension of V) is found, all calculations related to linear algebra on V can be translated into the world of the n -dimensional coordinate space \mathbb{K}^n , as we will see in this and subsequent chapters.

In order to see the subspace generated by two vectors, we randomly make 1000 linear combinations of the two vectors.

Program: lincombi.py

```
In [1]: 1 from vpython import vec, arrow, color, points
2 from numpy.random import normal
3
4 x = vec(*normal(0, 1, 3))
5 arrow(pos=vec(0, 0, 0), axis=x, color=color.red)
6 y = vec(*normal(0, 1, 3))
7 arrow(pos=vec(0, 0, 0), axis=y, color=color.red)
8 W = [a * x + b * y for (a, b) in normal(0, 1, (1000, 2))]
9 points(pos=W, radius=2)
```

Line 2: To generate random numbers subject to a normal distribution, use `normal` from a module called `random` in NumPy where various random numbers are defined.

Lines 4-7: `normal(0, 1, 3)` produces an array of three independent random numbers subject to the standard normal distribution (mean 0, standard deviation 1). `vec` requires three arguments, and if `p` is a list (or tuple, array) of length 3 and if we call it as `vec(*p)`, the three elements of `p` are decomposed and are passed to `vec` as the three arguments. In this way, vectors `x` and `y` of three-dimensional space are randomly generated and drawn as arrows.

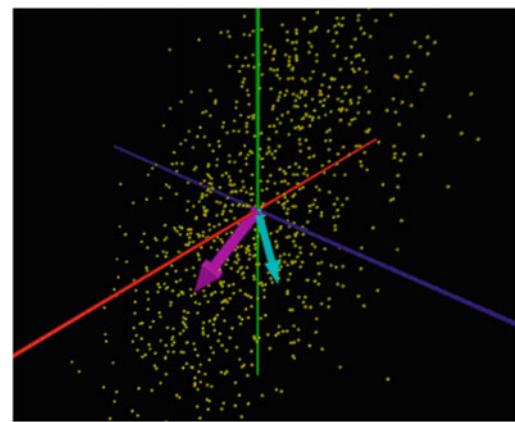
Line 8: Make 1000 pairs (a, b) subject to the two-dimensional standard normal distribution, and then make linear combinations $a * x + b * y$. `normal(0, 1, (1000, 2))` is an array that expresses a matrix of shape $(1000, 2)$, and regarding this as a nested array of length 1000 of arrays of length 2, make a list of linear combinations using the *intentional definition of lists* in the same way as sets.

Line 9: Draw the vectors in `W` as points (Fig. 3.1). Changing the viewpoint with the mouse, we can see that all 1000 points are on a certain plane, and the vectors `x` and `y` are also on that plane.

Consider the two-dimensional coordinate plane \mathbb{R}^2 and a vector $\vec{p} = (1, 2)$ in it. Scalar multiples $a\vec{p} = (a, 2a)$ of \vec{p} by $a \in \mathbb{R}$ all lie on the straight line $l : y = 2x$. Hence, the subspace $\langle \vec{p} \rangle$ generated by \vec{p} becomes this straight line l . Take another vector $\vec{q} = (2, 3)$ that is not on l . If $(x, y) \in \mathbb{R}^2$ is written as a linear combination of \vec{p} and \vec{q} , that is,

$$\begin{bmatrix} x \\ y \end{bmatrix} = a \begin{bmatrix} 1 \\ 2 \end{bmatrix} + b \begin{bmatrix} 2 \\ 3 \end{bmatrix},$$

Fig. 3.1 Linear combinations of two vectors in 3D



then we have a system of (simultaneous) equations

$$\begin{cases} x = a + 2b \\ y = 2a + 3b. \end{cases}$$

Solving it for a, b , we get

$$\begin{cases} a = -3x + 2y \\ b = 2x - y. \end{cases}$$

Thus, any $(x, y) \in \mathbb{R}^2$ can be written as a linear combination of \vec{p} and \vec{q} by determining a, b in this way, and hence the subspace $\langle \vec{p}, \vec{q} \rangle$ generated by \vec{p} and \vec{q} becomes whole \mathbb{R}^2 . Let us solve this equation by SymPy.

Program: eqn1.py

```
In [1]: 1 from sympy import solve
2 from sympy.abc import a, b, x, y
3
4 ans = solve([a + 2*b - x, 2*a + 3*b - y], [a, b])
5 print(ans)
```

Line 2: Import symbols used for constants and unknowns.

Line 4: Transform the equations into the form $\dots = 0$ and pass the list of its left-hand sides to the function `solve` as the first argument. The list of the second argument indicates the unknowns.



```
{a: -3*x + 2*y, b: 2*x - y}
```

The solution `ans` is a dictionary with the unknowns as keys. We can refer to it with keys as follows.

```
In [2]: ans[a]
```

```
Out[2]: -3*x + 2*y
```

```
In [3]: ans[b]
```

```
Out[3]: 2*x - y
```

Let us consider the three-dimensional coordinate space \mathbb{R}^3 and a vector $\vec{p} = (1, 2, 3)$ in it. Scalar multiples $a\vec{p} = (a, 2a, 3a)$ of \vec{p} by $a \in \mathbb{R}$ all lie on the straight line l passing through the origin and \vec{p} , and so $\langle \vec{p} \rangle$ becomes this line l . Take another vector $\vec{q} = (2, 3, 4)$ that is not on l . If $(x, y, z) \in \mathbb{R}^3$ is written as a linear combination of \vec{p} and \vec{q} , then

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = a \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + b \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

for $a, b \in \mathbb{R}$. So, it must satisfy the equations

$$\begin{cases} x = a + 2b \\ y = 2a + 3b \\ z = 3a + 4b. \end{cases}$$

Eliminating a , we have

$$\begin{cases} 2x - y = b \\ 3x - z = 2b, \end{cases}$$

and observing the right-hand sides we get $2(2x - y) = 3x - z$, that is,

$$x - 2y + z = 0.$$

This is the equation of the plane $C : x - 2y + z = 0$ passing through the origin, and we find that $\langle \vec{p}, \vec{q} \rangle$ equals C . Let us solve this by SymPy.

Program: eqn2.py

```
In [1]: 1 from sympy import solve
2 from sympy.abc import a, b, x, y, z
3
4 ans = solve([a + 2*b - x, 2*a + 3*b - y, 3*a + 4*b - z], [a, b])
5 print(ans)
```



[]

This implies that there is no solution (a, b) for general x, y , and z . Adding x as unknown, changing $[a, b]$ in Line 4 to $[a, b, x]$, we execute the program again. Then we should have

In [2]: ans

Out[2]: {x: 2*y - z, b: 3*y - 2*z, a: -4*y + 3*z}

which implies

$$\begin{cases} x = 2y - z \\ b = 3y - 2z \\ a = -4y + 3z. \end{cases}$$

The first line is the equation of plane C . Let $\vec{r} = (3, 4, 5)$. Then, because its coordinates satisfy the equation of the plane C , that is, $3 = 2 \cdot 4 - 5$, it lies on C . Therefore, $\langle \vec{p}, \vec{q}, \vec{r} \rangle = \langle \vec{p}, \vec{q} \rangle$ holds.

Next, let $\vec{r} = (3, 1, 2)$, then \vec{r} is not on C because $3 \neq 2 \cdot 1 - 2$. From

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = a \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + b \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} + c \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix},$$

we have the equations

$$\begin{cases} x = a + 2b + 3c \\ y = 2a + 3b + c \\ z = 3a + 4b + 2c. \end{cases}$$

Solving this system, we obtain

$$\begin{cases} a = -\frac{2}{3}x - \frac{8}{3}y + \frac{7}{3}z \\ b = \frac{1}{3}x + \frac{7}{3}y - \frac{5}{3}z \\ c = \frac{1}{3}x - \frac{2}{3}y + \frac{1}{3}z. \end{cases}$$

Thus, determining a, b, c in this way, any $(x, y, z) \in \mathbb{R}^3$ is written as a linear combination of $\vec{p}, \vec{q}, \vec{r}$, and hence we see $\langle \vec{p}, \vec{q}, \vec{r} \rangle = \mathbb{R}^3$. Let us solve the problem by SymPy.

Program: eqn3.py

```
In [1]: 1 from sympy import solve
2 from sympy.abc import a, b, c, x, y, z
3
4 ans = solve([a + 2*b + 3*c - x, 2*a + 3*b + c - y,
5             3*a + 4*b + 2*c - z], [a, b, c])
6
7 print(ans)
```

```
{a: -2*x/3 - 8*y/3 + 7*z/3, b: x/3 + 7*y/3 - 5*z/3, c: x/3 - 2*y/3 + z/3}
```

The solution for a, b, c is expressed in x, y, z . Let us find a solution for the substitution $x = 2, y = 3, z = 5$.

```
In [2]: N = [ans[k].subs([[x, 2], [y, 3], [z, 5]]) for k in [a, b, c]]; N
```

```
Out[2]: [7/3, -2/3, 1/3]
```

The solution contains fractions, which can be expressed as decimal numbers using `evalf`. The number of significant digits is specified by the argument of `evalf`.

```
In [3]: [n.evalf(2) for n in N]
```

```
Out[3]: [2.3, -0.67, 0.33]
```

Exercise 3.1 Express \vec{x} as a linear combination of the vectors in A in each case below, and check your solution by solving it by SymPy. Moreover, draw \vec{x} and the vectors in A using Matplotlib in case 1 and using VPython in case 2.

- (1) $\vec{x} = (17, -10)$, $A = \{(5, -4), (4, -5)\}$.
- (2) $\vec{x} = (-16, 1, 10)$, $A = \{(1, 0, 0), (1, 1, 0), (1, 1, 1)\}$.

3.2 Linear Dependence and Linear Independence

Let $A = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\} \subseteq V$, then we always have $0\mathbf{a}_1 + 0\mathbf{a}_2 + \dots + 0\mathbf{a}_n = \mathbf{0}$. If

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \dots + x_n\mathbf{a}_n = \mathbf{0} \quad \dots \quad (*)$$

holds for some $x_1, x_2, \dots, x_n \in \mathbb{K}$ at least one of which is not 0, we say that A is *linearly dependent*. For example, $\{2\mathbf{a}, 3\mathbf{a}\}$ for any vector \mathbf{a} is linearly dependent because

$$3 \cdot 2\mathbf{a} + (-2) \cdot 3\mathbf{a} = \mathbf{0}.$$

On the other hand, we say that A is *linearly independent* if it is not linearly dependent, that is, $(*)$ holds only when $x_1 = x_2 = \dots = x_n = 0$. For example, $\{\mathbf{a}\}$ is linearly independent if $\mathbf{a} \neq \mathbf{0}$.²

Let us examine if $A = \{(1, 2), (2, 3)\} \subseteq \mathbb{R}^2$ is linearly independent. Suppose

$$x \begin{bmatrix} 1 \\ 2 \end{bmatrix} + y \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

holds for $x, y \in \mathbb{K}$, that is,

$$\begin{cases} x + 2y = 0 \\ 2x + 3y = 0. \end{cases}$$

This system of equations has the unique solution $x = y = 0$ shown as below. Hence we find that A is linearly independent.

Program: eqn4.py

```
In [1]: 1 | from sympy import solve
2 | from sympy.abc import x, y
3 |
4 | ans = solve([x + 2*y, 2*x + 3*y], [x, y])
5 | print(ans)
```

```
{x: 0, y: 0}
```

Next, let us examine it for $A = \{(1, 2), (2, 4)\} \subseteq \mathbb{R}^2$. The problem is reduced to the equations

$$\begin{cases} x + 2y = 0 \\ 2x + 4y = 0, \end{cases}$$

² We showed that $x\mathbf{a} = \mathbf{0} \Rightarrow x = 0$ for $\mathbf{a} \neq \mathbf{0}$ in Sect. 2.1.

which have the solution $(x, y) = (2, -1)$ other than $(x, y) = (0, 0)$ for example. Hence, A is linearly dependent.

Program: eqn5.py

```
In [1]: 1 from sympy import solve
2 from sympy.abc import x, y
3
4 ans = solve([x + 2*y, 2*x + 4*y], [x, y])
5 print(ans)
```

{x: -2*y}



This implies that $x = -2y$ is the solution, where y is arbitrary.

The next program draws three vectors $\vec{a} = (1, 2)$, $\vec{b} = (2, 3)$, $\vec{c} = (2, 4)$ on a two-dimensional plane (Fig. 3.2, left). Two vectors \vec{a} and \vec{c} lie on the same line passing through the origin. Vector \vec{b} faces the different direction.

Program: arrow2d.py

```
In [1]: 1 import matplotlib.pyplot as plt
2
3 o, a, b, c = (0, 0), (1, 2), (2, 3), (2, 4)
4 arrows = [[o, a, 'r', 0.1], [o, b, 'g', 0.05], [o, c, 'b', 0.05]]
5 plt.axis('scaled'), plt.xlim(0, 5), plt.ylim(0, 5)
6 for p, v, c, w in arrows:
7     plt.quiver(p[0], p[1], v[0], v[1],
8                 units='xy', scale=1, color=c, width=w)
9 plt.show()
```

How about $A = \{(1, 2, 3), (2, 3, 4), (3, 4, 5)\}$ and $B = \{(1, 2, 3), (2, 3, 1), (3, 1, 2)\}$? The linear independence of A and B are reduced, respectively, to solving the equations

$$(a) \begin{cases} x + 2y + 3z = 0 \\ 2x + 3y + 4z = 0 \\ 3x + 4y + 5z = 0 \end{cases}$$

and

$$(b) \begin{cases} x + 2y + 3z = 0 \\ 2x + 3y + z = 0 \\ 3x + y + 2z = 0. \end{cases}$$

Exercise 3.2 Solve the equations (a) and (b) by hand, and decide if they are linearly independent.

Now, let us solve them with Python.

Program: eqn6.py

```
In [1]: 1 from sympy import solve
2 from sympy.abc import x, y, z
3
4 ans1 = solve([x + 2*y + 3*z, 2*x + 3*y + 4*z, 3*x + 4*y + 5*z],
5             [x, y, z])
6 print(ans1)
7
```

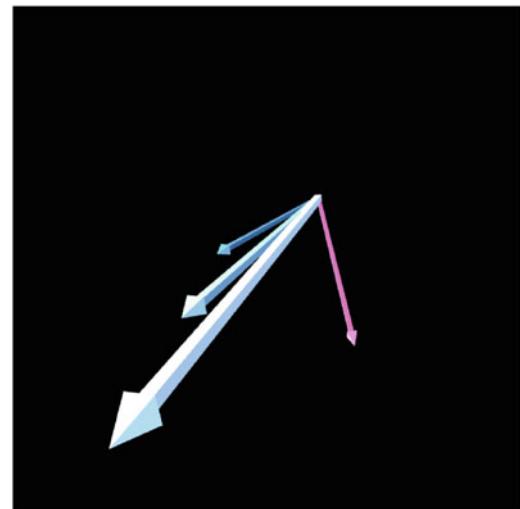
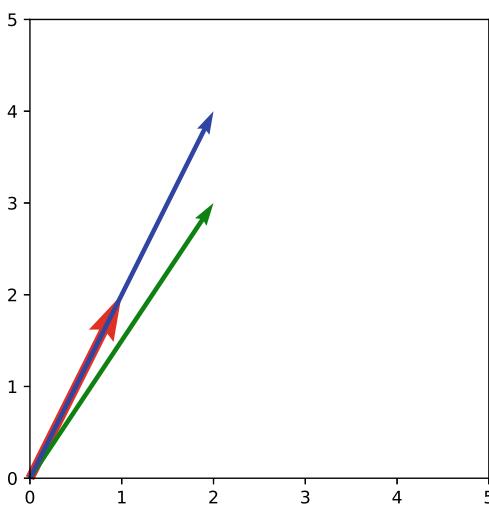


Fig. 3.2 `arrow2d.py`(left) and `arrow3d.py`(right)

```
In [1]: 8 | ans2 = solve([x + 2*y + 3*z, 2*x + 3*y + z, 3*x + y + 2*z],  
9 |           [x, y, z])  
10 | print(ans2)
```



```
{x: z, y: -2*z}  
{x: 0, z: 0, y: 0}
```

In case (a), $(x, y) = (z, -2z)$ is the solution, where z is arbitrary. For example, putting $z = 1$ we have $(x, y) = (1, -2)$, which is one of the solutions of (a). Hence, A is linearly dependent. In case (b), $(x, y, z) = (0, 0, 0)$ is the only solution and so B is linearly independent.

The following program draws four vectors $(1, 2, 3)$, $(2, 3, 4)$, $(3, 4, 5)$, and $(3, 1, 2)$ in three-dimensional space (Fig. 3.2, right). The first three vectors lie on the same plane, but the last one is not on it.

Program: `arrow3d.py`

```
In [1]: 1 | from vpython import vec, arrow, mag  
2 |  
3 | o = vec(0, 0, 0)  
4 | for p in [(1, 2, 3), (2, 3, 4), (3, 4, 5), (3, 1, 2)]:  
5 |     v = vec(*p)  
6 |     arrow(pos=o, axis=v, color=v, shaftwidth=mag(v) * 0.02)
```

Suppose that $A = \{a_1, a_2, \dots, a_n\}$ is linearly independent. If

$$x_1 a_1 + x_2 a_2 + \dots + x_n a_n = y_1 a_1 + y_2 a_2 + \dots + y_n a_n$$

for $x_1, \dots, x_n, y_1, \dots, y_n \in \mathbb{K}$, then

$$(x_1 - y_1) a_1 + (x_2 - y_2) a_2 + \dots + (x_n - y_n) a_n = \mathbf{0},$$

and hence

$$x_1 - y_1 = x_2 - y_2 = \dots = x_n - y_n = 0$$

because A is linearly independent. Thus, $(x_1, x_2, \dots, x_n) = (y_1, y_2, \dots, y_n)$. In other words, if the set of vectors is linearly independent, the coefficients of linear combinations of the vectors are unique.

Next, suppose that A is linearly dependent. Then,

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n = \mathbf{0}$$

holds for some $x_1, \dots, x_n \in \mathbb{K}$ not all zero. Because some of x_1, x_2, \dots, x_n is not 0, we may assume that $x_1 \neq 0$ without loss of generality.³ Then, we can write

$$\mathbf{a}_1 = \frac{-x_2}{x_1}\mathbf{a}_2 + \cdots + \frac{-x_n}{x_1}\mathbf{a}_n.$$

Conversely, if there is some vector among $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ which is written as a linear combination of the other vectors, we may assume that it is \mathbf{a}_1 by rearranging the vectors, that is,

$$\mathbf{a}_1 = x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n$$

for some $x_2, \dots, x_n \in \mathbb{K}$. Then because

$$(-1)\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n = \mathbf{0},$$

A is linearly dependent.

In short, linear dependence of A is equivalent to “there exists a vector in A that is a linear combination of the other vectors of A ”. The other way around, linear independence of A is equivalent to “any vector in A never belongs to the subspace generated by the other vectors but it”.

3.3 Basis and Representation

Let V be a linear space over \mathbb{K} and $X = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\} \subseteq V$. X is called a *basis* of V , if it is linearly independent and generates V .

Exercise 3.3 Let X be a basis of V . Prove the following facts:

- (1) A set obtained by adding a new vector to X or removing any vector of X is no longer a basis of V .
- (2) A set obtained by replacing any vector of X with its nonzero scalar multiple remains a basis.
- (3) A set obtained by replacing any vector of X with a sum of it and a scalar multiple of another vector of X remains a basis.

The set $\{\vec{e}_1, \vec{e}_2, \dots, \vec{e}_n\}$ given by

³The linear independence and the linear dependence do not depend on the order of arrangement of vectors, so by rearranging the vectors we may assume $x_1 \neq 0$.

$$\begin{aligned}\vec{e}_1 &= (1, 0, 0, \dots, 0) \\ \vec{e}_2 &= (0, 1, 0, \dots, 0) \\ &\vdots \\ \vec{e}_n &= (0, 0, 0, \dots, 1)\end{aligned}$$

is a basis of \mathbb{K}^n . We call this basis the *standard basis* of \mathbb{K}^n .

Suppose that V has a basis X . We serialize the vectors in X as $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$, and fix this order. Any $\mathbf{x} \in V$ can be expressed as

$$\mathbf{x} = x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \dots + x_n\mathbf{a}_n,$$

and this expression is unique as we stated in the last section. We call this expression the *expansion* of \mathbf{x} on the basis X . The vector

$$\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{K}^n$$

made of the expansion coefficients x_1, x_2, \dots, x_n is called the *representation* of \mathbf{x} on the basis X . For a vector $\vec{x} = (x_1, x_2, \dots, x_n)$ in \mathbb{K}^n , we can write

$$\vec{x} = x_1\vec{e}_1 + x_2\vec{e}_2 + \dots + x_n\vec{e}_n$$

with the standard basis $\{\vec{e}_1, \vec{e}_2, \dots, \vec{e}_n\}$ above. So, the representation of \vec{x} on the standard basis is \vec{x} itself.

Consider the mapping $f_X : \mathbf{x} \mapsto \vec{x}$ which sends a vector \mathbf{x} in V to its representation \vec{x} on X . Then, f_X is a bijective mapping from V to \mathbb{K}^n because of the uniqueness of representation. Let $\mathbf{x}, \mathbf{y} \in V$, $f_X(\mathbf{x}) = \vec{x}$ and $f_X(\mathbf{y}) = \vec{y}$, then

$$\begin{aligned}s\mathbf{x} + t\mathbf{y} &= s(x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \dots + x_n\mathbf{a}_n) + t(y_1\mathbf{a}_1 + y_2\mathbf{a}_2 + \dots + y_n\mathbf{a}_n) \\ &= (sx_1 + ty_1)\mathbf{a}_1 + (sx_2 + ty_2)\mathbf{a}_2 + \dots + (sx_n + ty_n)\mathbf{a}_n, \\ f_X(s\mathbf{x} + t\mathbf{y}) &= (sx_1 + ty_1, sx_2 + ty_2, \dots, sx_n + ty_n) \\ &= s\vec{x} + t\vec{y} = sf_X(\mathbf{x}) + tf_X(\mathbf{y}).\end{aligned}$$

Thus, f_X is a linear mapping, and hence f_X is a linear isomorphism and V is isomorphic to \mathbb{K}^n . We also call this isomorphism f_X the *representation* of V on X . Note that when $V = \mathbb{K}^n$ and X is chosen as the standard basis, f_X is just the identity mapping of \mathbb{K}^n .

Exercise 3.4 Let V and W be linear spaces and let $f : V \rightarrow W$ be a linear isomorphism. Let $X = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\} \subset V$. Prove the following facts:

- (1) If X generates V , then $f(X) = \{f(\mathbf{v}_1), f(\mathbf{v}_2), \dots, f(\mathbf{v}_k)\}$ generates W .
- (2) If X is linearly independent, then so is $f(X)$.
- (3) If X is a basis of V , then $f(X)$ is a basis of W .

Using VPython we choose two vectors \vec{a} and \vec{b} randomly⁴ in three-dimensional space. Then, $A = \{\vec{a}, \vec{b}\}$ is linearly independent with probability 1.⁵ Let us bring the data saved as a list of points on the xy -coordinate plane which are obtained by binarizing the pictures in Sect. 1.8. For a point (x, y) in the data we plot the point $\vec{w} = x\vec{a} + y\vec{b}$ in three-dimensional space.

Program: mypict4.py

```
In [1]: 1 from vpython import canvas, vec, curve, arrow, color, points
2 from numpy import array, linspace, sin, cos, pi, random
3
4 canvas(background=color.white, foreground=color.black)
5 for v in [vec(1, 0, 0), vec(0, 1, 0), vec(0, 0, 1)]:
6     curve(pos=[-v, v], color=v)
7
8 with open('mypict1.txt', 'r') as fd:
9     XY = eval(fd.read())
10
11 random.seed(123)
12 a = vec(*random.normal(0, 1, 3))
13 arrow(pos=vec(0, 0, 0), axis=a, shaftwidth=0.1)
14 b = vec(*random.normal(0, 1, 3))
15 arrow(pos=vec(0, 0, 0), axis=b, shaftwidth=0.1)
16 P = [x * a + y * b for (x, y) in XY]
17 Q = [cos(t) * a + sin(t) * b for t in linspace(0, 2 * pi, 101)]
18 points(pos=P, radius=2, color=color.cyan)
19 curve(pos=Q, color=color.magenta)
```

Line 4: Set the background of the 3D space white and the foreground black to make the results easier to see.

Lines 5,6: Draw x -axis, y -axis, and z -axis.

Lines 8,9: Read the list of points (x, y) into XY from the file mypict1.txt which was made executing mypict1.py in Chap. 1.

Line 11: Give a *seed* of random numbers. If we give the same seed (123 in this case), we will always get the same sequence of random numbers, and the results of experiments will be the same.⁶

Lines 12–15: Generate randomly two vectors \vec{a} and \vec{b} and display them as arrows.⁷

Line 16: P is the list of $x\vec{a} + y\vec{b}$ for points (x, y) in list XY .

Line 17: Q is the list of $x\vec{a} + y\vec{b}$ when point (x, y) ranges over the circle centering on the origin with radius 1 (in fact, the vertices of the regular 100-sided polygon).

Line 18: Draw the vectors in P as points.

Line 19: Draw the vectors in Q as a curve connecting points.

The execution results appear in Fig. 3.3. Repeat the experiment changing the seed of random numbers.

⁴ “Randomly” means here that we take two vectors independently according to the three-dimensional standard normal distribution (this “independence” is in the sense of probability theory and is not “linear independence”). We may think that two points are chosen arbitrarily from the points in Fig. 3.4 (right) in Sect. 3.6.

⁵ First choose \vec{a} randomly, then it is not the origin with probability 1. Next choose \vec{b} randomly, then it is not on the line passing through \vec{a} and the origin with probability 1.

⁶ If we do not call the function `seed`, the seed is taken from the internal clock of our computer. We get different results each time we run it.

⁷ Depending on the seed, one vector may look close to the zero vector or two vectors may look overlapped.

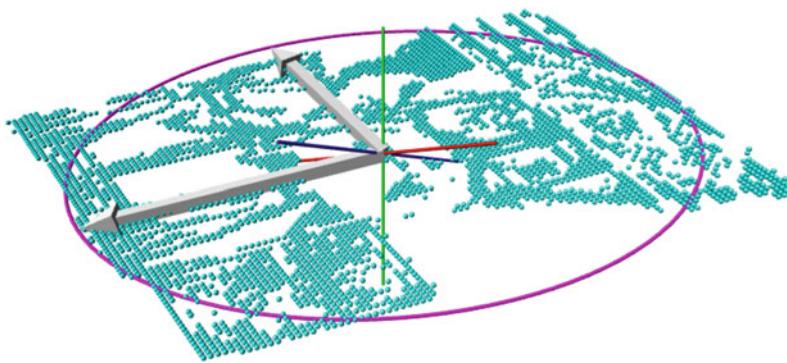


Fig. 3.3 Output

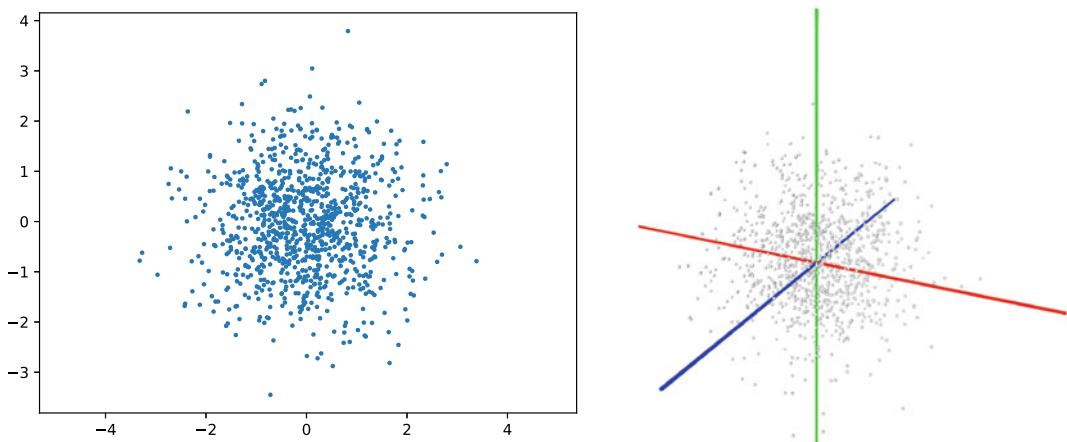


Fig. 3.4 Two-dimensional (left) and three-dimensional (right) standard normal distributions

Exercise 3.5 Let V be the linear space of all polynomials in x of degree up to 2 with real coefficients. Prove that $\{x^2 + 2x + 3, 2x^2 + 3x + 1, 3x^2 + x + 2\}$ becomes a basis of V . Moreover, represent $x^2 - 2x + 1$ as a vector in \mathbb{R}^3 on this basis.

3.4 Dimension and Rank

So far, we have proved various properties of linear spaces and linear mappings. Most of them are proved in a small number of steps from the definitions, and we did not dare call them theorems. For the first time in this section, two theorems will appear. They are so important that we call them the fundamental theorems of linear algebra.

Theorem 3.1 *If a linear space V has a basis consisting of m vectors, the size of the set of linearly independent vectors is always less than or equal to m . In particular, the size of any basis of V is equal to m .*

Proof Let $A_0 = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m\} \subseteq V$ and $B_0 = \{\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n\} \subseteq V$. Suppose that A_0 is a basis and that B_0 is linearly independent. Since A_0 generates V ,

$$\mathbf{b}_1 = x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \cdots + x_m \mathbf{a}_m \quad \dots \quad (**)$$

for some $x_1, \dots, x_m \in \mathbb{K}$. Here, some of x_1, x_2, \dots, x_m is not 0 because $\mathbf{b}_1 \neq \mathbf{0}$. We may assume $x_1 \neq 0$ by rearranging $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m$ if necessary. Replacing \mathbf{a}_1 by \mathbf{b}_1 in A_0 , we get a new set $A_1 = \{\mathbf{b}_1, \mathbf{a}_2, \dots, \mathbf{a}_m\}$, and removing \mathbf{b}_1 from B_0 , we get $B_1 = \{\mathbf{b}_2, \dots, \mathbf{b}_n\}$. Because $x_1 \neq 0$, \mathbf{a}_1 is expressed as a linear combination of A_1 by (**). Hence, A_1 generates V because so does A_0 .

Next, suppose

$$y_1 \mathbf{b}_1 + y_2 \mathbf{a}_2 + \cdots + y_m \mathbf{a}_m = \mathbf{0}$$

for $y_1, \dots, y_m \in \mathbb{K}$. By (**) we have

$$y_1 (x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \cdots + x_m \mathbf{a}_m) + y_2 \mathbf{a}_2 + \cdots + y_m \mathbf{a}_m = \mathbf{0}$$

and this is simplified as

$$(x_1 y_1) \mathbf{a}_1 + (x_2 y_1 + y_2) \mathbf{a}_2 + \cdots + (x_m y_1 + y_m) \mathbf{a}_m = \mathbf{0}.$$

Since A_0 is linearly independent, we obtain

$$x_1 y_1 = x_2 y_1 + y_2 = \cdots = x_m y_1 + y_m = 0.$$

Hence, $y_1 = 0$ because $x_1 \neq 0$, and so $y_2 = \cdots = y_m = 0$. Therefore, A_1 is linearly independent and is a basis of V .

We shall work similarly on the basis A_1 and the linearly independent set B_1 . Since A_1 generates V , we have

$$\mathbf{b}_2 = z_1 \mathbf{b}_1 + z_2 \mathbf{a}_2 + \cdots + z_m \mathbf{a}_m \quad \dots \quad (***)$$

for $z_1, \dots, z_m \in \mathbb{K}$. Because \mathbf{b}_1 and \mathbf{b}_2 are linearly independent, $z_2 = z_3 = \cdots = z_m = 0$ is impossible. Rearranging $\mathbf{a}_2, \mathbf{a}_3, \dots, \mathbf{a}_m$ if necessary, we may suppose $z_2 \neq 0$. Replacing \mathbf{a}_2 by \mathbf{b}_2 in A_1 , we get $A_2 = \{\mathbf{b}_1, \mathbf{b}_2, \mathbf{a}_3, \dots, \mathbf{a}_m\}$, and removing \mathbf{b}_2 from B_1 , we have $B_2 = \{\mathbf{b}_3, \dots, \mathbf{b}_n\}$. Because $z_2 \neq 0$, \mathbf{a}_2 is expressed as a linear combination of A_2 . Since A_1 generates V , so does A_2 .

Suppose

$$u_1 \mathbf{b}_1 + u_2 \mathbf{b}_2 + u_3 \mathbf{a}_3 + \cdots + u_m \mathbf{a}_m = \mathbf{0}$$

for $u_1, \dots, u_m \in \mathbb{K}$. By (***), we have

$$u_1 \mathbf{b}_1 + u_2 (z_1 \mathbf{b}_1 + z_2 \mathbf{a}_2 + \cdots + z_m \mathbf{a}_m) + u_3 \mathbf{a}_3 + \cdots + u_m \mathbf{a}_m = \mathbf{0},$$

which is simplified as

$$(u_1 + u_2 z_1) \mathbf{b}_1 + (u_2 z_2) \mathbf{a}_2 + (u_2 z_3 + u_3) \mathbf{a}_3 + \cdots + (u_2 z_m + u_m) \mathbf{a}_m = \mathbf{0}.$$

Since A_1 is linearly independent, we obtain

$$u_1 + u_2 z_1 = u_2 z_2 = u_2 z_3 + u_3 = \cdots = u_2 z_m + u_m = 0.$$

It follows that $u_2 = 0$ because $z_2 \neq 0$, and so $u_1 = u_3 = \cdots = u_m = 0$. Therefore A_2 is linearly independent and is a basis of V .

Repeating this argument, we obtain bases A_1, A_2, \dots of V and subsets B_1, B_2, \dots of B_0 , successively. This process must terminate in one of the following situations:

1. In the case $m < n$: $A_m = \{b_1, b_2, \dots, b_m\}$, $B_m = \{b_{m+1}, \dots, b_n\}$.
2. In the case $m = n$: $A_m = \{b_1, b_2, \dots, b_m\}$, $B_m = \{\}$.
3. In the case $m > n$: $A_n = \{b_1, b_2, \dots, b_n, a_{n+1}, \dots, a_m\}$, $B_n = \{\}$.

In case 1, because A_m generates V , b_{m+1} is written as a linear combination of A_m , but this contradicts the linear independence of B_0 . Hence, either case 2 or 3 happens, and so we find $n \leq m$. If B_0 is a basis of V , a_m is a linear combination of B_0 because B_0 generates V . Hence, case 3 cannot happen because A_n is linearly independent, consequently, $m = n$. ■

If V has a basis with m vectors, any basis of V consists of m vectors by Theorem 3.1. We call this invariant m the *dimension* of V over \mathbb{K} , and express as $m = \dim_{\mathbb{K}} V$. The real linear space \mathbb{R}^n is n -dimensional over \mathbb{R} (because it has the standard basis of size n). The complex linear space \mathbb{C}^n is n -dimensional over \mathbb{C} , while it is $2n$ -dimensional over \mathbb{R} if we consider it as a real linear space. When \mathbb{K} is fixed and there is no confusion, we write the dimension of V simply as $\dim V$. The dimension of the trivial space $\{\mathbf{0}\}$ is 0 because the empty set $\{\}$ is its basis.⁸

If V is finite dimensional then it has a finite basis. In fact, suppose that a finite set A generates V .⁹ If A is not a basis, that is, linearly dependent, some vector in A is a linear combination of the other vectors in A . The set A' obtained from A by removing it still generates V . Repeating this procedure, in the end, we obtain a linearly independent set which is a basis of V .

If V is finite dimensional, any linearly independent set is included in some basis of V . In fact, suppose that $A \subseteq V$ is linearly independent. If A does not generate V , there is a vector in V that is not written as a linear combination of vectors in A . If we add it to A , the new set remains linearly independent. We repeat this procedure, but cannot repeat it infinitely, because the size of a linearly independent set can never exceed the dimension of V . Thus, in the end, the obtained set generates V and it becomes a basis of V .

If we can find as many linearly independent vectors in V as we like, V is called an *infinite-dimensional linear space*. An infinite-dimensional space is never generated by a finite number of vectors. As some examples of infinite-dimensional linear space, we can give the space of all infinite sequences of numbers and the space of all polynomials.¹⁰

The dimension of the subspace generated by $A = \{a_1, a_2, \dots, a_n\}$ is called the *rank* of A and we denote it by $\text{rank}_{\mathbb{K}} A$ or simply $\text{rank } A$. From what we have discussed above, we can immediately see the following:

1. $\text{rank } A \leq n$.
2. If A is linearly independent, then $\text{rank } A = n$.
3. If A is linearly dependent, then $\text{rank } A < n$.

If A is a subset of an m -dimensional linear space V , then:

⁸As stated in Sect. 3.1, $\{\}$ is the smallest set generating $\{\mathbf{0}\}$.

⁹As we defined in Sect. 3.1, V is finite dimensional if it is generated by a finite number of vectors.

¹⁰See the supplementary remark on infinite dimension in Sect. 3.6.

1. $\text{rank } A \leq m$.
2. If A generates V , then $\text{rank } A = m$.
3. If A does not generate V , then $\text{rank } A < m$.

In particular, when $m = n$, A generates V if and only if A is linearly independent if and only if A is a basis of V .

Due to the above facts, in order to find whether a given set of vectors is linearly independent or generates V , it is enough to calculate its rank. We will discuss how to calculate the rank mathematically in the later chapters, but here we will introduce a calculation using NumPy. We use function `matrix_rank` defined in module `linalg`¹¹ of NumPy.

Program: `rank.py`

```
In [1]: 1 from numpy.linalg import matrix_rank
2
3 def f(*x): return matrix_rank(x)
4
5 a, b, c = (1, 2), (2, 3), (2, 4)
6 print(f(a, b), f(b, c), f(a, c), f(a, b, c))
7 a, b, c, d = (1, 2, 3), (2, 3, 4), (3, 4, 5), (3, 4, 4)
8 print(f(a, b), f(a, b, c), f(a, b, d), f(a, b, c, d))
```

Line 3: `matrix_rank` is redefined as `f` to shorten the name. At the same time we do a little trick.¹²

Lines 5–8: Define the vectors and give them to `f` to compute the rank.



```
2 2 1 2
2 2 3 3
```

For $\vec{a} = (1, 2)$, $\vec{b} = (2, 3)$, $\vec{c} = (2, 4)$, the ranks of $\{\vec{a}, \vec{b}\}$, $\{\vec{b}, \vec{c}\}$, $\{\vec{a}, \vec{c}\}$, and $\{\vec{a}, \vec{b}, \vec{c}\}$ are 2, 2, 1, and 2, respectively. Moreover, for $\vec{a} = (1, 2, 3)$, $\vec{b} = (2, 3, 4)$, $\vec{c} = (3, 4, 5)$, $\vec{d} = (3, 4, 4)$, the ranks of $\{\vec{a}, \vec{b}\}$, $\{\vec{a}, \vec{b}, \vec{c}\}$, $\{\vec{a}, \vec{b}, \vec{d}\}$, and $\{\vec{a}, \vec{b}, \vec{c}, \vec{d}\}$ are 2, 2, 3, and 3, respectively.

The following theorem concerns the dimensions of the image and the kernel of a linear mapping (cf. Fig. 2.7 and Exercises 2.15–2.16). In its proof, almost all the notions of linear space we learned appear, but it is almost straightforward without any special ideas. It may be a touchstone for how well the reader understands the story so far.

Theorem 3.2 (Dimension Theorem) *If $f : V \rightarrow W$ is a linear mapping and V is finite dimensional, then we have*

$$\dim V = \dim \text{kernel}(f) + \dim \text{range}(f).$$

Proof Let $X = \{v_1, v_2, \dots, v_n\}$ be a basis of V , then $\text{range}(f) = \langle f(X) \rangle$ is finite dimensional. Let $l = \dim \text{range}(f)$ and let $Y = \{y_1, y_2, \dots, y_l\}$ be a basis of $\text{range}(f)$. We can find $\{x_1, x_2, \dots, x_l\} \subseteq V$ such that

$$y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_l = f(x_l).$$

¹¹ In this module functions useful for various calculations on matrices are defined. We will learn them in Chap. 5.

¹² Like the built-in function `print`, the function `f` can have any numbers of arguments. For example, if we call it in the format `f(a, b, c)` for three arguments, the function `matrix_rank` is called in the format `matrix_rank([a, b, c])` for one argument of the list consisting of `a`, `b`, and `c`. If we omit `*` before the formal argument `x` in the definition of `f`, we need to call it in the format `f([a, b, c])`.

Let $k = \dim \text{kernel}(\mathbf{f})$ and let $Z = \{z_1, z_2, \dots, z_k\}$ be a basis of $\text{kernel}(\mathbf{f})$. We claim that $S = \{z_1, z_2, \dots, z_k, x_1, x_2, \dots, x_l\}$ forms a basis of V .

First, let us prove that S generates V . Let $\mathbf{v} \in V$ be arbitrary. Because $\mathbf{f}(\mathbf{v}) \in \text{range}(\mathbf{f})$ and Y generates $\text{range}(\mathbf{f})$, there exist $a_1, a_2, \dots, a_l \in \mathbb{K}$ such that

$$\mathbf{f}(\mathbf{v}) = a_1 \mathbf{y}_1 + a_2 \mathbf{y}_2 + \cdots + a_l \mathbf{y}_l.$$

Then, we have

$$\begin{aligned} & \mathbf{f}(\mathbf{v} - (a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \cdots + a_l \mathbf{x}_l)) \\ &= \mathbf{f}(\mathbf{v}) - \mathbf{f}(a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \cdots + a_l \mathbf{x}_l) \\ &= \mathbf{f}(\mathbf{v}) - (a_1 \mathbf{f}(\mathbf{x}_1) + a_2 \mathbf{f}(\mathbf{x}_2) + \cdots + a_l \mathbf{f}(\mathbf{x}_l)) \\ &= \mathbf{f}(\mathbf{v}) - (a_1 \mathbf{y}_1 + a_2 \mathbf{y}_2 + \cdots + a_l \mathbf{y}_l) = \mathbf{0}_W, \end{aligned}$$

and this implies $\mathbf{v} - (a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \cdots + a_l \mathbf{x}_l) \in \text{kernel}(\mathbf{f})$. Since Z generates $\text{kernel}(\mathbf{f})$, there exist $b_1, b_2, \dots, b_k \in \mathbb{K}$ such that

$$\mathbf{v} - (a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \cdots + a_l \mathbf{x}_l) = b_1 z_1 + b_2 z_2 + \cdots + b_k z_k,$$

or

$$\mathbf{v} = a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \cdots + a_l \mathbf{x}_l + b_1 z_1 + b_2 z_2 + \cdots + b_k z_k.$$

Hence, S generates V .

Next, we shall prove the linear independence of S . Suppose that

$$a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \cdots + a_l \mathbf{x}_l + b_1 z_1 + b_2 z_2 + \cdots + b_k z_k = \mathbf{0}_V$$

for $a_1, \dots, a_l, b_1, \dots, b_k \in \mathbb{K}$. Applying \mathbf{f} on both sides, we have

$$a_1 \mathbf{y}_1 + a_2 \mathbf{y}_2 + \cdots + a_l \mathbf{y}_l = \mathbf{0}_W,$$

because $\mathbf{f}(a_i \mathbf{x}_i) = a_i \mathbf{y}_i$ for $i = 1, \dots, l$ and $\mathbf{f}(b_j z_j) = \mathbf{0}_W$ for $j = 1, \dots, k$. By the linear independence of Y , we obtain $a_1 = a_2 = \cdots = a_l = 0$. Further, from this we have

$$b_1 z_1 + b_2 z_2 + \cdots + b_k z_k = \mathbf{0}_V,$$

and the linear independence of Z implies $b_1 = b_2 = \cdots = b_k = 0$.

Thus, S is a basis of V , and we have the desired equality $\dim V = k + l$. ■

Let \mathbf{f} be a linear mapping from a linear space V to a linear space W with the same dimension n . Then, by the benefit of this theorem we have

$$\begin{aligned}
 f \text{ is surjective} &\Leftrightarrow \text{range}(f) = W \\
 &\Leftrightarrow \dim \ker(f) = n - \dim \text{range}(f) = 0 \\
 &\Leftrightarrow f \text{ is injective.}
 \end{aligned}$$

That is, the bijectivity follows from only the condition of surjectivity or injectivity.

3.5 Direct Sums

Let W_1, W_2, \dots, W_k be subspaces of a linear space V . Define

$$W = \{\mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_k \mid \mathbf{x}_1 \in W_1, \mathbf{x}_2 \in W_2, \dots, \mathbf{x}_k \in W_k\},$$

then W is a subspace of V . We call W the *sum of subspaces* W_1, W_2, \dots, W_k and denote it by $W_1 + W_2 + \cdots + W_k$.

Exercise 3.6 Prove the following:

- (1) W above is a subspace of V .
- (2) W is equal to the subspace generated by $W_1 \cup W_2 \cup \cdots \cup W_k$.
- (3) For any i with $1 \leq i < k$,

$$W_1 + W_2 + \cdots + W_k = (W_1 + W_2 + \cdots + W_i) + (W_{i+1} + W_2 + \cdots + W_k).$$

We say that a family $\mathbb{W} = \{W_1, W_2, \dots, W_k\}$ of subspaces is *linearly independent*, if W_1, W_2, \dots, W_k are all nonzero and $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$ is linearly independent for any nonzero $\mathbf{x}_1 \in W_1, \mathbf{x}_2 \in W_2, \dots, \mathbf{x}_k \in W_k$. Because a subset of a linearly independent set is linearly independent, a subfamily of a linearly independent subspaces is linearly independent.

Suppose that \mathbb{W} is linearly independent and let $W = W_1 + W_2 + \cdots + W_k$. If $\mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_k = 0$ holds for $\mathbf{x}_1 \in W_1, \mathbf{x}_2 \in W_2, \dots, \mathbf{x}_k \in W_k$, then some of \mathbf{x}_i ($1 \leq i \leq k$) must be zero. We may assume that it is \mathbf{x}_1 , and we have $\mathbf{x}_2 + \cdots + \mathbf{x}_k = 0$. Then, again some of \mathbf{x}_i ($2 \leq i \leq k$) must be zero because the subfamily $\{W_2, \dots, W_k\}$ is linearly independent. Repeating this, we find that \mathbf{x}_i are all zero. Suppose that an element $\mathbf{x} \in W$ is written in two ways as

$$\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_k = \mathbf{y}_1 + \mathbf{y}_2 + \cdots + \mathbf{y}_k$$

with $\mathbf{x}_i, \mathbf{y}_i \in W_i$ ($i = 1, 2, \dots, k$), then

$$(\mathbf{x}_1 - \mathbf{y}_1) + (\mathbf{x}_2 - \mathbf{y}_2) + \cdots + (\mathbf{x}_k - \mathbf{y}_k) = 0.$$

It follows from the above discussion that $\mathbf{x}_i = \mathbf{y}_i$ for all i . This implies that every element \mathbf{x} of W is uniquely expressed as $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_k$ with $\mathbf{x}_1 \in W_1, \mathbf{x}_2 \in W_2, \dots, \mathbf{x}_k \in W_k$. We call W the *direct sum* of W_1, W_2, \dots, W_k (or of \mathbb{W}) and denote it by $W_1 \oplus W_2 \oplus \cdots \oplus W_k$.

Exercise 3.7 For nonzero subspaces W_1, W_2 of V , prove that the following statements are equivalent:

- (1) $\{W_1, W_2\}$ is linearly independent.
- (2) Any element x in $W_1 + W_2$ is uniquely written as $x = x_1 + x_2$ with $x_1 \in W_1$ and $x_2 \in W_2$.
- (3) $W_1 \cap W_2 = \{\mathbf{0}\}$.

Example 3.1 In three-dimensional space \mathbb{R}^3 , a one-dimensional subspace L (line passing through the origin) and a two-dimensional subspace P (plane passing through the origin) are linearly independent, if they intersect only at the origin. They are linearly dependent if L lies on P . Any two planes can never be linearly independent, because their intersection is at least one dimensional.

Let V and W be linear spaces over \mathbb{K} . We define addition and scalar multiplication on the direct product $V \times W = \{(\mathbf{v}, \mathbf{w}) \mid \mathbf{v} \in V, \mathbf{w} \in W\}$ by

$$\begin{aligned} (\mathbf{v}_1, \mathbf{w}_1) + (\mathbf{v}_2, \mathbf{w}_2) &\stackrel{\text{def}}{=} (\mathbf{v}_1 + \mathbf{v}_2, \mathbf{w}_1 + \mathbf{w}_2), \\ a(\mathbf{v}, \mathbf{w}) &\stackrel{\text{def}}{=} (a\mathbf{v}, a\mathbf{w}) \end{aligned}$$

for $a \in \mathbb{K}$, $\mathbf{v}, \mathbf{v}_1, \mathbf{v}_2 \in V$ and $\mathbf{w}, \mathbf{w}_1, \mathbf{w}_2 \in W$. Then, $V \times W$ becomes a linear space over \mathbb{K} . It is easy to see that $V' = \{(\mathbf{v}, \mathbf{0}) \mid \mathbf{v} \in V\}$ and $W' = \{(\mathbf{0}, \mathbf{w}) \mid \mathbf{w} \in W\}$ are subspaces of $V \times W$, and $\{V', W'\}$ is linearly independent and $V' \cup W'$ generates $V \times W$, and so we have $V \times W = V' \oplus W'$.

Exercise 3.8 Prove the results stated just above.

The mappings $f_1 : \mathbf{v} \mapsto (\mathbf{v}, \mathbf{0})$ and $f_2 : \mathbf{w} \mapsto (\mathbf{0}, \mathbf{w})$ are injections from V and W into $V \times W$ such that $\text{range}(f_1) = V'$ and $\text{range}(f_2) = W'$, respectively. With these embeddings we identify V' with V and W' with W and call $V \times W = V' \oplus W'$ the *direct sum* of V and W and write it as $V \oplus W$.¹³ We express an element (\mathbf{v}, \mathbf{w}) of $V \oplus W$ by $\mathbf{v} \oplus \mathbf{w}$, then the operations above are expressed as

$$\begin{aligned} (\mathbf{v}_1 \oplus \mathbf{w}_1) + (\mathbf{v}_2 \oplus \mathbf{w}_2) &= (\mathbf{v}_1 + \mathbf{v}_2) \oplus (\mathbf{w}_1 + \mathbf{w}_2), \\ a(\mathbf{v} \oplus \mathbf{w}) &= (a\mathbf{v}) \oplus (a\mathbf{w}), \end{aligned}$$

respectively.

When $V = \mathbb{K}^m$ and $W = \mathbb{K}^n$, $V \oplus W$ is linearly isomorphic to \mathbb{K}^{m+n} by the mapping

$$(v_1, v_2, \dots, v_m) \oplus (w_1, w_2, \dots, w_n) \mapsto (v_1, v_2, \dots, v_m, w_1, w_2, \dots, w_n)$$

for $(v_1, v_2, \dots, v_m) \in V$ and $(w_1, w_2, \dots, w_n) \in W$. So, the direct sum of vectors is considered to be the concatenation of vectors.

¹³ We sometimes distinguish the direct sum of subspaces as the *internal* direct sum and the direct sum of spaces as the *external* direct sum. However, they are essentially the same.

The direct sum is expressed in NumPy as follows.

```
In [1]: [1, 2] + [3, 4, 5]
Out[1]: [1, 2, 3, 4, 5]
In [2]: from numpy import array, concatenate
concatenate([array([1, 2]), array([3, 4, 5])])
Out[2]: array([1, 2, 3, 4, 5])
```

When we express vectors using Python's lists, the direct sum is given by addition. Actually, it is the concatenation of lists. When we express vectors using NumPy's array (*ndarray*), addition of two arrays is a componentwise addition, and to express the direct sum we use the `concatenate` function.

On the other hand, in SymPy the direct sum is expressed as follows.

```
In [3]: from sympy import Matrix
Matrix([1, 2]).col_join(Matrix([3, 4, 5]))
Out[3]: Matrix([
[1],
[2],
[3],
[4],
[5]])
In [4]: Matrix([[1, 2]]).row_join(Matrix([[3, 4, 5]]))
Out[4]: Matrix([[1, 2, 3, 4, 5]])
```

Use the `col_join` method for the direct sum of column vectors and `row_join` for row vectors.

3.6 Remarks on Dimension

An n -dimensional arrangement $[a_{i_1 i_2 \dots i_n}]_{i_1=1}^{k_1} {}_{i_2=1}^{k_2} \dots {}_{i_n=1}^{k_n}$ makes the element $a_{i_1 i_2 \dots i_n}$ correspond to the point (i_1, i_2, \dots, i_n) in n -dimensional space, in other words, the elements are arranged in n -dimensional space (recall Sect. 1.7). A one-dimensional arrangement is a sequence of elements arranged in a row, a two-dimensional arrangement is a matrix in which elements are arranged vertically and horizontally in a two-dimensional plane, and a three-dimensional arrangement is a layout of elements arranged vertically, horizontally, and depth-wise in three-dimensional space.

We can express n -dimensional arrangements by arrays in NumPy. In the following examples, A is a one-dimensional array (three-dimensional vectors), B is a two-dimensional array (2 by 3 matrices), and C is a three-dimensional array. We can regard B as an array consisting of two one-dimensional array (three-dimensional vectors) side by side, and C as an array consisting of two two-dimensional arrays (2 by 2 matrices) side by side. B is also regarded as a six-dimensional vector, and in the same way C is regarded as an eight-dimensional vector. When we use the word dimension, we need to be careful about what we are paying attention to.

Program: dim.py

```
In [1]: 1 | from numpy import array
2 | A = array([1, 2, 3])
```

```
In [1]: 4 | B = array([[1, 2, 3], [4, 5, 6]])
5 | C = array([[1, 2], [3, 4], [[5, 6], [7, 8]]])
6 | print(f'A={A}')
7 | print(f'B={B}')
8 | print(f'C={C}')
```



```
A=[[1 2 3]
B=[[1 2 3]
[4 5 6]]
C=[[1 2]
[3 4]
[[5 6]
[7 8]]]
```

Let us apply multidimensional arrays to see the multidimensional normal distributions. Program `random2d.py` plots 10000 points subject to the two-dimensional normal distribution on a 2-dimensional plane, and Program `random3d.py` plots 1000 points subject to the 3-dimensional normal distribution in 3-dimensional space.¹⁴

Program: `random2d.py`

```
In [1]: 1 | import matplotlib.pyplot as plt
2 | from numpy.random import normal
3 |
4 | P = normal(0, 1, (1000, 2))
5 | plt.scatter(P[:, 0], P[:, 1], s=4)
6 | plt.axis('equal'), plt.show()
```

Line 4: `P` is a two-dimensional array of shape $(1000, 2)$, and is considered as 1000 two-dimensional vectors lined up as $(x_1, y_1), (x_2, y_2), \dots, (x_{1000}, y_{1000})$.

Line 5: `plt.scatter` is a function to make a two-dimensional scatter diagram. In the same way as we used the function `plt.plot` to draw graphs before, to the first argument of this function we must give a sequence of $x_1, x_2, \dots, x_{1000}$ and to the second argument we give a sequence of $y_1, y_2, \dots, y_{1000}$. Considering `P` as a matrix of 1000 rows and 2 columns, `P[:, 0]` takes out $x_1, x_2, \dots, x_{1000}$ as an array of the first column of the matrix and `P[:, 1]` takes out $y_1, y_2, \dots, y_{1000}$ as an array of the second column of the matrix. This is a specific usage of *slice*.¹⁵

Program: `random3d.py`

```
In [1]: 1 | from vpython import canvas, color, vec, curve, points
2 | from numpy.random import normal
3 |
4 | canvas(background=color.white, foreground=color.black)
5 | for v in [vec(5, 0, 0), vec(0, 5, 0), vec(0, 0, 5)]:
6 |     curve(pos=[-v, v], color=v)
7 | P = normal(0, 1, (1000, 3))
8 | points(pos=[vec(*p) for p in P], radius=4)
```

Line 7: `P` is a matrix of shape $(1000, 3)$ and considered to be a sequence of 1000 three-dimensional vectors.

¹⁴ The image drawn by VPython is actually projected on the computer screen which is a two-dimensional plane. We feel that it is drawn in three-dimensional space as we change the viewpoint with the mouse. It is one of the important themes after Chap. 6 of this book to project a space with large dimension, not only 3D, into a smaller space (plane, straight line).

¹⁵ For objects of a class whose elements can be specified by subscript such as lists, tuples, and strings, slice is a mechanism referring to their elements.

Finally, let us make a supplementary remark about the infinite dimension. An infinite sequence

$$x_1, x_2, \dots, x_n, \dots$$

is written as $\{x_n\}_{n=1}^{\infty}$. We define a structure of linear space on the infinite sequences as follows:

(Vector sum) $\{x_n\}_{n=1}^{\infty} + \{y_n\}_{n=1}^{\infty} \stackrel{\text{def}}{=} \{x_n + y_n\}_{n=1}^{\infty}$.

(Scalar multiple) $a \{x_n\}_{n=1}^{\infty} \stackrel{\text{def}}{=} \{ax_n\}_{n=1}^{\infty}$.

(Zero vector) the sequence $0, 0, \dots, 0, \dots$

(Inverse vector) $-\{x_n\}_{n=1}^{\infty} \stackrel{\text{def}}{=} \{-x_n\}_{n=1}^{\infty}$.

Consider an infinite number of figures

$$\begin{aligned} & 1, 0, 0, \dots, 0, \dots \\ & 0, 1, 0, \dots, 0, \dots \\ & 0, 0, 1, \dots, 0, \dots \\ & \vdots \quad \vdots \quad \vdots \quad \ddots \quad \vdots \\ & 0, 0, 0, \dots, 1, \dots \\ & \vdots \quad \vdots \quad \vdots \quad \ddots \quad \vdots \end{aligned}$$

arranged two dimensionally (infinite matrix). Let e_1 be the infinite sequence in the first row of the above matrix, e_2 the infinite sequence in the second row, e_3 the infinite sequence in the third row, and so on. That is, e_n is the infinite sequence in which only the n -th term is 1 and all others are 0. It is easy to see that e_n cannot be represented by a linear combination of $\{e_1, e_2, \dots, e_{n-1}\}$. Hence, $\{e_1, e_2, \dots, e_n\}$ are linearly independent for any $n \in \mathbb{N}$. Therefore, the space of all infinite sequences is an infinite-dimensional linear space.

You might think that any infinite sequence $\{x_n\}_{n=1}^{\infty}$ is written as $x_1e_1 + x_2e_2 + x_3e_3 + \dots + x_ne_n + \dots = \sum_{n=1}^{\infty} x_n e_n$; however, the infinite summation of vectors is not defined in a linear space. In order to deal with such infinite sums, we need to clarify what the limit of vectors is.

The set of all polynomials in x is also an infinite-dimensional linear space, because for any $n \in \mathbb{N}$, $\{1, x, x^2, x^3, \dots, x^n\}$ is linearly independent. The Taylor expansion is an infinite sum of polynomials, but it does not add an infinite number of terms at once. We must also consider it as a limit.



Matrices

4

In this chapter, we will learn the matrix representation of a linear mapping, and matrix operations defined naturally through representation. The main themes in linear algebra are “choosing a suitable basis for the matrix representation to become simple” and “finding properties of linear mappings that do not depend on basis choice”.

We introduce two libraries, now standard in Python, NumPy for numerical computation and SymPy for symbolic computation. We need to use them selectively in accordance with our problems.

4.1 Matrix Operations

We consider an $m \times n$ matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

which is a two-dimensional arrangement of elements of \mathbb{K} . Taking out the columns a_1, a_2, \dots, a_n from it in order, we write it as $A = [a_1 \ a_2 \ \cdots \ a_n]$. The entry a_{ij} at the intersection of the i -th row and the j -th column is called the (i, j) -element of A . In particular, a matrix of shape (n, n) with the same number of rows and columns is called a *square matrix* of *order n*, and its (i, i) -elements a_{ii} ($i = 1, 2, \dots, n$) are called *diagonal elements*. A square matrix is called a *diagonal matrix*, if all its elements other than diagonal elements are zero. Two matrices A and B are equal if and only if they are of the same shape and the corresponding elements are equal.

For matrix calculation, we use `array` of NumPy in Python as standard.

```
In [1]: from numpy import array  
A = [[1, 2, 3], [4, 5, 6]]; A
```

```
Out[1]: [[1, 2, 3], [4, 5, 6]]
```

Matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ is expressed by a nested list. Note that a list expresses only an arrangement of elements of a matrix, so in order to use it for various matrix calculations described later in this chapter, we need to implement them by ourselves.

```
In [2]: A = array(A); A
```

```
Out[2]: array([[1, 2, 3],
 [4, 5, 6]])
```

Cast¹ list A to an array and let this array be A again. Without this assignment, A remains a list.

```
In [3]: print(A)
```



```
[[1 2 3]
 [4 5 6]]
```

The function `print` outputs the matrix.

```
In [4]: L = A.tolist(); L
```

```
Out[4]: [[1, 2, 3], [4, 5, 6]]
```

Make the list from the array using method `tolist`.

```
In [5]: B = A.copy(); B
```

```
Out[5]: array([[1, 2, 3],
 [4, 5, 6]])
```

Copying A, make another array B with the same contents. On the other hand, `B = A` means that A and B point to the same object, and the change of the elements of A is also reflected in B.² This happens because an array is a mutable object.

```
In [6]: A == B
```

```
Out[6]: array([[ True,  True,  True],
 [ True,  True,  True]])
```

```
In [7]: (A == B).all()
```

```
Out[7]: True
```

```
In [8]: B[0, 1] = 1
 (A == B).all()
```

```
Out[8]: False
```

`A == B` compares arrays and returns the array of Boolean values of componentwise comparison. The method `all` decides the equality of matrices. This method returns `True` if all Boolean values of comparison are `True`, whereas the method `any` returns `True` if at least one value is `True`.

Exercise 4.1 Observe what happens if we replace `A.tolist` in In [4] with `list(A)` in the above dialog. Moreover, observe how the results change if we replace `B = A.copy()` in In [5] with `B = A`.

¹ See Sect. 1.4 about cast.

² So to speak, this is the difference between pointing at a matrix written at some place in a notebook and rewriting the same matrix with a pencil in another place.

When we write a vector $\vec{x} \in \mathbb{K}^n$ in the column form $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, it is a matrix of shape $(n, 1)$ and

is called a *column vector*. On the other hand, a matrix $[x_1 \ x_2 \ \cdots \ x_n]$ of shape $(1, n)$ is called a *row vector*. They are treated as follows in NumPy.

```
In [1]: from numpy import array
A = array([1, 2, 3]); A
```

```
Out[1]: array([1, 2, 3])
```

```
In [2]: B = A.reshape((1, 3)); B
```

```
Out[2]: array([[1, 2, 3]])
```

```
In [3]: C = B.reshape((3, 1)); C
```

```
Out[3]: array([[1],
 [2],
 [3]])
```

```
In [4]: D = C.reshape((3,)); D
```

```
Out[4]: array([1, 2, 3])
```

```
In [5]: A[0] = 0; A
```

```
Out[5]: array([0, 2, 3])
```

```
In [6]: B[0, 0], C[0, 0], D[0]
```

```
Out[6]: (0, 0, 0)
```

Vector $(1, 2, 3)$ is expressed as in Out[1]. On the other hand, row vector $[1 \ 2 \ 3]$ is expressed as in Out[2], and column vector $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ is expressed as in Out[3] (or as `array([[1], [2], [3]])`). These can be converted to each other by method `reshape` of `array` as in In[2], In[3], and In[4]. However, it should be noted that when converted by method `reshape`, the corresponding element in each expression points to the same thing.³ For example, as seen in Out[5] and Out[6], if some element in A is changed, the corresponding elements in B, C, and D are affected. If we want to avoid such effects, we need to make B, C, and D as copies of the reshaped arrays.

The set $M_{\mathbb{K}}(m, n)$ of all matrices of the same shape (m, n) whose elements belong to \mathbb{K} forms a linear space over \mathbb{K} . For two matrices $A = [a_1 \ a_2 \ \cdots \ a_n]$ and $B = [b_1 \ b_2 \ \cdots \ b_n]$ of shape (m, n) and for $c \in \mathbb{K}$, define

³The method `reshape` just changes the way of reference to the same memory by subscripts of the array.

$$\begin{aligned} \mathbf{A} + \mathbf{B} &\stackrel{\text{def}}{=} [\mathbf{a}_1 + \mathbf{b}_1 \ \mathbf{a}_2 + \mathbf{b}_2 \cdots \mathbf{a}_n + \mathbf{b}_n] \\ c\mathbf{A} &\stackrel{\text{def}}{=} [c\mathbf{a}_1 \ c\mathbf{a}_2 \cdots c\mathbf{a}_n]. \end{aligned}$$

That is, the vector sum is defined by summing each pair of corresponding elements, and the scalar multiple is defined by multiplying each element by the same scalar. The matrix playing the role of the zero vector is one all whose elements are 0. We call this matrix the *zero matrix*. The matrix for the inverse vector is obtained by inverting the signs of all elements.⁴

In NumPy, the sum and the scalar multiple of matrices are expressed in the usual way. Define A and B as arrays. If we let A and B be lists, the calculations later would not work.

```
In [1]: from numpy import array
A = array([[1, 2, 3], [4, 5, 6]])
B = array([[1, 3, 5], [2, 4, 6]])
A + B
```

```
Out[1]: array([[ 2,  5,  8],
   [ 6,  9, 12]])
```

Calculate the matrix sum.

```
In [2]: 2 * A
```

```
Out[2]: array([[ 2,  4,  6],
   [ 8, 10, 12]])
```

Calculate the scalar multiple. We can calculate $\mathbf{A} * 2$ and $\mathbf{A} / 2$ too. We can also calculate $\mathbf{A} * \mathbf{B}$,⁵ but it is different from the matrix product which we will consider in Sect. 4.3.

```
In [3]: 0 * A
```

```
Out[3]: array([[0, 0, 0],
   [0, 0, 0]])
```

```
In [4]: -1 * A
```

```
Out[4]: array([[-1, -2, -3],
   [-4, -5, -6]])
```

Multiplying A by 0 gives the zero matrix. Multiplying A by -1 gives the matrix corresponding to the inverse vector.

Exercise 4.2 The following program outputs a calculation problem of matrices in a form of a code in math mode of L^AT_EX.⁶ Typesetting the obtained TeX code, we obtain Fig. 4.1. Create another problem by changing the seed 2021 and solve it.

⁴ We do not call this matrix the inverse matrix. This name will be used later for another meaning. Recall that for a number x the inverse of x is $1/x$ instead of $-x$.

⁵ Similar to the matrix sum, this is a componentwise product. This product is called the *Schur product* or *Hadamard product*. There are various products of matrices other than this.

⁶ The usage of L^AT_EX is explained in Sect. A.6.

Fig. 4.1 L^AT_EX output image with no margins using the [template.tex](#) in Appendix

$$\begin{bmatrix} -2 & -3 & 3 \\ 4 & 4 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 4 & 1 \\ 3 & 3 & 3 \end{bmatrix} =$$

Program: latex1.py

```
In [1]: 1 from numpy.random import seed, randint, choice
2 from sympy import Matrix, latex
3
4 seed(2021)
5 m, n = randint(2, 4, 2)
6 X = [-3, -2, -1, 1, 2, 3, 4, 5]
7 A = Matrix(choice(X, (m, n)))
8 B = Matrix(choice(X, (m, n)))
9 print(f'{latex(A)} + {latex(B)} = ')
```

```
\left[\begin{matrix}-2 & -3 & 3\end{matrix}\right] + \left[\begin{matrix}5 & 4 & 1\\3 & 3 & 3\end{matrix}\right] =
```

Here is the answer.

```
In [2]: A+B
```

```
Out[2]: Matrix([
 [3, 1, 4],
 [7, 7, 5]])
```

4.2 Matrices and Linear Mappings

For a given linear mapping $f : V \rightarrow W$, it is impossible to thoroughly investigate $y \in W$ satisfying $y = f(x)$ for all $x \in V$.⁷ It is convenient if we can express the relation $y = f(x)$ by a formula in x like $y = ax + b$ in the case of the usual numbers. Let us consider such a method.

Let V and W be linear spaces over \mathbb{K} of dimensions n and m and let $X = \{v_1, v_2, \dots, v_n\}$ and $Y = \{w_1, w_2, \dots, w_m\}$ be their bases, respectively. Because $f(v_1), f(v_2), \dots, f(v_n)$ are vectors in W and Y is a basis of W , we can write

$$\begin{aligned}f(v_1) &= a_{11}w_1 + a_{21}w_2 + \cdots + a_{m1}w_m \\f(v_2) &= a_{12}w_1 + a_{22}w_2 + \cdots + a_{m2}w_m \\&\vdots \\f(v_n) &= a_{1n}w_1 + a_{2n}w_2 + \cdots + a_{mn}w_m\end{aligned}$$

with $a_{ij} \in \mathbb{K}$. Here,

⁷Even if V is finite dimensional, it has an infinite number of vectors.

$$\mathbf{a}_1 \stackrel{\text{def}}{=} \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}, \quad \mathbf{a}_2 \stackrel{\text{def}}{=} \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}, \quad \dots, \quad \mathbf{a}_n \stackrel{\text{def}}{=} \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

are the representations of $f(\mathbf{v}_1), f(\mathbf{v}_2), \dots, f(\mathbf{v}_n)$ on basis Y , respectively. For $\mathbf{x} \in V$ let $\mathbf{y} = f(\mathbf{x})$. Let $\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{K}^n$ be the representation of \mathbf{x} on basis X and $\vec{y} = (y_1, y_2, \dots, y_m) \in \mathbb{K}^m$ be the representation of \mathbf{y} on basis Y , that is,

$$\mathbf{x} = x_1 \mathbf{v}_1 + x_2 \mathbf{v}_2 + \dots + x_n \mathbf{v}_n, \quad \mathbf{y} = y_1 \mathbf{w}_1 + y_2 \mathbf{w}_2 + \dots + y_m \mathbf{w}_m.$$

Then, we have

$$\begin{aligned} y_1 \mathbf{w}_1 + y_2 \mathbf{w}_2 + \dots + y_m \mathbf{w}_m &= f(x_1 \mathbf{v}_1 + x_2 \mathbf{v}_2 + \dots + x_n \mathbf{v}_n) \\ &= x_1 f(\mathbf{v}_1) + x_2 f(\mathbf{v}_2) + \dots + x_n f(\mathbf{v}_n) \\ &= (a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n) \mathbf{w}_1 \\ &\quad + (a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n) \mathbf{w}_2 \\ &\quad \vdots \\ &\quad + (a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n) \mathbf{w}_m. \end{aligned}$$

By the linear independence of Y we get

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix}.$$

Therefore, by defining the matrix

$$\mathbf{A} \stackrel{\text{def}}{=} [\mathbf{a}_1 \ \mathbf{a}_2 \ \dots \ \mathbf{a}_n]$$

and the product⁸

$$\mathbf{A}\vec{x} \stackrel{\text{def}}{=} x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \dots + x_n \mathbf{a}_n,$$

of matrix \mathbf{A} and column vector \vec{x} , we can express the above equation as

$$\vec{y} = \mathbf{A}\vec{x}.$$

We call this equation the *matrix representation* of $\mathbf{y} = f(\mathbf{x})$ (or of linear mapping f). Moreover, we call matrix \mathbf{A} in this equation the matrix representation (or *representation matrix*) of f .

In summary, we can calculate $\mathbf{y} = f(\mathbf{x})$ for given $\mathbf{x} \in V$ as follows:

1. Give bases $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ and $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_m\}$ of V and W , respectively.
2. Get the representation \mathbf{a}_j of $f(\mathbf{v}_j)$ on basis $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_m\}$ for each $j = 1, 2, \dots, n$.

⁸This is actually the matrix product which will be defined in the next section.

3. Get the representation $\vec{x} \in \mathbb{K}^n$ of $x \in V$ on basis $\{v_1, v_2, \dots, v_n\}$.
4. Letting $A = [\mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n]$, calculate $\vec{y} = A\vec{x}$.
5. We have $y = y_1 w_1 + y_2 w_2 + \cdots + y_m w_m$, where $\vec{y} = (y_1, y_2, \dots, y_m)$.

We consider the standard bases of $V = \mathbb{K}^n$ and $W = \mathbb{K}^m$. Let $f : V \rightarrow W$ be a linear mapping and let A be its representation matrix. Let $x \in V$ and let $y = f(x)$. Then the representations of x and y are the same x and y , respectively, as stated in Sect. 3.3. Hence, $y = Ax$ holds. Conversely, for any (m, n) -matrix A , the mapping $f : V \rightarrow W$ defined by $f(x) \stackrel{\text{def}}{=} Ax$ ($x \in V$) is a linear mapping, and the representation matrix of f is A itself. In this way, matrices of shape (m, n) have a one-to-one correspondence with linear mappings from V to W each other. Moreover, because

$$\begin{aligned}(aA + bB)x &= [aa_1 + bb_1 \ aa_2 + bb_2 \ \cdots \ aa_n + bb_n]x \\&= x_1(aa_1 + bb_1) + x_2(aa_2 + bb_2) + \cdots + x_n(aa_n + bb_n) \\&= a(x_1a_1 + x_2a_2 + \cdots + x_na_n) + b(x_1b_1 + x_2b_2 + \cdots + x_nb_n) \\&= a([\mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n]x) + b([\mathbf{b}_1 \mathbf{b}_2 \cdots \mathbf{b}_n]x) \\&= a(Ax) + b(Bx),\end{aligned}$$

the linear combination of matrices just corresponds to the linear combination of the corresponding linear mappings. This means that the mapping sending a linear mapping f to its representation matrix A is a linear isomorphism from the linear space $L(\mathbb{K}^n, \mathbb{K}^m)$ of linear mappings from \mathbb{K}^n to \mathbb{K}^m to the linear space $M_{\mathbb{K}}(m, n)$ of matrices of shape (m, n) .

Exercise 4.3 Let $V = \mathbb{K}^n$ and $W = \mathbb{K}^m$, and let $f : V \rightarrow W$ be a linear mapping and $A = [\mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n]$ be the representation matrix of f on the standard bases. Prove the following statements:

- (1) f is surjective if and only if $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ generates W .
- (2) f is injective if and only if $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ is linearly independent.
- (3) f is bijective if and only if $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ is a basis of W .

Exercise 4.4 Generate 1000 vectors subject to the two-dimensional standard normal distribution and multiply them by matrix (1) or (2), and display the vectors as points on a plane. Do a similar experiment using matrices (3) or (4) for the three-dimensional standard normal distribution.

$$(1) \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \quad (2) \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \quad (3) \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \quad (4) \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

Let V be the linear space consisting of all polynomials in x with real coefficients of degree at most 3 and W be the linear space consisting of all polynomials of degree at most 2. Consider the basis $\{1, x, x^2, x^3\}$ of V and the basis $\{1, x, x^2\}$ of W . The representation of the polynomial $a_3x^3 + a_2x^2 + a_1x + a_0 \in V$ is $(a_0, a_1, a_2, a_3) \in \mathbb{R}^4$. The differentiation $\frac{d}{dx}$ is a linear mapping from V to W . Because this linear mapping sends the basis of V as

$$\begin{cases} 1 & \mapsto 0 + 0x + 0x^2, \\ x & \mapsto 1 + 0x + 0x^2, \\ x^2 & \mapsto 0 + 2x + 0x^2, \\ x^3 & \mapsto 0 + 0x + 3x^2, \end{cases}$$

the matrix representation of the differentiation

$$b_2x^2 + b_1x + b_0 = \frac{d}{dx}(a_3x^3 + a_2x^2 + a_1x + a_0)$$

is given by

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

Exercise 4.5 Let V be the linear space consisting of all polynomials in x of degree at most 4 and W be the linear space consisting of all polynomials of degree at most 2. Find the matrix representation of the second-order differentiation $\frac{d^2}{dx^2}$, which is a linear mapping from V to W , on the bases $\{1, x, x^2, x^3, x^4\}$ and $\{1, x, x^2\}$ of V and W , respectively.

In NumPy we have the function `dot` or method `dot` of array for the operation corresponding to the product of a matrix and a vector.⁹ Let us learn how to use them.

```
In [1]: from numpy import array, dot
y = dot([[1, 2], [3, 4]], [5, 6]); y
```

```
Out[1]: array([17, 39])
```

We do not need to import the function `dot` in using the `dot` method of array. In general, if we import the name of a class, we can use all methods of the class. Moreover, even if we import `dot` only without importing `array`, we can execute `dot` and `y` can be defined as an array. The `dot` function computes the product of matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and vector $(5, 6)$. The actual argument can be an array or a list, and the return value is the vector $(17, 39)$ expressed by an array.

```
In [2]: A = array([[1, 2], [3, 4]])
A.dot([5, 6])
```

```
Out[2]: array([17, 39])
```

Defining `A` as an array, the `dot` method executes the same computation.

```
In [3]: A.dot([[5], [6]])
```

```
Out[3]: array([[17],
               [39]])
```

⁹The `dot` function is used in the format `dot(A, B)` and the `dot` method is used in the format `A.dot(B)`.

The dot method computes the product of the matrix and the column vector and return column vector $\begin{bmatrix} 17 \\ 39 \end{bmatrix}$ expressed by an array. The dot function will give the same result.

The mapping f that rotates the plane \mathbb{R}^2 by θ degrees counterclockwise around the origin is a linear mapping. Let us describe this linear mapping using the procedure given above. We express vectors of \mathbb{R}^2 as column vectors from the beginning.

1. We choose the standard basis in \mathbb{R}^2 .
2. We have $f\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix}$, $f\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} -\sin \theta \\ \cos \theta \end{bmatrix}$.
3. We have the representation matrix $A = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$ of f .
4. Compute $y = Ax$ with $x = \begin{bmatrix} x \\ y \end{bmatrix}$.
5. We get $y = \begin{bmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{bmatrix}$.

Therefore, the point (x, y) is mapped to the point $(x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta)$. The matrix A obtained here is called the *rotation matrix* around the origin of angle θ in \mathbb{R}^2 setting the counterclockwise direction as the positive direction.

The following program rotates an image on a plane using the rotation matrix. The file mypict1.txt made in Sect. 1.8 contains a list of arrays representing two-dimensional vectors written in the style of Python code.

Program: mypict5.py

```
In [1]: 1 from numpy import array, pi, sin, cos
2 import matplotlib.pyplot as plt
3
4 t = pi / 4
5 A = array([[cos(t), -sin(t)], [sin(t), cos(t)]])
6
7 with open('mypict1.txt', 'r') as fd:
8     P = eval(fd.read())
9
10 Q = [A.dot(p) for p in P]
11 x, y = zip(*Q)
12 plt.scatter(x, y, s=1)
13 plt.axis('equal'), plt.show()
```

Line 5: A is the rotation matrix of angle t.

Lines 7,8: P is the list read from mypict1.txt.

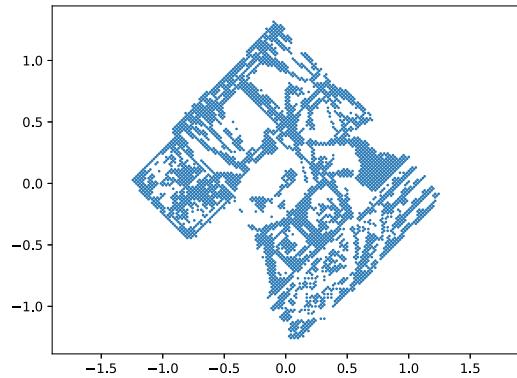
Line 10: Q is the list of points obtained by rotating the points in P.

Lines 11,12: Plot the points in Q on the coordinate plane.

Line 13: Figure 4.2 is obtained.

Exercise 4.6 Find the representation matrix of each of the following linear mappings on the standard bases:

Fig. 4.2 Execution result
of mypict5.py



- (1) Linear mapping $(x, y) \mapsto (x + y, x, y)$ from \mathbb{K}^2 to \mathbb{K}^3 .
- (2) Linear mapping $(x, y, z) \mapsto (x + y, y + z, x + z)$ from \mathbb{K}^3 to \mathbb{K}^3 .

4.3 Composition of Linear Mappings and Product of Matrices

Let \mathbf{A} and \mathbf{B} be matrices of shape (l, m) and of shape (m, n) , respectively, given by

$$\mathbf{A} = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_m], \quad \mathbf{B} = [\mathbf{b}_1 \ \mathbf{b}_2 \ \cdots \ \mathbf{b}_n].$$

Let \mathbf{g} be the linear mapping sending $\mathbf{x} \in \mathbb{K}^n$ to $\mathbf{Bx} = \mathbf{y} \in \mathbb{K}^m$ and \mathbf{f} be the linear mapping sending \mathbf{y} to $\mathbf{Ay} = \mathbf{z} \in \mathbb{K}^l$. The matrix representation of composite mapping $\mathbf{f} \circ \mathbf{g} : \mathbf{x} \mapsto \mathbf{z}$ is given by

$$\begin{aligned} \mathbf{z} &= \mathbf{A}(\mathbf{Bx}) \\ &= \mathbf{A}(x_1\mathbf{b}_1 + x_2\mathbf{b}_2 + \cdots + x_n\mathbf{b}_n) \\ &= x_1\mathbf{Ab}_1 + x_2\mathbf{Ab}_2 + \cdots + x_n\mathbf{Ab}_n \\ &= [\mathbf{Ab}_1 \ \mathbf{Ab}_2 \ \cdots \ \mathbf{Ab}_n]\mathbf{x} \end{aligned}$$

on the standard bases of \mathbb{K}^n and \mathbb{K}^m . Here, we define \mathbf{AB} by

$$\mathbf{AB} \stackrel{\text{def}}{=} [\mathbf{Ab}_1 \ \mathbf{Ab}_2 \ \cdots \ \mathbf{Ab}_n],$$

and we call this the *matrix product* of \mathbf{A} and \mathbf{B} . Then, we have

$$(\mathbf{AB})\mathbf{x} = \mathbf{A}(\mathbf{Bx}).$$

Hence, \mathbf{AB} is the representation matrix of the composition of the linear mappings whose representation matrices are \mathbf{A} and \mathbf{B} , in other words, regarding matrices as linear mappings the product of matrices is just the composition of mappings.

The (i, j) -element of \mathbf{AB} is, by definition, equal to

$$a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{im}b_{mj},$$

which is the sum of the products multiplying $a_{i1}, a_{i2}, \dots, a_{im}$ in the i -th row of \mathbf{A} by $b_{1j}, b_{2j}, \dots, b_{mj}$ in the j -th column of \mathbf{B} in order.¹⁰ The matrix product \mathbf{AB} is *definable* only when the number of columns of \mathbf{A} is equal to the number of rows of \mathbf{B} . If \mathbf{A} is of shape (l, m) and \mathbf{B} is of shape (m, n) , then \mathbf{AB} is of shape (l, n) .

The matrix product can be calculated in Numpy by using the `dot` function, `dot` method of array, or `matrix` class.

```
In [1]: 1 import numpy as np
2 A = [[1, 2, 3], [4, 5, 6]]
3 B = [[1, 2], [3, 4], [5, 6]]
4 np.dot(A, B)
```

```
Out[1]: array([[22, 28],
   [49, 64]])
```

```
In [2]: np.array(A).dot(B)
```

```
Out[2]: array([[22, 28],
   [49, 64]])
```

```
In [3]: np.matrix(A) * np.matrix(B)
```

```
Out[3]: matrix([[22, 28],
   [49, 64]])
```

Set $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, $\mathbf{B} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ and calculate \mathbf{AB} using the `dot` function in In[1]. Calculate \mathbf{AB} using the `dot` method in In[2]. In In[3] calculate \mathbf{AB} using the `matrix` class. The binary operation `*` for the product is available.

Exercise 4.7 Select two, allowing duplicates, from the matrices below, and compute the product of them if it is definable:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

The following program outputs all the solutions.

Program: problems.py

```
In [1]: 1 from numpy import array
2
3 A = array([[1, 2], [3, 4]])
4 B = array([[1, 2, 3], [4, 5, 6]])
5 C = array([[1, 2], [3, 4], [5, 6]])
6 D = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
7
8 for X in (A, B, C, D):
9     for Y in (A, B, C, D):
10         if X.shape[1] == Y.shape[0]:
11             print(f'{X}\n{Y}\n= {X.dot(Y)}\n')
```

¹⁰ Remember as “row \times column”, that is, “multiply the row of \mathbf{A} with the column of \mathbf{B} componentwise”.

Lines 8-11: For X and Y taken from tuple (A, B, C, D), if the product of the matrices is definable, it is calculated and displayed.

NumPy changes fractions into decimal and does not allow symbolic expressions. To avoid this inconvenience, we use SymPy.

```
In [1]: from sympy import Matrix
from sympy.abc import a, b, c, d
A = Matrix([[1, 2], [3, 4]])
A/2
```

```
Out[1]: Matrix([
[1/2, 1],
[3/2, 2]])
```

Use `Matrix` class for matrices in Python and import names used for symbolic expressions or for constants. Define matrix A whose elements are integers. The result of `A/2` is given with fractions. If we write

```
In [1]: 1 | A = Matrix([[1., 2.], [3., 4.]])
```

the elements of the matrix are regarded as real numbers and subsequent results will change.

Symbolic computations can be done as given below.

```
In [2]: B = Matrix([[a, b], [c, d]])
B/2
```

```
Out[2]: Matrix([
[a/2, b/2],
[c/2, d/2]])
```

```
In [3]: A + B
```

```
Out[3]: Matrix([
[a + 1, b + 2],
[c + 3, d + 4]])
```

```
In [4]: A * B
```

```
Out[4]: Matrix([
[a + 2*c, b + 2*d],
[3*a + 4*c, 3*b + 4*d]])
```

If we use integers for elements of `Matrix`, SymPy does computation of fractions, but if we write solely `2/3`, it does not become a fraction.

```
In [1]: from sympy import Integer, Rational
2 / 3
```

```
Out[1]: 0.6666666666666666
```

```
In [2]: Integer(2) / 3
```

Out[2]: 2 / 3

In [3]: Rational(2, 3)

Out[3]: 2 / 3

Because the rotation of angle $\alpha + \beta$ around the origin on \mathbb{R}^2 is a composition of the rotation of angle β and the rotation of angle α , we have

$$\begin{bmatrix} \cos(\alpha + \beta) & -\sin(\alpha + \beta) \\ \sin(\alpha + \beta) & \cos(\alpha + \beta) \end{bmatrix} = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} \cos \beta & -\sin \beta \\ \sin \beta & \cos \beta \end{bmatrix}$$

$$= \begin{bmatrix} \cos \alpha \cos \beta - \sin \alpha \sin \beta & -\cos \alpha \sin \beta - \sin \alpha \cos \beta \\ \sin \alpha \cos \beta + \cos \alpha \sin \beta & -\sin \alpha \sin \beta + \cos \alpha \cos \beta \end{bmatrix}.$$

Hence, the addition theorem of trigonometric functions is obtained from this matrix computation.

The zero matrix of shape (m, n) is denoted by O_{mn} , but is written simply as O if there is no fear of confusion. The zero matrix O_{mn} is the representation matrix of the zero linear mapping from an n -dimensional linear space V to an m -dimensional linear space W . This does not depend on the choice of bases of V and W . The diagonal matrix whose diagonal elements are all 1 is called the *unit matrix*. The unit matrix of order n is denoted by I_n , and is written as I if there is no confusion. The unit matrix I_n is the representation matrix of the identity mapping id_V on any basis of V .

NumPy and SymPy express the zero matrix and the unit matrix as follows:

In [1]: import numpy as np
np.zeros((2, 3))Out[1]: array([[0., 0., 0.],
[0., 0., 0.]])In [2]: import sympy as sp
sp.zeros(2, 3)Out[2]: Matrix([
[0, 0, 0],
[0, 0, 0]])

In [3]: np.eye(3)

Out[3]: array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]])

In [4]: sp.eye(3)

Out[4]: Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]]))

In the above example, all elements of the arrays of NumPy are real.

The following equalities from 1 to 6 on the operations among matrices hold, when all the matrix operations are definable:

1. $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$,
2. $(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C})$,
3. $\mathbf{A} + \mathbf{O} = \mathbf{O} + \mathbf{A} = \mathbf{A}$,
4. $\mathbf{AI} = \mathbf{A}$, $\mathbf{IA} = \mathbf{A}$,
5. $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$,
6. $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$, $(\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$.

The proofs of 1–4 can be reduced to the property of scalars if we focus on the elements of the matrices. We can prove 5 and 6 in a similar way, but it will take time though it is not difficult. The proof by regarding matrices as linear mappings is more elegant; using the fact that the product of matrices is the composition of linear mappings, we have

$$((\mathbf{AB})\mathbf{C})\mathbf{x} = (\mathbf{AB})(\mathbf{Cx}) = \mathbf{A}(\mathbf{B}(\mathbf{Cx})) = \mathbf{A}((\mathbf{BC})\mathbf{x}) = (\mathbf{A}(\mathbf{BC}))\mathbf{x}$$

for any vector \mathbf{x} . This implies that the equality in 5 holds. Similarly, the equalities in 6 come from the above and the fact that a sum of matrices is a sum of linear mappings. In fact,

$$\begin{aligned} (\mathbf{A}(\mathbf{B} + \mathbf{C}))\mathbf{x} &= \mathbf{A}((\mathbf{B} + \mathbf{C})\mathbf{x}) = \mathbf{A}(\mathbf{Bx} + \mathbf{Cx}) \\ &= \mathbf{A}(\mathbf{Bx}) + \mathbf{A}(\mathbf{Cx}) = (\mathbf{AB})\mathbf{x} + (\mathbf{AC})\mathbf{x} = (\mathbf{AB} + \mathbf{AC})\mathbf{x} \end{aligned}$$

holds for any vector \mathbf{x} . The second equation is similarly obtained.

Let l be a straight line passing through the origin with angle θ to the x -axis on the xy -coordinate plane. Let us find the matrix representation of the linear map that sends a point (x, y) to the symmetric point (x', y') with respect to l .

1. Find the rotation matrix that lays the line l over the x -axis.
2. Find the representation matrix of the linear map that sends a point to the symmetric point with respect to the x -axis.
3. Calculate the rotation matrix that lays the x -axis over the line l .
4. Calculate the product of the three matrices obtained above.

In addition, we will find the matrix that sends (x, y) to the point (x'', y'') on l with the shortest distance from (x, y) .¹¹ Here, we use the relation $(x'', y'') = \frac{(x, y) + (x', y')}{2}$.

Program: mat_product1.py

```
In [1]: 1 | from sympy import Matrix, sin, cos, eye
2 | from sympy.abc import theta
3 |
4 | A = Matrix([[cos(theta), sin(theta)],
5 |             [-sin(theta), cos(theta)]])
6 | B = Matrix([[1, 0],
7 |             [0, -1]])
8 | C = Matrix([[cos(theta), -sin(theta)],
```

¹¹ (x'', y'') is called the *orthogonal projection* (or *orthographic projection*) of (x, y) onto l . In Chap. 6, we will learn different methods to find the orthogonal projections and their matrix representations in general dimension.

```
In [1]: 9 |         [sin(theta), cos(theta)]])
10 | D = C * B * A
11 | E = (eye(2)+D) / 2
12 | print(f'D = {D}')
13 | print(f'E = {E}')
```

Line 2: `sympy.abc` also defines lowercase Greek letters as symbols. `theta` is the symbol for the rotation angle θ .

Lines 4, 5: `A` is the rotation matrix that lays l over the x -axis.

Lines 6, 7: `B` is the matrix that transforms the plane symmetrically with respect to the x -axis.

Lines 8, 9: `C` is the rotation matrix that lays the x -axis over l .

Line 10: `D = C * B * A12` is the matrix that transforms the points symmetrically with respect to l .

Line 11: Set $E = (I + D)/2$, where I is the unit matrix. `E` satisfies

$$E \begin{bmatrix} x \\ y \end{bmatrix} = \frac{1}{2} \left(I \begin{bmatrix} x \\ y \end{bmatrix} + D \begin{bmatrix} x \\ y \end{bmatrix} \right) = \frac{1}{2} \left(\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x' \\ y' \end{bmatrix} \right) = \begin{bmatrix} x'' \\ y'' \end{bmatrix}.$$

Lines 12–13: `D` and `E` are raw results of the matrix product.



```
D = Matrix([[-sin(theta)**2 + cos(theta)**2, 2*sin(theta)*cos(theta)], [2*
sin(theta)*cos(theta), sin(theta)**2 - cos(theta)**2]])
E = Matrix([[ -sin(theta)**2/2 + cos(theta)**2/2 + 1/2, sin(theta)*cos(theta)
], [sin(theta)*cos(theta), sin(theta)**2/2 - cos(theta)**2/2 + 1/2]])
```

The addition theorem seems to make the expression simpler. Let us use `simplify` which is the method for simplifying formulas. This is a destructive method that rewrites the expressions themselves.

```
In [2]: D.simplify(); D
```

```
Out[2]: Matrix([
[cos(2*theta), sin(2*theta)],
[sin(2*theta), -cos(2*theta)])]
```

```
In [3]: E.simplify(); E
```

```
Out[3]: Matrix([
[cos(theta)**2, sin(2*theta)/2],
[sin(2*theta)/2, sin(theta)**2]])
```

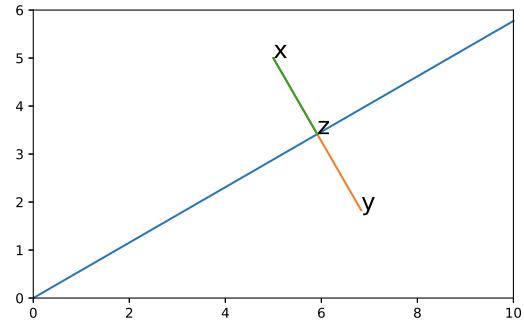
Using NumPy we compute the same matrices numerically and plot them on the xy -coordinate plane. We print the letters in the graph as well (Fig. 4.3). The matrices are expressed using `matrix`.

Program: mat_product2.py

```
In [1]: 1 | from numpy import matrix, sin, cos, tan, pi, eye
2 | import matplotlib.pyplot as plt
3 |
4 | t = pi / 6
5 | A = matrix([[cos(t), sin(t)], [-sin(t), cos(t)]])
6 | B = matrix([[1, 0], [0, -1]])
7 | C = matrix([[cos(t), -sin(t)], [sin(t), cos(t)]])
8 | D = C * B * A
9 | E = (eye(2)+D) / 2
10 | x = matrix([[5], [5]])
11 | y = D * x
12 | z = E * x
13 |
```

¹² Be careful about the order of making products of matrices.

Fig. 4.3 Execution result using NumPy and Matplotlib



```
In [1]: 14 plt.plot([0, 10], [0, 10 * tan(t)])
15 plt.plot([x[0, 0], y[0, 0]], [x[1, 0], y[1, 0]])
16 plt.plot([x[0, 0], z[0, 0]], [x[1, 0], z[1, 0]])
17 plt.text(x[0, 0], x[1, 0], 'x', fontsize=18)
18 plt.text(y[0, 0], y[1, 0], 'y', fontsize=18)
19 plt.text(z[0, 0], z[1, 0], 'z', fontsize=18)
20 plt.axis('scaled'), plt.xlim(0, 10), plt.ylim(0, 6), plt.show()
```

Lines 10–12: x is defined as a column vector (matrix of shape $(2, 1)$).

Lines 17–19: Print letters in the figure by specifying the x - and y -coordinates of the position (the default position is the lower left corner of the character), the character, and the font size.

The following program makes calculation problems of matrix multiplication and output them with the array environment¹³ of LATEX. It takes care not to make the problems too complicated or too easy.

Program: latex2.py

```
In [1]: 1 from numpy.random import seed, choice
2 from sympy import Matrix, latex
3
4 seed(2021)
5 template = r'''
6 \begin{array}{ll}
7 (1) & \text{\%s\%} = \\[0.5cm]
8 (2) & \text{\%s\%} = \\[0.5cm]
9 (3) & \text{\%s\%} = \\[0.5cm]
10 (4) & \text{\%s\%} = \\[0.5cm]
11 (5) & \text{\%s\%} =
12 \end{array}
13 '''
14
15 matrices= ()
16 for no in range(5):
17     m, e1, n = choice([2, 3], 3)
18     X = [-3, -2, -1, 1, 2, 3, 4, 5]
19     A = Matrix(choice(X, (m, e1)))
20     B = Matrix(choice(X, (e1, n)))
21     matrices += (latex(A), latex(B))
22 print(template % matrices)
```

Lines 5–13: In a character string enclosed in *triple quotation marks* (*quotes* in terminology of Python), line breaks are reflected as they are. Here, the code for LATEX array environment is written. The backslash \ is an escape sequence and followed by a character it means a special letter in Python code, but in a string with r before quotation marks, it is treated as an ordinary letter. The format operator % embeds the LATEX code of the 10 matrices made by the for loop into the %s part on Line 22.

¹³ An environment that outputs multiple expressions arranged in a given form.

Line 15: The tuple `matrices` stores the string made by typesetting the generated matrix in L^AT_EX math mode.

Lines 17, 18: Choose m, l, n from 2 or 3 randomly to make matrices of shape (m, l) and (l, n) , so that the problems are not complicated. The elements of matrices are selected from integers from -3 through 5 excluding 0 .

Lines 19, 20: Generate randomly two matrices satisfying the condition.

Line 21: The matrices in `matrices` and the two newly generated matrices are combined into new matrices. This is the same as writing `matrices = matrices + (latex(A), latex(B))`.¹⁴

Line 22: Embed matrices into template.

```
\begin{array}{ll}
(1) & \left(\begin{matrix} -3 & 3 & 4 \\ 4 & 2 & 5 \end{matrix}\right) \left(\begin{matrix} 4 & 1 & 3 \\ 3 & 3 & -3 \\ 2 & 4 & 4 \end{matrix}\right) = \left(\begin{matrix} 0.5 \\ 0.5cm \end{matrix}\right) \\
(2) & \left(\begin{matrix} 3 & 5 \\ -2 & 4 \end{matrix}\right) \left(\begin{matrix} -2 & 2 & 3 \\ -1 & -1 & -3 \end{matrix}\right) = \left(\begin{matrix} 0.5cm \end{matrix}\right) \\
(3) & \left(\begin{matrix} 1 & 2 & 3 \\ -3 & 4 & 5 \\ 2 & 3 & 4 \end{matrix}\right) \left(\begin{matrix} 5 & 3 & 1 \\ 1 & 2 & 5 \\ 3 & 4 & -2 \end{matrix}\right) = \left(\begin{matrix} 0.5cm \end{matrix}\right) \\
(4) & \left(\begin{matrix} 2 & 3 & 1 \\ -1 & 1 & -2 \\ -2 & -1 & -1 \end{matrix}\right) \left(\begin{matrix} 4 & 1 & 3 \\ 3 & 3 & -3 \\ 2 & 4 & 4 \end{matrix}\right) = \left(\begin{matrix} 0.5cm \end{matrix}\right) \\
(5) & \left(\begin{matrix} 5 & 3 & 1 \\ 1 & 2 & 5 \\ 3 & 4 & -2 \end{matrix}\right) \left(\begin{matrix} 2 & 3 & 1 \\ -1 & 1 & -2 \\ -2 & -1 & -1 \end{matrix}\right) = \left(\begin{matrix} 0.5cm \end{matrix}\right)
\end{array}
```

Exercise 4.8 Typeset the L^AT_EX code output by the program `latex2.py` to obtain five calculation problems (Fig. 4.4). Solve the problems.

Fig. 4.4 L^AT_EX output image with no margins using the `template.tex` in Appendix

$(1) \begin{bmatrix} -3 & 3 & 4 \\ 4 & 2 & 5 \end{bmatrix} \begin{bmatrix} 4 & 1 & 3 \\ 3 & 3 & -3 \\ 2 & 4 & 4 \end{bmatrix} =$
$(2) \begin{bmatrix} 3 & 5 \\ -2 & 4 \end{bmatrix} \begin{bmatrix} -2 & 2 & 3 \\ -1 & -1 & -3 \end{bmatrix} =$
$(3) \begin{bmatrix} -3 & -1 & 4 \\ 1 & 2 & 3 \\ -3 & 3 & -2 \end{bmatrix} \begin{bmatrix} 4 & -1 & 4 \\ 5 & 3 & 4 \\ -2 & 3 & 4 \end{bmatrix} =$
$(4) \begin{bmatrix} 2 & 3 \\ -1 & 1 \\ -2 & -1 \end{bmatrix} \begin{bmatrix} -1 & 5 \\ -3 & -1 \end{bmatrix} =$
$(5) \begin{bmatrix} 1 & -2 & 4 \\ -2 & -2 & -1 \end{bmatrix} \begin{bmatrix} 5 & 3 & 1 \\ 1 & 2 & 5 \\ 3 & 4 & -2 \end{bmatrix} =$

¹⁴ We can connect tuples with the operator `+` in the same way as lists. This does not mean the change of matrices, but the connected tuple is newly set as matrices.

4.4 Inverse Matrix, Basis Change, and Similarity of Matrices

Let $A = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n]$ be a square matrix of order n . A is said to be a *regular matrix* if it is bijective as a linear mapping. As stated at the end of Sect. 3.4, A is bijective if and only if it is injective. Hence, by Exercise 4.3, A is regular if and only if $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ is linearly independent.

Exercise 4.9 For the following matrices, see if they are regular matrices by checking the linear independence of their column vectors:

$$(1) \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \quad (2) \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \quad (3) \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \quad (4) \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

Let A be a regular matrix and let $f : \mathbb{K}^n \rightarrow \mathbb{K}^n$ be the linear mapping whose representation matrix is A , that is, $f(\mathbf{x}) = A\mathbf{x}$. Then, there exists the inverse mapping $f^{-1} : \mathbb{K}^n \rightarrow \mathbb{K}^n$. Because $f^{-1} \circ f$ and $f \circ f^{-1}$ are both equal to the identity mapping on \mathbb{K}^n ,

$$AA^{-1} = A^{-1}A = I$$

holds, where A^{-1} denotes the representation matrix of f^{-1} . We call A^{-1} the *inverse matrix* of A . In particular, a diagonal matrix is regular when all its diagonal elements are nonzero, and its inverse matrix is obtained by inverting all its diagonal elements.

For a square matrix A , if there exists a matrix B satisfying $AB = I$ or $BA = I$, A is a regular matrix and $B = A^{-1}$ holds. In fact, suppose $AB = I$. Regarding matrices as linear mappings, A is surjective because AB is surjective. Hence, A is bijective as we discussed above. Thus, A is a regular matrix and has its inverse A^{-1} . Multiplying both sides of $AB = I$ by A^{-1} from the left yields $B = A^{-1}AB = A^{-1}I = A^{-1}$. Similarly, we can deduce the same conclusion from $BA = I$.

For any regular matrices A and B we can deduce the following equalities from the properties of mapping discussed in Sect. 1.5:

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

Note that $AB = I$ may hold even for non-square matrices A and B . For example, let us find a, b, c, d, e, f satisfying

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Program: mat_product3.py

```
In [1]: 1 from sympy import Matrix, solve, eye
2 from sympy.abc import a, b, c, d, e, f
3
4 A = Matrix([[1, 2, 3], [2, 3, 4]])
5 B = Matrix([[a, b], [c, d], [e, f]])
6 ans = solve(A*B - eye(2), [a, b, c, d, e, f])
7 print(ans)
```



```
{a: e - 3, c: 2 - 2*e, b: f + 2, d: -2*f - 1}
```

The result solved in Line 6 is given as above. Actually, we can check $\mathbf{AB} = \mathbf{I}$ by substituting the result into \mathbf{B} .

```
In [2]: C = B.subs(ans); C
```

```
Out[2]: Matrix([
[e - 3, f + 2],
[2 - 2*e, -2*f - 1],
[e, f]])
```

```
In [3]: A * C
```

```
Out[3]: Matrix([
[1, 0],
[0, 1]]))
```

Exercise 4.10 Is it possible to make $\mathbf{BA} = \mathbf{I}$ hold for the same \mathbf{A} and \mathbf{B} as above?

In NumPy, the `linalg` module has the function `inv` for calculating the inverse matrix. Also, using `matrix` class, we can use the (-1) -th power and the squaring like mathematical expressions.

```
In [1]: A = [[1, 2], [2, 1]]
from numpy.linalg import inv
inv(A)
```

```
Out[1]: array([[-0.33333333,  0.66666667],
   [ 0.66666667, -0.33333333]])
```

```
In [2]: from numpy import matrix
matrix(A)**(-1)
```

```
Out[2]: matrix([[-0.33333333,  0.66666667],
   [ 0.66666667, -0.33333333]])
```

```
In [3]: matrix(A)**2
```

```
Out[3]: matrix([[5, 4],
   [4, 5]])
```

The following is a calculation example of the inverse matrix and the square in SymPy. We can use function `S` to define symbols.

```
In [1]: from sympy import Matrix, S
Matrix([[1, 2], [2, 1]]) ** (-1)
```

```
Out[1]: Matrix([
[-1/3, 2/3],
[2/3, -1/3]])
```

```
In [2]: A = Matrix([[S('a'), S('b')], [S('c'), S('d')]])
A**(-1)
```

```
Out[2]: Matrix([
 [ d/(a*d - b*c), -b/(a*d - b*c) ],
 [-c/(a*d - b*c), a/(a*d - b*c) ]])
```

```
In [3]: A**2
```

```
Out[3]: Matrix([
 [a**2 + b*c, a*b + b*d],
 [a*c + c*d, b*c + d**2]])
```

Let $\{v_1, v_2, \dots, v_n\}$ be a basis of \mathbb{K}^n . Let $x = (x_1, x_2, \dots, x_n) \in \mathbb{K}^n$ and let $x' = (x'_1, x'_2, \dots, x'_n)$ be the representation of x on the above basis, that is,

$$x = x'_1 v_1 + x'_2 v_2 + \cdots + x'_n v_n.$$

Set $V = [v_1 \ v_2 \ \cdots \ v_n]$, then because the right-hand side of the above equation is expressed as Vx' , we can write $x = Vx'$. Multiplying both sides by V^{-1} from the left, we have

$$x' = V^{-1}x.$$

Let $f : \mathbb{K}^n \rightarrow \mathbb{K}^n$ be a linear mapping and let A be its representation matrix on the standard basis. Suppose $y = f(x)$, that is, $y = Ax$. Let y' be the representation of y on $\{v_1, v_2, \dots, v_n\}$, then $y' = V^{-1}y$ and we have

$$y' = V^{-1}y = V^{-1}Ax = V^{-1}AVV^{-1}x = Bx',$$

where $B = V^{-1}AV$. Two square matrices A and B are said to be *similar* if there is a regular matrix V such that

$$B = V^{-1}AV.$$

We call the matrix V the *basis change matrix*. By changing basis from the standard basis to $\{v_1, v_2, \dots, v_n\}$, the representation of a vector x becomes $V^{-1}x$ and the representation matrix of a linear map f changes from A to $V^{-1}AV$, which is similar to A .

For a square matrix A , we define the *matrix power* inductively by

$$A^0 \stackrel{\text{def}}{=} I, \quad A^{p+1} \stackrel{\text{def}}{=} AA^p \quad (p = 0, 1, 2, \dots).$$

In particular, the p -th power of a diagonal matrix is obtained by raising each diagonal element to the p -th power.

Exercise 4.11 For regular matrices A and B prove the following:

- (1) $(A^p)^{-1} = (A^{-1})^p$ holds for a nonnegative integer p .
- (2) Define $A^{-p} \stackrel{\text{def}}{=} (A^p)^{-1}$ for positive integer p . Then, the *exponent law* $A^p A^q = A^{p+q}$ holds for any integers p and q .
- (3) If A and B are similar, so are A^p and B^p for any integer p .

4.5 Adjoint Matrix

For a matrix

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

of shape (m, n) , we define matrices of shape (n, m) as

$$\mathbf{A}^T \stackrel{\text{def}}{=} \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{A}^* \stackrel{\text{def}}{=} \begin{bmatrix} \overline{a_{11}} & \overline{a_{21}} & \cdots & \overline{a_{m1}} \\ \overline{a_{12}} & \overline{a_{22}} & \cdots & \overline{a_{m2}} \\ \vdots & \vdots & & \vdots \\ \overline{a_{1n}} & \overline{a_{2n}} & \cdots & \overline{a_{mn}} \end{bmatrix}.$$

They are called the *transposed matrix* and the *adjoint matrix* of \mathbf{A} , respectively.¹⁵ By easy calculations we can show that the following equalities hold; if the matrix operations in the left-hand side are definable, the matrix operations in the right-hand side are also definable and the equalities hold:

1. $(a\mathbf{A} + b\mathbf{B})^T = a\mathbf{A}^T + b\mathbf{B}^T$, $(a\mathbf{A} + b\mathbf{B})^* = \bar{a}\mathbf{A}^* + \bar{b}\mathbf{B}^*$.
2. $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$, $(\mathbf{AB})^* = \mathbf{B}^*\mathbf{A}^*$.

When \mathbf{A} is a regular matrix, by putting $\mathbf{B} = \mathbf{A}^{-1}$ in 2, we obtain the following:

3. $(\mathbf{A}^{-1})^T = (\mathbf{A}^T)^{-1}$, $(\mathbf{A}^{-1})^* = (\mathbf{A}^*)^{-1}$.

Exercise 4.12 Prove equalities 1–3.

Below are calculation examples of the transposed matrix and the adjoint matrix in Python.

```
In [1]: from numpy import *
A = array([[1 + 2j, 2 + 3j, 3 + 4j],
           [2 + 3j, 3 + 4j, 4 + 5j]])
A.T
```

```
Out[1]: array([[1.+2.j, 2.+3.j],
               [2.+3.j, 3.+4.j],
               [3.+4.j, 4.+5.j]])
```

```
In [2]: A.conj()
```

```
Out[2]: array([[1.-2.j, 2.-3.j, 3.-4.j],
               [2.-3.j, 3.-4.j, 4.-5.j]])
```

```
In [3]: A = matrix(A); A
```

```
Out[3]: matrix([[1.+2.j, 2.+3.j, 3.+4.j],
                [2.+3.j, 3.+4.j, 4.+5.j]])
```

¹⁵ \mathbf{A}^T and \mathbf{A}^* coincide for a real matrix \mathbf{A} .

```
In [4]: A.H
```

```
Out[4]: matrix([[1.-2.j, 2.-3.j],
 [2.-3.j, 3.-4.j],
 [3.-4.j, 4.-5.j]])
```

Other than `A.T` we can use `A.transpose()` for the transposed matrix too. Both `A.conj()` and `A.conjugate()` give the complex conjugate. To obtain the adjoint matrix for the array, we transpose it and then take its conjugate or vice versa. If we use `matrix`, `A.H` (`H` is the initial of Hermite) gives the adjoint matrix.

In Sympy the transpose matrix and adjoint matrix are calculated as follows:

```
In [1]: from sympy import Matrix
A = Matrix([[1 + 2j, 2 + 3j, 3 + 4j],
 [2 + 3j, 3 + 4j, 4 + 5j]]); A
```

```
Out[1]: Matrix([
 [1.0 + 2.0*I, 2.0 + 3.0*I, 3.0 + 4.0*I],
 [2.0 + 3.0*I, 3.0 + 4.0*I, 4.0 + 5.0*I]])
```

In SymPy, `I` is a symbol indicating the imaginary unit.¹⁶ In `Matrix` (`M` is capital) of SymPy, we can write the transposed matrix as `A.T` and the adjoint matrix as `A.H`.

```
In [2]: A.T
```

```
Out[2]: Matrix([
 [1.0 + 2.0*I, 2.0 + 3.0*I],
 [2.0 + 3.0*I, 3.0 + 4.0*I],
 [3.0 + 4.0*I, 4.0 + 5.0*I]])
```

```
In [3]: A.C
```

```
Out[3]: Matrix([
 [1.0 - 2.0*I, 2.0 - 3.0*I, 3.0 - 4.0*I],
 [2.0 - 3.0*I, 3.0 - 4.0*I, 4.0 - 5.0*I]])
```

```
In [4]: A.H
```

```
Out[4]: Matrix([
 [1.0 - 2.0*I, 2.0 - 3.0*I],
 [2.0 - 3.0*I, 3.0 - 4.0*I],
 [3.0 - 4.0*I, 4.0 - 5.0*I]]))
```

4.6 Measuring Matrix Computation Time

Let us define a function computing the product of matrices expressed by lists and measure the computation time.

¹⁶We need to import `I` from SymPy to use in Python code. We can use `1j` in default Python, but its type is different from `I`.

Program: mat_product4.py

```
In [1]: 1 def matrix_multiply(A, B):
2     m, el, n = len(A), len(A[0]), len(B[0])
3     C = [[sum([A[i][k] * B[k][j] for k in range(el)])
4           for j in range(n)] for i in range(m)]
5     return C
6
7 if __name__ == '__main__':
8     from numpy.random import normal
9     import matplotlib.pyplot as plt
10    from time import time
11
12    N = range(10, 210, 10)
13    T = []
14    for n in N:
15        A = normal(0, 1, (n, n)).tolist()
16        t0 = time()
17        matrix_multiply(A, A)
18        t1 = time()
19        print(n, end=', ')
20        T.append(t1 - t0)
21    plt.plot(N, T), plt.show()
```

Lines 1–5: Define a function `matrix_multiply` computing the product of matrices. Lines 3 and 4 make list C expressing the matrix whose (i, j) -element is $\sum_{k=1}^l a_{ik} b_{kj}$ using the intentional definition of a list. The multiplication is done n^3 times in the computation of $A[i][k] * B[k][j]$ inside C for square matrices A and B of order n . So, we can estimate that the execution of this function takes time approximately proportional to n^3 .¹⁷ On the other hand, the memory required for the calculation is mostly occupied by C and the size is proportional to n^2 . We say that the *time complexity* is n^3 and the *space complexity* is n^2 in computing the product of square matrices of order n .

Lines 7–21: Use the clock function `time` imported from the internal library `time` to measure the time to calculate the square of a square matrix A of order n whose elements are subject to the standard normal distribution for each $n = 10, 20, \dots, 200$ using Python. The difference between the times measured immediately before and after calling `matrix_multiply` is the time (in seconds) required for the calculation. The result is shown in the graph in Fig. 4.5 (left). It takes more computation time as n becomes larger. This curve looks a cubic function as mentioned above.

For the matrix computation NumPy calls a function written in C language and compiled in machine language, so it executes the computation at a much greater speed compared to the matrix calculation written in Python code which is an interpreter. Here, we measure the time $f(n)$ required to make a square matrix A of order n generated randomly in the same way as above, the time $g(n)$ required to calculate the square of A and the time $h(n)$ required to find the inverse matrix¹⁸ for $n = 100, 200, \dots, 2000$. The results are shown in Fig. 4.5 (right). It is expected that $f(n)$ is a quadratic function, and $g(n)$ and $h(n)$ are cubic functions.

Program: mat_product5.py

```
In [1]: 1 from numpy.random import normal
2 from numpy.linalg import inv
3 import matplotlib.pyplot as plt
4 from time import time
5
```

¹⁷ A computer takes much more time for multiplication than addition, so we ignore the computation time for addition.

¹⁸ There is the inverse matrix with probability 1.

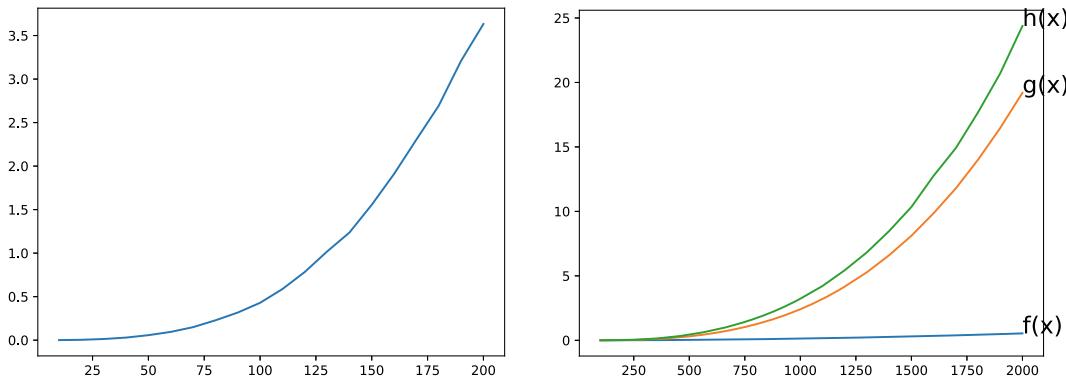


Fig. 4.5 Computation time using list (left) and array (right)

```
In [1]: 6 N = range(100, 2100, 100)
7 T = [[], [], []]
8
9 for n in N:
10    t0 = time()
11    A = normal(0, 1, (n, n))
12    t1 = time()
13    A.dot(A)
14    t2 = time()
15    inv(A)
16    t3 = time()
17    print(n, end=', ')
18    t = (t0, t1, t2, t3)
19    for i in range(3):
20        T[i].append(t[i + 1] - t[i])
21
22 label = ['f(x)', 'g(x)', 'h(x)']
23 for i in range(3):
24     plt.plot(N, T[i])
25     plt.text(N[-1], T[i][-1], label[i], fontsize=18)
26 plt.show()
```

Exercise 4.13 Draw the graphs of $\frac{g(n)}{f(n)}$ and $\frac{h(n)}{g(n)}$, and observe them. It is expected that the former is a linear function and the latter is a constant function. Is that true? If necessary, try to calculate up to $n = 3000$.



Elementary Operations and Matrix Invariants

5

In the previous chapter, we learned the matrix representation A of a linear mapping $f : V \rightarrow W$. The appearance of A changes depending on the choice of bases of V and W that cause the matrix representation of f . Some quantities of a matrix can be calculated only from the elements in it. A quantity that does not change even if we change the basis and the shape of A changes is called an *invariant*. The rank is an invariant in this sense. For linear mappings from V to itself, matrices representing the same linear mapping on different bases of V are said to be similar, as stated in Sect. 4.4. In this chapter, we study quantities that are invariant under the similarity of matrices, that is, common quantities among similar matrices. The determinant and the trace introduced in this chapter are invariants in this sense. We learn the elementary operation which gives a way to find the rank and the determinant of a matrix. It is also useful for solving simultaneous linear equations and for calculating inverse matrices.

Calculations with paper and pencil using elementary operations take time and it is easy to make mistakes. It is better to get help from Python. We give some programs that generate exercises and their answers. Let us try to solve the exercises also by hand calculation.

5.1 Elementary Matrices and Operations

Let I be the unit matrix of order n , and i and j be different positive integers less than or equal to n . The matrix obtained from I by replacing the (i, j) -element 0 by $c \in \mathbb{K}$ is denoted by $E_1^{(i,j,c)}$, that is,

$$E_1^{(i,j,c)} \stackrel{\text{def}}{=} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix},$$

where the first of the right-hand side is for the case $i < j$ and the second is for the case $i > j$. The matrix obtained from I by swapping the i -th row and j -th row of I is denoted by $E_2^{(i,j)}$, and the matrix obtained by replacing the (i, i) -element 1 by $c (\neq 0) \in \mathbb{K}$ is denoted by $E_3^{(i,c)}$, that is,

$$\mathbf{E}_2^{(i,j)} \stackrel{\text{def}}{=} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & & \ddots & & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}, \quad \mathbf{E}_3^{(i,c)} \stackrel{\text{def}}{=} \begin{bmatrix} 1 & \cdots & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & \cdots & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ \cdots & & \cdots & \cdots & 1 & \cdots & \cdots \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & \cdots & \cdots & 1 \end{bmatrix}.$$

We call these three types the *elementary matrices*. In particular, $\mathbf{E}_1^{(i,j,0)}$ and $\mathbf{E}_3^{(i,1)}$ coincide with the unit matrix \mathbf{I} . The elementary matrices are regular and have their inverse matrices. In fact,

$$(\mathbf{E}_1^{(i,j,c)})^{-1} = \mathbf{E}_1^{(i,j,-c)}, \quad (\mathbf{E}_2^{(i,j)})^{-1} = \mathbf{E}_2^{(i,j)}, \quad (\mathbf{E}_3^{(i,c)})^{-1} = \mathbf{E}_3^{(i,1/c)}.$$

Thus, the inverses of elementary matrices are also elementary.

Exercise 5.1 Confirm the above results by calculating the products $\mathbf{E}_1^{(i,j,c)} \mathbf{E}_1^{(i,j,-c)}$, $\mathbf{E}_2^{(i,j)} \mathbf{E}_2^{(i,j)}$, and $\mathbf{E}_3^{(i,c)} \mathbf{E}_3^{(i,1/c)}$.

Exercise 5.2 Check what happens in \mathbb{R}^2 if we apply elementary matrices as linear mappings to vector (x, y) . Also, check how matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ changes if we multiply it by elementary matrices from the left or right.

In order to see the role of elementary matrices in \mathbb{R}^3 , consider the unit cube with 8 vertices $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 0)$, $(0, 1, 1)$, $(1, 0, 0)$, $(1, 0, 1)$, $(1, 1, 0)$, and $(1, 1, 1)$. Let us see how this cube is transformed by the following elementary matrices:

$$\mathbf{E}_1^{(1,2,2)} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{E}_2^{(1,2)} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{E}_3^{(1,2)} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Program: elementary_vp.py

```
In [1]: 1 from vpython import *
2 import numpy as np
3
4 o = vec(0, 0, 0)
5 x, y, z = vec(1, 0, 0), vec(0, 1, 0), vec(0, 0, 1)
6 yz, zx, xy = [o, y, z, y+z], [o, z, x, z+x], [o, x, y, x+y]
7
8 def T(A, u): return vec(*np.dot(A, (u.x, u.y, u.z)))
9
10 E1 = [[1, 2, 0], [0, 1, 0], [0, 0, 1]]
11 E2 = [[0, 1, 0], [1, 0, 0], [0, 0, 1]]
12 E3 = [[2, 0, 0], [0, 1, 0], [0, 0, 1]]
13
14 def draw(E, filename):
15     scene = canvas(width=600, height=600)
16     scene.camera.pos = vec(3, 4, 5)
17     scene.camera.axis = -vec(3, 4, 5)
18     box(pos=(x+y+z)/2)
19     for axis in [x, y, z]:
```

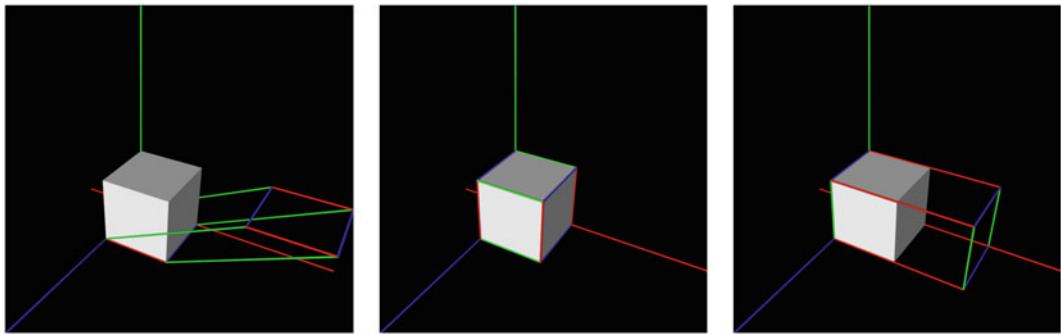


Fig. 5.1 Changes of unit cube by E1, E2, and E3

```
In [1]: 20 |         curve(pos=[-axis, 3*axis], color=axis)
21 |     for axis, face in [(x, yz), (y, zx), (z, xy)]:
22 |         for side in face:
23 |             A = E
24 |             curve(pos=[T(A, side), T(A, axis+side)], color=axis)
25 |         scene.capture(filename)
26 |
27 | draw(E1, 'E1')
```

Line 4: Let \mathbf{o} denote the zero vector.

Line 5: $\{x, y, z\}$ is the standard basis.

Line 6: yz , zx , and xy are the lists of vertices of three faces of the unit cube including the origin.

Line 8: Define a function that multiplies vector \mathbf{u} by matrix A .

Lines 10–12: $E1$, $E2$, and $E3$ represent the elementary matrices $E_1^{(1,2,2)}$, $E_2^{(1,2)}$, and $E_3^{(1,2)}$, respectively.

Line 14: Below this line, define a function that creates and saves a 3D shape in VPython.

Line 15: Determine the size of the screen on which the 3D image is projected.

Line 16: Decide the position of the camera.

Line 17: Decide the direction the lens points to.

Line 18: Draw the unit cube.

Lines 19, 20: Draw three axes with different colors.

Lines 21–24: Draw each side of the unit cube with the same color as the axis parallel to it in Line 20, and draw the corresponding side of the hexahedron transformed by $E1$ in Line 24. Changing $E1$ in Line 27 to $E2$ or $E3$, we have Fig. 5.1.

Line 25: Save the projected image to a file. The file is saved in the browser's download folder.

Line 27: Call the function defined above for the basic matrix $E1$ and the filename $E1.png$.

By $E_1^{(i,j,c)}$ the unit cube is transformed into a parallelepiped, but since the base area and height do not change, the volume does not change. $E_2^{(i,j)}$ just swaps the sides of the unit cube. By $E_3^{(i,c)}$ the unit cube is transformed to a cuboid, and the volume is multiplied by c .

Let us investigate the more general meaning of the elementary matrices.

Program: elementary_sp.py

```
In [1]: 1 from sympy import Matrix, var
2
3 var('x y a11 a12 a13 a21 a22 a23 a31 a32 a33')
4 E1 = Matrix([[1, x, 0], [0, 1, 0], [0, 0, 1]])
5 E2 = Matrix([[0, 1, 0], [1, 0, 0], [0, 0, 1]])
6 E3 = Matrix([[1, 0, 0], [0, y, 0], [0, 0, 1]])
7 A = Matrix([[a11, a12, a13], [a21, a22, a23], [a31, a32, a33]])
```

Line 3: Create `Symbol` class objects and their names all at once. So far, symbols are imported like `from sympy.abc import x, y, z`, etc., but with the `var` function, we can also define symbols with unique names if we want. The symbol can be quoted with Python's variable of the same name.¹

Lines 4–7: Define matrices E_1 , E_2 , E_3 , and A containing the symbols defined above:

$$E_1 = \begin{bmatrix} 1 & x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad E_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad E_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

After running the above program, we multiply matrix A by E_1 , E_2 , and E_3 from the left in interactive mode.

```
In [2]: E1 * A
Out[2]: Matrix([
[a11 + a21*x, a12 + a22*x, a13 + a23*x],
[a21, a22, a23],
[a31, a32, a33]])
```



```
In [3]: E2 * A
Out[3]: Matrix([
[a21, a22, a23],
[a11, a12, a13],
[a31, a32, a33]])
```



```
In [4]: E3 * A
Out[4]: Matrix([
[a11, a12, a13],
[a21*y, a22*y, a23*y],
[a31, a32, a33]]))
```

Next, multiply from the right.

```
In [5]: A * E1
Out[5]: Matrix([
[a11, a11*x + a12, a13],
[a21, a21*x + a22, a23],
[a31, a31*x + a32, a33]]))
```



```
In [6]: A * E2
Out[6]: Matrix([
[a11, a12, a13],
[a21, a22, a23],
[a31, a32, a33]]))
```

¹ Be careful that using `var('x')` overwrites the variable `x` even if it was previously defined for something else.

```
Out[6]: Matrix([
[a12, a11, a13],
[a22, a21, a23],
[a32, a31, a33]])
```

```
In [7]: A * E3
```

```
Out[7]: Matrix([
[a11, a12*y, a13],
[a21, a22*y, a23],
[a31, a32*y, a33]])
```

In general, we can state the following:

1. Multiplying a given matrix by elementary matrices from the left produces *elementary row operations* on the matrix as follows:
 - (a) Add to one row a scalar multiple of another row.
 - (b) Swap the positions of two rows.
 - (c) Multiply one row by a nonzero scalar.
2. Multiplying a given matrix by elementary matrices from the right produces *elementary column operations* on the matrix as follows:
 - (d) Add to one column a scalar product of another column.
 - (e) Swap the positions of two columns.
 - (f) Multiply one column by a nonzero scalar.

Together they are called the *elementary matrix operations*. More precisely, (a), (b), and (c) are realized by multiplying $E_1^{(i,j,c)}$, $E_2^{(i,j)}$, and $E_3^{(i,c)}$ from the left, respectively, and (d), (e), and (f) are realized by multiplying the corresponding elementary matrices from the right.

In Python, elementary operations can be performed by rewriting the contents of matrix A which is a mutable object.

```
In [8]: B = A.copy(); B[1, :] *= x; B
```

```
Out[8]: Matrix([
[a11, a12, a13],
[a21*x, a22*x, a23*x],
[a31, a32, a33]])
```

We make a copy B of A and apply elementary operations upon B in order to avoid rewriting A itself. The three commands are written on one line separated with semicolons to save space. $B[1, :]$ is the row vector fetched from the second row of B. Multiply this row by x using $\ast=$ ².

```
In [9]: B = A.copy(); B[:, 2] *= x; B
```

```
Out[9]: Matrix([
[a11, a12, a13*x],
[a21, a22, a23*x],
[a31, a32, a33*x]])
```

² In NumPy, the augmented assignment operators can be used for any slices of arrays.

Multiply the third column $B[:, 2]$ of B by x using the augmented assignment statement.

```
In [10]: B = A.copy(); B[0, :] , B[1, :] = B[1, :], B[0, :]; B
```

```
Out[10]: Matrix([
 [a21, a22, a23],
 [a11, a12, a13],
 [a31, a32, a33]])
```

Swap the first and second rows of B .

```
In [11]: B = A.copy(); B[:, 1], B[:, 2] = B[:, 2], B[:, 1]; B
```

```
Out[11]: Matrix([
 [a11, a13, a12],
 [a21, a23, a22],
 [a31, a33, a32]])
```

Swap the second and third columns of B .

```
In [12]: B = A.copy(); B[0, :] += y * B[1, :]; B
```

```
Out[12]: Matrix([
 [a11 + a21*y, a12 + a22*y, a13 + a23*y],
 [a21, a22, a23],
 [a31, a32, a33]])
```

Add the scalar product of the second row by y to the first row of B using $+=$.

```
In [13]: B = A.copy(); B[:, 1] += y * B[:, 2]; B
```

```
Out[13]: Matrix([
 [a11, a12 + a13*y, a13],
 [a21, a22 + a23*y, a23],
 [a31, a32 + a33*y, a33]])
```

Add the scalar product of the third column by y to the second column of B using $+=$.

We can also perform the elementary operations with NumPy. The row (or column) exchange is performed as follows:

```
In [1]: from numpy import array
A = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
B = A.copy(); B[[0, 1], :] = B[[1, 0], :]; B
```

```
Out[1]: array([[4, 5, 6],
 [1, 2, 3],
 [7, 8, 9]])
```

$B[[0, 1], :]$ refers to the first and second rows of B in this order, whereas $B[[1, 0], :]$ is the second and first rows of it. These two rows are swapped.

```
In [2]: B = A.copy(); B[:, [1, 2]] = B[:, [2, 1]]; B
```

```
Out[2]: array([[1, 3, 2],
 [4, 6, 5],
 [7, 9, 8]])
```

$B[:, [1, 2]]$ is the second and third columns of B in this order, and $B[:, [2, 1]]$ is the third and second columns. These two columns are swapped.

5.2 Rank

Let $A = [\mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n]$ be a matrix of shape (m, n) and let $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ be the set of vectors obtained by taking out all of the column vectors of A . The *rank* of the matrix A is defined to be the rank of set $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ and is denoted by $\text{rank}(A)$. This is equal to the dimension of $\text{range}(A) = \langle \mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \rangle$. Especially, for a square matrix A of order n , A is regular if and only if $\text{rank}(A) = n$ (recall Sect. 4.4).

Let B and C be regular matrices of order m and n , respectively. Then, because B and C are bijective as linear mappings, we have $\text{range}(BA) = \text{range}(AC) = \text{range}(A)$. It follows that

$$\text{rank}(BA) = \text{rank}(AC) = \text{rank}(A).$$

Therefore, the rank is an invariant in the strong sense, as stated at the beginning of this chapter.

We apply the elementary operations to calculate the rank of a matrix. Let E and E' be elementary matrices of order m and n , respectively. Since elementary matrices are regular, we have $\text{rank}(EA) = \text{rank}(AE') = \text{rank}(A)$, that is, the rank of A will not change if an elementary operation is applied to it.

The rank of T of the following form (*upper triangular matrix*) is equal to k (an arbitrary element of \mathbb{K} is acceptable for each *), because the first k columns of the matrix form a basis of $\text{range}(T)$:

$$T = \begin{bmatrix} a_{11} & * & * & * & \cdots & * \\ 0 & \ddots & \ddots & \vdots & \cdots & * \\ 0 & \ddots & a_{kk} & * & \cdots & * \\ 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 \end{bmatrix} \quad (a_{11}a_{22}\cdots a_{kk} \neq 0).$$

We try to transform a given matrix A into this shape by applying elementary operations. Because the rank of upper triangular matrix T is equal to the rank of its transposed matrix, we find that

$$\text{rank}(A) = \text{rank}(T) = \text{rank}(T^T) = \text{rank}(A^T).$$

Here we give a concrete example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \xrightarrow{\text{2nd row} - \text{1st row} \times 2} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 3 & 4 & 5 \end{bmatrix} \xrightarrow{\text{3rd row} - \text{1st row} \times 3} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 0 & -2 & -4 \end{bmatrix}.$$

In this way, both elements 2 and 3 in the first column below the diagonal are changed to 0. Next, -2 in the second column below the diagonal of the last matrix is changed to 0 as

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 0 & -2 & -4 \end{bmatrix} \xrightarrow{\text{3rd row} - 2\text{nd row} \times 2} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 0 & 0 & 0 \end{bmatrix}.$$

This last matrix is of the desired shape, and so we find that the rank of the original matrix is 2. In the process of elementary operations, we try to change all of the elements in the i -th column below the diagonal (i, i) to 0 for $i = 1, 2, \dots$. This (i, i) -element is called a *pivot*. Sometimes a pivot becomes 0. In this case we swap the row having the pivot by a row below it, or the column having the pivot by a column to the right. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 5 \end{bmatrix} \xrightarrow{\text{swap 2nd and 3rd columns}} \begin{bmatrix} 1 & 3 & 2 \\ 0 & 4 & 0 \\ 0 & 5 & 0 \end{bmatrix} \xrightarrow{\text{3rd row} - 2\text{nd row} \times \frac{5}{4}} \begin{bmatrix} 1 & 3 & 2 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

To calculate the rank with NumPy, we use the `matrix_rank` function defined in the `linalg` module, as already mentioned in Sect. 3.4.

```
In [1]: from numpy import *
A = array([[1, 2, 3], [2, 3, 4], [3, 4, 5]])
linalg.matrix_rank(A)
```

```
Out[1]: 2
```

In SymPy, use the `rank` method of the `Matrix` class.

```
In [1]: from sympy import *
A = Matrix([[1, 2, 3], [2, 3, 4], [3, 4, 5]])
A.rank()
```

```
Out[1]: 2
```

Exercise 5.3 Prove the following facts about the rank of a matrix:

- (1) $\text{rank } (\mathbf{A}) \leq \min \{m, n\}$ for $m \times n$ matrix \mathbf{A} .
- (2) $\text{rank } (\mathbf{AB}) \leq \min \{\text{rank } (\mathbf{A}), \text{rank } (\mathbf{B})\}$ for $l \times m$ matrix \mathbf{A} and for $m \times n$ matrix \mathbf{B} .

Exercise 5.4 The following program produces calculation problems of matrix rank. Find the rank of each matrix generated by this program.

Program: `prob_rank.py`

```
In [1]: 1 | from numpy.random import seed, choice, permutation
2 | from sympy import Matrix
3 |
4 | def f(P, m1, m2, n):
5 |     if n > min(m1, m2):
6 |         return Matrix(choice(P, (m1, m2)))
7 |     else:
8 |         while True:
9 |             X, Y = choice(P, (m1, n)), choice(P, (n, m2))
10 |            A = Matrix(X.dot(Y))
11 |            if A.rank() == n:
12 |                return A
13 |
```

```
In [1]: 14 | m1, m2 = 3, 4
15 | seed(2021)
16 | for i in permutation(max(m1, m2)):
17 |     print(f([-3, -2, -1, 1, 2, 3], m1, m2, i+1))
```

Lines 4–12: This function tries to randomly generate a matrix of shape $(m1, m2)$ whose rank is n . If the value of n is greater than $m1$ or $m2$, the rank will never be n . In this case, an $m1 \times m2$ matrix whose elements are randomly taken from \mathbb{P} is returned. Otherwise, create an $m1 \times m2$ matrix with rank n as follows. Create randomly an $m1 \times n$ matrix X and an $n \times m2$ matrix Y with elements in \mathbb{P} , and multiply them to make A . Then, the rank of A will be less than or equal to n (by Exercise 5.3). Remake A until the rank becomes exactly n and return the obtained A . To get a matrix with small rank relative to $m1$ and $m2$, this way is more efficient than simply generating a matrix of shape $(m1, m2)$ randomly.

Line 14: Assign values in $m1$ and $m2$ to determine the shape of the matrix.

Line 15: Changing the seed of the random number creates another problem.

Lines 16, 17: Generate problems with different ranks. The first argument of f is the list of numbers with relatively small absolute values to facilitate manual calculation, but 0 is excluded to avoid generating problems that are too easy.

```
Matrix([[[-3, 3, 2, 1], [3, 3, 3, -3], [2, -2, 3, -2]]])
Matrix([[5, -2, 1, -14], [-5, 2, -9, 4], [-4, 1, -3, 11]])
Matrix([[3, -1, -2, 2], [-7, 4, 7, -3], [1, 3, 4, 4]])
Matrix([[-2, -1, 3, 1], [4, 2, -6, -2], [4, 2, -6, -2]])
```

5.3 Determinant

A sequence made by extracting all elements from $\{1, 2, \dots, n\}$ and rearranging them without duplication is called a *permutation* of order n . P_n denotes the set of all permutations of order n , and we express its elements as (p_1, p_2, \dots, p_n) . In particular, $(1, 2, \dots, n)$ is called the *identity permutation*.

For $i \neq j$, the action that exchanges the i -th and j -th numbers in a permutation

$$(p_1, \dots, p_i, \dots, p_j, \dots, p_n) \mapsto (p_1, \dots, \underline{p_j}, \dots, \underline{p_i}, \dots, p_n)$$

is called a *transposition*. For example, the permutation $(5, 1, 2, 3, 4)$ of order 5 is obtained from the identity permutation by repeating transpositions four times as follows:

$$(1, 2, 3, 4, 5) \mapsto (\underline{5}, 2, 3, 4, \underline{1}) \mapsto (5, \underline{1}, 3, 4, \underline{2}) \mapsto (5, 1, \underline{2}, 4, \underline{3}) \mapsto (5, 1, 2, \underline{3}, \underline{4}).$$

It is easy to see that any permutation can be obtained from the identity permutation by repeating transpositions. A permutation obtained from the identity permutation by iterating transpositions even (resp. odd) times is called an *even* (resp. *odd*) permutation. Let us see that any permutation is either even or odd. Consider a function f in n variables x_1, x_2, \dots, x_n defined by

$$f(x_1, x_2, \dots, x_n) \stackrel{\text{def}}{=} \prod_{i < j} (x_i - x_j).$$

Exchanging two variables in f turns the sign of its value, that is,

$$f(\dots, x_i, \dots, x_j, \dots) = -f(\dots, x_j, \dots, x_i, \dots)$$

for $i \neq j$. Therefore, if a permutation (p_1, p_2, \dots, p_n) is obtained from the identity permutation by iterating transpositions s times, then

$$f(x_{p_1}, x_{p_2}, \dots, x_{p_n}) = (-1)^s f(x_1, x_2, \dots, x_n).$$

If this (p_1, p_2, \dots, p_n) is also obtained by iterating transpositions t times, then we have $(-1)^s = (-1)^t$. Hence both s and t are either even or odd. We call

$$\sigma(p_1, p_2, \dots, p_n) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{if } (p_1, p_2, \dots, p_n) \text{ is an even permutation} \\ -1, & \text{if } (p_1, p_2, \dots, p_n) \text{ is an odd permutation} \end{cases}$$

the *signature* of the permutation (p_1, p_2, \dots, p_n) .

Exercise 5.5 Enumerate all permutations of orders 2 and 3, and calculate their signatures.

Now we define the *determinant* $|A| \in \mathbb{K}$ of a square matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

over \mathbb{K} of order n by

$$|A| = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} \stackrel{\text{def}}{=} \sum_{(p_1, p_2, \dots, p_n) \in P_n} \sigma(p_1, p_2, \dots, p_n) a_{1p_1} a_{2p_2} \cdots a_{np_n},$$

where $\sum_{(p_1, p_2, \dots, p_n) \in P_n}$ means that the summation ranges over the all permutations of order n . The determinant of A is also denoted by $\det(A)$.

Exercise 5.6 Prove the following formulas for the determinants of matrices of orders 2 and 3:

$$(1) \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}.$$

$$(2) \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}.$$

Here we give a function producing all permutations of given order together with their signatures and a function computing the determinant of a matrix according to the definition.

Program: `determinant.py`

```
In [1]: 1 from functools import reduce
2
3 def P(n):
4     if n == 1:
5         return [[(0, 1)]]
```

In [1]:

```

6     else:
7         Q = []
8         for p, s in P(n-1):
9             Q.append((p + [n-1], s))
10            for i in range(n-1):
11                q = p + [n-1]
12                q[i], q[-1] = q[-1], q[i]
13                Q.append((q, -1*s))
14
15    return Q
16
17
18 def prod(L): return reduce(lambda x, y: x*y, L)
19
20 def det(A):
21     n = len(A)
22     a = sum([s * prod([A[i][p[i]] for i in range(n)])
23             for p, s in P(n)])
24     return a
25
26 if __name__ == '__main__':
27     A = [[1, 2], [2, 3]]
28     B = [[1, 2], [2, 4]]
29     C = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
30     D = [[1, 2, 3], [2, 3, 1], [3, 1, 2]]
31     print(det(A), det(B), det(C), det(D))

```

Lines 3–14: Recursively generate a list of the permutations of order n as follows. For a permutation $(p_1, p_2, \dots, p_{n-1})$ of order $n - 1$, get a permutation $(p_1, p_2, \dots, p_{n-1}, n)$ of order n with the same signature. Furthermore, from this permutation get $n - 1$ permutations

$$(n, p_2, \dots, p_{n-1}, p_1), (p_1, n, \dots, p_{n-1}, p_2), \dots, (p_1, p_2, \dots, n, p_{n-1})$$

of order n with the opposite signature by exchanging n with each of p_1, p_2, \dots, p_{n-1} . Letting $(p_1, p_2, \dots, p_{n-1})$ move to cover all permutations of order $n - 1$, get all permutations of order n .

Line 16: Define the function that returns the product of all the elements in a given list. It uses the function `reduce` defined in the standard library `functools`. This function with two arguments, say `reduce(f, L)` where the first argument is a two-variable function $x, y \mapsto f(x, y)$ and the second argument is a list $[a_1, a_2, \dots, a_n]$, returns $f\left(\dots f(f(a_1, a_2), a_3) \dots, a_n\right)$. If $f : x, y \mapsto xy$, then it returns $a_1 a_2 \dots a_n$.

Lines 18–22: Calculate the determinant according to the definition.



-1 0 0 -18

The determinant has the following properties.

- (1) Swapping two rows of a matrix changes the sign of its determinant:

$$\begin{array}{c} i\text{-th row} \\ j\text{-th row} \end{array} \begin{vmatrix} \vdots & \vdots \\ a_{i1} & \cdots & a_{in} \\ \vdots & \vdots \\ a_{j1} & \cdots & a_{jn} \\ \vdots & \vdots \end{vmatrix} = - \begin{vmatrix} \vdots & \vdots \\ a_{j1} & \cdots & a_{jn} \\ \vdots & \vdots \\ a_{i1} & \cdots & a_{in} \\ \vdots & \vdots \end{vmatrix}.$$

\therefore Suppose $i < j$ and let A' be the matrix formed by exchanging the i -th row and the j -th row of matrix A . Then, noting that the results of addition and multiplication of scalars do not depend on

their order, we have

$$\begin{aligned}\det(A) &= \sum_{(p_1, \dots, p_i, \dots, p_j, \dots, p_n) \in P_n} \sigma(p_1, \dots, p_i, \dots, p_j, \dots, p_n) a_{1p_1} \cdots a_{np_n} \\ &= - \sum_{(p_1, \dots, p_j, \dots, p_i, \dots, p_n) \in P_n} \sigma(p_1, \dots, p_j, \dots, p_i, \dots, p_n) a_{1p_1} \cdots a_{np_n} \\ &= -\det(A').\end{aligned}$$

- (2) The determinant of a matrix with two equal rows is 0.

\because Let A be a matrix in which the i -th row and the j -th row are equal. Swapping these two rows we get the same matrix A . By (1) we see $\det(A) = -\det(A)$, that is, $\det(A) = 0$.

- (3) Multiplying one row by scalar c multiplies the determinant by c :

$$i\text{-th row} \left| \begin{array}{ccc|cc} a_{11} & \cdots & a_{1n} & | & a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots & | & \vdots & & \vdots \\ ca_{i1} & \cdots & ca_{in} & | & a_{i1} & \cdots & a_{in} \\ \vdots & & \vdots & | & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} & | & a_{n1} & \cdots & a_{nn} \end{array} \right. = c \left| \begin{array}{ccc|cc} a_{11} & \cdots & a_{1n} & | & a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots & | & \vdots & & \vdots \\ a_{i1} & \cdots & a_{in} & | & a_{i1} & \cdots & a_{in} \\ \vdots & & \vdots & | & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} & | & a_{n1} & \cdots & a_{nn} \end{array} \right..$$

\because Let A' be a matrix obtained by multiplying the i -th row of A by c . Then, we have

$$\begin{aligned}\det(A') &= \sum_{(p_1, \dots, p_i, \dots, p_n) \in P_n} \sigma(p_1, \dots, p_i, \dots, p_n) a_{1p_1} \cdots (ca_{ip_i}) \cdots a_{np_n} \\ &= c \sum_{(p_1, \dots, p_i, \dots, p_n) \in P_n} \sigma(p_1, \dots, p_i, \dots, p_n) a_{1p_1} \cdots a_{ip_i} \cdots a_{np_n} \\ &= c \det(A).\end{aligned}$$

- (4) The determinant of a matrix whose one row is a sum of two rows is the sum of the determinants of two matrices made by dividing that row:

$$i\text{-th row} \left| \begin{array}{ccc|cc} a_{11} & \cdots & a_{1n} & | & a_{11} & \cdots & a_{1n} & | & a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots & | & \vdots & & \vdots & | & \vdots & & \vdots \\ b_1 + c_1 & \cdots & b_n + c_n & | & b_1 & \cdots & b_n & | & c_1 & \cdots & c_n \\ \vdots & & \vdots & | & \vdots & & \vdots & | & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} & | & a_{n1} & \cdots & a_{1n} & | & a_{n1} & \cdots & a_{nn} \end{array} \right. = \left| \begin{array}{ccc|cc} a_{11} & \cdots & a_{1n} & | & a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots & | & \vdots & & \vdots \\ a_{n1} & \cdots & a_{1n} & | & a_{n1} & \cdots & a_{nn} \end{array} \right| + \left| \begin{array}{ccc|cc} a_{11} & \cdots & a_{1n} & | & c_1 & \cdots & c_n \\ \vdots & & \vdots & | & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} & | & a_{n1} & \cdots & a_{nn} \end{array} \right|.$$

\because Suppose that the i -th row of A is a sum of two rows $b = (b_1, b_2, \dots, b_n)$ and $c = (c_1, c_2, \dots, c_n)$; $a_{ij} = b_j + c_j$ ($j = 1, 2, \dots, n$). Let A' and A'' be matrices obtained by replacing the i -th row of A by b and by c , respectively. Then we have

$$\begin{aligned}
\det(\mathbf{A}) &= \sum_{(p_1, \dots, p_i, \dots, p_n) \in P_n} \sigma(p_1, \dots, p_i, \dots, p_n) a_{1p_1} \cdots (b_{p_i} + c_{p_i}) \cdots a_{np_n} \\
&= \sum_{(p_1, \dots, p_i, \dots, p_n) \in P_n} \sigma(p_1, \dots, p_i, \dots, p_n) a_{1p_1} \cdots b_{p_i} \cdots a_{np_n} \\
&\quad + \sum_{(p_1, \dots, p_i, \dots, p_n) \in P_n} \sigma(p_1, \dots, p_i, \dots, p_n) a_{1p_1} \cdots c_{p_i} \cdots a_{np_n} \\
&= \det(\mathbf{A}') + \det(\mathbf{A}'').
\end{aligned}$$

(5) Adding a scalar multiple of one row to another row does not change the determinant;

$$\begin{array}{c|ccccc}
& \vdots & \vdots & & \vdots & \vdots \\
i\text{-th row} & a_{i1} & \cdots & a_{in} & a_{i1} + ca_{j1} & \cdots & a_{in} + ca_{jn} \\
& \vdots & \vdots & & \vdots & & \vdots \\
j\text{-th row} & a_{j1} & \cdots & a_{jn} & a_{j1} & \cdots & a_{jn} \\
& \vdots & \vdots & & \vdots & & \vdots
\end{array} = \begin{array}{c|ccccc}
& \vdots & & & \vdots & \vdots \\
& \vdots & & & \vdots & \vdots \\
& \vdots & & & \vdots & \vdots \\
& \vdots & & & \vdots & \vdots
\end{array}.$$

\because Let \mathbf{A}' be the matrix obtained from \mathbf{A} by adding the multiple of the j -th row by scalar c to the i -th row ($i \neq j$). Let \mathbf{A}'' be the matrix obtained from \mathbf{A} by replacing the i -th row with the j -th row, then $\det(\mathbf{A}'') = 0$ by (2). By (3) and (4) we have

$$\det(\mathbf{A}') = \det(\mathbf{A}) + c \det(\mathbf{A}'') = \det(\mathbf{A}).$$

In a similar and symmetric way, we see that (1)–(5) above hold even if all rows in the statements are changed to columns. We summarize the relations between the determinant and the elementary operations of a matrix.

- Elementary row operations:

- The determinant of a matrix formed by exchanging one row of \mathbf{A} with another row is equal to $-\det(\mathbf{A})$.
- The determinant of a matrix formed by multiplying one row of matrix \mathbf{A} by c is equal to $c \det(\mathbf{A})$.
- The determinant of a matrix formed by adding to one row a scalar multiple of another row of \mathbf{A} is equal to $\det(\mathbf{A})$.

- Elementary column operations:

- The determinant of a matrix formed by exchanging one column of \mathbf{A} with another column is equal to $-\det(\mathbf{A})$.
- The determinant of a matrix formed by multiplying one column of matrix \mathbf{A} by c is equal to $c \det(\mathbf{A})$.
- The determinant of a matrix formed by adding to one column a scalar multiple of another column of \mathbf{A} is equal to $\det(\mathbf{A})$.

These properties can be used to calculate the determinant. Applying elementary operations we reduce a given square matrix to an upper triangular matrix

$$\mathbf{T} = \begin{bmatrix} a_{11} & * & \cdots & * \\ 0 & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ 0 & \cdots & 0 & a_{nn} \end{bmatrix}.$$

Note that for any non-identity permutation (p_1, p_2, \dots, p_n) , there is i such that $i > p_i$ (so $a_{ip_i} = 0$ in \mathbf{T}). Hence,

$$|\mathbf{T}| = \sum_{(p_1, p_2, \dots, p_n) \in P_n} \sigma(p_1, p_2, \dots, p_n) a_{1p_1} a_{2p_2} \cdots a_{np_n} = a_{11} a_{22} \cdots a_{nn},$$

because all other terms than $a_{11} a_{22} \cdots a_{nn}$ in the summation are 0. The method of this reduction is the same as when we calculate the rank of a matrix with elementary operations, but it is an equality transformation. When rows or columns are exchanged, correct the value by multiplying -1 to keep the equality.

Here is an example:

$$\begin{array}{c|ccc} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{array} \xrightarrow{\text{2nd row} - \text{1st row} \times 2} \begin{array}{c|ccc} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 3 & 1 & 2 \end{array} \xrightarrow{\text{3rd row} - \text{1st row} \times 3} \begin{array}{c|ccc} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 0 & -5 & -7 \end{array} \\ \xrightarrow{\text{3rd row} - \text{2nd row} \times 5} \begin{array}{c|ccc} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 0 & 0 & 18 \end{array} = 1 \times (-1) \times 18 = -18. \end{array}$$

When the pivot reaches 0 along the way, swap rows or columns:

$$\begin{array}{c|ccc} 0 & 2 & 2 \\ 2 & 1 & 2 \\ 3 & 2 & 3 \end{array} \xrightarrow{\text{swap 1st and 2nd rows}} \begin{array}{c|ccc} 2 & 1 & 2 \\ 0 & 2 & 2 \\ 3 & 2 & 3 \end{array} \xrightarrow{\text{3rd row} - \text{1st row} \times \frac{3}{2}} \begin{array}{c|ccc} 2 & 1 & 2 \\ 0 & 2 & 2 \\ 0 & \frac{1}{2} & 0 \end{array} \\ \xrightarrow{\text{3rd row} - \text{2nd row} \times \frac{1}{4}} \begin{array}{c|ccc} 2 & 1 & 2 \\ 0 & 2 & 2 \\ 0 & 0 & -\frac{1}{2} \end{array} = -2 \times 2 \times \left(-\frac{1}{2}\right) = 2. \end{array}$$

Furthermore, because elementary row and column operations are caused by multiplying elementary matrices from the left and from the right, respectively, as stated in Sect. 5.1, any square matrix \mathbf{A} can be transformed into the following form by applying elementary row and column operations successively:

$$\mathbf{J}_1 \mathbf{A} \mathbf{J}_2 = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & 1 & \ddots & & \vdots \\ \vdots & & \ddots & 0 & \ddots & \vdots \\ \vdots & & & & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & 0 \end{bmatrix},$$

where \mathbf{J}_1 and \mathbf{J}_2 are the products of elementary matrices.

Let us see how this transformation is possible in the following example.

$$\begin{array}{c} \left[\begin{array}{ccc} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 1 & 0 & -7 \end{array} \right] \xrightarrow{\text{2nd row} - \text{1st row} \times 2} \left[\begin{array}{ccc} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 1 & 0 & -7 \end{array} \right] \xrightarrow{\text{3rd row} - \text{1st row}} \left[\begin{array}{ccc} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 0 & -2 & -10 \end{array} \right] \\ \xrightarrow{\text{multiply 2nd row by } -1} \left[\begin{array}{ccc} 1 & 2 & 3 \\ 0 & 1 & 5 \\ 0 & -2 & -10 \end{array} \right] \xrightarrow{\text{3rd row} + \text{2nd row} \times 2} \left[\begin{array}{ccc} 1 & 2 & 3 \\ 0 & 1 & 5 \\ 0 & 0 & 0 \end{array} \right] \xrightarrow{\text{2nd col} - \text{1st col} \times 2} \\ \left[\begin{array}{ccc} 1 & 0 & 3 \\ 0 & 1 & 5 \\ 0 & 0 & 0 \end{array} \right] \xrightarrow{\text{3rd col} - \text{1st col} \times 3} \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 5 \\ 0 & 0 & 0 \end{array} \right] \xrightarrow{\text{3rd col} - \text{2nd col} \times 5} \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array} \right]. \end{array}$$

Let \mathbf{J}_0 be the matrix in the right-hand side of the above equation; $\mathbf{J}_1 \mathbf{A} \mathbf{J}_2 = \mathbf{J}_0$. The rank of \mathbf{J}_0 is equal to the number of 1's on the diagonal, and it is equal to the rank of \mathbf{A} because \mathbf{J}_1 and \mathbf{J}_2 are regular matrices.

It is easy to see

$$\det(\mathbf{E}_1^{(i,j,c)}) = 1, \quad \det(\mathbf{E}_2^{(i,j)}) = -1, \quad \det(\mathbf{E}_3^{(i,c)}) = c.$$

By this together with the relationship between the elementary transformations and the determinant, we have

$$\det(\mathbf{A}\mathbf{E}) = \det(\mathbf{E}\mathbf{A}) = \det(\mathbf{E})\det(\mathbf{A}) \quad \dots \quad (*)$$

for any matrix \mathbf{A} and an elementary matrix \mathbf{E} . Moreover, these equalities hold even if \mathbf{E} is a product of elementary matrices (induction on the number of elementary matrices involved in \mathbf{E}).

By $(*)$ we have

$$\det(\mathbf{J}_0) = \det(\mathbf{J}_1 \mathbf{A} \mathbf{J}_2) = \det(\mathbf{J}_1)\det(\mathbf{A}\mathbf{J}_2) = \det(\mathbf{J}_1)\det(\mathbf{A})\det(\mathbf{J}_2).$$

Thus, we find that

$$\mathbf{A} \text{ is regular} \Leftrightarrow \mathbf{J}_0 \text{ is regular} \Leftrightarrow \det(\mathbf{J}_0) \neq 0 \Leftrightarrow \det(\mathbf{A}) \neq 0.$$

When \mathbf{A} is of order n , the above conditions are also equivalent to

$$\text{rank}(\mathbf{A}) = n.$$

Because \mathbf{J}_1 and \mathbf{J}_2 have inverses, we have $\mathbf{A} = \mathbf{J}_1^{-1}\mathbf{J}_0\mathbf{J}_2^{-1}$. Since the inverses of elementary matrices are also elementary matrices, \mathbf{A} can be expressed as a product of elementary matrices and \mathbf{J}_0 . In particular, if \mathbf{A} is a regular matrix, then \mathbf{J}_0 is the unit matrix \mathbf{I} , and thus \mathbf{A} is written as a product of elementary matrices.

Here, we give the following important *product formula* of determinant:

$$\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B}),$$

where \mathbf{A} and \mathbf{B} are square matrices of the same order. In fact, if \mathbf{A} or \mathbf{B} is not regular, then \mathbf{AB} is not regular and both sides of the formula are 0. If both \mathbf{A} and \mathbf{B} are regular, they are products of elementary matrices, and the formula holds due to (*).

The function `det` to calculate the determinant is implemented in NumPy.

```
In [1]: from numpy.linalg import det
A = [[1, 2], [2, 3]]
B = [[1, 2], [2, 4]]
C = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
D = [[1, 2, 3], [2, 3, 1], [3, 1, 2]]
det(A), det(B), det(C), det(D)
```

```
Out[1]: (-1.0, 0.0, 2.2204460492503185e-16, -18.000000000000004)
```

The determinant whose components are all integers should be an integer according to the definition of a determinant, but the results for C and D include some errors. In particular, the determinant of C must be 0. This is caused by a division error when the pivot is set to 1 in the process of elementary operations. Let us experiment with nonregular larger matrices.

Program: error.py

```
In [1]: 1 | from numpy.linalg import det, matrix_rank
2 | from numpy.random import seed, normal
3 |
4 | seed(123)
5 | n = 20
6 | F = normal(0, 1, (n, n-1))
7 | G = F.dot(F.T)
8 | print(f'shape = {G.shape}')
9 | print(f'det = {det(G)}')
10 | print(f'rank = {matrix_rank(G)}')
```

Line 7: A randomly generated G is a nonregular square matrix of order 20 whose rank is 19.



```
shape = (20, 20)
det = 33.21619942347684
rank = 19
```

The determinant of G must be 0 but the numerical result includes a very large error compared to the previous result, while `matrix_rank` returns the correct value. To decide whether a matrix is regular, we should use `matrix_rank` in NumPy. The reader may change the seed and the size of a matrix in the above code.

Sympy does not cause such errors as above. It also calculates a determinant containing symbols.

```
In [1]: from sympy import Matrix, symbols
A = Matrix([[1, 2], [2, 3]])
B = Matrix([[1, 2], [2, 4]])
```

```
In [2]: C = Matrix([[1, 2, 3], [2, 3, 4], [3, 4, 5]])
D = Matrix([[1, 2, 3], [2, 3, 1], [3, 1, 2]])
A.det(), B.det(), C.det(), D.det()
```

```
Out[2]: (-1, 0, 0, -18)
```

```
In [3]: a11, a12, a13 = symbols('a11, a12, a13')
a21, a22, a23 = symbols('a21, a22, a23')
a31, a32, a33 = symbols('a31, a32, a33')
E = Matrix([[a11, a12], [a21, a22]])
F = Matrix([[a11, a12, a13], [a21, a22, a23], [a31, a32, a33]])
E.det()
```

```
Out[3]: a11*a22 - a12*a21
```

```
In [4]: F.det()
```

```
Out[4]: a11*a22*a33 - a11*a23*a32 - a12*a21*a33
+ a12*a23*a31 + a13*a21*a32 - a13*a22*a31
```

Exercise 5.7 Write a program that randomly creates 10000 square matrices of a given order n whose elements are integers between 0 and 10. Count the number of regular matrices for several n .

Exercise 5.8 The following program generates two matrices. Calculate the determinant of generated matrices by hand using elementary operations.

Program: prob_det.py

```
In [1]: 1 from numpy.random import seed, choice, permutation
2 from sympy import Matrix
3
4 def f(P, m, p):
5     while True:
6         A = Matrix(choice(P, (m, m)))
7         if p == 0:
8             if A.det() == 0:
9                 return A
10            elif A.det() != 0:
11                return A
12
13 m = 3
14 seed(2021)
15 for p in permutation(2):
16     print(f([-3, -2, -1, 1, 2, 3], m, p))
```

Lines 15, 16: Randomly generate two matrices, one with zero determinant and the other with nonzero determinant.



```
Matrix([[3, -2, -3], [3, 2, 1], [3, 3, 3]])
Matrix([[1, -2, 1], [-2, -2, 1], [1, -2, 1]])
```

Lastly, we add some important results on determinants. For a square matrix A , we have

$$\det(A^T) = \det(A), \quad \det(A^*) = \overline{\det(A)}.$$

Let $A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$, then $A^T = \begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nn} \end{bmatrix}$. The determinant of A^T is obtained by replacing the row subscripts with the column subscripts in the expression of the determinant definition of A , that is,

$$|A^T| = \begin{vmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{vmatrix} = \sum_{(p_1, p_2, \dots, p_n) \in P_n} \sigma(p_1, p_2, \dots, p_n) a_{p_1 1} a_{p_2 2} \cdots a_{p_n n}.$$

Rearranging the pairs $\frac{i}{p_i}$ in the table on the left-hand side below, we can transform the table to the table on the right:

$$\begin{array}{|c|c|c|c|} \hline 1 & 2 & \cdots & n \\ \hline p_1 & p_2 & \cdots & p_n \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|c|} \hline q_1 & q_2 & \cdots & q_n \\ \hline 1 & 2 & \cdots & n \\ \hline \end{array}.$$

Then, $(q_1, q_2, \dots, q_n) \in P_n$ and $a_{p_1 1} a_{p_2 2} \cdots a_{p_n n} = a_{1q_1} a_{2q_2} \cdots a_{nq_n}$ holds. This transformation can be done by a sequence of iterative transpositions, which transforms the identity permutation $(1, 2, \dots, n)$ to (q_1, q_2, \dots, q_n) and at the same time (p_1, p_2, \dots, p_n) to $(1, 2, \dots, n)$. It follows that $\sigma(q_1, q_2, \dots, q_n) = \sigma(p_1, p_2, \dots, p_n)$. Thus, the rightmost formula of the above equation of $|A^T|$ is equal to the expression of the determinant definition of A , and we have $\det(A^T) = \det(A)$. Also we have

$$\det(A^*) = \overline{\det(A^T)} = \overline{\det(A)}.$$

For a regular matrix V we have

$$|V^{-1}| = |V|^{-1},$$

because $|V| |V^{-1}| = |VV^{-1}| = |I| = 1$. So, if A and B are similar, that is, $B = V^{-1}AV$, then

$$|B| = |V^{-1}AV| = |V|^{-1} |A| |V| = |A|.$$

Therefore, the determinant is a matrix invariant in the sense stated at the beginning of the chapter.

5.4 Trace

For a square matrix A of order n , the *trace* of A is defined to be the sum of all diagonal elements of A and is denoted by $\text{Tr}(A)$. For square matrices A and B of order n and for $c \in \mathbb{K}$, the following equalities hold:

$$1. \text{Tr}(A + B) = \text{Tr}(A) + \text{Tr}(B) \quad 2. \text{Tr}(cA) = c\text{Tr}(A)$$

$$3. \text{Tr}(A^*) = \overline{\text{Tr}(A)} \quad 4. \text{Tr}(AB) = \text{Tr}(BA).$$

The equalities 1–3 are immediate from the definition. Let us prove 4. Let E be a diagonal matrix or an elementary matrix of order n . A simple calculation leads to $\text{Tr}(AE) = \text{Tr}(EA)$. As described in the previous section, any square matrix B of order n can be expressed by the product of elementary matrices and a diagonal matrix, and using this fact repeatedly, we get $\text{Tr}(AB) = \text{Tr}(BA)$.

Next, we show that the trace is unique in some sense. Let $M_n(\mathbb{K})$ denote the linear space of all square matrices of order n over \mathbb{K} . Suppose that a linear mapping $\varphi : M_n(\mathbb{K}) \rightarrow \mathbb{K}$ satisfies $\varphi(AB) = \varphi(BA)$ for any $A, B \in M_n(\mathbb{K})$. Let us show that there exists $\alpha \in \mathbb{K}$ such that $\varphi(A) = \alpha \text{Tr}(A)$ for any

$A \in M_n(\mathbb{K})$. For $1 \leq i, j \leq n$ let $\mathbf{U}_{ij} \in M_n(\mathbb{K})$ be the matrix in which the only (i, j) -element is 1 and all of the others are 0. They are called *matrix units*. The set of all matrix units forms a basis of linear space $M_n(\mathbb{K})$. Note that $\text{Tr}(\mathbf{U}_{ij}) = \delta_{ij}$ holds.³ Hence, to prove our assertion, it is enough to show that $\varphi(\mathbf{U}_{ij}) = \alpha \delta_{ij}$ for some constant $\alpha \in \mathbb{K}$. Let $i \neq j$. Since $\mathbf{U}_{ij} = \mathbf{U}_{ij}\mathbf{U}_{jj}$ and $\mathbf{U}_{jj}\mathbf{U}_{ij} = \mathbf{O}$, we have

$$\varphi(\mathbf{U}_{ij}) = \varphi(\mathbf{U}_{ij}\mathbf{U}_{jj}) = \varphi(\mathbf{U}_{jj}\mathbf{U}_{ij}) = \varphi(\mathbf{O}) = 0.$$

Since $\mathbf{U}_{ii} = \mathbf{E}_2^{(i,j)}\mathbf{U}_{jj}\mathbf{E}_2^{(i,j)}$ and $\mathbf{E}_2^{(i,j)}\mathbf{E}_2^{(i,j)} = \mathbf{I}$, we have

$$\varphi(\mathbf{U}_{ii}) = \varphi\left(\mathbf{E}_2^{(i,j)}\mathbf{U}_{jj}\mathbf{E}_2^{(i,j)}\right) = \varphi\left(\mathbf{U}_{jj}\mathbf{E}_2^{(i,j)}\mathbf{E}_2^{(i,j)}\right) = \varphi(\mathbf{U}_{jj}).$$

Letting $\alpha = \varphi(\mathbf{U}_{ii})$, we have $\varphi(\mathbf{U}_{ij}) = \alpha \delta_{ij}$ for any i and j .

More generally, for any linear mapping $\varphi : M_n(\mathbb{K}) \rightarrow \mathbb{K}$ there exists $\mathbf{X} \in M_n(\mathbb{K})$ such that

$$\varphi(\mathbf{A}) = \text{Tr}(\mathbf{AX})$$

for all $\mathbf{A} \in M_n(\mathbb{K})$. Actually, \mathbf{X} is given by

$$\mathbf{X} = \begin{bmatrix} \varphi(\mathbf{U}_{11}) & \varphi(\mathbf{U}_{21}) & \cdots & \varphi(\mathbf{U}_{n1}) \\ \varphi(\mathbf{U}_{12}) & \varphi(\mathbf{U}_{22}) & \cdots & \varphi(\mathbf{U}_{n2}) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi(\mathbf{U}_{1n}) & \varphi(\mathbf{U}_{2n}) & \cdots & \varphi(\mathbf{U}_{nn}) \end{bmatrix}.$$

Exercise 5.9 There are some small gaps and omissions in the arguments above in this section. In the first part we say “are immediate from”, “a simple calculation”, and “using this fact repeatedly”, which contain some gaps. In the second half, we omit the proof of existence of \mathbf{X} . Fill these gaps one by one.

Lastly, we show that the trace is also a matrix invariant. In fact, for square matrices \mathbf{A} and \mathbf{V} of the same order such that \mathbf{V} is regular, we have

$$\text{Tr}(\mathbf{V}^{-1}\mathbf{AV}) = \text{Tr}(\mathbf{AVV}^{-1}) = \text{Tr}(\mathbf{A}).$$

5.5 Systems of Linear Equations

In this section, we give a method to solve a *system of linear equations* (or simply *linear system*)

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

³This δ_{ij} is called *Kronecker's delta* and is defined by $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ otherwise.

with coefficients $a_{ij} \in \mathbb{K}$ ($1 \leq i \leq m$, $1 \leq j \leq n$) and unknown variables x_j ($1 \leq j \leq n$). In order to find a solution of this system, we first make a table as follows:

$$\begin{array}{c|ccccc} x_1 & x_2 & \cdots & x_n & \\ \hline a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \\ \hline a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}$$

Here, we call $(a_{i1}, a_{i2}, \dots, a_{in}, b_i)$ the i -th row for $i = 1, 2, \dots, m$. Also, the j -th column consists of $x_j, a_{1j}, a_{2j}, \dots, a_{mj}$ for $j = 1, 2, \dots, n$. We will transform this table, applying the following rules:

1. All three types of elementary row operations.
 2. Exchanging operations for a pair of columns.

These operations correspond to the following transformations of the linear system:

- 1.(a) To one equation add a multiple by c of another equation.
 (b) Swap the positions of two equations.
 (c) Multiply one equation by $c \neq 0$.
 2. Swap the positions of the terms of x_i and x_j of the left-hand side for every equation ($1 \leq i < j \leq n$).

Applying these elementary operations several times we transform the table to the following form:

$$\begin{array}{cccccc|c} x'_1 & x'_2 & \cdots & x'_k & x'_{k+1} & \cdots & x'_n \\ \hline a'_{11} & a'_{12} & \cdots & a'_{1k} & a'_{k+1} & \cdots & a'_{1n} & b'_1 \\ 0 & a'_{22} & \cdots & a'_{2k} & a'_{2k+1} & \cdots & a'_{2n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{kk} & a'_{kk+1} & \cdots & a'_{kn} & b'_k & (a'_{11}a'_{22}\cdots a'_{kk} \neq 0) \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & b'_{k+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & b'_m \end{array}$$

Corresponding to this, the original linear system is transformed to the following equivalent linear system:

$$\left\{ \begin{array}{l} a'_{11}x'_1 + a'_{12}x'_2 + \cdots + a'_{1k}x'_k + a'_{1k+1}x'_{k+1} + \cdots + a'_{1n}x'_n = b'_1 \\ a'_{22}x'_2 + \cdots + a'_{2k}x'_k + a'_{2k+1}x'_{k+1} + \cdots + a'_{2n}x'_n = b'_2 \\ \vdots \quad \vdots \\ a'_{kk}x'_k + a'_{kk+1}x'_{k+1} + \cdots + a'_{kn}x'_n = b'_k \\ 0x'_{k+1} + \cdots + 0x'_n = b'_{k+1} \\ \vdots \quad \vdots \quad \vdots \\ 0x'_n = b'_m. \end{array} \right.$$

If at least one of b'_{k+1}, \dots, b'_m is not zero, then this system has no solution. When all b'_{k+1}, \dots, b'_m are 0, x'_{k+1}, \dots, x'_n can be arbitrary constants. In particular, when $k = n = m$, it has a unique solution.

Let us see it in the following example. We try to solve

$$\begin{cases} x + 2y + 3z = 6 \\ 2x + 3y + 4z = 9 \\ 3x + 4y + 5z = 10. \end{cases}$$

The table is transformed as follows:

$$\begin{array}{ccc|c} x & y & z & \\ \hline 1 & 2 & 3 & 6 \\ 2 & 3 & 4 & 9 \\ 3 & 4 & 5 & 10 \end{array} \xrightarrow{\text{2nd row} - \text{1st row} \times 2} \begin{array}{ccc|c} x & y & z & \\ \hline 1 & 2 & 3 & 6 \\ 0 & -1 & -2 & -3 \\ 3 & 4 & 5 & 10 \end{array} \quad \begin{array}{ccc|c} x & y & z & \\ \hline 1 & 2 & 3 & 6 \\ 0 & -1 & -2 & -3 \\ 0 & 2 & 4 & -8 \end{array} \xrightarrow{\text{3rd row} - \text{1st row} \times 3} \begin{array}{ccc|c} x & y & z & \\ \hline 1 & 2 & 3 & 6 \\ 0 & -1 & -2 & -3 \\ 0 & 0 & 0 & -2. \end{array} \xrightarrow{\text{3rd row} - \text{2nd row} \times 2} \begin{array}{ccc|c} x & y & z & \\ \hline 1 & 2 & 3 & 6 \\ 0 & -1 & -2 & -3 \\ 0 & 0 & 0 & -2. \end{array}$$

Therefore, the linear system is finally transformed to

$$\begin{cases} x + 2y + 3z = 6 \\ -y - 2z = -3 \\ 0z = -2. \end{cases}$$

This system has no solution. Next, consider the following system of equations:

$$\begin{cases} x + 2y + 3z = 6 \\ 2x + 3y + 4z = 9 \\ 3x + 4y + 5z = 12. \end{cases}$$

Applying elementary operations, the table is transformed in the end to

$$\begin{array}{ccc|c} x & y & z & \\ \hline 1 & 2 & 3 & 6 \\ 0 & -1 & -2 & -3 \\ 0 & 0 & 0 & 0. \end{array}$$

Hence, we have the equivalent linear system

$$\begin{cases} x + 2y + 3z = 6 \\ -y - 2z = -3 \\ 0z = 0. \end{cases}$$

Looking these equations from the bottom, z can be arbitrary constant. The middle equation gives $y = 3 - 2z$, which is then substituted into the top equation to get $x = z$. Thus the solution is

$$\begin{cases} x = z \\ y = 3 - 2z \\ z : \text{arbitrary.} \end{cases}$$

Finally, consider the linear system

$$\begin{cases} x + 2y + 3z = 6 \\ 2x + 3y + z = 9 \\ 3x + y + 2z = 12. \end{cases}$$

The table is transformed as follows:

$$\begin{array}{ccc|c} x & y & z & | & x & y & z & | & x & y & z & | \\ \hline 1 & 2 & 3 & | & 6 & & & | & 1 & 2 & 3 & | & 6 \\ 2 & 3 & 1 & | & 9 & & & | & 0 & -1 & -5 & | & -3 \\ 3 & 1 & 2 & | & 12 & & & | & 3 & 1 & 2 & | & 12 \end{array} \xrightarrow{\substack{\text{2nd row} - \text{1st row} \times 2}} \begin{array}{ccc|c} & & & | \\ & 0 & -1 & -5 & | -3 \\ & 0 & -5 & -7 & | -6 \end{array} \xrightarrow{\substack{\text{3rd row} - \text{1st row} \times 3}} \begin{array}{ccc|c} & & & | \\ & 0 & -1 & -5 & | -3 \\ & 0 & 0 & 18 & | 9 \end{array} \xrightarrow{\substack{\text{3rd row} - \text{2nd row} \times 5}} \begin{array}{ccc|c} & & & | \\ & 0 & 0 & 18 & | 9 \end{array}$$

Hence, we get the equivalent system

$$\begin{cases} x + 2y + 3z = 6 \\ -y - 5z = -3 \\ 18z = 9, \end{cases}$$

and solving this, we have a unique solution $(x, y, z) = \left(\frac{7}{2}, \frac{1}{2}, \frac{1}{2}\right)$.

The method explained above is called the *Gaussian elimination*. The first process of transforming tables is said to be the *forward elimination* and the second process of solving equations from bottom upward is said to be the *backward substitution*.

Using a matrix and vectors a linear system is represented as

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

with

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

If $\mathbf{A} = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n]$, it is also written as

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n = \mathbf{b}.$$

We call A the *coefficient matrix* and $A' = [\mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n \mathbf{b}]$ the *extended coefficient matrix* of the system of equations. With these notations, the system has a solution, if and only if $x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n = \mathbf{b}$ has a solution, if and only if $\mathbf{b} \in \langle \mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \rangle$, if and only if $\text{rank } A = \text{rank } A'$.

We can solve linear systems with SymPy as already explained in Sect. 3.1. We use the `solve` function of the `linalg` module, where a matrix A and a vector b are given as arguments. The system of equations must have a unique solution, which is returned as an array class object.

```
In [1]: from numpy.linalg import solve
solve([[1,2,3],[2,3,1],[3,1,2]],[6,9,12])
```

```
Out[1]: array([3.5, 0.5, 0.5])
```

Exercise 5.10 The following program randomly generates a linear system problem with unknown variables x, y, z . By changing the parameters, it generates a problem with or without solution, or with a unique or infinitely many solutions. It prints out the created problem in L^AT_EX math mode code. It also outputs the solution. Solve the output problem by hand.

Program: prob_eqn.py

```
1 | from numpy.random import seed, choice, shuffle
2 | from sympy import Matrix, latex, solve, zeros
3 | from sympy.abc import x, y, z
4 |
5 | def f(P, m, n):
6 |     while True:
7 |         A = Matrix(choice(P, (3, 4)))
8 |         if A[:, :3].rank() == m and A.rank() == n:
9 |             break
10 |     A, b = A[:, :3], A[:, 3]
11 |     u = Matrix([[x], [y], [z]])
12 |     print(f'{latex(A)}{latex(u)}={latex(b)}')
13 |     print(solve(A*u - b, [x, y, z]))
14 |
15 | seed(1234)
16 | m, n = 2, 2
17 | f(range(2, 10), m, n)
```

Lines 5–13: Variables m and n are the ranks of the coefficient matrix and the extended one, respectively. The elements of the extended coefficient matrix are taken from list P . Function f with these arguments randomly generates a problem and prints it out with the L^AT_EX style. Also the solution is printed out as a dictionary of Python.

Line 16: Assignment $m, n = 3, 3$ gives a linear system with unique solution, assignment $m, n = 2, 2$ gives one with infinitely many solutions, and $m, n = 2, 3$ gives one without solution.



```
\left[\begin{matrix}3 & 2 & 3 \\ 8 & 6 & 2 \\ 7 & 5 & 4\end{matrix}\right]\left[\begin{matrix}x \\ y \\ z\end{matrix}\right]=\left[\begin{matrix}7 \\ 2 \\ 8\end{matrix}\right]
```

Running the above program, we have the linear system

$$\begin{bmatrix} 3 & 2 & 3 \\ 8 & 6 & 2 \\ 7 & 5 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 7 \\ 2 \\ 8 \end{bmatrix}$$

with the Python dictionary `{x: 19 - 7*z, y: 9*z - 25}`, which means that the system has the solution $(x, y) = (9z - 25, 19 - 7z)$, where z is arbitrary.

5.6 Inverse Matrix

Elementary operations can also be used for calculating the inverse of a given matrix. Suppose that a square matrix A is transformed to the unit matrix I by iterating elementary operations of row. This means that there exists a sequence of elementary matrices J_1, J_2, \dots, J_n such that $J_n \cdots J_2 J_1 A = I$, that is,

$$A^{-1} = J_n \cdots J_2 J_1.$$

In order to obtain the inverse matrix using this property, we record the sequence $J_1, J_2 J_1, \dots, J_n \cdots J_2 J_1$ of the matrices in the process of operations. First, place A and I side by side in a table and apply the same sequence of elementary operations on both sides of the table.

We perform this with a simple example. We start by writing down the following table:

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 2 & 3 & 1 & 0 & 1 & 0 \\ 3 & 1 & 2 & 0 & 0 & 1 \end{array} \right].$$

The vertical line in the center is just a marker. The matrix to the left of the line is a given matrix A and to the right is the unit matrix I . We can do it by hand, but let us proceed with the help of SymPy. The starting matrix is put as follows:

```
In [1]: from sympy import Matrix
A = Matrix([[1, 2, 3, 1, 0, 0],
            [2, 3, 1, 0, 1, 0],
            [3, 1, 2, 0, 0, 1]])
```

In order to eliminate the elements at $(2, 1)$ and $(3, 1)$ in the first column, apply the operations “2nd row – 1st row $\times 2$ ” and “3rd row – 1st row $\times 3$ ”.

```
In [2]: A[1, :] -= A[0, :] * 2; A[2, :] -= A[0, :] * 3; A
```

```
Out[2]: Matrix([
[1, 2, 3, 1, 0, 0],
[0, -1, -5, -2, 1, 0],
[0, -5, -7, -3, 0, 1]])
```

Next, to normalize (to set to 1) the pivot at $(2, 2)$, execute the operation “2nd row $\div (-1)$ ”.

```
In [3]: A[1, :] /= -1; A
```

```
Out[3]: Matrix([
[1, 2, 3, 1, 0, 0],
[0, 1, 5, 2, -1, 0],
[0, -5, -7, -3, 0, 1]])
```

Executing “1st row – 2nd row $\times 2$ ” and “3rd row + 2nd row $\times 5$ ” eliminates the elements at $(1, 2)$ and $(3, 2)$ in the second column.

```
In [4]: A[0, :] -= A[1, :] * 2; A[2, :] += A[1, :] * 5; A
Out[4]: Matrix([
[1, 0, -7, -3, 2, 0],
[0, 1, 5, 2, -1, 0],
[0, 0, 18, 7, -5, 1]])
```

Executing “3rd row $\div 18$ ” normalizes the pivot at (3, 3).

```
In [5]: A[2, :] /= 18; A
Out[5]: Matrix([
[1, 0, -7, -3, 2, 0],
[0, 1, 5, 2, -1, 0],
[0, 0, 1, 7/18, -5/18, 1/18]])
```

Finally, execute “1st row + 3rd row $\times 7$ ” and “2nd row – 3rd row $\times 5$ ”, to eliminate the elements at (1, 3) and (2, 3).

```
In [6]: A[0, :] += A[2, :] * 7; A[1, :] -= A[2, :] * 5; A
Out[6]: Matrix([
[1, 0, 0, -5/18, 1/18, 7/18],
[0, 1, 0, 1/18, 7/18, -5/18],
[0, 0, 1, 7/18, -5/18, 1/18]])
```

In this way we obtain the matrix

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & -\frac{5}{18} & \frac{1}{18} & \frac{7}{18} \\ 0 & 1 & 0 & \frac{1}{18} & \frac{7}{18} & -\frac{5}{18} \\ 0 & 0 & 1 & \frac{7}{18} & -\frac{5}{18} & \frac{1}{18} \end{array} \right],$$

where the left side of the vertical line is the unit matrix and the right is the inverse matrix of A .

The difference from elementary operations for ranks, determinants, and linear systems is that the pivot always must be set to 1, and the elements of the matrix to the left of the vertical line not only below the diagonal line but also above the diagonal must be eliminated. Furthermore, only elementary row operations are allowed. We call this algorithm for obtaining the inverse matrix the *sweeping method*. If we get stuck in the middle of the process, that is, no nonzero pivot⁴ is found even by exchanging rows, then the given matrix is not regular and the inverse matrix does not exist.

As we already explained in Sect. 4.4 how to calculate the inverse matrix numerically with NumPy and symbolically with SymPy, use them for practical purposes.

Exercise 5.11 Generate matrices using program `prob_det.py` in Exercise 6.8 and calculate their inverse matrices by the sweeping method.

Exercise 5.12 Write a program using NumPy to calculate the inverse of a matrix of any order by the sweeping method. Compare its calculation time with the function `inv` provided in NumPy for randomly generated matrices of large order.

⁴ Because floating-point calculation cannot avoid numerical errors, the zero judgment of pivot is critical. If it is erroneous, computation of matrix inverse will also be erroneous.

Let A be a square matrix of order n . The (i, j) -minor A_{ij} of A is the matrix of order $n - 1$ which is obtained by removing all $2n - 1$ elements in the i -th row and j -th column from A . The (i, j) -cofactor of A is defined to be $(-1)^{i+j} |A_{ij}|$, and is denoted by Δ_{ij} . The square matrix

$$A' \stackrel{\text{def}}{=} \begin{bmatrix} \Delta_{11} & \Delta_{21} & \cdots & \Delta_{n1} \\ \Delta_{12} & \Delta_{22} & \cdots & \Delta_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ \Delta_{1n} & \Delta_{2n} & \cdots & \Delta_{nn} \end{bmatrix}$$

of order n is called the *cofactor matrix* of A . Then, we have

$$AA' = A'A = \det(A) I.$$

Exercise 5.13 Prove the equalities above for the cofactor matrix.

This means that, if A is a regular matrix, that is, $|A| \neq 0$, A^{-1} is calculated through its cofactor matrix as $A^{-1} = \frac{A'}{|A|}$. Here is a program that numerically confirms this.

Program: `inv.py`

```
In [1]: 1 from numpy import array, linalg, random
2
3 n = 5
4 A = random.randint(0, 10, (n, n))
5 K = [[j for j in range(n) if j != i] for i in range(n)]
6 B = array([[(-1)**(i+j) * linalg.det(A[K[i], :][:, K[j]]) for i in range(n)] for j in range(n)])
7
8 print(A.dot(B/linalg.det(A)))
```

```
[ [ 1.00000000e+00  1.42108547e-14 -1.42108547e-14  9.23705556e-14
   1.59872116e-14]
 [ 3.10862447e-15  1.00000000e+00 -2.13162821e-14  4.97379915e-14
   1.06581410e-14]
 [ 1.77635684e-15  7.99360578e-15  1.00000000e+00  2.22044605e-14
   1.28785871e-14]
 [ 1.77635684e-15  1.06581410e-14  0.00000000e+00  1.00000000e+00
   3.55271368e-15]
 [ 2.22044605e-16  1.06581410e-14 -5.32907052e-15  1.06581410e-14
   1.00000000e+00]]
```

Calculating the inverse matrix by this method is not realistic because the calculation becomes more complicated and time-consuming as the size of the matrix increases. However, the existence of such a formula for an inverse matrix is a very important mathematical property. It also gives a formula for a system of equations. The solution of a linear system $Ax = b$ for regular matrix A is expressed as $x = A^{-1}b = \frac{1}{|A|}A'b$, from which we can derive well-known **Cramer's formula**.⁵

⁵ The reader interested in this formula is advised to consult the textbooks on linear algebra mentioned in the afterword of this book.

Exercise 5.14 Give the formula for the solution of the system of equations $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

where A is regular. That is, express explicitly the solutions for unknowns x_1, x_2, x_3 in terms of coefficients a_{ij} and b_i ($1 \leq i, j \leq 3$).



Inner Product and Fourier Expansion

6

In this chapter, we consider a scalar-valued binary operation on a linear space called an inner product. It leads to the concepts of the length of a vector and the orthogonality between vectors, which give to a linear space the structure of Euclidean geometry. Also, we learn the meaning of orthogonality between functions in a function space. In Chap. 2, we considered the function space of sounds and made a chord on a scale as a sum of pure tones. In this chapter using the inner product in the space of sounds, we consider the opposite task. We extract pure tones from a given sound using orthogonality of vectors and observe their ratio, which is called the power spectrum of the sound. This technique is known as Fourier analysis, which provides us with various interesting topics in signal processing, probability, statistics, and others. In Python many tools providing numerical and symbolical calculations for it are available.

6.1 Norm and Inner Product

Let V be a linear space over \mathbb{K} . Suppose that a real value $\|x\| \in \mathbb{R}$ is defined for every $x \in V$ and satisfies the following properties called **axioms of the norm**:

- N1. **positivity**: $\|x\| \geq 0$ and the equality holds if and only if $x = \mathbf{0}$,
- N2. **absolute homogeneity**: $\|ax\| = |a| \|x\|$,
- N3. **sub-additivity or triangular inequality**: $\|x + y\| \leq \|x\| + \|y\|$,

for any $a \in \mathbb{K}$ and $x, y \in V$. We call $\|x\|$ the *norm* of x and the function $x \mapsto \|x\|$ is said to be a norm on V . A linear space equipped with norm is called a *normed space*.

We consider that the norm is a measure of the size (or length) of a vector and the amount $\|x - y\|$ is the distance between vectors x and y in the space. The name triangular inequality comes from the fact that for any triangle the sum of the lengths of two sides must be greater than the length of the remaining side.

For $x = (x_1, x_2, \dots, x_n) \in \mathbb{K}^n$, define

$$\|x\|_1 \stackrel{\text{def}}{=} |x_1| + |x_2| + \cdots + |x_n|$$

and

$$\|\mathbf{x}\|_{\infty} \stackrel{\text{def}}{=} \max \{|x_1|, |x_2|, \dots, |x_n|\}.$$

Then, $\|\cdot\|_1$ and $\|\cdot\|_{\infty}$ are norms on the n -dimensional vector space \mathbb{K}^n , and are called the l^1 -norm and l^{∞} -norm respectively.

Exercise 6.1 Prove that $\|\cdot\|_1$ and $\|\cdot\|_{\infty}$ satisfy axioms N1–N3 of norm.

If a value $\langle \mathbf{x} \mid \mathbf{y} \rangle \in \mathbb{K}$ is defined for every pair $\mathbf{x}, \mathbf{y} \in V$ and satisfies the following properties called **axioms of the inner product**, then we say that the binary function $(\mathbf{x}, \mathbf{y}) \mapsto \langle \mathbf{x} \mid \mathbf{y} \rangle$ is an *inner product* on V , or say that $\langle \mathbf{x} \mid \mathbf{y} \rangle$ is the inner product of \mathbf{x} and \mathbf{y} :

- I1. **positivity**: $\langle \mathbf{x} \mid \mathbf{x} \rangle \geq 0$, where equality holds if and only if $\mathbf{x} = \mathbf{0}$.
- I2. **Hermitian property**: $\langle \mathbf{y} \mid \mathbf{x} \rangle = \overline{\langle \mathbf{x} \mid \mathbf{y} \rangle}$.
- I3. **homogeneity**: $\langle \mathbf{x} \mid a\mathbf{y} \rangle = a\langle \mathbf{x} \mid \mathbf{y} \rangle$.
- I4. **additivity or distributive law**: $\langle \mathbf{x} \mid \mathbf{y} + \mathbf{z} \rangle = \langle \mathbf{x} \mid \mathbf{y} \rangle + \langle \mathbf{x} \mid \mathbf{z} \rangle$.

A linear space equipped with an inner product is called an *inner product space*.

\mathbb{K}^n is an inner product space with the binary function defined by

$$\langle \mathbf{x} \mid \mathbf{y} \rangle \stackrel{\text{def}}{=} \mathbf{x}^* \mathbf{y} = \begin{bmatrix} \overline{x_1} & \overline{x_2} & \cdots & \overline{x_n} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^n \overline{x_i} y_i$$

for $\mathbf{x}, \mathbf{y} \in \mathbb{K}^n$, where vectors in \mathbb{K}^n are treated as column vectors. When $\mathbb{K} = \mathbb{R}$, it becomes simply $\langle \mathbf{x} \mid \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \sum_{i=1}^n x_i y_i$. We call this inner product on \mathbb{K}^n the *standard inner product*.

Exercise 6.2 Show that the standard inner product satisfies axioms of the inner product.

Exercise 6.3 Prove the following properties by using axioms I1–I4:

- I5. **linearity**: $\langle \mathbf{x} \mid a\mathbf{y} + b\mathbf{z} \rangle = a\langle \mathbf{x} \mid \mathbf{y} \rangle + b\langle \mathbf{x} \mid \mathbf{z} \rangle$.
- I6. **conjugate homogeneity**: $\langle a\mathbf{x} \mid \mathbf{y} \rangle = \bar{a}\langle \mathbf{x} \mid \mathbf{y} \rangle$.
- I7. **additivity or distributive law**: $\langle \mathbf{x} + \mathbf{y} \mid \mathbf{z} \rangle = \langle \mathbf{x} \mid \mathbf{z} \rangle + \langle \mathbf{y} \mid \mathbf{z} \rangle$.
- I8. **conjugate linearity**: $\langle a\mathbf{x} + b\mathbf{y} \mid \mathbf{z} \rangle = \bar{a}\langle \mathbf{x} \mid \mathbf{z} \rangle + \bar{b}\langle \mathbf{y} \mid \mathbf{z} \rangle$.

When $\mathbb{K} = \mathbb{R}$, properties I2, I6, and I8 can be rewritten as follows, respectively:

- I2'. **symmetry**: $\langle \mathbf{y} \mid \mathbf{x} \rangle = \langle \mathbf{x} \mid \mathbf{y} \rangle$.
- I6'. **homogeneity**: $\langle a\mathbf{x} \mid \mathbf{y} \rangle = a\langle \mathbf{x} \mid \mathbf{y} \rangle$.
- I8'. **linearity**: $\langle a\mathbf{x} + b\mathbf{y} \mid \mathbf{z} \rangle = a\langle \mathbf{x} \mid \mathbf{z} \rangle + b\langle \mathbf{y} \mid \mathbf{z} \rangle$.

Let V be an inner product space. Define

$$\|x\| \stackrel{\text{def}}{=} \sqrt{\langle x | x \rangle}$$

for $x \in V$.

Exercise 6.4 Prove the identity

$$\|x + y\|^2 = \|x\|^2 + 2\operatorname{Re}\langle x | y \rangle + \|y\|^2.$$

When $\mathbb{K} = \mathbb{R}$, it becomes

$$\|x + y\|^2 = \|x\|^2 + 2\langle x | y \rangle + \|y\|^2.$$

Theorem 6.1 (Schwarz's inequality) *Let V be an inner product space. We have the inequality*

$$|\langle x | y \rangle| \leq \|x\| \|y\|$$

for $x, y \in V$, where the equality holds if and only if x or y is a scalar multiple of the other; that is, $\{x, y\}$ is linearly dependent.

Proof If $x = \mathbf{0}$, then both sides are 0, and also $x = 0y$. Hence the assertion of the theorem is valid. Suppose $x \neq \mathbf{0}$, and let $t = \langle x | y \rangle / \|x\|^2$. Using I1, I4, I8, and I2, we have

$$\begin{aligned} 0 &\leq \langle y - tx | y - tx \rangle = \langle y | y \rangle - t\langle y | x \rangle - \bar{t}\langle x | y \rangle + \bar{t}t\langle x | x \rangle \\ &= \|y\|^2 - \frac{\langle x | y \rangle \langle y | x \rangle}{\|x\|^2} = \|y\|^2 - \frac{|\langle x | y \rangle|^2}{\|x\|^2}. \end{aligned}$$

The desired inequality follows from this. If the equality holds, then $\langle y - tx | y - tx \rangle = 0$, and so $y = tx$. Conversely, if $y = tx$ for some $t \in \mathbb{K}$, then

$$|\langle x | y \rangle| = |t\langle x | x \rangle| = \sqrt{t\langle x | x \rangle^2} = \sqrt{\langle x | x \rangle \langle tx | tx \rangle} = \|x\| \|y\|. \quad \blacksquare$$

From this inequality, we can show that an inner product space is a normed space with the norm $\|\cdot\|$ defined above. N1 and N2 are easily driven from axioms I1, I2, and I3 of the inner product. Since

$$\begin{aligned} \|x + y\|^2 &= \|x\|^2 + 2\operatorname{Re}(\langle x | y \rangle) + \|y\|^2 && \text{(Exercise 5.4)} \\ &\leq \|x\|^2 + 2|\langle x | y \rangle| + \|y\|^2 && (|\operatorname{Re} z| \leq |z| \text{ for complex } z) \\ &\leq \|x\|^2 + 2\|x\| \|y\| + \|y\|^2 && \text{(by Schwarz's inequality)} \\ &\leq (\|x\| + \|y\|)^2, \end{aligned}$$

we can get N3 by taking the square root of both sides.

For $x, y \in V$ we say that x is *orthogonal* to y , if $\langle x | y \rangle = 0$, and denote it by $x \perp y$. We see $x \perp y \Leftrightarrow y \perp x$ by I2. For $x \in V$ and a subset $S \subseteq V$, if x is orthogonal to all vectors in S , we say that x is orthogonal to S and denote it by $x \perp S$.

Exercise 6.5 For $x, y \in V$, prove that $x = y$ if and only if $\langle x | z \rangle = \langle y | z \rangle$ holds for every $z \in V$.

Exercise 6.6 Show the following formulas:

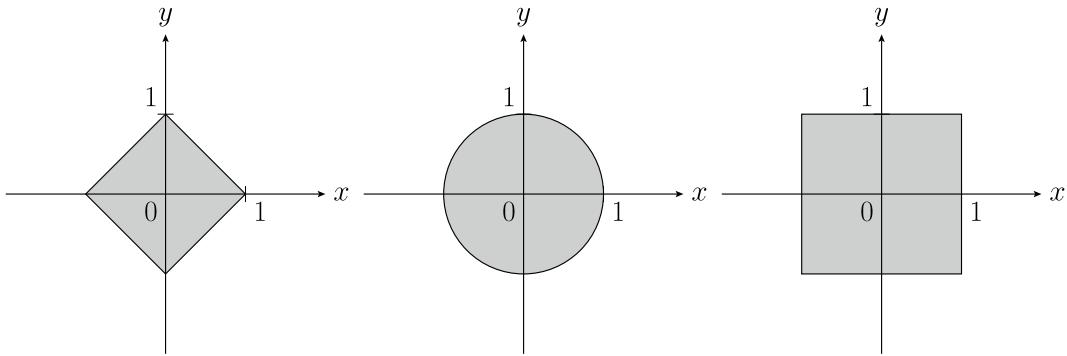


Fig. 6.1 The unit disk for l^1 , l^2 , and l^∞ from left to right

(1) **Pythagorean theorem:**

$$\mathbf{x} \perp \mathbf{y} \Rightarrow \|\mathbf{x} + \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2.$$

(2) **parallelogram law:**

$$\|\mathbf{x} + \mathbf{y}\|^2 + \|\mathbf{x} - \mathbf{y}\|^2 = 2\|\mathbf{x}\|^2 + 2\|\mathbf{y}\|^2.$$

(3) **polarization identity:**

$$\langle \mathbf{x} | \mathbf{y} \rangle = \frac{1}{4} \left(\|\mathbf{x} + \mathbf{y}\|^2 - \|\mathbf{x} - \mathbf{y}\|^2 + i \|\mathbf{x} + i\mathbf{y}\|^2 - i \|\mathbf{x} - i\mathbf{y}\|^2 \right),$$

and when $\mathbb{K} = \mathbb{R}$,

$$\langle \mathbf{x} | \mathbf{y} \rangle = \frac{1}{4} \left(\|\mathbf{x} + \mathbf{y}\|^2 - \|\mathbf{x} - \mathbf{y}\|^2 \right).$$

On \mathbb{K}^n we have another norm $\|\cdot\|_2$ than $\|\cdot\|_1$ or $\|\cdot\|_\infty$ induced from the standard inner product:

$$\|\mathbf{x}\|_2 \stackrel{\text{def}}{=} \sqrt{\mathbf{x}^* \mathbf{x}} = \left(\sum_{i=1}^n |x_i|^2 \right)^{1/2}$$

for $\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{K}^n$. We call this norm the *l^2 -norm* or the *Euclidean norm*. The unit disk (the set of all vectors whose norm is less than or equal to 1) on \mathbb{R}^2 with respect to each of these three norms is illustrated in Fig. 6.1.

NumPy has several functions and methods for calculating the standard inner product $\mathbf{x}^T \mathbf{y}$ on \mathbb{R}^n such as the `inner` function, `dot` function, and `dot` method of array class. When we calculate the standard inner product $\mathbf{x}^* \mathbf{y}$ on \mathbb{C}^n , we must also use the complex conjugate operation, but if we use the `vdot` function,¹ we do not need it.

```
In [1]: from numpy import array, dot, inner, conj, vdot
x, y = array([1+2j, 3+4j]), array([4+3j, 2+1j]); x, y
```

```
Out[1]: (array([1.+2.j, 3.+4.j]), array([4.+3.j, 2.+1.j]))
```

¹ A method corresponding to the `vdot` function is not defined as a method of `array` class.

```
In [2]: conj(x).dot(y), dot(x.conj(), y), inner(conj(x), y)
```

```
Out[2]: ((20-10j), (20-10j), (20-10j))
```

```
In [3]: vdot(x, y), vdot(1j*x, y), vdot(x, 1j*y)
```

```
Out[3]: ((20-10j), (-10-20j), (10+20j))
```

We use the function `norm` defined in the module `linalg` of NumPy to find the norms.

```
In [4]: from numpy.linalg import norm
norm(x), norm(y)
```

```
Out[4]: (5.477225575051661, 5.477225575051661)
```

If we give 2 or nothing to the keyword argument `ord` of this function, we get the l^2 -norm, and if we give 1 (resp. `inf`), we get the l^1 -norm (resp. l^∞ -norm). The name `inf` in NumPy means ∞ in mathematics.

```
In [5]: norm([1, 2, 3], ord=1)
```

```
Out[5]: 6.0
```

```
In [6]: norm([1, 2, 3], ord=2)
```

```
Out[6]: 3.7416573867739413
```

```
In [7]: from numpy import inf
norm([1, 2, 3], ord=inf)
```

```
Out[7]: 3.0
```

Exercise 6.7 On the standard inner product and the Euclidean norm in \mathbb{R}^2 , prove

$$\langle \mathbf{x} | \mathbf{y} \rangle = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \omega,$$

where ω is the angle between vectors \mathbf{x} and \mathbf{y} in \mathbb{R}^2 .

6.2 Orthonormal Systems and Fourier Transform

Let V be an inner product space and let A be a (not necessarily finite) subset of V . We say that A is linearly independent if any finite subset of A is linearly independent. If A does not contain the zero vector and any two distinct vectors in it are orthogonal, we say that A is an *orthogonal system*. If moreover every vector in A is of norm 1, we call it an *orthonormal system* or *normal orthogonal system*.

Let A be an orthogonal system and let $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ be any finite number of distinct elements of A . If

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n = \mathbf{0}$$

for $x_1, x_2, \dots, x_n \in \mathbb{K}$, then taking inner products of both sides with \mathbf{a}_i and expanding the left-hand side, we have

$$x_1 \langle \mathbf{a}_i | \mathbf{a}_1 \rangle + x_2 \langle \mathbf{a}_i | \mathbf{a}_2 \rangle + \cdots + x_n \langle \mathbf{a}_i | \mathbf{a}_n \rangle = 0$$

for every $i = 1, 2, \dots, n$. Because $\langle \mathbf{a}_i | \mathbf{a}_j \rangle = 0$ for any $i \neq j$, it follows that $x_i = 0$ for every i . Hence, we see that an orthogonal system is linearly independent.

Let $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\} \subseteq V$ be a finite orthonormal system in V and let W be the subspace generated by it. For any $\mathbf{x} \in V$ we define $\mathbf{y} \in W$ by

$$\mathbf{y} \stackrel{\text{def}}{=} \sum_{i=1}^n \langle \mathbf{e}_i | \mathbf{x} \rangle \mathbf{e}_i.$$

Since

$$\langle \mathbf{e}_j | \mathbf{x} - \mathbf{y} \rangle = \langle \mathbf{e}_j | \mathbf{x} \rangle - \sum_{i=1}^n \langle \mathbf{e}_j | \mathbf{e}_i \rangle \langle \mathbf{e}_i | \mathbf{x} \rangle = \langle \mathbf{e}_j | \mathbf{x} \rangle - \langle \mathbf{e}_j | \mathbf{x} \rangle = 0$$

for each $j = 1, 2, \dots, n$, we find that $\mathbf{x} - \mathbf{y}$ is orthogonal to W . On the other hand, let \mathbf{y}' be any vector in W . Then, because $\mathbf{x} - \mathbf{y}$ is orthogonal to $\mathbf{y} - \mathbf{y}' \in W$, by the Pythagorean theorem we have

$$\|\mathbf{x} - \mathbf{y}'\|^2 = \|\mathbf{x} - \mathbf{y} + \mathbf{y} - \mathbf{y}'\|^2 = \|\mathbf{x} - \mathbf{y}\|^2 + \|\mathbf{y} - \mathbf{y}'\|^2 \geq \|\mathbf{x} - \mathbf{y}\|^2.$$

This means that \mathbf{y} is a vector of W with the shortest distance from \mathbf{x} . Also, such \mathbf{y} is unique, because if the equality holds above, then we must have $\|\mathbf{y} - \mathbf{y}'\| = 0$, that is, $\mathbf{y}' = \mathbf{y}$. We denote this unique \mathbf{y} by $\text{proj}_W(\mathbf{x})$ and call it the *orthogonal projection* of \mathbf{x} onto W .

Now we consider a nonzero subspace W of V . Pick up a nonzero vector $\mathbf{x} \in W$ and put $\mathbf{e}_1 \stackrel{\text{def}}{=} \frac{\mathbf{x}}{\|\mathbf{x}\|}$, then \mathbf{e}_1 is of norm 1 and has the same direction as \mathbf{x} . This operation is said to *normalize* \mathbf{x} . The singleton set $\{\mathbf{e}_1\}$ is an orthonormal system contained in W . This is the first step of induction. Suppose that after the k -th step we have an orthonormal system $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_k\}$ in W . If this system does not generate W , put $W' \stackrel{\text{def}}{=} \langle \mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_k \rangle$. Pick up any \mathbf{x} from $W \setminus W'$, then $\mathbf{y} = \mathbf{x} - \text{proj}_{W'}(\mathbf{x})$ is a nonzero vector orthogonal to W' . Normalizing \mathbf{y} to $\mathbf{e}_{k+1} = \mathbf{y}/\|\mathbf{y}\|$, we obtain an orthonormal system $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_k, \mathbf{e}_{k+1}\}$ in W . This induction will terminate whenever W is finite dimensional and at the final stage we have an orthonormal system generating W . This system is a basis of W and we call it an *orthonormal basis* of W . This method is called the *Gram–Schmidt orthogonalization*.

The discussion above shows that every finite-dimensional linear space has an orthonormal basis. Furthermore, we have the following important facts:

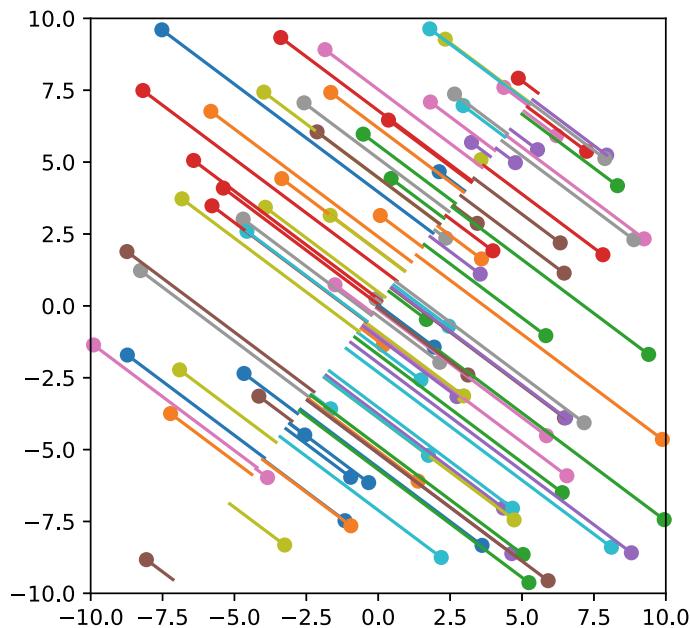
1. The orthogonal projection $\text{proj}_W : V \rightarrow V$ with $\text{range}(\text{proj}_W) = W$ exists for any finite-dimensional subspace W of V ²
2. There exists an algorithm³ to get an orthonormal basis of the subspace generated by any finite set of vectors.

Let W be a finite-dimensional subspace of V . From the discussion above we see the following statements:

² W must be finite dimensional but V can be infinite dimensional in general. If W is infinite dimensional, it may not have the orthogonal projection.

³ A procedure that achieves a certain purpose for a finite number of operation steps is called an *algorithm*.

Fig. 6.2 Orthogonal projection onto a line through the origin



- The expression $\mathbf{proj}_W(x) = \sum_{i=1}^n \langle e_i \mid x \rangle e_i$ does not depend on the choice of orthonormal basis $\{e_1, e_2, \dots, e_n\}$ of W .
 - $x - \mathbf{proj}_W(x) \perp W$.
 - $\|x - \mathbf{proj}_W(x)\| = \min_{y \in W} \|x - y\|$.

Exercise 6.8 Show that the orthogonal projection $\text{proj}_W : V \rightarrow V$ is a linear mapping.

Exercise 6.9 Let a and b be real numbers with $a^2 + b^2 = 1$ and consider the subspace $W = \left\{ \begin{bmatrix} ax \\ bx \end{bmatrix} \mid x \in \mathbb{R} \right\}$ of \mathbb{R}^2 generated by $e = \begin{bmatrix} a \\ b \end{bmatrix}$. Prove that the representation matrix of the linear mapping proj_W on the standard basis of \mathbb{R}^2 is given by $e e^T = \begin{bmatrix} a^2 & ab \\ ab & b^2 \end{bmatrix}$.

The program `proj.py` below, choosing $a = 3/5$ and $b = 4/5$ in Exercise 6.9, draws Fig. 6.2 which illustrates the correspondence between points x of \mathbb{R}^2 randomly picked up and their orthogonal projections $\text{proj}_W(x)$ on W .

Program: proj.py

```
In [1]: 1 from numpy import random, array, inner, sqrt  
2 import matplotlib.pyplot as plt  
3  
4 random.seed(2021)  
5 v = array([3, 4])  
6 e = v / sqrt(inner(v, v))  
7 plt.axis('scaled'), plt.xlim(-10, 10), plt.ylim(-10, 10)  
8 for n in range(100):  
9     x = random.uniform(-10, 10, 2)  
10    plt.scatter(*x)  
11    y = inner(e, x) * e  
12    #P = dot(e.reshape((2, 1)), e.reshape((1, 2)))
```

In [1]:

```

13 |     #y = dot(P, x)
14 |     plt.plot(*zip(x, y))
15 |     plt.show()

```

Line 10: `plt.scatter(*x)` draws a dot at $(x[0], v[1])$.

Lines 11–13: If we use the representation matrix in Exercise 5.9, comment out Line 11 and uncomment Lines 12 and 13, then we should have the same result.

Line 14: The function `zip(x, y)` generates the list $[[x[0], y[0]], [x[1], y[1]]]$ and `plt.plot(*zip(x, y))` draws the line segment connecting x and y .

Let $\{e_1, e_2, \dots, e_n\}$ be an orthonormal basis of V , that is, it is orthonormal and generates V . Because for any $x \in V$ the nearest vector to it in V is x itself, the orthogonal projection proj_V is the identity mapping. Hence, x can be expanded as

$$x = \sum_{i=1}^n \langle e_i | x \rangle e_i.$$

It is called the *Fourier expansion* of x and each $x_i = \langle e_i | x \rangle$ is called the *Fourier coefficient*. For another vector $y \in V$ with Fourier coefficients y_1, y_2, \dots, y_n , we have

$$\langle x | y \rangle = \left\langle \sum_{i=1}^n x_i e_i \mid \sum_{j=1}^n y_j e_j \right\rangle = \sum_{i=1}^n \sum_{j=1}^n \overline{x_i} y_j \langle e_i | e_j \rangle = \sum_{i=1}^n \overline{x_i} y_i.$$

In particular, letting $x = y$, we have $\|x\|^2 = \sum_{i=1}^n |x_i|^2$. We summarize these results in the following:

Theorem 6.2 Let $\{e_1, e_2, \dots, e_n\}$ be an orthonormal basis of V . For any $x, y \in V$ we have

1. **Fourier expansion:** $x = \sum_{i=1}^n \langle e_i | x \rangle e_i$.
2. **Parseval's identity:** $\langle x | y \rangle = \sum_{i=1}^n \langle x | e_i \rangle \langle e_i | y \rangle$.
3. **Riesz–Fischer identity:** $\|x\|^2 = \sum_{i=1}^n |\langle e_i | x \rangle|^2$.

This theorem means that every finite-dimensional inner product space is isomorphic to \mathbb{K}^n equipped with the standard inner product and the Euclidean norm not only as a linear space but also as an inner product space and as a normed space.

Exercise 6.10 Show the inequality $\|\text{proj}_W(x)\| \leq \|x\|$ for any vector $x \in V$ and any finite-dimensional subspace W of V .

We explain the Gram–Schmidt orthogonalization in the following program using NumPy.

Program: gram_schmidt.py

```
In [1]: 1 from numpy import array, vdot, sqrt
2
3 def proj(x, E, inner=vdot):
4     return sum([inner(e, x) * e for e in E])
5
6 def gram_schmidt(A, inner=vdot):
7     E = []
8     while A != []:
9         a = array(A.pop(0))
10        b = a - proj(a, E, inner)
11        normb = sqrt(inner(b, b))
12        if normb >= 1.0e-15:
13            E.append(b / normb)
14    return E
15
16 if __name__ == '__main__':
17     A = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
18     E = gram_schmidt(A)
19     for n, e in enumerate(E):
20         print(f'e{n+1} = {e}')
21     print(array([[vdot(e1, e2) for e2 in E] for e1 in E]))
```

Lines 3,4: Define the function `proj(x, E, inner=vdot)` which returns the orthogonal projection of x for orthonormal basis E with respect to inner product `inner`. The default inner product is the standard inner product function `vdot`. In the later chapters, we will use this program as a library and consider other inner products.

Lines 6–14: Define a function `gram_schmidt(A, inner=vdot)` which performs the Gram–Schmidt orthogonalization. The argument A is a list of vectors.⁴ Starting with empty list E , Lines 9–13 repeat operations of removing one vector from A , orthonormalizing it with respect to E , and adding it to E . The method `A.pop(0)` removes the first element of A and returns it. Set it to a as an array. Orthogonalize a and put it in b . If b is not zero, then append the normalized b at the end of E , otherwise do nothing. Decide that b is 0 if the norm `normb` of b is less than 10^{-15} . This is because `normb` may contain computational errors. When A becomes empty, return E .

Lines 16–21: These will be executed when this program is run as a main program, not as a library. We apply the orthogonalization to the set $A = \{(1, 2, 3), (2, 3, 4), (3, 4, 5)\}$ of vectors in \mathbb{R}^3 .



```
e1 = [0.26726124 0.53452248 0.80178373]
e2 = [ 0.87287156 0.21821789 -0.43643578]
[[1.0000000e+00 6.51017134e-17]
 [6.51017134e-17 1.0000000e+00]]
```

The rank of A is equal to 2. Therefore, the obtained orthonormal system consists of two vectors. The inner product of the vectors that should be orthogonal is not exactly 0, with error about 10^{-16} .

Using libraries of VPython and Matplotlib, let us experiment with the orthogonal projection of a 3D figure onto a 2D plane (Fig. 6.3).

Program: proj2d.py

```
In [1]: 1 from vpython import proj, color, curve, vec, hat, box, arrow, \
2     label
3
4 def proj2d(x, e): return x - proj(x, e)
5
6 def draw_cube(o, x, y, z):
7     axes, cols = [x, y, z], [color.red, color.green, color.blue]
```

⁴The order of vectors may affect the result of an orthonormal system.

```
In [1]: 8     planes = [(y, z), (z, x), (x, y)]
9     for ax, col, p in zip(axes, cols, planes):
10        face = [o, o + p[0], o + p[0] + p[1], o + p[1]]
11        for vertex in face:
12            curve(pos=[vertex, vertex + ax], color=col)
13
14 o, e, u = vec(0, 0, 0), hat(vec(1, 2, 3)), vec(5, 5, 5)
15 x, y, z = vec(1, 0, 0), vec(0, 1, 0), vec(0, 0, 1)
16 box(pos=-0.1 * e, axis=e, up=proj2d(y, e),
17      width=20, height=20, length=0.1, opacity=0.5)
18 arrow(axis=3 * e)
19 for ax, lbl in [(x, 'x'), (y, 'y'), (z, 'z')]:
20    curve(pos=[-5 * ax, 10 * ax], color=vec(1, 1, 1) - ax)
21    label(pos=10 * ax, text=lbl)
22    curve(pos=[proj2d(-5 * ax, e), proj2d(10 * ax, e)], color=ax)
23    label(pos=proj2d(10 * ax, e), text=f"{lbl}'")
24 c1 = [u, 2*x, 2*y, 2*z]
25 c2 = [proj2d(v, e) for v in c1]
26 draw_cube(*c1), draw_cube(*c2)
```

Lines 1,2: VPython has the function `hat` to normalize a vector and function `proj` to project a vector orthogonally upon a line through the origin. Sometimes a line becomes too long and the program is difficult to read. In this case, a backslash \ can be placed at the end of a line to connect it to the next line.

Line 4: `proj2d(x, e)` is the orthogonal projection of `x` onto a plane whose normal vector (the vector of norm 1 orthogonal to the plane) is `e`.

Lines 6–12: This function draws the figure specified by argument `c`.

Line 14: `o` is the origin, `e` is the normal vector of the plane for projection, and `u` is the position of the image.

Line 15: Three unit vectors on the coordinate axes.

Lines 16–18: Draw the plane and its normal vector. The plane is drawn as a semi-transparent rectangular parallelepiped with almost no thickness.

Lines 19–23: Draw three axes and their projections onto the plane with labels and colors.

Line 24: `c1` is a list of vectors translated from `c` by `u`.

Line 25: `c2` is a list of vectors projected from `c1` onto the plane.

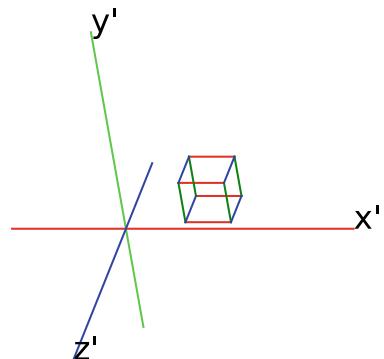
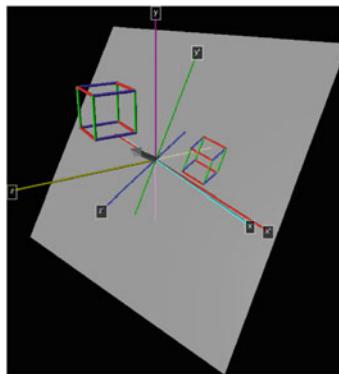
Line 26: Draw the figures specified by `c1` and `c2` with function `draw_fig`.

Next we transform the figure (Fig. 6.3, left) drawn by VPython on a 2D plane lying in 3D space to a figure (Fig. 6.3, right) in \mathbb{R}^2 using Matplotlib. For this purpose, we choose an orthonormal system on the plane in space.

Program: screen.py

```
In [1]: 1  from numpy import array
2  import matplotlib.pyplot as plt
3  from gram_schmidt import gram_schmidt
4
5  def curve(pos, col=(0, 0, 0)):
6      plt.plot(*zip(*pos), color=col)
7
8  def draw_cube(o, x, y, z):
9      axes, cols = [x, y, z], ['r', 'g', 'b']
10     planes = [(y, z), (z, x), (x, y)]
11     for ax, col, p in zip(axes, cols, planes):
12        face = [o, o + p[0], o + p[0] + p[1], o + p[1]]
13        for vertex in face:
14            curve(pos=[vertex, vertex + ax], col=col)
15
16 o, e, u = array([0, 0, 0]), array([1, 2, 3]), array([5, 5, 5])
```

Fig. 6.3 Orthogonal projection of 3D figure onto 2D plane



```
In [1]: 17 x, y, z = array([1, 0, 0]), array([0, 1, 0]), array([0, 0, 1])
18 E = array(gram_schmidt([e, x, y, z])[1:])
19 plt.axis('equal'), plt.axis('off')
20 for ax, lbl in [(x, "x'"), (y, "y'"), (z, "z'")]:
21     curve(pos=[E.dot(-5 * ax), E.dot(10 * ax)], col=ax)
22     plt.text(*E.dot(10 * ax), lbl, fontsize=24)
23 c1 = [u, 2*x, 2*y, 2*z]
24 c2 = [E.dot(v) for v in c1]
25 draw_cube(*c2), plt.show()
```

Lines 5–14: Define a function with the same specifications as `draw_fig` in `proj2d.py`.

Lines 16, 17: Define the same vectors as in `proj2d.py`. However, since there is no convenient function such as `hat` of VPython, use unnormalized vector `a` for `e`. We leave its normalization to the Gram–Schmidt orthogonalization.

Line 18: Applying the Gram–Schmidt orthogonalization method to the vectors `a`, `x`, `y`, and `z`, get an orthonormal system of \mathbb{R}^3 . The first vector is the normal vector of the 2D plane, and the other two vectors form an orthonormal system on the plane. `E` is an array expressing a $(2, 3)$ -matrix whose row vectors are these two vectors. So, `E.dot(v)` is the vector in \mathbb{R}^2 which represents the vector obtained by projecting `v` onto the plain.

Line 19: Set attributes of the axes of the 2D plane.

Lines 20–25: Almost the same as Lines 18–25 of `proj2d.py`. Matplotlib is used for drawing.

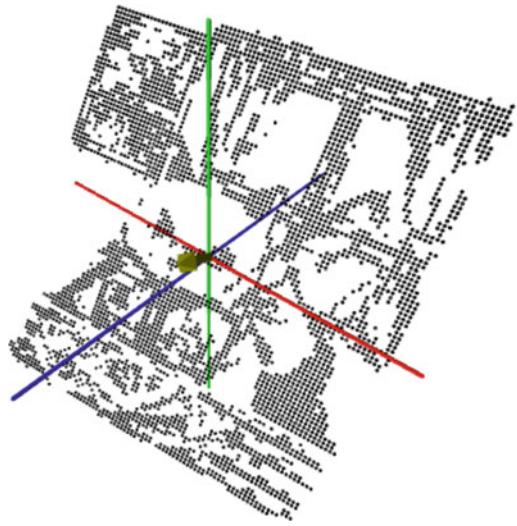
Exercise 6.11 Use VPython to paste the image data created in Sect. 1.8 onto a 2D plane in 3D space without distortion (Fig. 6.4).

Let V and W be inner product spaces over \mathbb{K} with inner products $\langle \cdot | \cdot \rangle_1$ and $\langle \cdot | \cdot \rangle_2$, respectively. Assume that V is finite dimensional and $\{v_1, v_2, \dots, v_n\}$ is an orthonormal basis of V . For a linear mapping $f : V \rightarrow W$ we define a mapping $f^* : W \rightarrow V$ by

$$f^*(y) \stackrel{\text{def}}{=} \sum_{i=1}^n \langle f(v_i) | y \rangle_2 v_i$$

for $y \in W$. Then, f^* is also a linear mapping, and the equality $\langle f^*(y) | x \rangle_1 = \langle y | f(x) \rangle_2$ holds for any $x \in V$ and $y \in W$, because

Fig. 6.4 Pasting the image onto a plane



$$\begin{aligned}\langle f^*(y) | x \rangle_1 &= \left\langle \sum_{i=1}^n \langle f(v_i) | y \rangle_2 v_i \mid x \right\rangle_1 = \sum_{i=1}^n \langle y \mid f(v_i) \rangle_2 \langle v_i \mid x \rangle_1 \\ &= \left\langle y \mid f \left(\sum_{i=1}^n \langle v_i \mid x \rangle_1 v_i \right) \right\rangle_2 = \langle y \mid f(x) \rangle_2.\end{aligned}$$

We call f^* the *conjugate linear mapping* of f .

Exercise 6.12 Prove the following assertions:

- (1) f^* is a linear mapping and does not depend on the choice of orthonormal basis of V .
- (2) Assume that W is also finite dimensional. If A is the representation matrix of f on orthonormal bases of V and W , then the representation matrix of f^* on the same bases is the adjoint matrix A^* of A , and $\langle A^*y | x \rangle_1 = \langle y | Ax \rangle_2$ holds for any $x \in V$ and $y \in W$.

For a subset S of a linear space V with inner product $\langle \cdot | \cdot \rangle$, we define another subset S^\perp by

$$S^\perp \stackrel{\text{def}}{=} \{v \in V \mid \langle x | v \rangle = 0 \text{ for all } x \in S\}.$$

Then S^\perp is a subspace of V and we call it the *orthocomplemented subspace* or *orthogonal complement* of S in V . In fact, $\mathbf{0} \in S^\perp$ because $\langle x | \mathbf{0} \rangle = 0$ for any $x \in S$. In addition, for $a, b \in \mathbb{K}$ and $u, v \in S^\perp$ we see $au + bv \in S^\perp$ because

$$\langle x | au + bv \rangle = a\langle x | u \rangle + b\langle x | v \rangle = 0$$

for any x . Moreover, if W is a finite-dimensional subspace of V , then the following equalities hold:

1. $V = W \oplus W^\perp$,
2. $W^{\perp\perp} = W$.

If $v \in W \cap W^\perp$, then $\langle v | v \rangle = 0$, so $v = \mathbf{0}$. Thus, we have $W \cap W^\perp = \{\mathbf{0}\}$. Moreover, for any $v \in V$ let $w = \text{proj}_W(v)$ and $u = v - w$. Then, $v = w + u$ and $w \in W$ and $u \in W^\perp$. This implies $W \cup W^\perp$

generates V . Hence, $V = W \oplus W^\perp$ by Exercise 3.6. Next, let us show 2. Because $\langle \overline{\mathbf{w}} \mid \mathbf{v} \rangle = \langle \mathbf{v} \mid \mathbf{w} \rangle = 0$ for any $\mathbf{w} \in W$ and $\mathbf{v} \in W^\perp$, we see $W \subset W^{\perp\perp}$. Conversely, any $\mathbf{v} \in W^{\perp\perp}$ is decomposed as $\mathbf{v} = \mathbf{w} + \mathbf{u}$ with $\mathbf{w} \in W$ and $\mathbf{u} \in W^\perp$ by 1. Then $\mathbf{u} = \mathbf{v} - \mathbf{w} \in W^{\perp\perp}$, that is, $\mathbf{u} \in W^\perp \cap W^{\perp\perp}$. Because $W^\perp \cap W^{\perp\perp} = \{\mathbf{0}\}$ by 1, we have $\mathbf{u} = \mathbf{0}$ and hence $\mathbf{v} = \mathbf{w} \in W$.

6.3 Function Spaces

Let $a, b \in \mathbb{R}$ with $a < b$. It is well known that any continuous function $f : [a, b] \rightarrow \mathbb{R}$ has the definite integral $\int_a^b f(x) dx < \infty$. It is also well known that the integral satisfies the *positivity property*

$$f \geqq 0 \Rightarrow \int_a^b f(x) dx \geqq 0,$$

where the equality holds if and only if $f = 0$,⁵ and the *linearity property*

$$\int_a^b (\alpha f(x) + \beta g(x)) dx = \alpha \int_a^b f(x) dx + \beta \int_a^b g(x) dx$$

for another continuous function $g : [a, b] \rightarrow \mathbb{R}$ and for $\alpha, \beta \in \mathbb{K}$.⁶

Let $C([a, b], \mathbb{K})$ denote the set of all continuous functions from $[a, b]$ to \mathbb{K} , which is a subspace of the linear space $\mathbb{K}^{[a, b]}$ of all functions on $[a, b]$ over \mathbb{K} . The definite integral for a complex-valued function $f \in C([a, b], \mathbb{C})$ is defined by

$$\int_a^b f(x) dx \stackrel{\text{def}}{=} \int_a^b \operatorname{Re} f(x) dx + i \int_a^b \operatorname{Im} f(x) dx.$$

We introduce an inner product $(f, g) \mapsto \langle f \mid g \rangle$ on $C([a, b], \mathbb{K})$ by

$$\langle f \mid g \rangle \stackrel{\text{def}}{=} \int_a^b \overline{f(x)} g(x) dx.$$

for $f, g \in C([a, b], \mathbb{K})$.

Exercise 6.13 Prove that the binary function $\langle \cdot \mid \cdot \rangle$ above satisfies axioms of the inner product. Also prove that each function defined by the following is a norm of $C([a, b], \mathbb{K})$:

$$\begin{aligned} \|f\|_1 &\stackrel{\text{def}}{=} \int_a^b |f(x)| dx, \\ \|f\|_2 &\stackrel{\text{def}}{=} \left(\int_a^b |f(x)|^2 dx \right)^{1/2}, \\ \|f\|_\infty &\stackrel{\text{def}}{=} \max_{a \leq x \leq b} |f(x)|. \end{aligned}$$

⁵ $f \geqq 0$ (resp. $f = 0$) means that $f(x) \geqq 0$ (resp. $f(x) = 0$) for all $x \in [a, b]$.

⁶ See a textbook on calculus. In order to analyze a function space with a more elaborate theory, the concept of Lebesgue integration, which is a more general-purpose integration method, is needed rather than Riemann integration of a definite integral in calculus.

We call $\|\cdot\|_1$, $\|\cdot\|_2$, and $\|\cdot\|_\infty$ the L^1 -norm, the L^2 -norm and the L^∞ -norm, respectively. In particular, the L^2 -norm is the norm induced from the inner product above.

Exercise 6.14 Solve the following:

- (1) Consider the monomial functions $f_0, f_1, f_2 \in C([0, 1], \mathbb{R})$ defined by $f_0(x) \stackrel{\text{def}}{=} 1$, $f_1(x) \stackrel{\text{def}}{=} x$ and $f_2(x) \stackrel{\text{def}}{=} x^2$ for $x \in [0, 1]$. Find the inner products $\langle f_m | f_n \rangle$ for $m, n = 0, 1, 2$ and the norms $\|f_n\|_1, \|f_n\|_2, \|f_n\|_\infty$ for $n = 0, 1, 2$.
- (2) Consider the exponential functions $e_n(x) \stackrel{\text{def}}{=} e^{2\pi i n x} \in C([0, 1], \mathbb{C})$ for $x \in [0, 1]$ and $n \in \mathbb{Z}$, where i is the imaginary unit. Find the inner products $\langle e_m | e_n \rangle$ for $m, n = 0, \pm 1, \pm 2, \dots$.

We compute $\langle \sin | \cos \rangle$ and $\|f_1\|_2$ by NumPy.

Program: integral.py

```
In [1]: 1  from numpy import array, sqrt
2
3  def integral(f, dom):
4      N = len(dom) - 1
5      w = (dom[-1] - dom[0]) / N
6      x = array([(dom[n] + dom[n + 1]) / 2 for n in range(N)])
7      return sum(f(x)) * w
8
9  def inner(f, g, dom):
10     return integral(lambda x: f(x).conj() * g(x), dom)
11
12 norm = {
13     'L1': lambda f, dom: integral(lambda x: abs(f(x)), dom),
14     'L2': lambda f, dom: sqrt(inner(f, f, dom)),
15     'Loo': lambda f, dom: max(abs(f(dom))),
16 }
17
18 if __name__ == '__main__':
19     from numpy import linspace, pi, sin, cos
20     dom = linspace(0, pi, 1001)
21     print(f'<sin|cos> = {inner(sin, cos, dom)}')
22     print(f'||f_1||_2 = {norm["L2"] (lambda x: x, dom)}')
```

Lines 3–7: `integral(f, dom)` calculates the definite integral $\int_a^b f(x) dx$. Its arguments are the function f and array dom of $N + 1$ points including both ends, which divides the interval $[a, b]$ into N equal parts. We use the *midpoint formula* for numerical integration.⁷ This method approximates the integral by the total of the areas of N rectangles, each of which is the value of f at the midpoint of each small section multiplied by the width of the section.

Lines 9, 10: Define the inner product using `integral`.

Lines 12–16: Define the L^1 -norm, L^2 -norm, and L^∞ -norm according to mathematical definitions of these norms.

Lines 18–22: Calculate the inner product of trigonometric functions $\sin x$ and $\cos x$, and the L^2 -norm of monomial x on $C([0, \pi], \mathbb{R})$, and print them out.

```
<sin|cos> = -3.6738427362813575e-18
||f_1||_2 = 3.214875266047432
```

The above results contain numerical errors. The exact values are given as

⁷ Other numerical integration methods include the *trapezoidal rule* and *Simpson's rule*. See the textbook on numerical analysis [15] in the Bibliography.

$$\langle \sin | \cos \rangle = \int_0^\pi \sin(x) \cos(x) dx = 0, \quad \|f_1\|_2 = \sqrt{\int_0^\pi x^2 dx} = \sqrt{\frac{\pi^3}{3}}.$$

We can get these integrals symbolically using SymPy. Note that we must give the integrand as a string.

```
In [1]: from sympy import integrate, pi, sin, cos, sqrt
from sympy.abc import x
integrate('sin(x) * cos(x)', [x, 0, pi])
```

```
Out[1]: 0
```

```
In [2]: sqrt(integrate('x**2', [x, 0, pi]))
```

```
Out[2]: sqrt(3)*pi**(3/2)/3
```

6.4 Least Squares, Trigonometric Series, and Fourier Series

We define \mathbf{x}^p for $\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$ and $p = 0, 1, 2, \dots, k$ by

$$\mathbf{x}^p \stackrel{\text{def}}{=} (x_1^p, x_2^p, \dots, x_n^p).$$

The problem to find $a_0, a_1, \dots, a_k \in \mathbb{R}$ that minimize the value

$$\left\| a_0 \mathbf{x}^0 + a_1 \mathbf{x}^1 + \cdots + a_k \mathbf{x}^k - \mathbf{y} \right\|_2$$

for given $\mathbf{x} = (x_1, x_2, \dots, x_n)$ and $\mathbf{y} = (y_1, y_2, \dots, y_n)$ in \mathbb{R}^n is called the *least squares approximation* with polynomials. Since this is nothing but finding $\mathbf{y}_0 = \mathbf{proj}_W(\mathbf{y})$ where $W \stackrel{\text{def}}{=} \langle \mathbf{x}^0, \mathbf{x}^1, \dots, \mathbf{x}^k \rangle$, we can use the Gram–Schmidt orthogonalization.

Program: lstsqr.py

```
In [1]: 1 from numpy import linspace, cumsum, vdot, sort
2 from numpy.random import seed, uniform, normal
3 from gram_schmidt import gram_schmidt, proj
4 import matplotlib.pyplot as plt
5
6 n = 20
7 seed(2021)
8 x = sort(uniform(-1, 1, n))
9 z = 4 * x**3 - 3 * x
10 sigma = 0.2
11 y = z + normal(0, sigma, n)
12 E = gram_schmidt([x**0, x**1, x**2, x**3])
13 y0 = proj(z, E)
14
15 plt.figure(figsize=(15,5))
16 plt.errorbar(x, z, yerr=sigma, fmt='ro')
17 plt.plot(x, y, color='k', linestyle = 'solid')
18 plt.plot(x, y0, color='k', linestyle = 'dotted')
19 plt.show()
```

Line 8: Generate randomly n numbers $x_1 < x_2 < \cdots < x_n$ between -1 and 1 , and set $\mathbf{x} = (x_1, x_2, \dots, x_n)$.

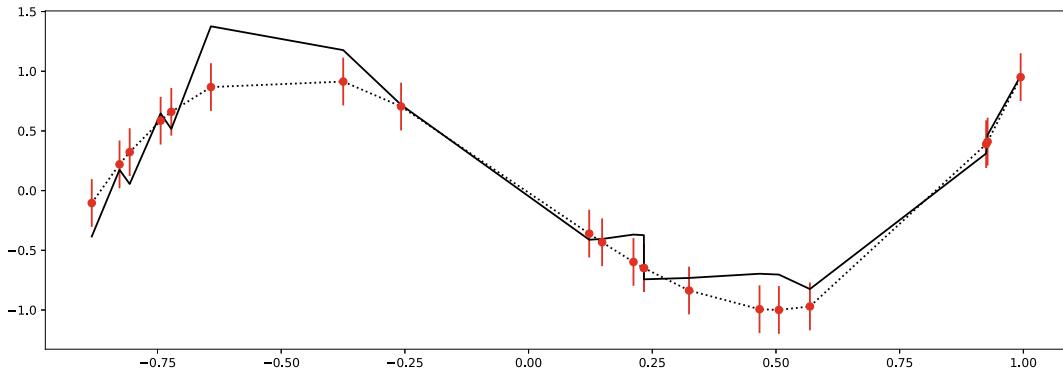


Fig. 6.5 Least squares approximation

Lines 9–11: Let $z = 4x^3 - 3x$ and let y be z plus errors which are n -dimensional vectors of mutually independent random numbers following the normal distribution with mean 0 and standard deviation $\sigma = 0.2$. In Fig. 6.5, the points with error bars of 1σ represent z and a relatively zig-zag line (solid) represents y .

Lines 12, 13: y_0 is a smooth curve obtained by least squares approximation of y with a cubic function. Such a curve is called a *regression curve* (or *regression line* if it is a straight line). In the graph, we can see that the curve (dotted) of y_0 is closer to z than y .

We consider the inner product introduced at the beginning of Sect. 6.3 on the function space $C([0, 1], \mathbb{R})$. Let

$$e_k(t) \stackrel{\text{def}}{=} \begin{cases} \sqrt{2} \sin(2\pi kt) & \text{if } k < 0, \\ 1 & \text{if } k = 0, \\ \sqrt{2} \cos(2\pi kt) & \text{if } k > 0, \end{cases}$$

for $k = 0, \pm 1, \pm 2, \dots$, then the set $E \stackrel{\text{def}}{=} \{\dots, e_{-1}, e_0, e_1, \dots\}$ becomes an (infinite) orthonormal system of $C([0, 1], \mathbb{R})$ (Exercise 6.15). The larger the absolute value of k becomes, the higher the frequency and the shorter the wavelength of e_k become as observed in its graph (Fig. 6.6). For any $f \in C([0, 1], \mathbb{R})$ and $K \in \mathbb{N}$ we put $f_K \stackrel{\text{def}}{=} \sum_{k=-K}^K \langle e_k | f \rangle e_k$. The convergence $\lim_{K \rightarrow \infty} \|f - f_K\|_2 = 0$ is known⁸ and we have the *trigonometric series expansion*

$$f = \sum_{k=-\infty}^{\infty} \langle e_k | f \rangle e_k$$

of f in the sense of convergence in L^2 -norm. The transform $f \mapsto f_K$ is called a *low-pass filter* with *cutoff frequency* K which is the orthogonal projection of $C([0, 1], \mathbb{R})$ onto the subspace generated by $E_K \stackrel{\text{def}}{=} \{e_{-K}, \dots, e_0, \dots, e_K\}$. The function f_K can be thought of as removing the high-frequency components of f .

Exercise 6.15 Prove that E is an orthonormal system of $C([0, 1], \mathbb{R})$, that is, $\langle e_i | e_j \rangle = \delta_{ij}$ for all $i, j = 0, \pm 1, \pm 2, \dots$

⁸ See the textbooks on functional analysis in the Bibliography.

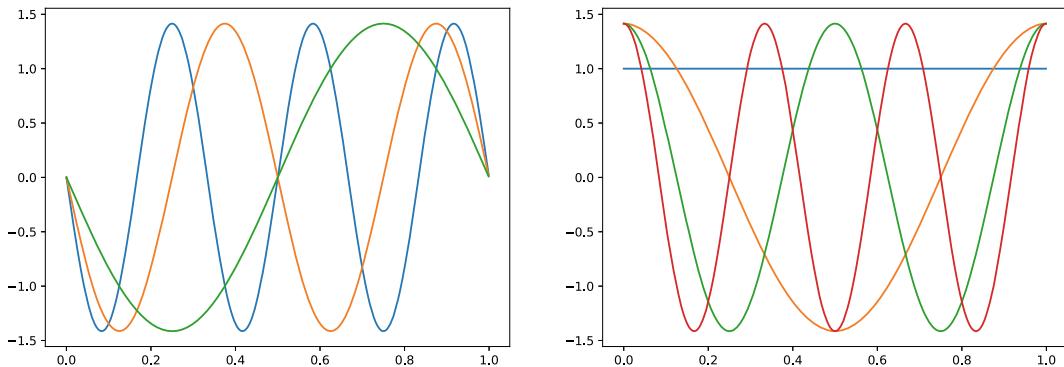


Fig. 6.6 Trigonometric functions e_{-1}, e_{-2}, e_{-3} (left) and $1, e_1, e_2, e_3$ (right)

To solve the problem by computer, we take n points $0 = t_0 < t_1 < \dots < t_{n-1} < 1$ dividing equally the domain $[0, 1]$ of f and consider the vector $\mathbf{f} = (f(t_0), f(t_1), \dots, f(t_{n-1}))$ instead of f itself. This process is called *sampling*, and \mathbf{f} is the sampling vector of f with respect to the sampling points t_0, t_1, \dots, t_{n-1} . It is considered as projecting the infinite-dimensional linear space $C([0, 1], \mathbb{R})$ to the n -dimensional space \mathbb{K}^n , whose inner product is the standard inner product multiplied by $1/n$.

Program: trigonometric.py

```
In [1]: 1  from numpy import inner, pi, sin, cos, sqrt, ones
2  from gram_schmidt import proj
3
4  def e(k, t):
5      if k < 0:
6          return sin(2 * k * pi * t) * sqrt(2)
7      elif k == 0:
8          return ones(len(t))
9      elif k > 0:
10         return cos(2 * k * pi * t) * sqrt(2)
11
12 def lowpass(K, t, f):
13     n = len(t)
14     E_K = [e(k, t) for k in range(-K, K + 1)]
15     return proj(f, E_K, inner=lambda x, y: inner(x, y) / n)
16
17 if __name__ == '__main__':
18     from numpy import arange
19     import matplotlib.pyplot as plt
20     t = arange(0, 1, 1 / 1000)
21     plt.figure(figsize=(15, 5))
22     plt.subplot(121)
23     for k in range(-3, 0):
24         plt.plot(t, e(k, t))
25     plt.subplot(122)
26     for k in range(4):
27         plt.plot(t, e(k, t))
28     plt.show()
```

Lines 4–10: $e(k, t)$ returns the sampling vector of e_k with respect to array t of sampling points.

Lines 12–15: $lowpass(K, t, f)$ returns the low-pass filter of f for cutoff frequency K .

Lines 17–28: Draw Fig. 6.6 when this is run as a main program, not as a library.

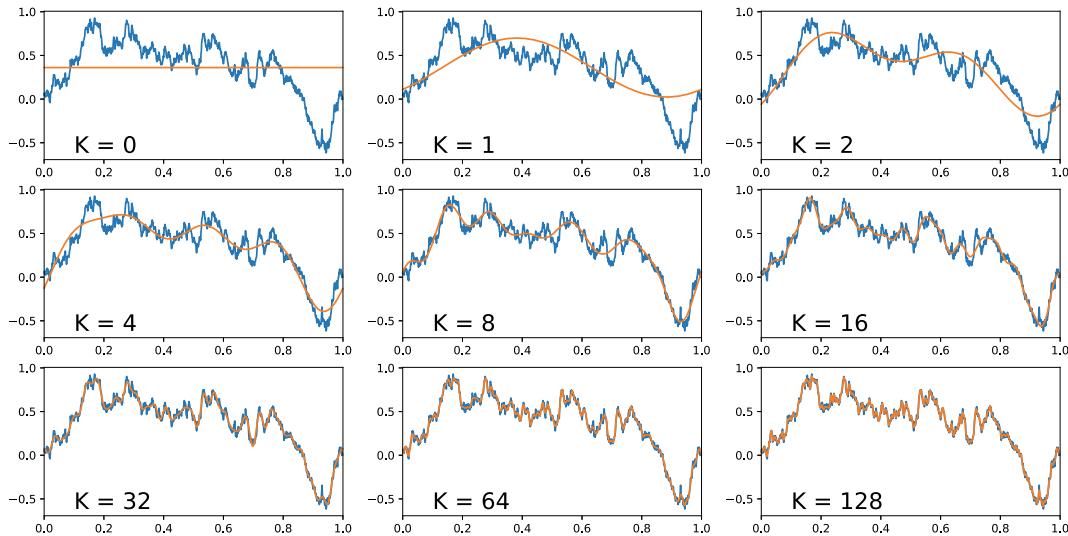


Fig. 6.7 Low-pass filters

Program: brown.py

```
In [1]: 1 from numpy import arange, cumsum, sqrt
2 from numpy.random import seed, normal
3 from trigonometric import lowpass
4 #from fourier import lowpass
5 import matplotlib.pyplot as plt
6
7 seed(2021)
8 n = 1000
9 dt = 1 / n
10 t = arange(0, 1, dt)
11 f = cumsum(normal(0, sqrt(dt), n))
12
13 fig, ax = plt.subplots(3, 3, figsize=(16, 8), dpi=100)
14 for k, K in enumerate([0, 1, 2, 4, 8, 16, 32, 64, 128]):
15     i, j = divmod(k, 3)
16     f_K = lowpass(K, t, f)
17     ax[i][j].plot(t, f), ax[i][j].plot(t, f_K)
18     ax[i][j].text(0.1, -0.5, f'K = {K}', fontsize = 20)
19     ax[i][j].set_xlim(0, 1)
20 plt.show()
```

Line 3: Import `lowpass` from the previous program `trigonometric.py` as a library.

Lines 7–11: Create a continuous function f containing high-frequency components.⁹

Lines 13–20: Draw the graph of g_K for each $K = 0, 1, 2, 4, 8, 16, 32, 64, 128$ overlaid with the graph of f (Fig. 6.7).

In the complex linear space $C([0, 1], \mathbb{C})$ equipped with the inner product introduced in Sect. 6.3,

$$E \stackrel{\text{def}}{=} \left\{ e_k \mid e_k(t) \stackrel{\text{def}}{=} e^{2\pi i k t}, k = \dots, -2, -1, 0, 1, 2, \dots \right\}$$

⁹ We use here a sample function of one-dimensional Brownian motion created by simulation using the module `random` of NumPy.

becomes an orthonormal system (cf. Exercise 6.14 (2)). The Fourier coefficients of $f \in C([0, 1], \mathbb{C})$ are given by

$$\langle e_k | f \rangle = \int_0^1 e^{-2\pi i k t} f(t) dt$$

for $k = \dots, -2, -1, 0, 1, 2, \dots$. As in the case of trigonometric series, the infinite series expansion

$$f \stackrel{\text{def}}{=} \sum_{k=-\infty}^{\infty} \langle e_k | f \rangle e_k$$

holds in the sense of L^2 -norm convergence. This expansion is called the *Fourier series expansion* and

$f_K \stackrel{\text{def}}{=} \sum_{k=-K}^K \langle e_k | f \rangle e_k$ is a low-pass filter with cutoff frequency K of f .

When dealing with this problem on a computer, we replace a function of $C([0, 1], \mathbb{C})$ with a sampling vector of \mathbb{C}^n as in the case of trigonometric series. Let n be a natural number and put $dt = 1/n$ and $t_j = jdt$. We replace e_k in E with the vector

$$e_k = (e_k(t_0), e_k(t_1), \dots, e_k(t_{n-1})).$$

Since $e^{2\pi i t}$ is a function of t with period 1, that is, $e^{2\pi i t} = e^{2\pi i(t-1)}$, we have

$$e_{n-k}(t_j) = e^{2\pi i(n-k)j/n} = e^{2\pi i(-k)j/n} = e_{-k}(t_j).$$

Therefore, $e_{n-k} = e_{-k}$.

The program for the low-pass filter is given as follows:

Program: fourier.py

```
In [1]: 1 from numpy import arange, exp, pi, vdot
2 from gram_schmidt import proj
3
4 def e(k, t): return exp(2j * pi * k * t)
5
6 def lowpass(K, t, z):
7     dt = 1 / len(t)
8     E_K = [e(k, t) for k in range(-K, K + 1)]
9     return proj(z, E_K, inner=lambda x, y: vdot(x, y) * dt)
10
11 if __name__ == '__main__':
12     import matplotlib.pyplot as plt
13     from mpl_toolkits.mplot3d import Axes3D
14
15     fig = plt.figure(figsize=(20, 8))
16     t = arange(0, 1, 1/1000)
17     for n, k in enumerate([0, 1, 2, 100, 300, 500, 700, 900, 998, 999]):
18         x = [z.real for z in e(k, t)]
19         y = [z.imag for z in e(k, t)]
20         ax = fig.add_subplot(2, 5, n+1, projection="3d")
21         ax.set_xlim(0, 1), ax.set_ylim(-1, 1), ax.set_zlim(-1, 1)
22         ax.plot(t, x, y)
23         ax.text(0, 1, 1, f'k = {k}', fontsize = 20)
24     plt.show()
```

Draw a 3D graph of the complex-valued sampled function e_k for given k . The graphs for $k = 0, 1, 2, 100, 300, 500, 700, 900, 998, 999$ are shown in Fig. 6.8. This figure is drawn as a line graph, but if we

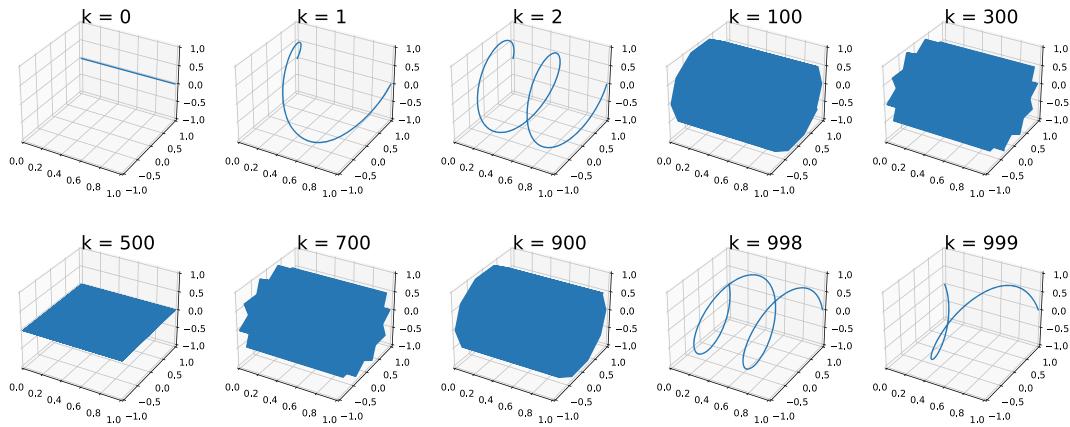


Fig. 6.8 Fourier series

want to see a scatter plot, we delete Line 22 or comment it out (insert # before `ax.plot`). We can observe $e_{1000-k} = e_{-k}$.

Exercise 6.16 In program `brown.py`, in order to use this program `fourier.py` as a library and import this `lowpass`, comment out Line 3 and uncomment Line 4, and execute the program. The modified `brown.py` will display a warning at Line 17 because `f_K` takes a complex value. If we use `f_K.real` instead of `f_K` at Line 17, the warning will disappear. Display also `f_K.imag` at the same time.

6.5 Orthogonal Function Systems

Let X be an interval of \mathbb{R} and let $w : X \rightarrow [0, \infty)$ be a continuous function taking positive values on X except for a finite number of $x \in X$. We call w a *weight function*. Define

$$\langle f | g \rangle \stackrel{\text{def}}{=} \int_X \overline{f(x)} g(x) w(x) dx$$

for $f, g \in \mathbb{C}^X$, whenever the finite definite integral on the right-hand side exists. The problem of the existence of this integral is beyond the scope of this book and we do not discuss it in depth here.¹⁰ In this section, we only consider polynomials for $f(x)$ and $g(x)$, but if the integration interval is open or infinite, we need an improper integral.

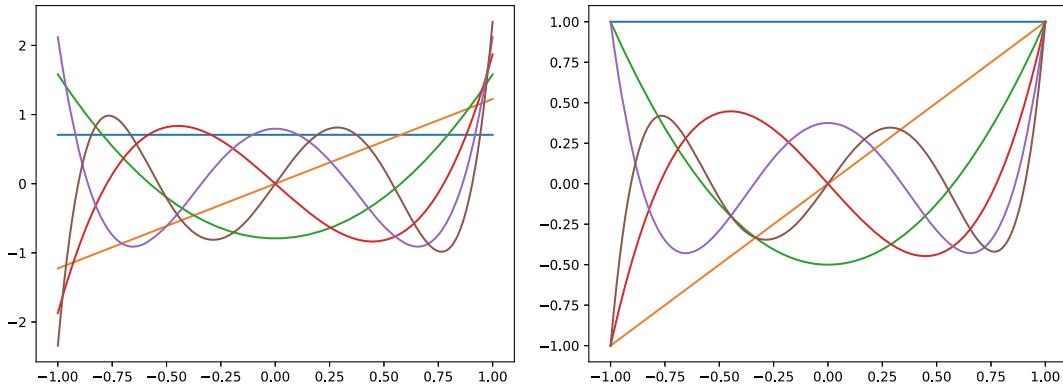
The set V of all polynomials on X is an inner product space over \mathbb{K} equipped with the inner product above which depends on w . The orthogonal system obtained by applying the Gram–Schmidt orthogonalization method to $\{1, x, x^2, \dots, x^n\} \subseteq V$ is called an *orthogonal polynomial* system. Orthogonal polynomials differ depending on the combination of X and w (Table 6.1).

Let us use the library `gram_schmidt` we already made for Gram–Schmidt orthogonalization. Since it applies to vectors of \mathbb{K}^n , we must treat polynomials as sampling vectors. Here we create Legendre polynomials and Chebyshev polynomials of the first and second kinds, which have finite domains.

¹⁰ This is a problem deeply related to the definition of an integral.

Table 6.1 Orthogonal polynomials and weight functions

Polynomials	X	ω
Legendre—	$-1 \leq x \leq 1$	1
Chebyshev—of the first kind	$-1 < x < 1$	$\frac{1}{\sqrt{1-x^2}}$
Chebyshev—of the second kind	$-1 \leq x \leq 1$	$\sqrt{1-x^2}$
Laguerre—	$0 \leq x < \infty$	e^{-x}
Hermite—	$-\infty < x < \infty$	e^{-x^2}

**Fig. 6.9** Graph of Legendre polynomials (left: $\|e_n\| = 1$, right: $e_n(1) = 1$)**Program:** poly_np1.py

```

In [1]: 1  from numpy import array, linspace, sqrt, ones, pi
2  import matplotlib.pyplot as plt
3  from gram_schmidt import gram_schmidt
4
5  m = 10000
6  D = linspace(-1, 1, m + 1)
7  x = array([(D[n] + D[n+1])/2 for n in range(m)])
8
9  inner = {
10     'Legendre': lambda f, g: f.dot(g) * 2/m,
11     'Chebyshev1': lambda f, g: f.dot(g/sqrt(1 - x**2)) * 2/m,
12     'Chebyshev2': lambda f, g: f.dot(g*sqrt(1 - x**2)) * 2/m,
13 }
14
15 A = [x**n for n in range(6)]
16 E = gram_schmidt(A, inner=inner['Legendre'])
17 for e in E:
18     plt.plot(x, e)
19 plt.show()

```

Lines 5–7: We divide the interval between -1 and 1 into 10000 equal parts and use the midpoints of subintervals as sampling points. With these, we use the midpoint formula for numerical integration and prevent division by 0 of the integrand at the ends of the interval of the Chebyshev polynomials of the first kind.

Lines 9–13: Make a dictionary whose values are lambda expressions of the inner products by the weight functions for Legendre polynomials, Chebyshev polynomials of the first and second kinds, respectively.

Line 15: List of vectorized monomials of degree up to 5.

Line 16: Apply the Gram–Schmidt orthogonalization. Obtain the desired orthogonal polynomials by assigning the value in the name argument. In this example, seek for the Legendre polynomials.

Lines 17–19: Draw the graphs of the resulting orthogonal polynomials (Fig. 6.9, left).

In the module `polynomial` of NumPy, the orthogonal polynomials are defined.¹¹ Using them, let us draw the graphs of the orthogonal polynomials.

Program: `poly_np2.py`

```
In [1]: 1 from numpy import linspace, exp, sqrt
2 from numpy.polynomial.legendre import Legendre
3 from numpy.polynomial.chebyshev import Chebyshev
4 from numpy.polynomial.laguerre import Laguerre
5 from numpy.polynomial.hermite import Hermite
6 import matplotlib.pyplot as plt
7
8 x1 = linspace(-1, 1, 1001)
9 x2 = x1[1:-1]
10 x3 = linspace(0, 10, 1001)
11 x4 = linspace(-3, 3, 1001)
12
13 f, x, w = Legendre, x1, 1
14 # f, x, w = Chebyshev, x2, 1
15 # f, x, w = Chebyshev, x2, 1/sqrt(1 - x2**2)
16 # f, x, w = Laguerre, x3, 1
17 # f, x, w = Laguerre, x3, exp(-x3)
18 # f, x, w = Hermite, x4, 1
19 # f, x, w = Hermite, x4, exp(-x4**2)
20 for n in range(6):
21     e = f.basis(n)(x)
22     plt.plot(x, e * w)
23 plt.show()
```

Lines 2–5: The `polynomial` module not only defines each of the orthogonal polynomials but also contains some tools for numerical calculations. Here, we import only the necessities for drawing the graphs.

Line 8: Array `x1` is the integration interval of Legendre polynomials.

Line 9: `x2`, removing both ends from `x1`, is the integration interval of Chebyshev polynomials of type 1.

Line 10: The integration interval of Laguerre polynomials is from 0 to ∞ , but set the upper bound of the interval as $x = 10$, neglecting the integration over 10 because the values of weight there do not exceed e^{-10} .

Line 11: The integration interval of Hermite polynomials is from $-\infty$ to ∞ , but we regard the interval as $-3 \leq x \leq 3$ neglecting the outside of it.

Lines 13–19: Uncomment one of these lines and run. We can choose either polynomials themselves or functions that multiply polynomials by the weight function. Hermite polynomials may be better characterized by the latter.

Lines 20–23: Draw the graphs of six Legendre polynomials up to degree 5 (Fig. 6.9, right). These are different from Legendre polynomials made by the original Gram–Schmidt orthogonalization (Fig. 6.9, left). Instead of normalizing the norm of the function (to norm 1), the value at $x = 1$ is aligned to 1. This form is more commonly referred to as the Legendre polynomials.

¹¹ These are also defined in SciPy.

Exercise 6.17 Draw the graphs of other polynomials and compare them with each other.

The Gram–Schmidt orthogonalization of polynomials can be executed also by symbolic calculation. It is hard to do it by hand, so get the help of SymPy.

Program: poly_sp1.py

```
In [1]: 1 from sympy import integrate, sqrt, exp, oo
2 from sympy.abc import x
3
4 D = {
5     'Legendre': ((x, -1, 1), 1),
6     'Chebyshev1': ((x, -1, 1), 1 / sqrt(1 - x**2)),
7     'Chebyshev2': ((x, -1, 1), sqrt(1 - x**2)),
8     'Laguerre': ((x, 0, oo), exp(-x)),
9     'Hermite': ((x, -oo, oo), exp(-x**2)),
10 }
11 dom, weight = D['Legendre']
12
13 def inner(expr1, expr2):
14     return integrate(expr1*expr2*weight, dom)
15
16 def gram_schmidt(A):
17     E = []
18     while A != []:
19         a = A.pop(0)
20         b = a - sum([inner(e, a) * e for e in E])
21         c = sqrt(inner(b, b))
22         E.append(b / c)
23     return E
24
25 E = gram_schmidt([1, x, x**2, x**3])
26 for n, e in enumerate(E):
27     print(f'e{n}(x) = {e}' )
```

Line 1: In SymPy, ∞ means ∞ .

Line 2: Use symbol x as the integral variable.

Lines 4–10: D is a dictionary whose values are tuples of integration intervals and weights.

Line 11: Choose the integration interval and the weight functions of Legendre polynomials.

Lines 13, 14: Arguments $expr1$ and $expr2$ are polynomials in x and the inner product of these two is calculated by a symbolic definite integration. Even if the integration interval is infinite or the integrand diverges at an end point, the definite integration is correctly processed with improper integral.

Lines 16–23: Define the function to apply the Gram–Schmidt orthogonalization.

Lines 25–27: Find Legendre polynomials up to degree 3.



```
e0(x) = sqrt(2)/2
e1(x) = sqrt(6)*x/2
e2(x) = 3*sqrt(10)*(x**2 - 1/3)/4
e3(x) = 5*sqrt(14)*(x**3 - 3*x/5)/4
```

Orthogonal polynomials are also built-in in SymPy. By running the following program, we can observe the difference from the previous results as in the experiment with NumPy.

Program: poly_sp2.py

```
In [1]: 1 from sympy.polys.orthopolys import (
2     legendre_poly,
3     chebyshev1_poly,
4     chebyshev2_poly,
5     laguerre_poly,
6     hermite_poly,
```

In [1]:

```

7  )
8  from sympy.abc import x
9
10 e = legendre_poly
11 for n in range(4):
12     print(f'e{n}(x) = {e(n, x)}')

```



```

e0(x) = 1
e1(x) = x
e2(x) = 3*x**2/2 - 1/2
e3(x) = 5*x**3/2 - 3*x/2

```

Exercise 6.18 Experiment with other polynomials than Legendre's. Compare the results by NumPy and by SymPy.

6.6 Convergence of Vector Sequences

Consider an infinite sequence

$$x_1, x_2, \dots, x_k, \dots$$

of vectors $x_k = (x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}) \in \mathbb{K}^n$. It is called a *vector sequence* and denoted by $\{x_k\}_{k=1}^\infty$. For $x \in \mathbb{K}^n$, the sequence $\{x_k\}_{k=1}^\infty$ is said to *converge* to x as $k \rightarrow \infty$, if the following conditions 1 to 4, which are equivalent to each other as shown below, are satisfied. In this case x is called the *limit* of $\{x_k\}_{k=1}^\infty$ and denoted as $x = \lim_{k \rightarrow \infty} x_k$.

1. $\lim_{k \rightarrow \infty} \|x - x_k\|_1 = 0$,
2. $\lim_{k \rightarrow \infty} \|x - x_k\|_2 = 0$,
3. $\lim_{k \rightarrow \infty} \|x - x_k\|_\infty = 0$,
4. $\lim_{k \rightarrow \infty} |x_i - x_i^{(k)}| = 0$ for every $i = 1, 2, \dots, n$.

The equivalence is proved as follows. From the inequalities

$$\left(\max_{1 \leq i \leq n} |x_i| \right)^2 = \max_{1 \leq i \leq n} |x_i|^2 \leq \sum_{i=1}^n |x_i|^2 \leq \left(\sum_{i=1}^n |x_i| \right)^2 \leq \left(n \max_{1 \leq i \leq n} |x_i| \right)^2,$$

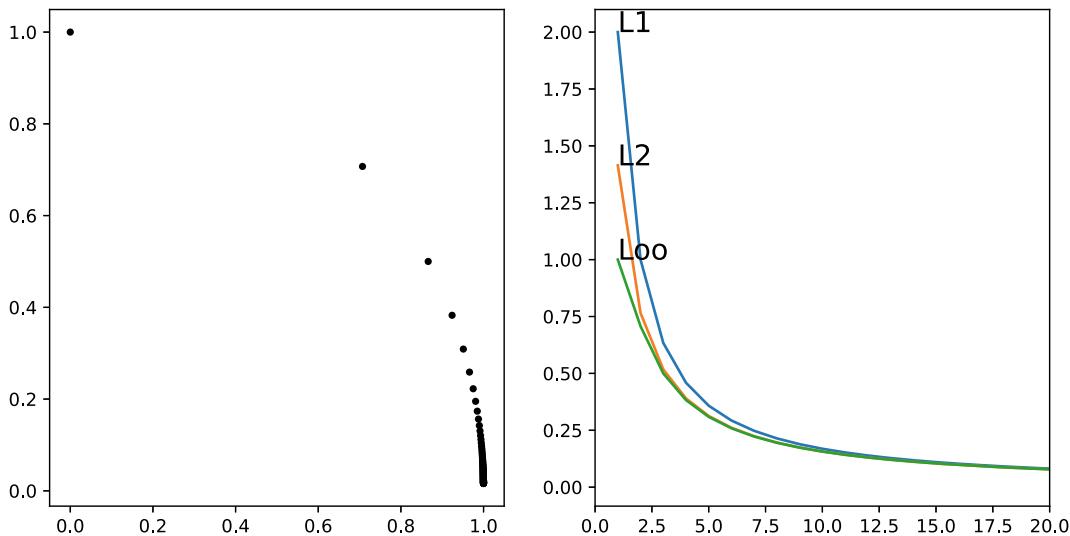
we have

$$\|x\|_\infty \leq \|x\|_2 \leq \|x\|_1 \leq n \|x\|_\infty$$

for $x = (x_1, x_2, \dots, x_n) \in \mathbb{K}^n$. From this $1 \Rightarrow 2$ and $2 \Rightarrow 3$ follow, and also $3 \Rightarrow 1$ follows because n is a fixed number. Hence, 1, 2, and 3 are equivalent to each other. Further, we have $1 \Rightarrow 4$ because $|x_i| \leq \|x\|_1$ for every $i = 1, 2, \dots, n$. Finally, $4 \Rightarrow 1$ holds because

$$\lim_{k \rightarrow \infty} \|x - x_k\|_1 = \lim_{k \rightarrow \infty} \sum_{i=1}^n |x_i - x_i^{(k)}| = \sum_{i=1}^n \lim_{k \rightarrow \infty} |x_i - x_i^{(k)}| = 0.$$

The following program draws how a vector sequence converges and how the three norms change as the sequence converges (Fig. 6.10).

**Fig. 6.10** Convergence of vector sequence and change of norms**Program:** limit.py

```
In [1]: 1  from numpy import array, sin, cos, pi, inf
2  from numpy.linalg import norm
3  import matplotlib.pyplot as plt
4
5  def A(t):
6      return array([[cos(t), -sin(t)], [sin(t), cos(t)]])
7
8  x = array([1, 0])
9  P, L1, L2, Loo = [], [], [], []
10 M = range(1, 100)
11 for m in M:
12     xm = A(pi / 2 / m).dot(x)
13     P.append(xm)
14     L1.append(norm(x - xm, ord=1))
15     L2.append(norm(x - xm))
16     Loo.append(norm(x - xm, ord=inf))
17
18 plt.figure(figsize=(10, 5))
19 plt.subplot(121)
20 for p in P:
21     plt.plot(p[0], p[1], marker='.', color='black')
22 plt.subplot(122), plt.xlim(0, 20)
23 plt.plot(M, L1), plt.text(1, L1[0], 'L1', fontsize=16)
24 plt.plot(M, L2), plt.text(1, L2[0], 'L2', fontsize=16)
25 plt.plot(M, Loo), plt.text(1, Loo[0], 'Loo', fontsize=16)
26 plt.show()
```

When $\lim_{k \rightarrow \infty} x_k = x$, the following results hold:

1. $\lim_{k \rightarrow \infty} \langle x_k | y \rangle = \langle x | y \rangle$ for any $y \in \mathbb{K}^n$,
2. $\lim_{k \rightarrow \infty} Ax_k = Ax$ for any (m, n) -matrix A .

Using the Schwarz inequality we have

$$0 \leq |\langle \mathbf{x} | \mathbf{y} \rangle - \langle \mathbf{x}_k | \mathbf{y} \rangle| = |\langle \mathbf{x} - \mathbf{x}_k | \mathbf{y} \rangle| \leq \|\mathbf{x} - \mathbf{x}_k\| \|\mathbf{y}\|,$$

from which 1 follows. We have 2 because $\mathbf{A}\mathbf{x}_k$ converges to $\mathbf{A}\mathbf{x}$ componentwise.

6.7 Fourier Analysis

In mathematics, it is common to represent a space by another space or project it onto a subspace and to analyze the represented or the projected space. Just one example is to choose a basis of an n -dimensional linear space V over \mathbb{K} and to represent V by the isomorphic space \mathbb{K}^n . In that situation, it is important to take a basis convenient for the problem to be analyzed. *Fourier analysis* in a wide sense is the idea of representing a target space as \mathbb{K}^n and projecting it to its subspace, by considering various inner products and orthonormal bases. It contains the topics of least squares approximation and orthogonal function systems covered in this chapter, and also their applications and more developed ideas (treated in Chap. 10).

On the other hand, Fourier analysis in a narrow sense means the basic theory and its application¹² concerning the Fourier series $F(t) = \sum_{k \in \mathbb{Z}} x_k e^{-2\pi i k t}$ ($t \in [0, 1]$) and the *Fourier integral* $F(\omega) = \int_{-\infty}^{\infty} e^{-2\pi i \omega t} x(t) dt$ ($\omega \in \mathbb{R}$). Representing a given function $x(t)$ in a Fourier series is called *Fourier series expansion*. This is a special case of Fourier expansions in a general inner product space. Also, making a function $x(t)$ in time t into a function $F(\omega)$ in frequency ω by a Fourier integral is called the *Fourier transform*.

When calculating the Fourier series on a computer, we must approximate an infinite series with a finite series, and as we did in Sect. 6.4, a function was sampled at a finite number of points and was replaced with a finite-dimensional vector. For given $f \in C([0, 1], \mathbb{C})$ and $k \in \mathbb{Z}$, we approximate the definite integral $\int_0^1 e^{-2\pi i k t} f(t) dt$ by

$$\sum_{l=0}^{n-1} e^{-2\pi i k t_l} f(t_l) \Delta t,$$

where n is a sufficiently large natural number, $\Delta t = \frac{1}{n}$ and $t_l = l \Delta t$ for $l = 0, 1, \dots, n-1$. The function $e_k(t) = e^{2\pi i k t}$ in t with $0 \leq t < 1$ for $k \in \mathbb{Z}$ is replaced by the vector

$$\mathbf{e}_k = \frac{1}{\sqrt{n}} (e^{2\pi i k t_0}, e^{2\pi i k t_1}, \dots, e^{2\pi i k t_{n-1}})$$

of \mathbb{C}^n and inner product $\langle e_k | e_l \rangle$ is replaced by the standard inner product

$$\langle \mathbf{e}_k | \mathbf{e}_l \rangle = \frac{1}{n} \sum_{j=0}^{n-1} e^{2\pi i (l-k)t_j}$$

¹² Basic theory is mainly the work of mathematicians. Applications are the work of engineers, and typical applications include the design of filters that eliminate noise in electrical engineering and acoustics. Physicists are just halfway between mathematicians and engineers. Ideas from physics often contribute to both mathematical theory and engineering applications. Fourier analysis is a typical example.

on \mathbb{C}^n for $k, l = 0, 1, \dots, n - 1$. The right-hand side above is 1 if $k = l$, and otherwise, it is 0 because its terms locate evenly at vertices of a regular n polygon on the unit circle of the complex plane. Therefore $\{\mathbf{e}_0, \mathbf{e}_1, \dots, \mathbf{e}_{n-1}\}$ is an orthonormal basis of \mathbb{C}^n .¹³

Let $\mathbf{x} \in \mathbb{C}^n$, then its representation

$$\widehat{\mathbf{x}} \stackrel{\text{def}}{=} (\langle \mathbf{e}_0 | \mathbf{x} \rangle, \langle \mathbf{e}_1 | \mathbf{x} \rangle, \dots, \langle \mathbf{e}_{n-1} | \mathbf{x} \rangle)$$

with respect to the basis is a tuple of Fourier coefficients and is called the *discrete Fourier transform* of \mathbf{x} . The sequence of the squares of absolute values of the Fourier coefficients

$$|\langle \mathbf{e}_0 | \mathbf{x} \rangle|^2, |\langle \mathbf{e}_1 | \mathbf{x} \rangle|^2, \dots, |\langle \mathbf{e}_{n-1} | \mathbf{x} \rangle|^2$$

is called the *power spectrum* of \mathbf{x} . By the Riesz–Fischer identity in Theorem 6.2, the sum of power spectra equals $\|\mathbf{x}\|_2^2$.

On the other hand, for $\mathbf{y} = (y_0, y_1, \dots, y_{n-1}) \in \mathbb{C}^n$, define $\widetilde{\mathbf{y}}$ by

$$\widetilde{\mathbf{y}} \stackrel{\text{def}}{=} y_0 \mathbf{e}_0 + y_1 \mathbf{e}_1 + \dots + y_{n-1} \mathbf{e}_{n-1}$$

and call it the *discrete inverse Fourier transform* of \mathbf{y} . The identity $\mathbf{x} = \widehat{\mathbf{x}}$ is no more than the Fourier expansion of \mathbf{x} . The mapping $\mathbf{x} \mapsto \widehat{\mathbf{x}}$ is an isomorphism of \mathbb{C}^n onto itself and $\mathbf{y} \mapsto \widetilde{\mathbf{y}}$ is its inverse mapping. From Parseval's identity and the Riesz–Fischer identity, this isomorphism reserves the norm and the inner product.

Exercise 6.19 Find the matrix representation of the discrete Fourier transform, that is, the basis change matrix from the standard basis to $\{\mathbf{e}_0, \mathbf{e}_1, \dots, \mathbf{e}_{n-1}\}$.

The graphs of $\vec{e}_0, \vec{e}_1, \dots, \vec{e}_{n-1}$ were shown in Fig. 6.8 in Sect. 6.4. Identities $\mathbf{e}_{-1} = \mathbf{e}_{n-1}$, $\mathbf{e}_{-2} = \mathbf{e}_{n-2}, \dots$ hold, and ones around the center of the aligned $\mathbf{e}_0, \mathbf{e}_1, \dots, \mathbf{e}_{n-1}$ have a higher frequency (number of rotations on the unit circle). The low-pass filter

$$\mathbf{x} \mapsto \sum_{k=-K}^K \langle \mathbf{e}_k | \mathbf{x} \rangle \mathbf{e}_k$$

is a projection onto the subspace generated by $\{\mathbf{e}_{-K}, \dots, \mathbf{e}_0, \dots, \mathbf{e}_K\}$ removing the high frequency.

Let us experiment with actual voice data. If the sampling rate of 22050 is used for audio data, for example, even a sound 1 second long can be 22050 dimensional. A low-pass filter for such a high-dimensional vector will take a considerable amount of time to calculate, if `fourier.py` in this chapter is used as a library. Let us use a method called the *fast Fourier transform*¹⁴ which finds the Fourier coefficient at high speed. The module `fft` of NumPy defines functions `fft` and `ifft` implementing the fast Fourier transform algorithm. The former is a function to find the Fourier coefficients (the fast Fourier transform) of a vector, and the latter is a function to convert the Fourier coefficients to the vector (the inverse fast Fourier transform).

¹³ The Fourier transform of a function defined on $(-\infty, \infty)$ is similarly calculated by considering its domain as the set of a sufficiently large number of sampling points of sufficiently wide finite interval.

¹⁴ The discrete Fourier transform is obtained by multiplying a vector of dimension n by the basis transformation matrix, so if calculated normally, it takes time proportional to the number n^2 of elements of the matrix of order n . In contrast, the fast Fourier transform is known to take time proportional to $n \log n$. This method utilizes the fact that the basis transformation matrix has a special form. See the textbook on numerical analysis [15] in the Bibliography.

Table 6.2 Object variables

Expression	Contents
S.len	Number of sampling points (integer)
S.tmax	Recording time in seconds (real number)
S.time	Sampling points on time axis (array of real numbers)
S.data	Audio data (array of real numbers x with $-1 \leq x < 1$)
S.fft	Fourier coefficients (array of complex numbers)

Table 6.3 Class methods

Expression	Function
S.power_spectrum(rng)	Return the power spectra of the audio data to display in a graph with Matplotlib. If the name argument <code>rng</code> is assigned with (r_1, r_2) , the spectra are included for this interval, otherwise for the entire length by default
S.lowpass(K)	Return the low-pass filter with the cutoff frequency K . At the same time, the audio data passing the filter is saved as a WAV file of the original file name with the cutoff frequency

Let us define a class `Sound` that not only reads a WAV format audio data file and holds it, but also has methods to find its power spectrum and low-pass filter. First, we explain its specifications. If we have a file `example.wav` with monaural WAV format, then by setting `S = Sound('example')`, the audio data will be read into the variable `S`. Various information affiliated with the audio data can be referred to by the object variables listed in Table 6.2.¹⁵ In addition, the two methods¹⁶ listed in Table 6.3 are defined. We show two examples using the `Sound` class in Programs `power_spectrum.py` and `lowpass.py`.

Program: power_spectrum.py

```
In [1]: 1 import matplotlib.pyplot as plt
2 from sound import Sound
3
4 sound1, sound2 = Sound('CEG'), Sound('mono')
5
6 plt.figure(figsize=(15, 5))
7 plt.subplot(121)
8 plt.plot(*sound1.power_spectrum((-1000, 1000)))
9 plt.subplot(122)
10 plt.plot(*sound2.power_spectrum())
11 plt.show()
```

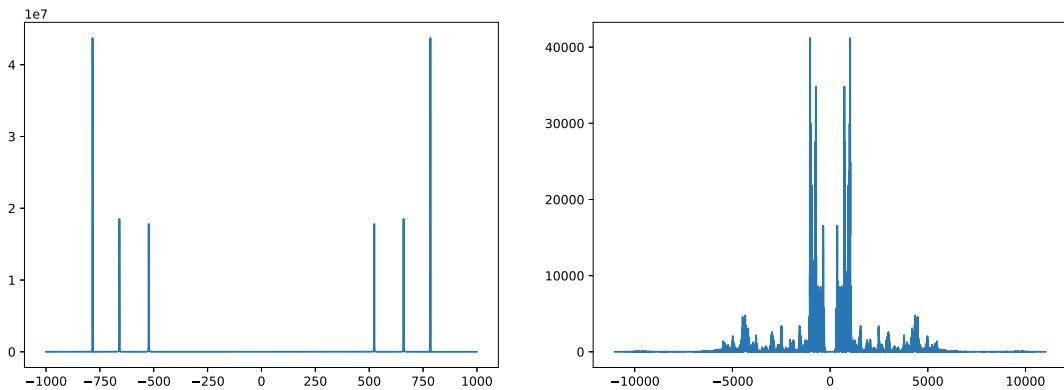
Line 4: `CEG.wav` is the chord created in the experiment in Chap. 2, and `mono.wav` is a monaural sound of a human voice. Using the file name without the extension as an argument, create two objects of `Sound` class for comparison, and refer to them with the names `sound1` and `sound2`.

Lines 7,8: Draw the power spectra of `sound1` below frequency 1000 (Fig. 6.11, left). The peaks appear at the places of the notes of the C, E, and G.

Lines 9,10: Draw the power spectra of `sound2` of a human voice (Fig. 6.11, right). The power spectra are 0 near the center and at both ends, and are concentrated in a certain frequency band.

¹⁵ These are peculiar variables of the object which are referred to with the object name as a prefix, such as `S.something`.

¹⁶ A method is a function defined in a class of objects. Label it with the name of the object as a prefix, such as `S.doing`.

**Fig. 6.11** Power spectra

Exercise 6.20 Though the sounds of the notes C, E, and G were added at the same ratio, the peak pitches are different. This is because their frequencies are not integers. Change Line 8 of the above program to zoom in near the peaks of frequency and observe.

Exercise 6.21 Give a reason why the power spectrum appears symmetrically around 0. (Hint: this is generally true for the discrete Fourier transform $\hat{x} \in \mathbb{C}^n$ of real vector $x \in \mathbb{R}^n$, though it is not for complex vector $x \in \mathbb{C}^n$.)

Program: lowpass.py

```
In [1]: 1 import matplotlib.pyplot as plt
2 from sound import Sound
3
4 sound = Sound('mono')
5 X, Y = sound.time, sound.data
6 Y3000 = sound.lowpass(3000)
7
8 plt.figure(figsize=(10, 5))
9 plt.subplot(121), plt.ylim(-1, 1)
10 plt.plot(X, Y), plt.plot(X, Y3000)
11 plt.subplot(122), plt.xlim(0.2, 0.21), plt.ylim(-1, 1)
12 plt.plot(X, Y), plt.plot(X, Y3000)
13 plt.show()
```

Line 6: Set the cutoff frequency to 3000, create the sound data of the low-pass filter, and save it in `mono3000.wav`.

Lines 9, 10: The waveforms of the original sound and the sound passed through the low-pass filter are displayed together (Fig. 6.12, left).

Lines 11, 12: Enlarge and display a part of it (Fig. 6.12, right). The waveform of the sound passed through the low-pass filter is a relatively smooth curve with sharp peaks containing high-frequency components removed.

Exercise 6.22 Listen to the sound of `mono3000.wav` and compare it with the original sound `mono.wav`. Also, display the power spectrum of the sound of `mono3000.wav`. Moreover, experiment with different cutoff frequencies to observe what happens.

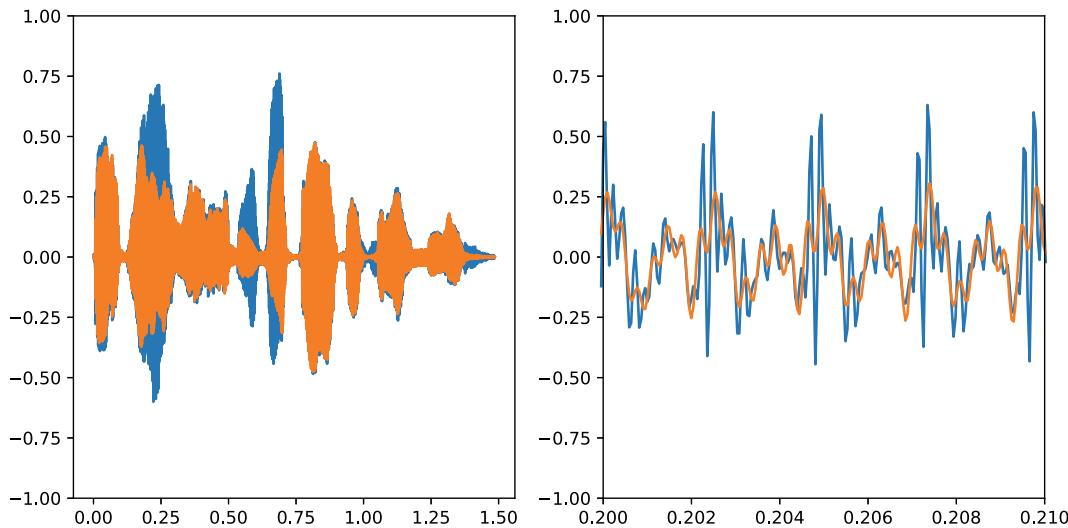


Fig. 6.12 Low-pass filter of audio data

The following program is an implementation example of Sound class.

Program: sound.py

```
In [1]: 1  from numpy import arange, fft
2  import scipy.io.wavfile as wav
3
4  class Sound:
5      def __init__(self, wavfile):
6          self.file = wavfile
7          self.rate, Data = wav.read(f'{wavfile}.wav')
8          dt = 1 / self.rate
9          self.len = len(Data)
10         self.tmax = self.len / self.rate
11         self.time = arange(0, self.tmax, dt)
12         self.data = Data.astype('float') / 32768
13         self.fft = fft.fft(self.data)
14
15     def power_spectrum(self, rng=None):
16         spectrum = abs(self.fft) ** 2
17         if rng is None:
18             r1, r2 = -self.len / 2, self.len / 2
19         else:
20             r1, r2 = rng[0] * self.tmax, rng[1] * self.tmax
21         R = arange(int(r1), int(r2))
22         return R / self.tmax, spectrum[R]
23
24     def lowpass(self, K):
25         k = int(K * self.tmax)
26         U = self.fft.copy()
27         U[range(k + 1, self.len - k)] = 0
28         V = fft.ifft(U).real
29         Data = (V * 32768).astype('int16')
30         wav.write(f'{self.file}{K}.wav', self.rate, Data)
31         return V
```

Lines 4–31: Define a class Sound, which uses three methods.

Lines 5–13: The first method called the *class initialization method*¹⁷ is executed when an object is created. Define here object variables that determine the attributes of the object. `Self` is a temporary name that will be replaced with the object name when an object of this class is created. The fast Fourier transform is also performed here.

Lines 15–22: Define `power_spectrum` method. A method for a class is the same as for a general function, but the name of the object that calls the method is passed to the first argument. We customarily use temporary name `self` as the first argument. When we actually call the `power_spectrum` method for the object, the object name is used as the prefix of the method and the first argument is omitted. Within a class definition, we may use the temporary prefix `self` like an object variable. Names with prefix `self` are common names throughout the class definition. Names without the prefix are local names effective within the method where the name is used.

Lines 24–31: Define the `low-pass` method. Make a copy on Line 26 so that the original Fourier transform data is not destroyed by the low-pass filter. Suppose that the elements of `U` are lined up like u_0, u_1, \dots, u_{n-1} . In Line 27, letting $u_i = 0$ for $k + 1 \leq i < n - k$, change array `U` like

$$u_0, u_1, \dots, u_k, 0, 0, \dots, 0, u_{n-k}, \dots, u_{n-1}.$$

Get the inverse Fourier transform of `U`. It becomes an array whose elements are complex numbers with imaginary part 0, but since they are still of complex type, we take only the real parts out. This is the data that has passed through the low-pass filter.

Exercise 6.23 Check that the inverse Fourier transform of `U` in Line 28 is an array of complex numbers with imaginary part 0 by drawing the graph of `fft.ifft(U).imag`. Consider the mathematical reason why the low-pass filter of a real vector by the Fourier transform and its inverse are also real vectors.

¹⁷ In the context of object-oriented programming, this is called a *constructor*, and an object created in the class is called its *instance*.



Eigenvalues and Eigenvectors

7

This chapter deals with the matrix eigenvalue problem, another major theme in linear algebra as important as the theory of linear equations. This problem is based on the *fundamental theorem of algebra*, which states that any polynomial with complex coefficients has a complex root. We will give a short proof of it. Though the proof needs some advanced knowledge of analysis, the reader will be able to grasp it with a little effort.

The matrix eigenvalue problem is very important also from a viewpoint of applications. We will study some concrete applications in subsequent chapters. In this chapter, we learn a process to compute eigenvalues and eigenvectors using Python in the relatively easy case where the orders of matrices are two or three. Moreover, we explain diagonalization of matrices and a method to realize it. Finally, we introduce the matrix norm and its related topics, where again we need some advanced knowledge of analysis.

7.1 Unitary Matrices and Hermitian Matrices

We denote by **diag** ($\lambda_1, \lambda_2, \dots, \lambda_n$) the diagonal matrix

$$\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

of order n whose diagonal elements are $\lambda_1, \lambda_2, \dots, \lambda_n$.

A diagonal matrix is expressed in NumPy and SymPy in the following manner:

```
In [1]: import numpy as np  
import sympy as sp  
np.diag([1, 2, 3])
```

```
Out[1]: array([[1, 0, 0],  
               [0, 2, 0],  
               [0, 0, 3]])
```

```
In [2]: sp.diag(1, 2, 3)
```

```
Out[2]: Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

Let A be a square matrix and A^* its adjoint matrix. If $A^* = A$, A is called a *Hermitian matrix*, and if $A^T = A$, A is called a *symmetric matrix*. If $A^*A = AA^* = I$, that is, $A^{-1} = A^*$, then A is called a *unitary matrix*. A *real matrix* is a matrix whose elements are all real numbers. For real matrices, a Hermitian matrix is identical to a symmetric matrix. A real unitary matrix A is usually called an *orthogonal matrix* and is specified by $A^{-1} = A^T$.

We consider the standard inner product $\langle \cdot | \cdot \rangle$ on \mathbb{K}^n and a square matrix A of order n . For any $x, y \in \mathbb{K}^n$, we have

$$\langle Ax | y \rangle = (Ax)^* y = (x^* A^*) y = x^* (A^* y) = \langle x | A^* y \rangle$$

by the definition of an inner product and the adjoint of matrix product (Exercise 4.12).

If A is Hermitian, using the above equality we have

$$\langle Ax | y \rangle = \langle x | A^* y \rangle = \langle x | Ay \rangle.$$

This equality characterizes that A is Hermitian. In fact, suppose that $\langle Ax | y \rangle = \langle x | Ay \rangle$ holds for any $x, y \in \mathbb{K}^n$. Then we have $\langle x | A^* y - Ay \rangle = 0$, and letting $x = A^* y - Ay$, we get $\|A^* y - Ay\|^2 = 0$, and so $A^* y = Ay$ for all $y \in \mathbb{K}^n$. Consequently, $A^* = A$ and A is Hermitian.

Unitary matrices are deeply related to the inner product and the norm. If A is a unitary matrix, then

$$\langle Ax | Ay \rangle = \langle x | A^* Ay \rangle = \langle x | y \rangle$$

for any $x, y \in \mathbb{K}^n$. From this, letting $y = x$, we also have $\|Ax\| = \|x\|$ for all $x \in \mathbb{K}^n$. In other words, a unitary matrix preserves the inner product and the norm, and we call such mapping $x \mapsto Ax$ a *unitary transformation*. A rotation matrix in \mathbb{R}^2 is an example of unitary matrices (orthogonal matrices).

Exercise 7.1 Prove that A is a unitary matrix if it preserves the inner product.¹

Write $A = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n]$ with column vectors $\mathbf{a}_1, \mathbf{a}_2, \dots$ and \mathbf{a}_n , then

$$\langle \mathbf{a}_i | \mathbf{a}_j \rangle = \mathbf{a}_i^* \mathbf{a}_j = (A^* A)_{ij},$$

where $(A^* A)_{ij}$ denotes the (i, j) -element of $A^* A$. Thus, $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ is an orthonormal basis of \mathbb{K}^n , if and only if $(A^* A)_{ij} = \delta_{ij}$, if and only if $A^* A = I$, if and only if A is unitary.

A square matrix A of order n is said to be *positive semi-definite* (or *nonnegative definite*) if $\langle x | Ax \rangle \geq 0$ for all $x \in \mathbb{K}^n$, and is said to be *positive definite* if $\langle x | Ax \rangle > 0$ for all $x \neq \mathbf{0}$.

Exercise 7.2 For a diagonal matrix $A = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$, show the following equivalences:

- (1) A is a Hermitian matrix $\Leftrightarrow \lambda_1, \dots, \lambda_n \in \mathbb{R}$,

¹ Moreover, it can be proved that A is a unitary matrix if A preserves the norm, using the polarization identity given in Sect. 6.1.

- (2) \mathbf{A} is a unitary matrix $\Leftrightarrow |\lambda_1| = \cdots = |\lambda_n| = 1$,
- (3) \mathbf{A} is a positive definite matrix $\Leftrightarrow \lambda_1, \dots, \lambda_n > 0$,
- (4) \mathbf{A} is a positive semi-definite matrix $\Leftrightarrow \lambda_1, \dots, \lambda_n \geq 0$.

When $\mathbb{K} = \mathbb{C}$, let us show that any positive semi-definite matrix is Hermitian. Let \mathbf{A} be a positive semi-definite matrix. Then, by the linearity of the inner product, we have

$$\langle \mathbf{x} + \mathbf{y} \mid \mathbf{A}(\mathbf{x} + \mathbf{y}) \rangle - \langle \mathbf{x} - \mathbf{y} \mid \mathbf{A}(\mathbf{x} - \mathbf{y}) \rangle = 2\langle \mathbf{x} \mid \mathbf{Ay} \rangle + 2\langle \mathbf{y} \mid \mathbf{Ax} \rangle$$

for all $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$. The value on the left-hand side is a real number. So the right-hand side is also a real number and must be equal to its complex conjugate

$$2\overline{\langle \mathbf{x} \mid \mathbf{Ay} \rangle} + 2\overline{\langle \mathbf{y} \mid \mathbf{Ax} \rangle} = 2\langle \mathbf{Ay} \mid \mathbf{x} \rangle + 2\langle \mathbf{Ax} \mid \mathbf{y} \rangle = 2\langle \mathbf{y} \mid \mathbf{A}^*\mathbf{x} \rangle + 2\langle \mathbf{x} \mid \mathbf{A}^*\mathbf{y} \rangle.$$

Hence, we have

$$\langle \mathbf{x} \mid (\mathbf{A} - \mathbf{A}^*)\mathbf{y} \rangle + \langle \mathbf{y} \mid (\mathbf{A} - \mathbf{A}^*)\mathbf{x} \rangle = 0.$$

Replacing \mathbf{y} by $i\mathbf{y}$, we have

$$i\langle \mathbf{x} \mid (\mathbf{A} - \mathbf{A}^*)\mathbf{y} \rangle - i\langle \mathbf{y} \mid (\mathbf{A} - \mathbf{A}^*)\mathbf{x} \rangle = 0.$$

Multiplying this by i and subtracting the result from the preceding equation, we obtain

$$2\langle \mathbf{x} \mid (\mathbf{A} - \mathbf{A}^*)\mathbf{y} \rangle = 0.$$

Since $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$ are arbitrary, we get $\mathbf{A} = \mathbf{A}^*$.

Remark that the positive semi-definiteness or the positive definiteness of matrices may vary depending on whether \mathbb{K} is \mathbb{R} or \mathbb{C} . For example, the matrix $\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ is positive definite over \mathbb{R} , since

$$\left\langle \begin{bmatrix} x \\ y \end{bmatrix} \mid \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \right\rangle = \left\langle \begin{bmatrix} x \\ y \end{bmatrix} \mid \begin{bmatrix} x-y \\ x+y \end{bmatrix} \right\rangle = x^2 + y^2.$$

But, it is not so over \mathbb{C} because it is not Hermitian (symmetric). In fact, we see²

$$\left\langle \begin{bmatrix} 1 \\ i \end{bmatrix} \mid \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ i \end{bmatrix} \right\rangle = \left\langle \begin{bmatrix} 1 \\ i \end{bmatrix} \mid \begin{bmatrix} 1-i \\ 1+i \end{bmatrix} \right\rangle = (1-i) - i(1+i) = 2 - 2i.$$

In particular, a positive definite matrix over \mathbb{R} is not necessarily Hermitian.

Exercise 7.3 Let $a, b, c \in \mathbb{R}$ and consider a symmetric matrix $\mathbf{A} = \begin{bmatrix} a & c \\ c & b \end{bmatrix}$. Prove that \mathbf{A} is positive definite over \mathbb{R} if and only if $a > 0$, $b > 0$ and $ab > c^2$.

² Remember to take the complex conjugates of the components of the first vector for the inner product in \mathbb{C}^2 .

Let W be a subspace of \mathbb{K}^n , then the orthogonal projection \mathbf{proj}_W onto W is a linear mapping from V to V (Exercise 6.8), which has the representation matrix denoted by \mathbf{P} with respect to the standard basis of \mathbb{K}^n . Let $A' = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k\}$ be an orthonormal basis of W and $A = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ be an orthonormal basis of \mathbb{K}^n including A' ($k \leq n$), then we have

$$\mathbf{P}\mathbf{a}_i = \mathbf{proj}_W(\mathbf{a}_i) = \begin{cases} \mathbf{a}_i & \text{if } i = 1, 2, \dots, k \\ \mathbf{0} & \text{if } i = k+1, \dots, n. \end{cases}$$

Thus, on this basis A the representation matrix of \mathbf{proj}_W is $\mathbf{D} = \mathbf{diag}(\underbrace{1, \dots, 1}_k, 0, \dots, 0)$, that is, $\mathbf{A}^{-1}\mathbf{P}\mathbf{A} = \mathbf{D}$ with the matrix $\mathbf{A} = [\mathbf{a}_1 \mathbf{a}_2 \dots \mathbf{a}_n]$ consisting of column vectors in A . Because A is orthonormal, A is a unitary matrix, and so $\mathbf{P} = \mathbf{A}\mathbf{D}\mathbf{A}^*$.³ Hence, because

$$\mathbf{P}^* = (\mathbf{A}\mathbf{D}\mathbf{A}^*)^* = \mathbf{A}\mathbf{D}^*\mathbf{A}^* = \mathbf{P},$$

\mathbf{P} is Hermitian. Moreover, we have

$$\mathbf{P}^2 = \mathbf{A}\mathbf{D}\mathbf{A}^*\mathbf{A}\mathbf{D}\mathbf{A}^* = \mathbf{A}\mathbf{D}^2\mathbf{A}^* = \mathbf{A}\mathbf{D}\mathbf{A}^* = \mathbf{P}.$$

Conversely, the equations $\mathbf{P} = \mathbf{P}^* = \mathbf{P}^2$ characterize that \mathbf{P} is the representation matrix of an orthogonal projection. Let \mathbf{P} be a square matrix satisfying this property and let $W = \text{range}(\mathbf{P})$. For $\mathbf{w} \in W$ take $\mathbf{w}' \in \mathbb{K}^n$ such that $\mathbf{P}\mathbf{w}' = \mathbf{w}$. Then, we have $\mathbf{P}\mathbf{w} = \mathbf{P}^2\mathbf{w}' = \mathbf{P}\mathbf{w}' = \mathbf{w}$. Now, for $\mathbf{x}, \mathbf{y} \in \mathbb{K}^n$, putting $\mathbf{w} = \mathbf{proj}_W(\mathbf{x})$, we have

$$\begin{aligned} \langle \mathbf{P}\mathbf{x} - \mathbf{w} \mid \mathbf{y} \rangle &= \langle \mathbf{P}\mathbf{x} \mid \mathbf{y} \rangle - \langle \mathbf{w} \mid \mathbf{y} \rangle \\ &= \langle \mathbf{P}\mathbf{x} \mid \mathbf{y} \rangle - \langle \mathbf{P}\mathbf{w} \mid \mathbf{y} \rangle \quad (\because \mathbf{P}\mathbf{w} = \mathbf{w}) \\ &= \langle \mathbf{x} \mid \mathbf{P}\mathbf{y} \rangle - \langle \mathbf{w} \mid \mathbf{P}\mathbf{y} \rangle \quad (\because \mathbf{P}^* = \mathbf{P}) \\ &= \langle \mathbf{x} - \mathbf{proj}_W(\mathbf{x}) \mid \mathbf{P}\mathbf{y} \rangle \\ &= 0. \quad (\because \mathbf{P}\mathbf{y} \in W \text{ and } (\mathbf{x} - \mathbf{proj}_W(\mathbf{x})) \perp W) \end{aligned}$$

Since \mathbf{y} is arbitrary, we have $\mathbf{P}\mathbf{x} = \mathbf{w} = \mathbf{proj}_W(\mathbf{x})$, that is, \mathbf{P} is the representation matrix of \mathbf{proj}_W . We thus refer to the matrix \mathbf{P} satisfying $\mathbf{P} = \mathbf{P}^* = \mathbf{P}^2$ also as an *orthogonal projection*.

Exercise 7.4 Let \mathbf{P} be an orthogonal projection in \mathbb{K}^n and let $\mathbf{x} (\neq \mathbf{0}) \in \mathbb{K}^n$. Show that a scalar $a \in \mathbb{K}$ satisfying $\mathbf{P}\mathbf{x} = a\mathbf{x}$ is only 0 or 1.

7.2 Eigenvalues

For a square matrix \mathbf{A} of order n , if there exist $\lambda \in \mathbb{K}$ and $\mathbf{x} \in \mathbb{K}^n$ not equal to $\mathbf{0}$ such that $\mathbf{Ax} = \lambda\mathbf{x}$, then we call λ an *eigenvalue* of \mathbf{A} and \mathbf{x} an *eigenvector* associated with the eigenvalue λ . The condition $\mathbf{x} \neq \mathbf{0}$ is crucial because $\mathbf{A}\mathbf{0} = \lambda\mathbf{0}$ always holds for any λ . On the other hand, an eigenvalue may be 0. For example, the matrix $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ has two eigenvalues 1 and 0, where $(1, 0)$ and $(0, 1)$ are eigenvectors associated with the eigenvalues 1 and 0, respectively. Since $\mathbf{Ax} = \lambda\mathbf{x}$ implies $\mathbf{A}\mathbf{ax} = \lambda\mathbf{ax}$ with $a \in \mathbb{K}$, an eigenvector permits any nonzero scalar multiplication.

³ This is an example of diagonalization with a unitary matrix. We will discuss it more generally in Sect. 7.3.

The expression $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ is transformed to $(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}$. Therefore, it is expressed as

$$\begin{bmatrix} a_{11} - \lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} - \lambda & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} - \lambda \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

with $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$. The value λ is an eigenvalue of \mathbf{A} if and only if this system of linear equations has a nonzero solution. It is equivalent to that $\mathbf{A} - \lambda\mathbf{I}$ is not a regular matrix, that is, $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$. Hence, λ is an eigenvalue of \mathbf{A} if and only if it satisfies

$$\begin{vmatrix} a_{11} - \lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} - \lambda & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} - \lambda \end{vmatrix} = 0.$$

The left-hand side is a polynomial in λ of degree n called the *characteristic polynomial* or the *eigen-polynomial*, and we call this equation the *characteristic equation* or *eigenequation* of \mathbf{A} .

Theorem 7.1 (The fundamental theorem of algebra) *A polynomial equation of degree n in λ with complex coefficients*

$$c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_1\lambda + c_0 = 0 \quad (c_n \neq 0)$$

has exactly n roots $\alpha_1, \alpha_2, \dots, \alpha_n$ (including multiple roots) in \mathbb{C} , and it can be factored into

$$c_n(\lambda - \alpha_1)(\lambda - \alpha_2) \cdots (\lambda - \alpha_n) = 0.$$

Proof Put $f(x) \stackrel{\text{def}}{=} c_nx^n + c_{n-1}x^{n-1} + \cdots + c_1x + c_0$ and suppose $n \geq 1$. We will show that $f(x) = 0$ has a solution in \mathbb{C} . Since $|f(x)| \geq 0$, $|f(x)|$ takes the minimum⁴ at $x = a$ for some $a \in \mathbb{C}$. For the sake of contradiction, assume $|f(a)| > 0$.⁵ We have

$$\begin{aligned} f(x+a) &= c_n(x+a)^n + \cdots + c_1(x+a) + c_0 \\ &= (\text{a polynomial in } x \text{ with terms of degree not less than 1}) + f(a). \end{aligned}$$

Hence by putting $g(x) = \frac{f(x+a)}{f(a)}$, we have

$$g(x) = c'_n x^n + \cdots + c'_1 x + 1$$

⁴ The existence of this minimum is not trivial. This is proved from the result that any real-valued continuous function on a bounded closed subset of \mathbb{C} attains the minimum.

⁵ The reader familiar with the complex function theory may immediately reach a contradiction. Let $g(x) = 1/f(x)$, then $|g(x)| \leq 1/|f(a)|$ by assumption, so $g(x)$ would be a bounded holomorphic function defined on the entire complex plane. It must be a constant function by Liouville's theorem, and hence $f(x)$ must also be constant, a contradiction.

with $c'_n, \dots, c'_1 \in \mathbb{C}$ ($c'_n \neq 0$), and $|g(x)|$ takes the minimum 1 at $x = 0$.

Let m be the least number such that $c'_m \neq 0$ and let $\alpha = \sqrt[m]{-c'_m}$ be the m -th root of $-c'_m$. Put $h(x) = g\left(\frac{x}{\alpha}\right)$, then we have

$$h(x) = c''_nx^n + \dots + c''_{m+1}x^{m+1} - x^m + 1$$

with $c''_n, \dots, c''_{m+1} \in \mathbb{C}$, and $|h(x)|$ takes the minimum 1 at $x = 0$.⁶ Let x be a real number, then we have

$$|h(x)|^2 = \overline{h(x)}h(x) = x^{m+1}p(x) - 2x^m + 1,$$

where $p(x)$ is a polynomial in x with real coefficients. Since $xp(x) \rightarrow 0$ as $x \rightarrow 0$, we can realize $xp(x) < 2$ for a sufficiently small $x > 0$. Consequently, we have reached

$$|h(x)|^2 = x^m(xp(x) - 2) + 1 < 1,$$

which contradicts that the minimum of $|h(x)|$ is 1. ■

This theorem only guarantees the existence of a solution, but does not provide any method to find a solution. There is another great theorem which states that there does not exist a formula⁷ for the solutions of a fifth (and thus higher) degree polynomial equation.⁸ This means that there does not exist a general procedure to find a solution by a finite number of steps of ordinary algebraic operations. On the other hand, a system of linear equations can be solved by a finite number of steps of addition, multiplication, and division, using Gaussian elimination. Moreover, as stated at the end of Chap. 5 we have Cramer's formula for the solution.

An eigenvalue can be found by solving the characteristic equation, which is a polynomial equation, and so the fundamental theorem of algebra guarantees the existence of eigenvalue. Once an eigenvalue is found, an eigenvector associated with the eigenvalue can be obtained by solving a system of linear equations. When dealing with the matrix eigenvalue problem, we usually consider it in a complex linear space, that is, $\mathbb{K} = \mathbb{C}$, because even a real matrix may have an imaginary eigenvalue.

Exercise 7.5

Prove that a matrix A is regular if and only if A has no zero eigenvalue.

Now, let us consider a square matrix A of order 2. If $\text{rank } A = 0$, then A is the zero matrix and its eigenvalue is only 0. In this case, any nonzero vector x is an eigenvector associated with eigenvalue 0. If $\text{rank } A = 1$, then any nonzero vector in $\text{kernel}(A)$ is an eigenvector associated with eigenvalue 0, and since the dimension of $\text{range}(A)$ is 1, any vector in $\text{range}(A)$, if it is not in $\text{kernel}(A)$, is an eigenvector associated with another eigenvalue than 0. Typically, the orthogonal projection onto a one-dimensional subspace has two eigenvalues 0 and 1 (Exercise 7.4) and eigenvectors associated with the respective eigenvalues are orthogonal to each other.

⁶If $n = m$, then $c''_n = \dots = c''_{m+1} = 0$ and $h(x) = -x^m + 1$.

⁷The formula for a quadratic equation is well known. There also exist the formulas for cubic and quartic equations, but they are not so frequently used.

⁸More precisely, in general, a root cannot be described by a finite number of combinations of arithmetic operations and radicals (inverse operations of exponential power) of the coefficients of a given equation. Its proof needs a profound and beautiful theory known as *Galois theory*.

Next, suppose that $\text{rank } A = 2$. Let us first consider the case where A has only one eigenvalue $a \neq 0$. In this case, the characteristic equation is of the form $(\lambda - a)^2 = 0$. Matrix A with one of the following forms falls in this case:

$$(1) \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \quad (2) \begin{bmatrix} a & b \\ 0 & a \end{bmatrix} \text{ or } \begin{bmatrix} a & 0 \\ b & a \end{bmatrix} (b \neq 0).$$

In either case, a is the only eigenvalue of A . In case (1), any nonzero vector can be an eigenvector. In case (2), by putting $x = \begin{bmatrix} x \\ y \end{bmatrix}$, $Ax = ax$ is expanded as

$$\begin{cases} ax + by = ax \\ 0x + ay = ay \end{cases} \quad \text{or} \quad \begin{cases} ax + 0y = ax \\ bx + ay = ay, \end{cases}$$

and we see that $(1, 0)$ and $(0, 1)$ are the unique eigenvectors up to scalar multiplication for the matrices left and right in (2), respectively. If there are two nonzero distinct eigenvalues, the respective eigenvectors must be linearly independent and so the matrix must be of rank 2. When A is a real matrix, there are two cases where it has two real eigenvalues and where it has a pair of complex eigenvalues which are conjugate to each other. The following program randomly generates a real matrix, then finds its eigenvalues and eigenvectors.

Program: prob1.py

```
In [1]: 1 from sympy import Matrix
2 from numpy.random import seed, choice
3
4 N = [-3, -2, -1, 1, 2, 3]
5 seed(2021)
6
7 def f(truth):
8     while True:
9         A = Matrix(choice(N, (2, 2)))
10        eigenvals= A.eigenvals()
11        if len(eigenvals) == 2 and not 0 in eigenvals:
12            if all([x.is_real for x in eigenvals]) == truth:
13                print(eigenvals)
14                return A
```

Line 4: N is the list of values chosen for elements of a matrix to be generated.

Lines 7–14: Set Boolean value `True` or `False` in the argument, depending on whether the eigenvalues are two distinct real numbers or imaginary numbers. Repeat the process until the given condition is satisfied. Line 9 randomly generates a square matrix A with elements from N . Line 10 uses a method for computing the eigenvalues of A , which returns a dictionary whose key is an eigenvalue and value is its multiplicity.⁹ Line 11 checks that there are two nonzero eigenvalues and Line 12 checks that the eigenvalues satisfy the condition indicated by the argument of `f`. The program prints the dictionary of eigenvalues and finally returns the matrix.

```
In [2]: f(True)
```



```
{3 - sqrt(2): 1, sqrt(2) + 3: 1}
Matrix([
[3, 2],
[1, 3]])
```

⁹ Multiplicity will be explained later in this section.

In [3]: `f(False)`

 {5/2 - sqrt(35)*I/2: 1, 5/2 + sqrt(35)*I/2: 1}
`Matrix([`
`[3, 3],`
`[-3, 2]])`

Exercise 7.6 Compute the eigenvalues and eigenvectors of a matrix generated by the program above without the aid of a computer. Next, solve the same problem with the aid of a computer.

For a square matrix of order 3, the computation may be more complicated, so let us proceed step

by step with the aid of a computer. Let $A = \begin{bmatrix} 3 & -4 & 2 \\ 2 & -3 & 2 \\ 3 & -6 & 4 \end{bmatrix}$.

In [1]: `from sympy import *`
`A = Matrix([[3, -4, 2], [2, -3, 2], [3, -6, 4]])`
`f = det(A - var('lmd') * eye(3)); f`

Out[1]: `-lmd**3 + 4*lmd**2 - 5*lmd + 2`

We can use `var` also inside a sentence. The symbol `eye(3)` denotes the unit matrix $I = I_3$, and `f` represents $\det(A - \lambda I)$.

In [2]: `factor(f)`

Out[2]: `-(lmd - 2)*(lmd - 1)**2`

The function `factor` gives the factorization of `f`, which shows that $\lambda = 1, 2$ are eigenvalues.

Next, let us compute eigenvectors associated with the respective eigenvalues λ , namely, find nonzero solutions of $(A - \lambda I)v = \mathbf{0}$. Let $v = (x, y, z)$ and $w(\lambda) = (A - \lambda I)v$. We use the function `Lambda` in SymPy, which makes a lambda expression with symbols as variables.

In [3]: `v = Matrix([var('x'), var('y'), var('z')]); v`

Out[3]: `Matrix([`
`[x],`
`[y],`
`[z]])`

In [4]: `w = Lambda(lmd, (A - lmd * eye(3)) * v); w`

Out[4]: `Lambda(lmd, Matrix([
[x*(3 - lmd) - 4*y + 2*z],
[2*x + y*(-lmd - 3) + 2*z],
[3*x - 6*y + z*(4 - lmd)])))`

Solving $w(1) = (A - 1 \cdot I)v = \mathbf{0}$ in x, y, z , substitute the obtained result for v .

In [5]: `ans = solve(w(1)); ans`

Out[5]: `{x: 2*y - z}`

```
In [6]: v.subs(ans)
```

```
Out[6]: Matrix([
[2*y - z],
[y],
[z]])
```

Any nonzero vector on the plane $x = 2y - z$ is an eigenvector associated with eigenvalue 1. From this we can take two linearly independent eigenvectors $(2, 1, 0)$ by letting $z = 0$, $y = 1$ and $(-1, 0, 1)$ by letting $z = 1$, $y = 0$.

For eigenvalue $\lambda = 2$, it gives $x = y = \frac{2z}{3}$ as follows. By letting $z = 3$, for example, we obtain an eigenvector $(2, 2, 3)$.

```
In [7]: ans = solve(w(2)); ans
```

```
Out[7]: {y: 2*z/3, x: 2*z/3}
```

Sympy provides a method to directly compute eigenvalues.

```
In [8]: A.eigenvals()
```

```
Out[8]: {2: 1, 1: 2}
```

A solution is of a dictionary form, whose key is an eigenvalue and value is its multiplicity.

There is also a method for computing eigenvalues and eigenvectors simultaneously in SymPy.

```
In [9]: A.eigenvects()
```

```
Out[9]: [(1, 2, [Matrix([
[2],
[1],
[0]]), Matrix([
[-1],
[0],
[1]])), (2, 1, [Matrix([
[2/3],
[2/3],
[-1]])])]
```

As a result, the list of two tuples $(1, 2, [(2, 1, 0), (-1, 0, 1)])$ and $(2, 1, (2/3, 2/3, 1))$ has been returned. Each tuple is the list of eigenvalue, multiplicity, and eigenvector(s) in order.

The *multiplicity* of an eigenvalue of A means the number of occurrences of it in the complete factorization of the characteristic polynomial of A . If eigenvalues are distinct, eigenvectors associated with the respective eigenvalues are linearly independent (see the next section). If there is more than one linearly independent eigenvector associated with one eigenvalue λ , any nonzero vector in the subspace spanned by those eigenvectors is also an eigenvector. We call this subspace W the *eigenspace* of A associated with λ :

$$W \stackrel{\text{def}}{=} \{x \in \mathbb{K}^n \mid Ax = \lambda x\}.$$

The dimension of the eigenspace does not necessarily coincide with the multiplicity of the eigenvector. More details will be discussed in the next chapter pertaining to Jordan normal form.

The module `linalg` of NumPy has two functions `eig` and `eigh` to compute eigenvalues and eigenvectors. The former can be used for any matrices, while the latter for Hermitian matrices. Here, let us use `eig`.

```
In [1]: from numpy.linalg import eig, norm
A = [[3, -2, 2], [2, -1, 4], [2, -2, 1]]
lmd, vec = eig(A)
lmd

Out[1]: array([1.+0.j, 1.+2.j, 1.-2.j])

In [2]: vec[:, 0]

Out[2]: array([ 7.07106781e-01+0.j, 7.07106781e-01+0.j, -3.35214071e-16+0.j])

In [3]: vec[:, 1]

Out[3]: array([-0.47434165-0.15811388j, -0.79056942+0.j, -0.15811388-0.31622777j])

In [4]: vec[:, 2]

Out[4]: array([-0.47434165+0.15811388j, -0.79056942-0.j, -0.15811388+0.31622777j])

In [5]: [norm(vec[:, n]) for n in range(3)]

Out[5]: [1.0, 1.0, 1.0]
```

Eigenvectors are normalized.

For a matrix having an eigenvalue of multiplicity more than one, let us compare the results obtained by solving the eigenvalue problem in SymPy and Numpy.

Program: eig1.py

```
In [1]: 1 import sympy as sp
2
3 A = [[1, 1], [0, 1]]
4 a = sp.Matrix(A).eigenvals()
5 print(f'''eigen value: {a[0][0]}
6 multiplicity: {a[0][1]}
7 eigen vector:
8 {a[0][2][0]}''')
```

Line 3: $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ is the matrix to be investigated.

Line 4: a is the result of the `eigenvals` method of the `Matrix` class of SymPy.



```
eigen value: 1
multiplicity: 2
eigen vector:
Matrix([[1], [0]])
```

Program: eig2.py

```
In [1]: 1 import numpy as np
2
3 A = [[1, 1], [0, 1]]
4 b = np.linalg.eig(A)
5 print(f'''eigen values: {b[0][0]}, {b[0][1]}
6 eigen vectors:
7 {b[1][:, 0]}
8 {b[1][:, 1]}'''')
```

Line 4: b is the result of the `eig` function in the `linalg` module of NumPy.



```
eigen values: 1.0, 1.0
eigen vectors:
[1. 0.]
[-1.0000000e+00  2.22044605e-16]
```

The matrix A has the unique eigenvalue 1 of multiplicity 2 and the unique associated eigenvector $(1, 0)$ up to scalar multiplication. SymPy returns the correct answer. On the other hand, NumPy returns two equal eigenvalues 1 and 1 and two eigenvectors associated with them; one is $(1, 0)$ and the other is almost equal to $(-1, 0)$. We have to be careful when determining the multiplicity with NumPy.

Exercise 7.7 The program below randomly generates a square matrix of order 3 whose characteristic polynomial can be factorized into linear factors over \mathbb{Z} , and so the eigenvalues are all integers. Calculate the eigenvalues and eigenvectors of a matrix generated by the program without the aid of a computer.

Program: prob2.py

```
In [1]: 1 from sympy import Matrix, Symbol, factor_list, factor
2 from numpy.random import choice, seed
3
4 seed(2021)
5 D = [-5, -4, -3, -2, -1, 1, 2, 3, 4, 5]
6
7 def f():
8     while True:
9         A = Matrix(choice(D, (3, 3)))
10        cp = A.charpoly(Symbol('lmd'))
11        F = factor_list(cp)
12        if len(F[1]) == 3:
13            print(f'det(A - lmd*I) = {factor(cp.expr)}\nA = {A}')
14            return A
```

Line 4: If the argument of `seed` is changed, the generated matrix will change.

Line 5: Choose elements of a matrix from this list.

Line 10: `charpoly` is a method of computing the characteristic equation. The argument is a symbol used in the equation.

Line 11: `factor_list` returns an ordered pair whose first component `F[0]` is a constant factor and the second component `F[1]` is the list of pairs of a non-constant irreducible factor and its degree.

Line 12: Since a generated matrix is of order 3, the characteristic equation is cubic. Therefore, when the number `len(F[1])` of distinct irreducible factors is 3, the characteristic equation has three distinct integer roots.

Line 13: Print the factorized characteristic polynomial.

In [2]:

```
f()

det(A - lmd*I) = lmd*(lmd - 2)*(lmd + 4)
A = Matrix([[5, -5, -1], [3, -2, -2], [4, -1, -5]])
Matrix([
[5, -5, -1],
[3, -2, -2],
[4, -1, -5]])
```



7.3 Diagonalization

Let $\lambda_1, \lambda_2, \dots, \lambda_k$ be the distinct eigenvalues of A and $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k$ eigenvectors associated with them, respectively. We will prove that these vectors are linearly independent by induction on k . When $k = 1$, the assertion is true because \mathbf{v}_1 is not zero. Suppose $k > 1$ and assume that $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_{k-1}\}$ is linearly independent. We will bring a contradiction, by assuming that $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ is linearly dependent. Then \mathbf{v}_k is a linear combination of $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_{k-1}$, that is,

$$\mathbf{v}_k = a_1 \mathbf{v}_1 + \cdots + a_{k-1} \mathbf{v}_{k-1} \quad \cdots \quad (*)$$

for $a_1, \dots, a_{k-1} \in \mathbb{K}$. Operating A on both sides, we have

$$\lambda_k \mathbf{v}_k = a_1 \lambda_1 \mathbf{v}_1 + \cdots + a_{k-1} \lambda_{k-1} \mathbf{v}_{k-1},$$

and eliminating \mathbf{v}_k by using $(*)$, we obtain

$$a_1 (\lambda_k - \lambda_1) \mathbf{v}_1 + \cdots + a_{k-1} (\lambda_k - \lambda_{k-1}) \mathbf{v}_{k-1} = \mathbf{0}.$$

Since $\{\mathbf{v}_1, \dots, \mathbf{v}_{k-1}\}$ is linearly independent, all the coefficients must be 0. However, by the assumption that λ_k is not equal to any of $\lambda_1, \lambda_2, \dots, \lambda_{k-1}$, we have $a_1 = \cdots = a_{k-1} = 0$, which yields $\mathbf{v}_k = \mathbf{0}$ by $(*)$, a contradiction. Consequently, $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ is linearly independent.

A square matrix A is said to be *diagonalizable* if there exists a regular matrix V such that $V^{-1}AV$ is a diagonal matrix. We shall show that a necessary and sufficient condition for A to be diagonalizable is that the set of eigenvectors of A forms a basis of \mathbb{K}^n .

To prove the necessity, suppose that there exists a regular matrix V such that $V^{-1}AV = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$. Then, with the standard basis $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$, we have $V^{-1}AV\mathbf{e}_i = \lambda_i \mathbf{e}_i$ for $i = 1, 2, \dots, n$. Applying V from the left on both sides, we have $A\mathbf{V}\mathbf{e}_i = \lambda_i V\mathbf{e}_i$ and hence $\{V\mathbf{e}_1, V\mathbf{e}_2, \dots, V\mathbf{e}_n\}$ is a basis consisting of the eigenvectors of A .

To prove the sufficiency, suppose that $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ is a basis consisting of eigenvectors \mathbf{v}_i of A ; $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$ for $i = 1, \dots, n$. Make the matrix $V = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n]$ by arranging these vectors as column vectors. Then, V is a regular matrix and we have $V^{-1}\mathbf{v}_i = \mathbf{e}_i$ for $i = 1, \dots, n$. Hence, we have

$$V^{-1}AV = V^{-1}[\lambda_1 \mathbf{v}_1 \ \lambda_2 \mathbf{v}_2 \ \cdots \ \lambda_n \mathbf{v}_n] = [\lambda_1 \mathbf{e}_1 \ \lambda_2 \mathbf{e}_2 \ \cdots \ \lambda_n \mathbf{e}_n].$$

Therefore, A is diagonalized by V .

As discussed in Sect. 3.4, V is a basis change matrix. In general, if $B = V^{-1}AV$ with regular matrix V , that is, A and B are similar, then

$$|B - \lambda I| = |V^{-1}AV - \lambda I| = |V^{-1}| |A - \lambda I| |V| = |A - \lambda I|,$$

and so the characteristic polynomials of A and B coincide. Therefore, their eigenvalues coincide including multiplicities. In this sense, the eigenvalues are matrix invariants.

When A is diagonalized to $\text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$, then $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of A and the same eigenvalue occurs as many times as the multiplicity. Moreover, we see that the multiplicity of an eigenvalue is equal to the dimension of the eigenspace. If, in particular, A has n distinct eigenvalues, then the eigenvectors associated with the respective eigenvalues are linearly independent and form a basis. Therefore, A is diagonalizable.

Executing `prob2.py` used in Exercise 7.7, we get a matrix with three distinct eigenvalues.

```
In [1]: A = f()
```

```
Out[1]: det(A - lmd*I) = lmd*(lmd - 2)*(lmd + 4)
A = Matrix([[5, -5, -1], [3, -2, -2], [4, -1, -5]])
```

To diagonalize this matrix, we first compute its eigenvectors and then make the matrix V by arranging them as column vectors.

```
In [2]: X = A.eigenvects()
u, v, w = [e for x in X for e in x[2]]
V = u.row_join(v).row_join(w); V
```

```
Out[2]: Matrix([
[4/11, 8/5, 2],
[5/11, 7/5, 1],
[ 1,   1,  1]])
```

A can be diagonalized by V .

```
In [3]: V**(-1) * A * V
```

```
Out[3]: Matrix([
[-4, 0, 0],
[ 0, 0, 0],
[ 0, 0, 2]])
```

Here the elements on the diagonal line are the eigenvalues of A . As the order of eigenvectors is changed, the order of eigenvalues is accordingly changed.

Exercise 7.8 Make another matrix with `prob2.py`, and diagonalize it in the same manner as above.

Exercise 7.9 Suppose that a matrix A is diagonalized as $V^{-1}AV = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ by a regular matrix V . Prove the following results from (1) to (4):

- (1) $\text{rank}(A)$ is equal to the number of nonzero elements of $\lambda_1, \lambda_2, \dots, \lambda_n$,
- (2) $\det(A) = \lambda_1\lambda_2 \cdots \lambda_n$,
- (3) $\text{Tr}(A) = \lambda_1 + \lambda_2 + \cdots + \lambda_n$,
- (4) $A^{-1} = V \text{diag}(\lambda_1^{-1}, \lambda_2^{-1}, \dots, \lambda_n^{-1}) V^{-1}$, if A is regular.

A matrix A is called a *normal matrix* if it satisfies $A^*A = AA^*$. Hermitian matrices and unitary matrices are examples of normal matrices. If A is a real matrix, then A is a normal matrix if and only if $A^T A = AA^T$. Real symmetric matrices and orthogonal matrices are normal matrices.

When A is a real matrix of order 2, is there any other normal matrix? Putting $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, let us solve the equation $A^T A = AA^T$ in a, b, c , and d .

```
In [1]: from sympy import *
from sympy.abc import a, b, c, d
A = Matrix([[a, b], [c, d]])
solve(A.T * A - A * A.T)
```

```
Out[1]: [{b: c}, {b: -c, a: d}, {b: 0, c: 0}]
```

It tells us that there are three cases. The first case is that $b = c$, which implies that A is symmetric. In the second case where $b = -c$ and $a = d$, the column vectors (and also the row vectors) of A form an orthogonal system, and if moreover they form an orthonormal system, A is an orthogonal matrix. In the last case where $b = c = 0$, A is diagonal.

Exercise 7.10 Confirm the result above without using SymPy.

Now let us investigate properties of a normal matrix A over $\mathbb{K} = \mathbb{C}$. First, for any vectors $x, y \in \mathbb{K}^n$ we have

$$\langle Ax | Ay \rangle = \langle A^*Ax | y \rangle = \langle AA^*x | y \rangle = \langle A^*x | A^*y \rangle.$$

Let λ be an eigenvalue of A and v be an eigenvector associated with it. Since $Av - \lambda v = \mathbf{0}$, we have

$$\begin{aligned} 0 &= \|Av - \lambda v\|^2 = \langle Av - \lambda v | Av - \lambda v \rangle \\ &= \langle Av | Av \rangle - \langle Av | \lambda v \rangle - \langle \lambda v | Av \rangle + \langle \lambda v | \lambda v \rangle \\ &= \langle A^*v | A^*v \rangle - \langle \bar{\lambda}v | A^*v \rangle - \langle A^*v | \bar{\lambda}v \rangle + \langle \bar{\lambda}v | \bar{\lambda}v \rangle \\ &= \langle A^*v - \bar{\lambda}v | A^*v - \bar{\lambda}v \rangle = \|A^*v - \bar{\lambda}v\|^2, \end{aligned}$$

and hence we find $A^*v - \bar{\lambda}v = \mathbf{0}$. It follows that

$$\begin{aligned} \lambda &\text{ is an eigenvalue of } A \text{ and } v \text{ is an eigenvector associated with it} \\ \Rightarrow \bar{\lambda} &\text{ is an eigenvalue of } A^* \text{ and } v \text{ is an eigenvector associated with it.} \end{aligned}$$

Let λ and μ be two distinct eigenvalues of A , and v and w be eigenvectors associated with them, respectively. Then, because

$$\lambda \langle v | w \rangle = \langle \bar{\lambda}v | w \rangle = \langle A^*v | w \rangle = \langle v | Aw \rangle = \langle v | \mu w \rangle = \mu \langle v | w \rangle$$

and $\lambda \neq \mu$, we obtain $\langle v | w \rangle = 0$. Namely, we see the following:

Eigenvectors associated with distinct eigenvalues of A are orthogonal to each other.

Let $\lambda_1, \dots, \lambda_k$ be all the distinct eigenvalues of A and W_1, \dots, W_k the eigenspaces associated with them, respectively. We make an orthonormal basis of W_i for each $i = 1, \dots, k$ and gather all of them together, then the resulting set becomes an orthonormal system of \mathbb{K}^n , consisting of eigenvectors of A . We claim that this forms a basis of \mathbb{K}^n . Let W be the subspace spanned by this set, and we shall prove that $W = \mathbb{K}^n$ by contradiction. Assume $W \neq \mathbb{K}^n$, then, we have $W^\perp \neq \{\mathbf{0}\}$. For any $v \in W^\perp$ and $w \in W_i$ we have

$$\langle w | Av \rangle = \langle A^*w | v \rangle = \langle \bar{\lambda}_i w | v \rangle = \lambda_i \langle w | v \rangle = 0.$$

Hence $Av \in W_i^\perp$ for $i = 1, \dots, k$, and so $Av \in W^\perp$. Thus, the mapping $v \mapsto Av$ induces a linear mapping from W^\perp to itself. Let $\{v_1, \dots, v_l\}$ be a basis of W^\perp and let B be the matrix representing this linear mapping based on this basis. By the fundamental theorem of algebra, B has an eigenvalue μ and eigenvector $x \in W^\perp$, that is, $Bx = \mu x$. Then, A also maps x to μx . Therefore, x is an eigenvector

associated with the eigenvalue μ of A as well. However, $x \notin W$ and this contradicts the way of construction of W . Consequently, we see the following:

There is an orthonormal basis $\{u_1, u_2, \dots, u_n\}$ of \mathbb{K}^n consisting of eigenvectors of A .

Thus, we have a unitary matrix $U \stackrel{\text{def}}{=} [u_1 \ u_2 \ \cdots \ u_n]$ arranging these column vectors, and U^*AU becomes a diagonal matrix. Conversely, if A is diagonalized as $U^*AU = \text{diag}(\lambda_1, \dots, \lambda_n)$ with unitary matrix U , then A is normal. In fact, if A is so diagonalized, then $A = UDU^*$ and $A^* = U\bar{D}U^*$ with $D = \text{diag}(\lambda_1, \dots, \lambda_n)$ and $\bar{D} = \text{diag}(\bar{\lambda}_1, \dots, \bar{\lambda}_n)$, and we have

$$A^*A = U\bar{D}U^*UDU^* = U\bar{D}DU^* = UDDU^* = UDU^*U\bar{D}U^* = AA^*.$$

Consequently, we conclude the following:

A is diagonalizable by a unitary matrix over \mathbb{C} if and only if it is normal.

Let us diagonalize $A = \begin{bmatrix} i & i \\ -i & i \end{bmatrix}$ which is normal but not real nor Hermitian. First, we check that it is certainly a normal matrix.

```
In [1]: from sympy import *
A = Matrix([[I, I], [-I, I]])
A * A.H - A.H * A
```

```
Out[1]: Matrix([
[0, 0],
[0, 0]])
```

Next, we compute its eigenvalues and eigenvectors.

```
In [2]: X = A.eigenvecs(); X
```

```
Out[2]: [(-1 + I, 1, [Matrix([
[-I],
[1]])), (1 + I, 1, [Matrix([
[I],
[1]]))])]
```

Finally, we make a basis change matrix U by normalizing each eigenvector, and obtain U^*AU as the result of diagonalization, where the eigenvalues appear on the diagonal line.

```
In [3]: B = [v / v.norm() for x in X for v in x[2]]
U = B[0].row_join(B[1]); U
```

```
Out[3]: Matrix([
[-sqrt(2)*I/2, sqrt(2)*I/2],
[sqrt(2)/2, sqrt(2)/2]])
```

```
In [4]: simplify(U.H * A * U)
```

```
Out[4]: Matrix([
[-1 + I, 0],
[0, 1 + I]])
```

The `Matrix` class of SymPy provides a method for diagonalizing a normal matrix by a regular matrix.

```
In [5]: A.diagonalize()
```

```
Out[5]: (Matrix([
[-I, I],
[1, 1]]), Matrix([
[1 - I, 0],
[0, 1 + I]]))
```

As defined in Chap. 4, matrices \mathbf{A} and \mathbf{B} are similar if there exists a regular matrix \mathbf{V} such that $\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{B}$. If it is possible to take a unitary matrix for \mathbf{V} , then \mathbf{A} and \mathbf{B} are said to be *unitarily equivalent*. We can rephrase the last result as

\mathbf{A} is unitarily equivalent to a diagonal matrix if and only if it is normal.

A common property among similar matrices is called an *invariant*, for example, the value of determinant or rank and the property of being regular or diagonalizable are invariants. A property invariant under unitary equivalence is said to be a *unitary invariant*.

Exercise 7.11 Prove that the properties of being (1) normal, (2) Hermitian, (3) unitary, (4) positive (semi-)definite, and (5) an orthogonal projection are all unitary invariants over \mathbb{C} .

Let \mathbf{A} be a normal matrix. Suppose that $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of \mathbf{A} with multiplicity, that is, the same eigenvalue occurs as many times as the multiplicity. Then, by Exercises 7.2, 7.3, and 7.11 and the invariance of the eigenvalues, we have the following equivalences over \mathbb{C} :

1. \mathbf{A} is a Hermitian matrix $\Leftrightarrow \lambda_1, \dots, \lambda_n \in \mathbb{R}$,
2. \mathbf{A} is a unitary matrix $\Leftrightarrow |\lambda_1| = \dots = |\lambda_n| = 1$,
3. \mathbf{A} is a positive definite matrix $\Leftrightarrow \lambda_1, \dots, \lambda_n > 0$,
4. \mathbf{A} is a positive semi-definite matrix $\Leftrightarrow \lambda_1, \dots, \lambda_n \geq 0$,
5. \mathbf{A} is an orthogonal projection $\Leftrightarrow \lambda_1, \dots, \lambda_n \in \{0, 1\}$.

Let \mathbf{A} be a positive semi-definite matrix over $\mathbb{K} = \mathbb{C}$ or a positive semi-definite symmetric matrix over $\mathbb{K} = \mathbb{R}$. Then, \mathbf{A} is diagonalized by a unitary matrix \mathbf{U} , that is, $\mathbf{U}^* \mathbf{A} \mathbf{U} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$. Since the last matrix is also positive semi-definite, we see $\lambda_1, \dots, \lambda_n \geq 0$. For $k \in \mathbb{N}$ let $\mathbf{B} \stackrel{\text{def}}{=} \mathbf{U} \text{diag}(\sqrt[k]{\lambda_1}, \sqrt[k]{\lambda_2}, \dots, \sqrt[k]{\lambda_n}) \mathbf{U}^*$. Then, \mathbf{B} is a positive semi-definite matrix over \mathbb{K} , and

$$\mathbf{B}^k = \mathbf{U} \left(\text{diag}(\sqrt[k]{\lambda_1}, \sqrt[k]{\lambda_2}, \dots, \sqrt[k]{\lambda_n}) \right)^k \mathbf{U}^* = \mathbf{U} \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n) \mathbf{U}^* = \mathbf{A}.$$

We call this \mathbf{B} the k -th root of \mathbf{A} and denote it by $\sqrt[k]{\mathbf{A}}$ (the square root $\sqrt[2]{\mathbf{A}}$ is simply written $\sqrt{\mathbf{A}}$). Suppose that \mathbf{A} satisfies $\langle \mathbf{x} | \mathbf{A}\mathbf{x} \rangle = 0$ for all $\mathbf{x} \in \mathbb{K}^n$. Then, we have

$$\langle \sqrt{k}\mathbf{x} | \sqrt{k}\mathbf{x} \rangle = \langle \mathbf{x} | \mathbf{A}\mathbf{x} \rangle = 0.$$

It follows that $Ax = \sqrt{A}(\sqrt{A}x) = \mathbf{0}$ for all $x \in \mathbb{K}^n$, and hence $A = \mathbf{0}$. Thus, the zero matrix is the only positive semi-definite matrix A over \mathbb{C} (or positive semi-definite symmetric matrix A over \mathbb{R}) satisfying $x \perp Ax$ all $x \in \mathbb{K}^n$. Over \mathbb{R} the assumption of symmetry is required. For example, $A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ is a nonzero positive semi-definite matrix over \mathbb{R} satisfying $x \perp Ax$ for all $x \in \mathbb{R}^2$.

Any symmetric real matrix A can be diagonalized by an orthogonal matrix. This is because A is normal and any eigenvalue of A is real since A is Hermitian and an eigenvector can be found in \mathbb{R}^n . A unitary matrix U diagonalizing A is also real, and so it is an orthogonal matrix.

Since an orthogonal matrix is normal, it is diagonalizable, but its eigenvalues are not always real. For example, a rotation matrix $\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$ is orthogonal, but it does not have an eigenvector in \mathbb{R}^2 unless $\theta = n\pi$ with integer n . Hence, a (complex) unitary matrix is needed to diagonalize an orthogonal matrix in general.

Let us diagonalize $A = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix}$ which is a symmetric matrix.

```
In [1]: from sympy import *
A = Matrix([[0, 1, 2], [1, 2, 0], [2, 0, 1]])
X = A.eigenvects()
[x[0] for x in X]
```

```
Out[1]: [3, -sqrt(3), sqrt(3)]
```

We get three eigenvalues 3 and $\pm\sqrt{3}$. We make the orthogonal matrix U by normalizing each eigenvector.

```
In [2]: B = [simplify(v) for x in X for v in x[2]]
C = [simplify(b / b.norm()) for b in B]
U = C[0].row_join(C[1]).row_join(C[2]); U
```

```
Out[2]: Matrix([
[sqrt(3)/3, -1/2 - sqrt(3)/6, 1/2 - sqrt(3)/6],
[sqrt(3)/3, 1/2 - sqrt(3)/6, -1/2 - sqrt(3)/6],
[sqrt(3)/3, sqrt(3)/3, sqrt(3)/3]])
```

The matrix A can be diagonalized by U .

```
In [3]: simplify(U.T * A * U)
```

```
Out[3]: Matrix([
[3, 0, 0],
[0, -sqrt(3), 0],
[0, 0, sqrt(3)]])
```

We use the `eig` function of the `linalg` module to compute eigenvalues and eigenvectors, which can be applied to diagonalization by a regular matrix. Also, for a Hermitian matrix, the `eigh` function to compute eigenvalues and eigenvectors for an orthonormal basis exists, which can be applied to diagonalization by a unitary matrix.

```
In [1]: from numpy import *
A = array([[1, 2, 3], [2, 3, 4], [3, 4, 5]])
Lmd, V = linalg.eig(A); Lmd
```

```

Out[1]: array([ 9.62347538e+00, -6.23475383e-01,  5.02863969e-16])

In [2]: linalg.inv(V).dot(A.dot(V))

Out[2]: array([[ 9.62347538e+00, -3.55271368e-15,  2.09591804e-15],
   [ 4.44089210e-16, -6.23475383e-01, -4.38613236e-16],
   [ 1.33226763e-15,  1.66533454e-16,  3.94430453e-31]])

In [3]: B = array([[1j, 1j], [-1j, 1j]])
Lmd, V = linalg.eigh(B); Lmd

Out[3]: array([-1.,  1.])

In [4]: V.T.conj().dot(B.dot(V))

Out[4]: array([-1.+1.j,  0.+0.j],
   [ 0.+0.j,  1.+1.j])

```

Exercise 7.12 The following program randomly generates a real normal matrix of order 2. Diagonalize a matrix generated by this program by a unitary matrix.

Program: prob3.py

```

1 | from sympy import *
2 | from numpy.random import choice, seed
3 |
4 | seed(2021)
5 | N = [-3, -2, -1, 1, 2, 3]
6 |
7 | def g(symmetric=True):
8 |     if symmetric:
9 |         a, b, d = choice(N, 3)
10 |        return Matrix([[a, b], [b, d]])
11 |    else:
12 |        a, b = choice(N, 2)
13 |        return Matrix([[a, b], [-b, a]])

```

If the argument of `g` is `True` (default value), then the program generates a symmetric matrix, otherwise an asymmetric matrix.

```

In [2]: g()

Out[2]: Matrix([
 [2, 3],
 [3, -2]])

In [3]: g(False)

Out[3]: Matrix([
 [-3, 3],
 [-3, -3]])

```

7.4 Matrix Norm and Matrix Functions

In what follows, we only consider the l^2 -norm (Euclidean norm) $\|\cdot\|_2$ in \mathbb{K}^n , which is written simply as $\|\cdot\|$.

Let A be a square matrix of order n . We define the *matrix norm* $\|A\|$ of A by

$$\|A\| \stackrel{\text{def}}{=} \sup_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|,$$

the supremum¹⁰ of $\|A\mathbf{x}\|$ as $\mathbf{x} \in \mathbb{K}^n$ moves on the unit sphere¹¹ $\|\mathbf{x}\| = 1$. In general, we have the inequality

$$\|A\mathbf{x}\| \leq \|A\| \|\mathbf{x}\|$$

for all $\mathbf{x} \in \mathbb{K}^n$. When $\mathbf{x} = \mathbf{0}$, it is obvious. When $\mathbf{x} \neq \mathbf{0}$, since $\left\| \frac{\mathbf{x}}{\|\mathbf{x}\|} \right\| = 1$, we have $\left\| A \frac{\mathbf{x}}{\|\mathbf{x}\|} \right\| \leq \|A\|$ by the definition of $\|A\|$, which yields the above inequality.

For a real matrix A , the range $\|\mathbf{x}\| = 1$ in the definition of the matrix norm depends on whether we consider it in \mathbb{R}^n or in \mathbb{C}^n . Remark that $\mathbf{z} \in \mathbb{C}^n$ is decomposed as $\mathbf{z} = \mathbf{x} + i\mathbf{y}$ with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $\|\mathbf{z}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2$ holds. So we have a decomposition $A\mathbf{z} = A\mathbf{x} + iA\mathbf{y}$ of $A\mathbf{z}$ into the real part and the imaginary part. Let $\|A\|_{\mathbb{R}}$ denote the matrix norm of A defined in \mathbb{R}^n and $\|A\|_{\mathbb{C}}$ the matrix norm defined in \mathbb{C}^n . The inequality $\|A\|_{\mathbb{R}} \leq \|A\|_{\mathbb{C}}$ is obvious. The converse also holds as

$$\begin{aligned} \|A\|_{\mathbb{C}}^2 &= \max_{\|\mathbf{z}\|^2=1} \|A\mathbf{z}\|^2 = \max_{\|\mathbf{x}\|^2+\|\mathbf{y}\|^2=1} (\|A\mathbf{x}\|^2 + \|A\mathbf{y}\|^2) \\ &\leq \max_{\|\mathbf{x}\|^2+\|\mathbf{y}\|^2=1} (\|A\|_{\mathbb{R}}^2 \|\mathbf{x}\|^2 + \|A\|_{\mathbb{R}}^2 \|\mathbf{y}\|^2) = \|A\|_{\mathbb{R}}^2. \end{aligned}$$

Therefore, the matrix norms over \mathbb{R}^n and over \mathbb{C}^n coincide.

For some real square matrices A of order 2, let us illustrate where $A\mathbf{x}$ moves in the two-dimensional coordinate plane when $\mathbf{x} \in \mathbb{R}^2$ moves on the unit circle $\|\mathbf{x}\| = 1$. We draw the segments connecting several points \mathbf{x} on the unit circle and their destinations $A\mathbf{x}$ together with the arrows expressing the normalized eigenvectors multiplied by the corresponding eigenvalues when A has real eigenvalues.

Program: unitcircle.py

```
In [1]: 1 from numpy import array, arange, pi, sin, cos, isreal
2 from numpy.linalg import eig
3 import matplotlib.pyplot as plt
4
5 def arrow(p, v, c=(0, 0, 0), w=0.02):
6     plt.quiver(p[0], p[1], v[0], v[1], units='xy', scale=1,
7                 color=c, width=w)
8
9 n = 3
10 A = [array([[1, -2], [2, 2]]),
11      array([[3, 1], [1, 3]]),
12      array([[2, 1], [0, 2]]),
```

¹⁰ It actually becomes the maximum because there exists \mathbf{x} attaining the supremum. To find such an \mathbf{x} numerically, we solve the maximum problem of the quadratic polynomial $\|A\mathbf{x}\|^2$ subject to the constraint of the quadratic equation $\|\mathbf{x}\|^2 = 1$. To show only the existence elegantly, we use the facts that the unit sphere in a finite-dimensional normed space is compact and that any real-valued continuous function on a compact set attains the maximum.

¹¹ This range can be replaced by the unit ball $\|\mathbf{x}\| \leq 1$.

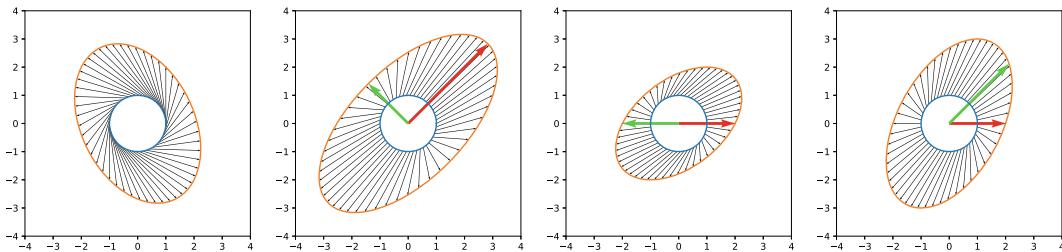


Fig. 7.1 Transforming the unit circle by A_0, A_1, A_2, A_3 (from left to right)

```
In [1]: 13     array([[2, 1], [0, 3]]])
14 plt.axis('scaled'), plt.xlim(-4, 4), plt.ylim(-4, 4),
15 T = arange(0, 2 * pi, pi / 500)
16 U = array([(cos(t), sin(t)) for t in T])
17 plt.plot(U[:,0], U[:,1])
18 V = array([A[n].dot(u) for u in U])
19 plt.plot(V[:,0], V[:,1])
20 for u, v in zip(U[::20], V[::20]):
21     arrow(u, v - u)
22 o = array([0, 0])
23 Lmd, Vec = eig(A[n])
24 if isreal(Lmd[0]):
25     arrow(o, Lmd[0] * Vec[:, 0], c=(1, 0, 0), w=0.1)
26 if isreal(Lmd[1]):
27     arrow(o, Lmd[1] * Vec[:, 1], c=(0, 1, 0), w=0.1)
28 plt.show()
```

Lines 9–13: For each $n = 0, 1, 2, 3$, $A[n]$ denotes the matrix A_n below:

$$A_0 = \begin{bmatrix} 1 & -2 \\ 2 & 2 \end{bmatrix}, A_1 = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}, A_2 = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}, A_3 = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}.$$

Lines 15–17: U is an array of 1000 points on the unit circle. Plot these points.

Lines 18, 19: V is the array of the points transformed from the points of U by A . Plot these points.

Lines 20, 21: Illustrate the correspondence between the points in U and in V with arrows. To make it easier to see, it is shown every 20 arrows.

Lines 24–27: When the matrix has real eigenvalues, illustrate the normalized eigenvectors multiplied by the eigenvalue with arrows.

The results for A_n with $n = 0, 1, 2, 3$ are shown in Fig. 7.1. Matrix A_0 has no real eigenvalue, and A_0x is not a real scalar multiple of x on the unit circle. A_1 is a symmetric matrix which has two real eigenvalues and the associated real eigenvectors orthogonal to each other. The characteristic equation of A_2 has a double root and it has only one eigenvalue, whose eigenspace is one dimensional. Matrix A_3 has two real eigenvalues, but the associated eigenvectors are not orthogonal to each other because the matrix is not normal. For each of these matrices, the matrix norm is equal to the longest radius of the resulting ellipse in the figure.

Suppose $\mathbb{K} = \mathbb{C}$. The following (in)equalities from 1 to 6 hold on the matrix norm:

1. $\|A\| \geq 0$, and the equality holds if and only if $A = \mathbf{O}$,
2. $\|cA\| = |c| \|A\|$,
3. $\|A + B\| \leq \|A\| + \|B\|$,

4. $\|AB\| \leq \|A\| \|B\|$,
5. $\|A^*\| = \|A\|$,
6. $\|A^*A\| = \|A\|^2$.

Proofs of 1–4 are left as exercises for the reader. Here we prove 5 and 6. For all $x \in \mathbb{K}^n$, we have

$$\|Ax\|^2 = \langle Ax | Ax \rangle = \langle A^*Ax | x \rangle \leq \|A^*Ax\| \|x\| \leq \|A^*\| \|A\| \|x\|^2.$$

Hence, we find $\|A\|^2 \leq \|A^*\| \|A\|$ by moving x in the range $\|x\| = 1$, and thus we have $\|A\| \leq \|A^*\|$. By replacing A by A^* , we have $\|A^*\| \leq \|A^{**}\| = \|A\|$, and 5 has been proved. Finally, 6 can be shown by

$$\begin{aligned} \|A\|^2 &= \left(\max_{\|x\|=1} \|Ax\| \right)^2 = \max_{\|x\|=1} \|Ax\|^2 = \max_{\|x\|=1} \langle Ax | Ax \rangle = \max_{\|x\|=1} \langle A^*Ax | x \rangle \\ &\leq \max_{\|x\|=1} \|A^*Ax\| \|x\| \leq \|A^*A\| \leq \|A^*\| \|A\| = \|A\|^2. \end{aligned}$$

Let λ be an eigenvalue of A and x be an associated normalized eigenvector. Then we have $|\lambda|^2 = \|\lambda x\|^2 = \|Ax\|^2 \leq \|A\|^2$. Hence, letting $\rho(A)$ denote the maximum absolute value of the eigenvalues of A , we have $\rho(A) \leq \|A\|$. We call $\rho(A)$ the *spectral radius* of A .

When A is a normal matrix, $\|A\| = \rho(A)$ holds. In fact, from the normality of A , we can take an orthonormal basis $\{x_1, x_2, \dots, x_n\}$ of \mathbb{K}^n consisting of eigenvectors of A . Let λ_i be the eigenvalue corresponding to the eigenvector x_i ($i = 1, 2, \dots, n$). Let $x = a_1x_1 + a_2x_2 + \dots + a_nx_n$ with $a_1, \dots, a_n \in \mathbb{K}$ be a vector with norm 1, that is, $|a_1|^2 + |a_2|^2 + \dots + |a_n|^2 = 1$. We have

$$\begin{aligned} \|A\|^2 &= \max_{\|x\|=1} \|Ax\|^2 = \max_{|a_1|^2 + \dots + |a_n|^2 = 1} \|a_1\lambda_1x_1 + a_2\lambda_2x_2 + \dots + a_n\lambda_nx_n\|^2 \\ &= \max_{|a_1|^2 + \dots + |a_n|^2 = 1} \left(|a_1|^2 |\lambda_1|^2 + |a_2|^2 |\lambda_2|^2 + \dots + |a_n|^2 |\lambda_n|^2 \right). \end{aligned}$$

Let λ_i be an eigenvalue with maximum absolute value, then $|a_1|^2 |\lambda_1|^2 + |a_2|^2 |\lambda_2|^2 + \dots + |a_n|^2 |\lambda_n|^2$ reaches the maximum $|a_i|^2 |\lambda_i|^2 = |\lambda_i|^2$ when $x = x_i$, that is, $a_i = 1$ and $a_j = 0$ for every $j \neq i$. It follows that

$$\|A\| = |\lambda_i| = \rho(A).$$

When A is a Hermitian matrix, all the eigenvalues are real and $\|A\| = \rho(A)$, so $\|A\|$ or $-\|A\|$ is an eigenvalue of A and $\|A\| = \|Ax\|$ for the normalized eigenvector x associated with it.

We define the *numerical radius* $N(A)$ of A by

$$N(A) \stackrel{\text{def}}{=} \max_{\|x\|=1} |\langle x | Ax \rangle|.$$

For all $x \in \mathbb{K}^n$, we have

$$|\langle x | Ax \rangle| \leq \|x\| \|Ax\| \leq \|A\| \|x\|^2,$$

which implies that $N(A) \leq \|A\|$.

If A is Hermitian, then the equality $N(A) = \|A\|$ holds. In fact, because A has an eigenvalue λ such that $|\lambda| = \|A\|$ with corresponding normalized eigenvector x , we have

$$N(\mathbf{A}) \geq |\langle \mathbf{x} | \mathbf{Ax} \rangle| = |\langle \mathbf{x} | \lambda \mathbf{x} \rangle| = \|\mathbf{A}\|.$$

The following program compares the numerical radius, spectral radius, and matrix norm.

Program: matrixnorm.py

```
In [1]: 1 from numpy import array, arange, pi, sin, cos
2 from numpy.linalg import eig, norm
3
4 M = [array([[1, 2], [2, 1]]),
5      array([[1, 2], [-2, 1]]),
6      array([[1, 2], [3, 4]])]
7 T = arange(0, 2 * pi, pi / 500)
8 U = array([(cos(t), sin(t)) for t in T])
9 for A in M:
10    r1 = max([abs((A.dot(u)).dot(u)) for u in U])
11    r2 = max([abs(e) for e in eig(A)[0]])
12    r3 = max([norm(A.dot(u)) for u in U])
13    print(f'{A}: num={r1:.2f}, spec={r2:.2f}, norm={r3:.2f}'')
```

Lines 4–6: List of a Hermitian matrix, a normal but non-Hermitian matrix and a non-normal matrix.

Lines 9–13: List of the numerical radius, spectral radius, and matrix norm of each matrix, calculated to 2 decimal places.



```
[[1 2]
 [2 1]]: num=3.00, spec=3.00, norm=3.00
[[ 1  2]
 [-2  1]]: num=1.00, spec=2.24, norm=2.24
[[1 2]
 [3 4]]: num=5.42, spec=5.37, norm=5.46
```

For the first Hermitian matrix, the three values coincide. For the second normal but non-Hermitian matrix, only the spectral radius and matrix norm coincide. For the last non-normal matrix, the three values are distinct.

Let a_{ij} be the (i, j) -element of \mathbf{A} . The following inequalities hold for the matrix norm and the matrix elements:

$$\frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n |a_{ij}| \leq \|\mathbf{A}\| \leq \sum_{i=1}^n \sum_{j=1}^n |a_{ij}|.$$

Let $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$ be the standard basis of \mathbb{K}^n . The first inequality follows from

$$|a_{ij}| = |\langle \mathbf{Ae}_i | \mathbf{e}_j \rangle| \leq \|\mathbf{Ae}_i\| \|\mathbf{e}_j\| \leq \|\mathbf{A}\|$$

for $i, j = 1, 2, \dots, n$. On the other hand, for any $\mathbf{x} = (x_1, x_2, \dots, x_n)$ with $\|\mathbf{x}\| = 1$, we have

$$\|\mathbf{Ax}\|^2 = \sum_{i=1}^n \left| \sum_{j=1}^n a_{ij} x_j \right|^2 \leq \left(\sum_{i=1}^n \left| \sum_{j=1}^n a_{ij} x_j \right| \right)^2 \leq \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}| |x_j| \right)^2 \leq \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}| \right)^2.$$

So, taking the supremum of the leftmost side in the range $\|\mathbf{x}\| = 1$, we obtain the second inequality.

An infinite sequence $\{\mathbf{A}_k\}_{k=1}^\infty$ of matrices is said to *converge* to a matrix \mathbf{A}_∞ if

$$\lim_{k \rightarrow \infty} \|\mathbf{A}_\infty - \mathbf{A}_k\| = 0.$$

This convergence holds if and only if the componentwise convergence holds, that is, the sequence $\{a_{ij}^{(k)}\}_{k=1}^{\infty}$ converges to $a_{ij}^{(\infty)}$ in \mathbb{K} for every $i, j = 1, \dots, n$, where $a_{ij}^{(k)}$ and $a_{ij}^{(\infty)}$ are the (i, j) -elements of \mathbf{A}_k and \mathbf{A}_{∞} , respectively, that is,

$$\lim_{k \rightarrow \infty} |a_{ij}^{(\infty)} - a_{ij}^{(k)}| = 0$$

holds. This can be shown by the inequalities proved above.¹²

A sequence $\{\mathbf{A}_k\}_{k=1}^{\infty}$ is called a *Cauchy sequence* if it satisfies $\lim_{k, k' \rightarrow \infty} \|\mathbf{A}_{k'} - \mathbf{A}_k\| = 0$. Similar to the above, it holds that $\{\mathbf{A}_k\}_{k=1}^{\infty}$ is a Cauchy sequence if and only if for each $i, j = 1, 2, \dots, n$, the sequence $\{a_{ij}^{(k)}\}_{k=1}^{\infty}$ is a Cauchy sequence in \mathbb{K} , that is, $\lim_{k, k' \rightarrow \infty} |a_{ij}^{(k')} - a_{ij}^{(k)}| = 0$. Then, from the completeness¹³ of \mathbb{K} , $\{a_{ij}^{(k)}\}_{k=1}^{\infty}$ converges to some $a_{ij}^{(\infty)} \in \mathbb{K}$ for each $i, j = 1, 2, \dots, n$. Thus, letting \mathbf{A}_{∞} be the matrix whose (i, j) -element is $a_{ij}^{(\infty)}$, we see $\lim_{k \rightarrow \infty} \|\mathbf{A}_{\infty} - \mathbf{A}_k\| = 0$. From this property, we say that the space of all square matrices of order n is *complete* as well.

For a polynomial $p(x) = c_k x^k + c_{k-1} x^{k-1} + \dots + c_1 x + c_0$ in a variable x with coefficients in \mathbb{K} and a square matrix \mathbf{A} , the *matrix polynomial* $p(\mathbf{A})$ in \mathbf{A} is defined by the expression

$$p(\mathbf{A}) \stackrel{\text{def}}{=} c_k \mathbf{A}^k + c_{k-1} \mathbf{A}^{k-1} + \dots + c_1 \mathbf{A} + c_0 \mathbf{I}$$

substituting x by \mathbf{A} . In particular, for a diagonal matrix $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$, we have

$$p(\mathbf{\Lambda}) = \text{diag}(p(\lambda_1), p(\lambda_2), \dots, p(\lambda_n)).$$

Furthermore, if \mathbf{A} can be diagonalized to $\mathbf{\Lambda} = \mathbf{V}^{-1} \mathbf{A} \mathbf{V}$ by a regular matrix \mathbf{V} , then we have

$$p(\mathbf{A}) = \mathbf{V} p(\mathbf{\Lambda}) \mathbf{V}^{-1}.$$

Suppose that a sequence $\{\mathbf{A}_k\}_{k=1}^{\infty}$ satisfies $\sum_{k=1}^{\infty} \|\mathbf{A}_k\| < \infty$. In this case, if $K < K'$, then we have

$$\left\| \sum_{k=1}^{K'} \mathbf{A}_k - \sum_{k=1}^K \mathbf{A}_k \right\| = \left\| \sum_{k=K+1}^{K'} \mathbf{A}_k \right\| \leq \sum_{k=K+1}^{K'} \|\mathbf{A}_k\| \rightarrow 0 \quad (K, K' \rightarrow \infty).$$

Hence, the sequence $\left\{ \sum_{k=1}^K \mathbf{A}_k \right\}_{K=1}^{\infty}$ of finite partial sums is a Cauchy sequence and thus has a limit.

We denote this limit by $\sum_{k=1}^{\infty} \mathbf{A}_k$.

Since $\|\mathbf{A}^k\| \leq \|\mathbf{A}\|^k$ for any $k \geq 0$, we have

¹² Thus, the convergence of matrices is the same as the convergence of vectors discussed in Sect. 6.6.

¹³ Completeness is a property of a metric space where every Cauchy sequence in the space converges to a point inside it. Consult textbooks on topology.

$$\sum_{k=0}^K \left\| \frac{\mathbf{A}^k}{k!} \right\| \leq \sum_{k=0}^K \frac{\|\mathbf{A}\|^k}{k!},$$

and letting $K \rightarrow \infty$ we obtain

$$\sum_{k=0}^{\infty} \left\| \frac{\mathbf{A}^k}{k!} \right\| \leq \sum_{k=0}^{\infty} \frac{\|\mathbf{A}\|^k}{k!} = e^{\|\mathbf{A}\|} < \infty.$$

Here the equality in the middle is the Maclaurin expansion of the exponential function given in Sect. 1.2.

Therefore, the limit $\sum_{k=0}^{\infty} \frac{\mathbf{A}^k}{k!}$ exists, and we call this the *matrix exponential* of \mathbf{A} , written $e^{\mathbf{A}}$ or $\exp(\mathbf{A})$.

We shall show the equality

$$e^{\mathbf{A} + \mathbf{B}} = e^{\mathbf{A}} e^{\mathbf{B}}$$

for square matrices \mathbf{A} and \mathbf{B} of the same order that *commute*, that is, $\mathbf{AB} = \mathbf{BA}$.

Let $\mathbf{A}_K = \sum_{k=0}^K \frac{\mathbf{A}^k}{k!}$, $\mathbf{B}_K = \sum_{k=0}^K \frac{\mathbf{B}^k}{k!}$ and $\mathbf{C}_K = \sum_{k=0}^K \frac{(\mathbf{A} + \mathbf{B})^k}{k!}$ be the finite partial sums of $e^{\mathbf{A}}$, $e^{\mathbf{B}}$ and $e^{\mathbf{A} + \mathbf{B}}$, respectively. We have

$$\begin{aligned} \|e^{\mathbf{A}} e^{\mathbf{B}} - e^{\mathbf{A} + \mathbf{B}}\| &= \|e^{\mathbf{A}} e^{\mathbf{B}} - \mathbf{A}_K \mathbf{B}_K + \mathbf{A}_K \mathbf{B}_K - \mathbf{C}_K + \mathbf{C}_K - e^{\mathbf{A} + \mathbf{B}}\| \\ &\leq \|e^{\mathbf{A}} e^{\mathbf{B}} - \mathbf{A}_K \mathbf{B}_K\| + \|\mathbf{A}_K \mathbf{B}_K - \mathbf{C}_K\| + \|\mathbf{C}_K - e^{\mathbf{A} + \mathbf{B}}\|. \end{aligned}$$

The third term in the rightmost side will converge to 0 when $K \rightarrow \infty$. Also, the first term converges to 0 as $K \rightarrow \infty$, since

$$\begin{aligned} \|e^{\mathbf{A}} e^{\mathbf{B}} - \mathbf{A}_K \mathbf{B}_K\| &\leq \|e^{\mathbf{A}} e^{\mathbf{B}} - e^{\mathbf{A}} \mathbf{B}_K\| + \|e^{\mathbf{A}} \mathbf{B}_K - \mathbf{A}_K \mathbf{B}_K\| \\ &= \|e^{\mathbf{A}} (e^{\mathbf{B}} - \mathbf{B}_K)\| + \|(e^{\mathbf{A}} - \mathbf{A}_K) \mathbf{B}_K\| \\ &\leq \|e^{\mathbf{A}}\| \|e^{\mathbf{B}} - \mathbf{B}_K\| + \|e^{\mathbf{A}} - \mathbf{A}_K\| \|\mathbf{B}_K\|. \end{aligned}$$

By assumption $\mathbf{AB} = \mathbf{BA}$, we have

$$\sum_{k=0}^K \frac{\mathbf{A}^k}{k!} \cdot \sum_{m=0}^K \frac{\mathbf{B}^m}{m!} - \sum_{k=0}^K \frac{(\mathbf{A} + \mathbf{B})^k}{k!} = \sum_{(k,m) \in I(K)} \frac{\mathbf{A}^k \mathbf{B}^m}{k! m!},$$

where $I(K) = \{(k, m) \in \mathbb{N}^2 \mid k \leq K, m \leq K, k + m > K\}$. Therefore, we have

$$\begin{aligned} &\left\| \sum_{k=0}^K \frac{\mathbf{A}^k}{k!} \cdot \sum_{m=0}^K \frac{\mathbf{B}^m}{m!} - \sum_{k=0}^K \frac{(\mathbf{A} + \mathbf{B})^k}{k!} \right\| \leq \sum_{(k,m) \in I(K)} \left\| \frac{\mathbf{A}^k \mathbf{B}^m}{k! m!} \right\| \\ &\leq \sum_{(k,m) \in I(K)} \frac{\|\mathbf{A}\|^k \|\mathbf{B}\|^m}{k! m!} = \sum_{k=0}^K \frac{\|\mathbf{A}\|^k}{k!} \cdot \sum_{m=0}^K \frac{\|\mathbf{B}\|^m}{m!} - \sum_{k=0}^K \frac{(\|\mathbf{A}\| + \|\mathbf{B}\|)^k}{k!}. \end{aligned}$$

This last term converges to 0 as $K \rightarrow \infty$ because $e^{\|A\|} e^{\|B\|} = e^{\|A\|+\|B\|}$ holds for the ordinary exponential. Hence the second term $\|A_K B_K - C_K\|$ above also converges to 0, and consequently we have $\|e^{A+B} - e^A e^B\| = 0$, that is, $e^{A+B} = e^A e^B$.

Exercise 7.13 Let A and V be square matrices of order n , where V is regular. Prove the equality

$$e^{V^{-1}AV} = V^{-1}e^AV.$$

If A can be diagonalized as $V^{-1}AV = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$, then by Exercise 7.13 we have

$$e^A = \exp(A) = V \text{diag}(e^{\lambda_1}, e^{\lambda_2}, \dots, e^{\lambda_n}) V^{-1}.$$

This suggests that to compute e^A it would be better to diagonalize A first.

Let us compute e^A in NumPy for a given Hermitian matrix A by means of both power series and diagonalization.

Program: exp_np.py

```
In [1]: 1 from numpy import matrix, e, exp, diag
2 from numpy.linalg import eigh
3
4 A = matrix([[1, 2], [2, 1]])
5 m, B = 1, 0
6 for n in range(10):
7     B += A ** n / m
8     m *= n + 1
9 print(B)
10
11 a = eigh(A)
12 S, V = diag(e**a[0]), a[1]
13 print(V * S * V.H)
14
15 print(exp(A))
```

Line 4: Instead of an array, the `matrix` class is used to easily express matrix operations.

Lines 5–9: Calculation with power series. An infinite sum is truncated at the 9th term. If the upper bound of n is changed, the truncation error will change.

Lines 11–13: Calculation by diagonalization of a Hermitian matrix.

Line 15: Investigate whether NumPy supports a matrix exponential.



```
[[10.21563602  9.84775683]
 [ 9.84775683 10.21563602]]
[[10.22670818  9.85882874]
 [ 9.85882874 10.22670818]]
[[2.71828183  7.3890561 ]
 [7.3890561   2.71828183]]
```

When $n = 9$ is the upper bound, the truncation errors about 0.01 occur. In Numpy `exp(A)` does not give a matrix exponential, but gives a componentwise exponential. Here we add that `e**A` induces a program error.

Exercise 7.14 Diagonalize $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ first, and then compute a general term of A^n and $\exp(A)$.

Here is a result by SymPy.

```
In [1]: from sympy import Matrix, exp, var
var('n')

Out[1]: n

In [2]: A = Matrix([[1, 2], [2, 1]])
A**n

Out[2]: Matrix([
[ (-1)**n/2 + 3**n/2, -(-1)**n/2 + 3**n/2],
[-(-1)**n/2 + 3**n/2, (-1)**n/2 + 3**n/2]])

In [3]: exp(A)

Out[3]: Matrix([
[ exp(-1)/2 + exp(3)/2, -exp(-1)/2 + exp(3)/2],
[-exp(-1)/2 + exp(3)/2, exp(-1)/2 + exp(3)/2]])
```

Exercise 7.15 For a square matrix A with $\|A\| < 1$, show that $I - A$ is regular and

$$(I - A)^{-1} = \sum_{n=0}^{\infty} A^n$$

holds. (Hint: first prove that $\sum_{n=0}^{\infty} A^n$ converges and then show the equality $(I - A) \sum_{n=0}^{\infty} A^n = I$.)



Jordan Normal Form and Spectrum

8

We have seen that a necessary and sufficient condition for a matrix to be diagonalizable is that a set of its eigenvectors forms a basis of the underlying linear space. In the first half of this chapter, we study the Jordan normal form and the Jordan decomposition which generalize the above fact for arbitrary matrices not necessarily diagonalizable. We explain how to compute them in Python for large matrices which may be hard and cumbersome using only paper and pencil. We also make a program which generates classroom or examination problems.

In the second half of this chapter, we observe the shape of the spectrum, the set of all eigenvalues of a matrix, in the complex plane. The ultimate goal is to prove the Perron–Frobenius theorem, a significant fact about the spectrum of a real matrix whose elements are all positive. The proof, which needs analytic properties of matrices, is far more difficult than the proofs of the other theorems we deal with in this book. The Perron–Frobenius theorem has various applications, an example of which will be given in the next chapter.

8.1 Direct Sum Decomposition

Let V be a linear space over \mathbb{K} and $f : V \rightarrow V$ be a linear mapping. A subspace $W \subseteq V$ is called an *invariant subspace* of f if $f(W) \subseteq W$. Then, restricting the domain to W we get a linear mapping $f|_W : W \rightarrow W$ defined by $f|_W(x) = f(x)$ for $x \in W$.

Let $\mathbb{X} = \{X_1, X_2, \dots, X_k\}$ be a linearly independent family of subspaces of V and let $W = X_1 \oplus X_2 \oplus \dots \oplus X_k$ be the direct sum of them. As stated in Sect. 3.5, any $x \in W$ is uniquely written as $x = x_1 + x_2 + \dots + x_n$ with $x_1 \in X_1, x_2 \in X_2, \dots, x_n \in X_n$. With this x , for linear mappings $f_i : X_i \rightarrow X_i$ ($i = 1, 2, \dots, k$), we define the linear mapping $f_1 \oplus f_2 \oplus \dots \oplus f_k : W \rightarrow W$ by

$$(f_1 \oplus f_2 \oplus \dots \oplus f_k)(x) \stackrel{\text{def}}{=} f_1(x_1) + f_2(x_2) + \dots + f_k(x_k).$$

We call this the *direct sum of linear mappings*. Suppose that each subspace X_i is finite dimensional with basis E_i and the linear mapping f_i is represented by a matrix A_i based on E_i for $i = 1, 2, \dots, k$. Then, $E = E_1 \cup E_2 \cup \dots \cup E_k$ is a basis of W , and on this basis, the linear mapping $f_1 \oplus f_2 \oplus \dots \oplus f_k$ is represented by the matrix

$$\left[\begin{array}{c|c|c|c} A_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \hline \mathbf{0} & A_2 & \ddots & \vdots \\ \hline \vdots & \ddots & \ddots & \mathbf{0} \\ \hline \mathbf{0} & \cdots & \mathbf{0} & A_k \end{array} \right].$$

We call this matrix the *direct sum of matrices*, which is written as $A_1 \oplus A_2 \oplus \cdots \oplus A_k$.

Let $f : W \rightarrow W$ be a linear mapping such that each X_i is an invariant subspace of f for $i = 1, 2, \dots, k$. In this case we say that \mathbb{X} decomposes f . Then, we have

$$f = f|_{X_1} \oplus f|_{X_2} \oplus \cdots \oplus f|_{X_k},$$

and we call this the *direct sum decomposition* of f .

For a linear mapping $f : V \rightarrow V$, we define $f^k : V \rightarrow V$ inductively by

$$f^k \stackrel{\text{def}}{=} \begin{cases} \mathbf{I} & (k=0) \\ f \circ f^{k-1} & (k=1, 2, \dots), \end{cases}$$

where \mathbf{I} is the identity mapping on V . If V is finite dimensional with basis E and A is the representation matrix of f on E , then A^k is a representation matrix of f^k on E .

We define the subspaces $K^{(k)}$ and $R^{(k)}$ of V by

$$K^{(k)} \stackrel{\text{def}}{=} \text{kernel}(f^k) \text{ and } R^{(k)} \stackrel{\text{def}}{=} \text{range}(f^k),$$

for $k = 0, 1, 2, \dots$. We have the inclusion relations

$$\begin{aligned} \{\mathbf{0}\} &= K^{(0)} \subseteq K^{(1)} \subseteq K^{(2)} \subseteq \cdots \subseteq K^{(k)} \subseteq \cdots, \\ V &= R^{(0)} \supseteq R^{(1)} \supseteq R^{(2)} \supseteq \cdots \supseteq R^{(k)} \supseteq \cdots. \end{aligned}$$

Put

$$K \stackrel{\text{def}}{=} \bigcup_{k=0}^{\infty} K^{(k)} \text{ and } R \stackrel{\text{def}}{=} \bigcap_{k=0}^{\infty} R^{(k)}.$$

Exercise 8.1 Prove that K and R are invariant subspaces of f .

Suppose that V is a finite-dimensional space. Then there exist k_1 and k_2 such that

$$\begin{aligned} K^{(k_1)} &= K^{(k_1+1)} = K^{(k_1+2)} = \cdots = K, \\ R^{(k_2)} &= R^{(k_2+1)} = R^{(k_2+2)} = \cdots = R. \end{aligned}$$

We will show $V = K \oplus R$. Let $k_0 \stackrel{\text{def}}{=} \max\{k_1, k_2\}$, then $K = \text{kernel}(f^{k_0})$ and $R = \text{range}(f^{k_0})$. Then the linear mapping $f|_R : R \rightarrow R$ is surjective, and actually it is bijective as stated in Sect. 3.4. Hence, $(f|_R)^{k_0} : R \rightarrow R$ is also bijective. In particular, we see $K \cap R = \text{kernel}((f|_R)^{k_0}) = \{\mathbf{0}\}$ and so $\{K, R\}$ is linearly independent. For any $\mathbf{u} \in V$, put $\mathbf{v} = f^{k_0}(\mathbf{u})$. Since $\mathbf{v} \in R$, there exists $\mathbf{w} \in R$ such that $\mathbf{v} = f^{k_0}(\mathbf{w})$. We have $\mathbf{u} - \mathbf{w} \in \text{kernel}(f^{k_0}) = K$ since $f^{k_0}(\mathbf{u} - \mathbf{w}) = \mathbf{0}$. Therefore, any $\mathbf{u} \in V$ can be written as the sum of a vector in K and a vector in R as $\mathbf{u} = (\mathbf{u} - \mathbf{w}) + \mathbf{w}$. Hence we have the

direct sum decomposition $V = K \oplus R$. Moreover, f also can be decomposed as $f = f|_K \oplus f|R$, since $\{K, R\}$ decomposes f by Exercise 8.1.

For $k < k_1$ suppose $K^{(k-1)} \subsetneq K^{(k)}$. Take a linearly independent set $\{\mathbf{x}_1, \dots, \mathbf{x}_i\} \subseteq K^{(k)} \setminus K^{(k-1)}$ of vectors such that the subspaces $\langle \mathbf{x}_1, \dots, \mathbf{x}_i \rangle$ and $K^{(k-1)}$ are linearly independent. We will show that

$$\left\{ \mathbf{x}_1, \dots, \mathbf{x}_i, f(\mathbf{x}_1), \dots, f(\mathbf{x}_i), \dots, f^{k-1}(\mathbf{x}_1), \dots, f^{k-1}(\mathbf{x}_i) \right\} \subseteq K^{(k)}$$

is linearly independent. Assume that

$$a_1\mathbf{x}_1 + \dots + a_i\mathbf{x}_i + b_1f(\mathbf{x}_1) + \dots + b_if(\mathbf{x}_i) + c_1f^2(\mathbf{x}_1) + \dots = \mathbf{0} \quad \dots (*)$$

for $a_1, \dots, a_i, b_1, \dots, b_i, c_1, \dots \in \mathbb{K}$. Then, operating f^{k-1} on both sides, we have

$$f^{k-1}(a_1\mathbf{x}_1 + \dots + a_i\mathbf{x}_i) = \mathbf{0}$$

because $f^{k'}(\mathbf{x}_1) = \dots = f^{k'}(\mathbf{x}_i) = \mathbf{0}$ for $k' \geq k$, and thus $a_1\mathbf{x}_1 + \dots + a_i\mathbf{x}_i \in K^{(k-1)}$. Since $\langle \mathbf{x}_1, \dots, \mathbf{x}_i \rangle \cap K^{(k-1)} = \{0\}$, we have $a_1\mathbf{x}_1 + \dots + a_i\mathbf{x}_i = \mathbf{0}$ and hence $a_1 = \dots = a_i = 0$. Next, operating f^{k-2} on both sides of (*), we have

$$f^{k-1}(b_1\mathbf{x}_1 + \dots + b_i\mathbf{x}_i) = \mathbf{0}$$

and hence $b_1 = \dots = b_i = 0$ in a similar manner. Repeating this argument we get the desired result.

8.2 Jordan Normal Form

From now on in this chapter, $\mathbb{K} = \mathbb{C}$ and A is a square matrix of order n . We consider A as a linear mapping on \mathbb{K}^n . Suppose that A has m distinct eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_m$ with multiplicities n_1, n_2, \dots, n_m , respectively, that is, the characteristic equation of A is given by

$$(\lambda_1 - \lambda)^{n_1} (\lambda_2 - \lambda)^{n_2} \cdots (\lambda_m - \lambda)^{n_m} = 0,$$

where $n_1 + n_2 + \dots + n_m = n$. For $k = 0, 1, 2, \dots$, let

$$K_i^{(k)} \stackrel{\text{def}}{=} \text{kernel}((A - \lambda_i I)^k) \quad \text{and} \quad R_i^{(k)} \stackrel{\text{def}}{=} \text{range}((A - \lambda_i I)^k),$$

and define

$$K_i \stackrel{\text{def}}{=} \bigcup_{k=0}^{\infty} K_i^{(k)} \quad \text{and} \quad R_i \stackrel{\text{def}}{=} \bigcap_{k=0}^{\infty} R_i^{(k)}.$$

As we have seen in the previous section, $\{K_i, R_i\}$ decomposes $A - \lambda_i I$ and hence $\{K_i, R_i\}$ also decomposes A .

Since K_i contains all eigenvectors for λ_i , R_i does not contain any eigenvector for λ_i . Moreover, K_i does not contain any eigenvector for other eigenvalues than λ_i . In fact, if there is a nonzero vector $\mathbf{x} \in K_i$ such that $A\mathbf{x} = \lambda_j \mathbf{x}$ with $j \neq i$, then since

$$(A - \lambda_i I)\mathbf{x} = \lambda_j \mathbf{x} - \lambda_i \mathbf{x} = (\lambda_j - \lambda_i) \mathbf{x},$$

we have

$$(A - \lambda_i I)^k x = (\lambda_j - \lambda_i)^k x \neq \mathbf{0}$$

for all $k = 1, 2, \dots$. This implies $x \notin K_i$, which is a contradiction.

Since $\{K_1, R_1\}$ decomposes A , the linear mapping A is decomposed into the direct sum as¹

$$A = A|_{K_1} \oplus A|_{R_1}.$$

Suppose that $n' = \dim K_1$ and let $\{e_1, e_2, \dots, e_{n'}\}$ be a basis of K_1 . Let A_1 denote the matrix representing $A|_{K_1}$ based on this basis. Also, let $\{e_{n'+1}, e_{n'+2}, \dots, e_n\}$ be a basis of R_1 , and R_1 be the matrix representing $A|_{R_1}$ on the basis. We define the regular matrix V_1 aligning these two bases as column vectors by

$$V_1 \stackrel{\text{def}}{=} [e_1 \ e_2 \ \cdots \ e_{n'} \ e_{n'+1} \ e_{n'+2} \ \cdots \ e_n],$$

then we can express the direct sum decomposition of A as

$$V_1^{-1} A V_1 = A_1 \oplus R_1,$$

which is the direct sum of matrices.

Here, A_1 has the unique eigenvalue λ_1 and R_1 has all other eigenvalues $\lambda_2, \dots, \lambda_m$. The characteristic polynomial of A is the product of the characteristic polynomials of A_1 and R_1 , and hence the characteristic equation of A_1 should be $(\lambda - \lambda_1)^{n_1} = 0$ and A_1 turns out to be a square matrix of order n_1 . Thus, $n' = \dim K_1 = n_1$.

By replacing A by R_1 and carrying out a similar argument to the above, we get the direct sum decomposition

$$V_2^{-1} R_1 V_2 = A_2 \oplus R_2,$$

where A_2 is a matrix whose characteristic equation is $(\lambda - \lambda_2)^{n_2} = 0$, R_2 is a matrix whose characteristic equation is $(\lambda_3 - \lambda)^{n_3} \cdots (\lambda_m - \lambda)^{n_m} = 0$, and V_2 is a regular matrix. We repeat this argument inductively to get square matrices A_1, A_2, \dots, A_m of orders n_1, \dots, n_m , respectively. Let V be the matrix made by aligning the bases of K_1, K_2, \dots, K_m as column vectors in order, then we have

$$V^{-1} A V = A_1 \oplus A_2 \oplus \cdots \oplus A_m.$$

Let k_i be the smallest number k such that $K_i^{(k)} = K_i$. Since $K_i^{(k_i-1)}$ is a proper subspace of $K_i^{(k_i)}$, there exists $x \in K_i^{(k_i)} \setminus K_i^{(k_i-1)}$. As seen at the end of the previous section,

$$\{x, (A - \lambda_i I)x, \dots, (A - \lambda_i I)^{k_i-1}x\} \subseteq K_i^{(k_i)}$$

is linearly independent. This implies $n_i = \dim K_i \geq k_i$ and thus, $K_i = K_i^{(k_i)} = K_i^{(n_i)}$.

We call K_i the *generalized eigenspace* of A for eigenvalue λ_i . We have

$$K_i = K_i^{(n_i)} = \text{kernel}((A - \lambda_i I)^{n_i}).$$

¹ This equality means that both sides are equal as linear mappings because the right-hand side is not a matrix but a linear mapping.

The subspace $K_i^{(1)}$ of K_i is nothing but the eigenspace associated with eigenvalue λ_i . Because $(A - \lambda_i I)^{n_i} \mathbf{x} = \mathbf{0}$ for all $\mathbf{x} \in K_i$ we have $(A_i - \lambda_i I_i)^{n_i} = \mathbf{O}$, where I_i is the identity matrix of order n_i .

The following is a summary of what we have discussed above:

1. \mathbb{K}^n is decomposed to the direct sum

$$\mathbb{K}^n = K_1 \oplus K_2 \oplus \cdots \oplus K_m$$

of the generalized eigenspaces K_i for eigenvalues λ_i of A , where the dimension of K_i is equal to the corresponding multiplicity n_i for $i = 1, \dots, m$.

2. The family $\{K_1, K_2, \dots, K_m\}$ decomposes A , and with a regular matrix V , A is transformed into

$$V^{-1}AV = A_1 \oplus A_2 \oplus \cdots \oplus A_m,$$

where A_i is a square matrix of order n_i which has the unique eigenvalue λ_i and satisfies $(A_i - \lambda_i I_i)^{n_i} = \mathbf{O}$ for $i = 1, 2, \dots, m$. We call $V^{-1}AV$ the *Jordan normal form* of A .

Using a matrix A generated by the following program, we will explain how to compute the generalized eigenspaces and the Jordan normal form of A .

Program: jordan.py

```
In [1]: 1 from sympy import *
2 from numpy.random import seed, permutation
3 from functools import reduce
4
5 A = diag(1, 2, 2, 2, 2, 3, 3, 3, 3)
6 A[1, 2] = A[3, 4] = A[5, 6] = A[7, 8] = A[8, 9] = 1
7
8 seed(123)
9 for n in range(10):
10    P = permutation(10)
11    for i, j in [(P[2*k], P[2*k + 1]) for k in range(5)]:
12        A[:, j] += A[:, i]
13        A[i, :] -= A[j, :]
14
15 B = Lambda(S('lmd'), A - S('lmd') * eye(10))
16 x = Matrix(var('x0, x1, x2, x3, x4, x5, x6, x7, x8, x9'))
17 y = Matrix(var('y0, y1, y2, y3, y4, y5, y6, y7, y8, y9'))
18 z = Matrix(var('z0, z1, z2, z3, z4, z5, z6, z7, z8, z9'))
```

Lines 5–13: Generate a square matrix A of order 10, by starting with a Jordan normal form on Lines 5 and 6 and applying elementary operations randomly on Lines 8–13.

Lines 15–18: Define the following:

$$\begin{aligned} B(\lambda) &\stackrel{\text{def}}{=} A - \lambda I, \\ x &\stackrel{\text{def}}{=} (x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9), \\ y &\stackrel{\text{def}}{=} (y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9), \\ z &\stackrel{\text{def}}{=} (z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9). \end{aligned}$$

After executing this program, we will proceed our computations in an interactive mode. We first display the generated matrix A and compute its eigenvalues.

In [2]: A

```
Out[2]: Matrix([
[ 35, 28, 24, 6, 26, 16, -6, 14, 26, 42],
[ -5, -8, -9, -24, -21, -2, -5, -3, -5, -17],
[ 11, 14, 10, 5, 7, 8, 2, 6, 13, 13],
[ 5, 2, 4, 2, 5, 1, -3, 1, 2, 7],
[ 1, 4, 4, 6, 8, 0, -1, 2, 0, 5],
[ 19, 11, 14, 19, 25, 13, 6, 7, 16, 31],
[ 8, 11, 5, 7, 6, 7, 7, 5, 11, 10],
[ 6, 16, 14, 40, 34, 2, 8, 7, 6, 26],
[-27, -19, -16, -6, -19, -16, -2, -11, -22, -33],
[-20, -21, -19, -11, -22, -9, 5, -10, -15, -28]])
```

In [3]: P = A.charpoly(); P

```
Out[3]: PurePoly(lambda**10 - 24*lambda**9 + 257*lambda**8 -
1616*lambda**7 + 6603*lambda**6 - 18304*lambda**5 +
34827*lambda**4 - 44856*lambda**3 + 37368*lambda**2 -
18144*lambda + 3888, lambda, domain='ZZ')
```

In [4]: factor(P.expr)

```
Out[4]: (lambda - 3)**5*(lambda - 2)**4*(lambda - 1)
```

The eigenvalues are $\lambda_1 = 1$, $\lambda_2 = 2$, $\lambda_3 = 3$ and the corresponding multiplicities are $n_1 = 1$, $n_2 = 4$, $n_3 = 5$, respectively.

First, we compute the generalized eigenspace K_1 for $\lambda_1 = 1$. Solving $B(1)x = \mathbf{0}$, we get a solution $x = a_1$, which is an eigenvector associated with eigenvalue 1.

In [5]: a1 = x.subs(solve(B(1) * x)); a1

```
Out[5]: Matrix([
[ -x9/3],
[ 5*x9/3],
[ -x9],
[ 0],
[ -x9/3],
[ 8*x9/3],
[ -x9/3],
[-8*x9/3],
[ -2*x9],
[ x9]])
```

This a_1 is a general form of the vectors in the eigenspace $K_1^{(1)}$ for eigenvalue 1. Because a_1 contains one arbitrary constant x_9 , $K_1^{(1)}$ is one dimensional. Since $n_1 = 1$, we see $K_1 = K_1^{(1)}$ and so a_1 is a general form of the vectors in K_1 too.

Next, we compute the generalized eigenspace K_2 for $\lambda_2 = 2$. The solution $x = a_2$ of $B(2)x = \mathbf{0}$ is an eigenvector associated with eigenvalue 2.

In [6]: a2 = x.subs(solve(B(2) * x)); a2

```
In [6]: Matrix([
[ -x8/3 - x9],
[ -5*x8/12 + x9/4],
[ 17*x8/12 - 5*x9/4],
[ -3*x8/4 + x9/4],
[ x8/12 - x9/4],
[-29*x8/12 + 5*x9/4],
[ 4*x8/3 - x9],
[ 5*x8/6 - x9/2],
[ x8],
[ x9]])
```

Again \mathbf{a}_2 is a general form of the vectors in the eigenspace $K_2^{(1)}$ for eigenvalue 2. Because \mathbf{a}_2 contains two arbitrary constants x_8 and x_9 , $K_2^{(1)}$ is two dimensional. Since $n_2 = 4$, $K_2^{(1)}$ is a proper subset of K_2 .

Solving $\mathbf{B}(2)\mathbf{y} = \mathbf{a}_2$, we get a solution $\mathbf{y} = \mathbf{b}_2$. That is, $(\mathbf{A} - 2\mathbf{I})^2 \mathbf{b}_2 = \mathbf{0}$.

```
In [7]: b2 = y.subs(solve(B(2) * y - a2)); b2
```

```
Out[7]: Matrix([
[ 2*y7 - 2*y8],
[y6/6 + y7/12 - 17*y8/24 + 11*y9/24],
[5*y6/6 + 5*y7/12 - y8/24 - 5*y9/24],
[-y6/2 + 5*y7/4 - 9*y8/8 + 3*y9/8],
[-2*y7 + 7*y8/4 - 5*y9/4],
[-y6 - 5*y7/2 + y8 - y9],
[ y6],
[ y7],
[ y8],
[ y9]])
```

Because \mathbf{b}_2 is a general form of the vectors in $K_2^{(2)}$ and contains four arbitrary constants y_6 to y_9 , $K_2^{(2)}$ is four dimensional. Since $n_2 = 4$, we have $K_2^{(2)} = K_2$ and so \mathbf{b}_2 is also a general form of the vectors in K_2 .

Finally, we compute the generalized eigenspace K_3 for $\lambda_3 = 3$. Solving $\mathbf{B}(3)\mathbf{x} = \mathbf{0}$, we have a solution $\mathbf{x} = \mathbf{a}_3$.

```
In [8]: a3 = x.subs(solve(B(3) * x)); a3
```

```
Out[8]: Matrix([
[ -2*x9],
[ -x8/3 + x9/3],
[ -2*x8/3 + 5*x9/3],
[ x8/3 - 4*x9/3],
[ 0],
[ -x8],
[ -2*x8/3 + 5*x9/3],
[ 2*x8/3 - 2*x9/3],
[ x8],
[ x9]])
```

Since \mathbf{a}_3 is a general form of the vectors in $K_3^{(1)}$ and contains two arbitrary constants x_8 and x_9 , the dimension of $K_3^{(1)}$ is 2, which is less than the multiplicity $n_3 = 5$.

Now we solve $\mathbf{B}(3)\mathbf{y} = \mathbf{a}_3$ to get a solution $\mathbf{y} = \mathbf{b}_3$, that is, $(\mathbf{A} - 3\mathbf{I})^2 \mathbf{b}_3 = \mathbf{0}$.

```
In [9]: b3 = y.subs(solve(B(3) * y - a3)); b3
```

```
Out[9]: Matrix([
[ -y6/2 + y7 - y8 - y9/2],
```

```
Out[9]: [ -y7/2] ,
[ 7*y6/6 - y7/3 + y8/3 - y9/2] ,
[ -2*y6/3 + y7/3 - y8/3] ,
[ y6/6 - y7/3 + y8/3 - y9/2] ,
[ -y6/3 - 5*y7/6 - 2*y8/3] ,
[ y6] ,
[ y7] ,
[ y8] ,
[ y9]])
```

Since the vector \mathbf{b}_3 is a general form of the vectors in $K_3^{(2)}$ and contains four arbitrary constants y_6 to y_9 , the dimension of $K_3^{(2)}$ is 4, which is still less than the multiplicity $n_3 = 5$.

Further, we solve $\mathbf{B}(3)\mathbf{z} = \mathbf{b}_3$ and obtain a solution $\mathbf{z} = \mathbf{c}_3$. We have $(\mathbf{A} - 3\mathbf{I})^3 \mathbf{c}_3 = \mathbf{0}$.

```
In [10]: c3 = z.subs(solve(B(3) * z - b3)); c3
```

```
Out[10]: Matrix([ [-15*z5/11 - 21*z6/22 - 3*z7/22 - 21*z8/11 - z9/2] ,
[ 3*z5/11 + z6/11 - 3*z7/11 + 2*z8/11] ,
[ -z5/11 + 25*z6/22 - 9*z7/22 + 3*z8/11 - z9/2] ,
[ -8*z5/11 - 10*z6/11 - 3*z7/11 - 9*z8/11] ,
[ z5 + z6/2 + z7/2 + z8 - z9/2] ,
[ z5] ,
[ z6] ,
[ z7] ,
[ z8] ,
[ z9]]))
```

Because \mathbf{c}_3 is a general form of the vectors in $K_3^{(3)}$ and contains five arbitrary constants z_5 to z_9 , $K_3^{(3)}$ is now five dimensional. Since $n_3 = 5$, we have $K_3^{(2)} = K_3$ and so \mathbf{c}_3 is also a general form of the vectors in K_3 .

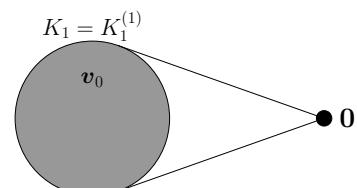
Coming back to K_1 , as we observed, $K_1 = K_1^{(1)}$ is one dimensional and consists of vectors of the form \mathbf{a}_1 with arbitrary constant x_9 . Letting \mathbf{v}_0 be the vector obtained by substituting 1 for x_9 in \mathbf{a}_1 , we have a basis $\{\mathbf{v}_0\}$ of K_1 (see Fig. 8.1).

```
In [11]: v0 = a1.subs({x9:1})
```

Next, the subspace $K_2^{(1)}$ is two dimensional, whereas $K_2^{(2)} = K_2$ is four dimensional. Thus, we should be able to find two linearly independent vectors from $K_2^{(2)} \setminus K_2^{(1)}$. The vectors in $K_2^{(1)}$ are represented by \mathbf{a}_2 with arbitrary constants x_8 and x_9 , whereas the vectors in $K_2^{(2)}$ are represented by \mathbf{b}_2 with arbitrary constants y_6 , y_7 , y_8 , and y_9 . Any vector in $K_2^{(2)}$ represented by \mathbf{b}_2 with $y_8 = y_9 = 0$ has 0 in the last two coordinates. A vector in $K_2^{(1)}$ whose last two coordinates are 0 is the zero vector because a vector represented by \mathbf{a}_2 with $x_8 = x_9 = 0$ is $\mathbf{0}$. Therefore, if we choose a nonzero \mathbf{b}_2 with $y_8 = y_9 = 0$, then it does not belong to $K_2^{(1)}$. Let \mathbf{v}_1 be the vector obtained by substitutions $y_6 = 1$ and $y_7 = y_8 = y_9 = 0$ in \mathbf{b}_2 , and put $\mathbf{v}_2 = \mathbf{B}(2)\mathbf{v}_1$.

```
In [12]: v1 = b2.subs({y6:1, y7:0, y8:0, y9:0})
v2 = B(2) * v1
```

Fig. 8.1 Relation between the eigenspace and generalized eigenspace for eigenvalue 1



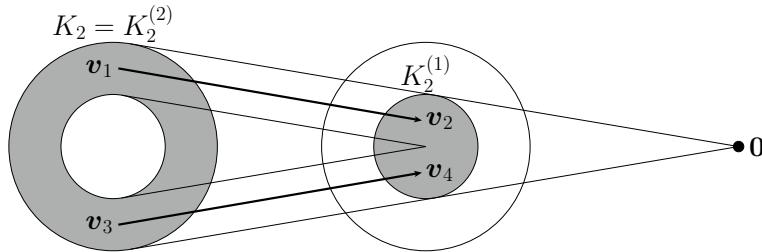


Fig. 8.2 Relation between the eigenspace and generalized eigenspace for eigenvalue 2

As one more vector in $K_2^{(2)} \setminus K_2^{(1)}$ linearly independent from v_2 , choose the vector v_3 obtained by substitutions $y_6 = 0, y_7 = 1, y_8 = y_9 = 0$ in \mathbf{b}_2 . Put $v_4 = \mathbf{B}(2)v_3$.

```
In [13]: v3 = b2.subs({y6:0, y7:1, y8:0, y9:0})
v4 = B(2) * v3
```

The set $\{v_1, v_2, v_3, v_4\}$ is linearly independent and forms a basis of K_2 (see Fig. 8.2).

Finally, let us observe K_3 . The dimensions of $K_3^{(1)}$, $K_3^{(2)}$, and $K_3^{(3)} = K_3$ are 2, 4, and 5, respectively. Since the difference between the dimensions of $K_3^{(3)}$ and $K_3^{(2)}$ is 1, we choose one vector from $K_3^{(3)} \setminus K_3^{(2)}$. Comparing the inside forms of \mathbf{c}_3 and \mathbf{b}_3 , let v_5 be the vector obtained by substitutions $z_5 = 1, z_6 = z_7 = z_8 = z_9 = 0$ in \mathbf{c}_3 . Put $v_6 = \mathbf{B}(3)v_5$ and $v_7 = \mathbf{B}(3)v_6$.

```
In [14]: v5 = c3.subs({z5: 1, z6: 0, z7: 0, z8: 0, z9: 0})
v6 = B(3) * v5
v7 = B(3) * v6
```

We have $v_6 \in K_3^{(2)} \setminus K_3^{(1)}$ and $v_7 \in K_3^{(1)}$. Since $K_3^{(1)}$ is two dimensional and $K_3^{(2)}$ is four dimensional, we find one more vector from $K_3^{(2)} \setminus K_3^{(1)}$ linearly independent from v_6 . Let v_8 be the vector obtained by substitutions $y_6 = 1, y_7 = y_8 = y_9 = 0$ in \mathbf{b}_3 and put $v_9 = \mathbf{B}(3)v_8$.

```
In [15]: v8 = b3.subs({y6: 1, y7: 0, y8: 0, y9: 0})
v9 = B(3) * v8
```

Then, $\{v_5, v_6, v_7, v_8, v_9\}$ is linearly independent and forms a basis of K_3 (see Fig. 8.3).

Now, putting

$$\mathbf{V} \stackrel{\text{def}}{=} [v_0 \ v_1 \ v_2 \ v_3 \ v_4 \ v_5 \ v_6 \ v_7 \ v_8 \ v_9],$$

we compute $\mathbf{V}^{-1}\mathbf{A}\mathbf{V}$.

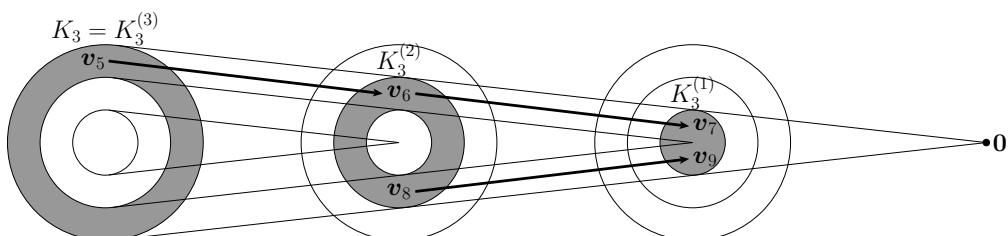


Fig. 8.3 Relation between the eigenspace and generalized eigenspace for eigenvalue 3

```
In [16]: L = [v0, v1, v2, v3, v4, v5, v6, v7, v8, v9]
V = reduce(lambda x, y: x.row_join(y), L)
V**(-1) * A * V
```

```
Out[16]: Matrix([
[1, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 2, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 2, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 2, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 2, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 3, 0, 0, 0],
[0, 0, 0, 0, 0, 1, 3, 0, 0],
[0, 0, 0, 0, 0, 0, 1, 3, 0],
[0, 0, 0, 0, 0, 0, 0, 3, 0],
[0, 0, 0, 0, 0, 0, 0, 1, 3]])
```

We obtain the matrix with the eigenvalues on the diagonal line and with some 1's on the subdiagonal line.

From the ten relations

$$\begin{aligned} \mathbf{A}\mathbf{v}_0 - \mathbf{v}_0 &= \mathbf{B}(1)\mathbf{v}_0 = 0\mathbf{v}_0, \\ \mathbf{A}\mathbf{v}_1 - 2\mathbf{v}_1 &= \mathbf{B}(2)\mathbf{v}_1 = 0\mathbf{v}_1 + 1\mathbf{v}_2 + 0\mathbf{v}_3 + 0\mathbf{v}_4, \\ \mathbf{A}\mathbf{v}_2 - 2\mathbf{v}_2 &= \mathbf{B}(2)\mathbf{v}_2 = 0\mathbf{v}_1 + 0\mathbf{v}_2 + 0\mathbf{v}_3 + 0\mathbf{v}_4, \\ \mathbf{A}\mathbf{v}_3 - 2\mathbf{v}_3 &= \mathbf{B}(2)\mathbf{v}_3 = 0\mathbf{v}_1 + 0\mathbf{v}_2 + 0\mathbf{v}_3 + 1\mathbf{v}_4, \\ \mathbf{A}\mathbf{v}_4 - 2\mathbf{v}_4 &= \mathbf{B}(2)\mathbf{v}_4 = 0\mathbf{v}_1 + 0\mathbf{v}_2 + 0\mathbf{v}_3 + 0\mathbf{v}_4, \\ \mathbf{A}\mathbf{v}_5 - 3\mathbf{v}_5 &= \mathbf{B}(3)\mathbf{v}_5 = 0\mathbf{v}_5 + 1\mathbf{v}_6 + 0\mathbf{v}_7 + 0\mathbf{v}_8 + 0\mathbf{v}_9, \\ \mathbf{A}\mathbf{v}_6 - 3\mathbf{v}_6 &= \mathbf{B}(3)\mathbf{v}_6 = 0\mathbf{v}_5 + 0\mathbf{v}_6 + 1\mathbf{v}_7 + 0\mathbf{v}_8 + 0\mathbf{v}_9, \\ \mathbf{A}\mathbf{v}_7 - 3\mathbf{v}_7 &= \mathbf{B}(3)\mathbf{v}_7 = 0\mathbf{v}_5 + 0\mathbf{v}_6 + 0\mathbf{v}_7 + 0\mathbf{v}_8 + 0\mathbf{v}_9, \\ \mathbf{A}\mathbf{v}_8 - 3\mathbf{v}_8 &= \mathbf{B}(3)\mathbf{v}_8 = 0\mathbf{v}_5 + 0\mathbf{v}_6 + 0\mathbf{v}_7 + 0\mathbf{v}_8 + 1\mathbf{v}_9, \\ \mathbf{A}\mathbf{v}_9 - 3\mathbf{v}_9 &= \mathbf{B}(3)\mathbf{v}_9 = 0\mathbf{v}_5 + 0\mathbf{v}_6 + 0\mathbf{v}_7 + 0\mathbf{v}_8 + 0\mathbf{v}_9, \end{aligned}$$

we have

$$\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \left[\begin{array}{c|c|c} \mathbf{A}_1 & \mathbf{O} & \mathbf{O} \\ \hline \mathbf{O} & \mathbf{A}_2 & \mathbf{O} \\ \hline \mathbf{O} & \mathbf{O} & \mathbf{A}_3 \end{array} \right] = \left[\begin{array}{c|ccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 3 \end{array} \right].$$

Focusing on \mathbf{A}_2 and \mathbf{A}_3 , they are further divided as

$$A_2 = \left[\begin{array}{cc|cc} 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ \hline 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 2 \end{array} \right] \text{ and } A_3 = \left[\begin{array}{ccc|cc} 3 & 0 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 \\ \hline 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 & 3 \end{array} \right].$$

A square matrix of the form

$$\left[\begin{array}{cccccc|c} \lambda & 0 & \cdots & \cdots & 0 \\ 1 & \ddots & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & \lambda \end{array} \right]$$

is called a *Jordan block*. Any square matrix can be transformed by a basis change into a Jordan normal form in which Jordan blocks are aligned on the diagonal line.

The Jordan normal form may vary with the order of listing the vectors v_0 to v_9 in making a basis change matrix. For example, if we take

$$U = [v_9 \ v_8 \ v_7 \ v_6 \ v_5 \ v_4 \ v_3 \ v_2 \ v_1 \ v_0]$$

for the basis change matrix, then $U^{-1}AU$ becomes a Jordan normal form of the following appearance.

```
In [17]: L = [v9, v8, v7, v6, v5, v4, v3, v2, v1, v0]
U = reduce(lambda x, y: x.row_join(y), L)
U**(-1) * A * U
```

```
Out[17]: Matrix([
[3, 1, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 3, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 3, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 3, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 3, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 2, 1, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 2, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 2, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 2, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])
```

In this case, the number 1 appears on the superdiagonal line of each Jordan block. In addition, the order in which Jordan blocks appear is reversed.

Exercise 8.2 The following program generates an exercise for Jordan normal form and Jordan decomposition (see the next section). Solve the problem generated by this program by hand.

Program: jordan2.py

```
In [1]: 1 from sympy import Matrix, diag
2 from numpy.random import permutation, seed
3
4 X = Matrix([[1, 1, 0], [0, 1, 0], [0, 0, 2]])
5 Y = Matrix([[2, 1, 0], [0, 2, 1], [0, 0, 2]])
6 Z = Matrix([[2, 1, 0], [0, 2, 0], [0, 0, 2]])
7
8 seed(2021)
```

In [1]:

```

9  while True:
10     A = X.copy()
11     while 0 in A:
12         i, j, _ = permutation(3)
13         A[:, j] += A[:, i]
14         A[i, :] -= A[j, :]
15         if max(abs(A)) >= 10:
16             break
17         if max(abs(A)) < 10:
18             break
19
20 U, J = A.jordan_form()
21 print(f'A = {A}')
22 print(f'U = {U}')
23 print(f'U**(-1)*A*U = {J}')
24 C = U * diag(J[0, 0], J[1, 1], J[2, 2]) * U**(-1)
25 B = A - C
26 print(f'B = {B}')
27 print(f'C = {C}')

```

Lines 4–6: Prepare three non-similar Jordan normal forms of order 3.

Lines 8–18: Generate randomly a matrix similar to each Jordan normal form. Addition of the i -th column vector to the j -th one and subtraction from the i -th row vector to the j -th one preserve similarity of matrices. These transformations are randomly performed several times. If every element of the generated matrix is nonzero and its absolute value is less than 10, we adopt it as a problem. If we replace X on Line 10 by Y or Z, we can obtain a problem of different Jordan normal form.

Line 20: Compute the Jordan normal form by the method `jordan_form` of the `Matrix` class.

Lines 24–27: Compute the Jordan decomposition, which will be explained in the next section.



```

A = Matrix([[2, 4, 4], [-4, 3, -1], [2, -4, -1]])
U = Matrix([[24/7, -4/7, -1/2], [30/7, 1, -1], [-36/7, 0, 1]])
U**(-1)*A*U = Matrix([[1, 1, 0], [0, 1, 0], [0, 0, 2]])
B = Matrix([[8, 8, 12], [10, 10, 15], [-12, -12, -18]])
C = Matrix([[-6, -4, -8], [-14, -7, -16], [14, 8, 17]])

```

8.3 Jordan Decomposition and Matrix Power

A square matrix \mathbf{B} is a *nilpotent matrix* if $\mathbf{B}^k = \mathbf{O}$ for some $k \geq 1$. The matrix $\mathbf{A}_i - \lambda_i \mathbf{I}_i$ of order n_i given in the previous section is a nilpotent matrix, because $(\mathbf{A}_i - \lambda_i \mathbf{I}_i)^{n_i} = \mathbf{O}$. From the decomposition of a square matrix \mathbf{A} given there, we have

$$\begin{aligned}\mathbf{A} &= \mathbf{V} (\mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \cdots \oplus \mathbf{A}_m) \mathbf{V}^{-1} \\ &= \mathbf{V} ((\mathbf{A}_1 - \lambda_1 \mathbf{I}_1) \oplus (\mathbf{A}_2 - \lambda_2 \mathbf{I}_2) \oplus \cdots \oplus (\mathbf{A}_m - \lambda_m \mathbf{I}_m)) \mathbf{V}^{-1} \\ &\quad + \mathbf{V} (\lambda_1 \mathbf{I}_1 \oplus \lambda_2 \mathbf{I}_2 \oplus \cdots \oplus \lambda_m \mathbf{I}_m) \mathbf{V}^{-1}.\end{aligned}$$

Hence, \mathbf{A} is decomposed as $\mathbf{A} = \mathbf{B} + \mathbf{C}$, where $\mathbf{B} = \mathbf{V} ((\mathbf{A}_1 - \lambda_1 \mathbf{I}_1) \oplus (\mathbf{A}_2 - \lambda_2 \mathbf{I}_2) \oplus \cdots \oplus (\mathbf{A}_m - \lambda_m \mathbf{I}_m)) \mathbf{V}^{-1}$ is nilpotent and $\mathbf{C} = \mathbf{V} (\lambda_1 \mathbf{I}_1 \oplus \lambda_2 \mathbf{I}_2 \oplus \cdots \oplus \lambda_m \mathbf{I}_m) \mathbf{V}^{-1}$ is diagonalizable. Moreover, \mathbf{B} and \mathbf{C} commute, that is, $\mathbf{BC} = \mathbf{CB}$ because \mathbf{A}_i and $\lambda_i \mathbf{I}_i$ commute.

We shall prove that such a decomposition of \mathbf{A} is unique. Assume that $\mathbf{A} = \mathbf{B}' + \mathbf{C}'$ for a nilpotent matrix \mathbf{B}' and a diagonalizable matrix \mathbf{C}' which commute. Then, $\mathbf{A} - \lambda_i \mathbf{I}$ commutes with \mathbf{B}' because $\mathbf{A} = \mathbf{B}' + \mathbf{C}'$ and \mathbf{I} commute with \mathbf{B}' . Thus, for all $x \in K_i$, we have

$$(A - \lambda_i I)^{n_i} B' x = B' (A - \lambda_i I)^{n_i} x = \mathbf{0},$$

and hence $B' x \in K_i$ for all $i = 1, 2, \dots, m$. Therefore, the family $\{K_1, K_2, \dots, K_m\}$ of the generalized eigenspaces decomposes B' and we have the decomposition $V^{-1} B' V = B_1 \oplus B_2 \oplus \dots \oplus B_m$. Because it also decomposes $C' = A - B'$, we have $V^{-1} C' V = C_1 \oplus C_2 \oplus \dots \oplus C_m$. From

$$\begin{aligned} A_1 \oplus A_2 \oplus \dots \oplus A_m &= V^{-1} A V = V^{-1} (B' + C') V \\ &= (B_1 + C_1) \oplus (B_2 + C_2) \oplus \dots \oplus (B_k + C_k), \end{aligned}$$

we have $A_i = B_i + C_i$, and thus

$$(A_i - \lambda_i I_i) - B_i = C_i - \lambda_i I_i$$

for each i . The matrix on the left-hand side of this equality is nilpotent because $A_i - \lambda_i I_i$ and B_i are nilpotent and commute. On the other hand, the matrix on the right-hand side is diagonalizable. They are equal if and only if both are zero. Therefore, we find $B_i = A_i - \lambda_i I_i$ and $C_i = \lambda_i I_i$ for each i , that is, $B' = B$ and $C' = C$.

We refer to such a decomposition of A into the sum of a nilpotent matrix B and a diagonalizable matrix C which commute as the *Jordan decomposition*.

For the diagonal matrix $C = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$, we have

$$C^k = \text{diag}(\lambda_1^k, \lambda_2^k, \dots, \lambda_n^k)$$

for $k \in \mathbb{N}$. On the other hand, for a nilpotent matrix

$$B = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 \\ 1 & 0 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 & 0 \end{bmatrix}$$

of order n we have

$$B^2 = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 \\ 0 & 0 & \ddots & & \vdots \\ 1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 1 & 0 & 0 \end{bmatrix}, \quad \dots, \quad B^{n-1} = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 \\ 0 & 0 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 1 & \dots & 0 & 0 & 0 \end{bmatrix}$$

and finally $B^n = O$. If J is the Jordan block

$$\mathbf{J} = \begin{bmatrix} a & 0 & \cdots & \cdots & 0 \\ 1 & a & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & a \end{bmatrix}$$

with $a \in \mathbb{K}$, then we have the Jordan decomposition $\mathbf{J} = a\mathbf{I} + \mathbf{B}$ of \mathbf{J} , where \mathbf{B} is the nilpotent matrix above. By the binomial expansion formula, we have

$$\mathbf{J}^k = (a\mathbf{I} + \mathbf{B})^k = \sum_{i=0}^k {}_k C_i a^{k-i} \mathbf{B}^i,$$

where ${}_k C_i = \frac{k!}{i!(k-i)!}$ are the binomial coefficients. For $k \geq n$ we have

$$\mathbf{J}^k = \sum_{i=0}^{n-1} {}_k C_i a^{k-i} \mathbf{B}^i,$$

because $\mathbf{B}^i = \mathbf{O}$ for $i \geq n$. Thus

$$\mathbf{J}^k = \begin{bmatrix} a^k & 0 & \cdots & \cdots & 0 \\ {}_k C_1 a^{k-1} & a^k & \ddots & & \vdots \\ {}_k C_2 a^{k-2} & {}_k C_1 a^{k-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ {}_k C_{n-1} a^{k-n+1} & \cdots & {}_k C_2 a^{k-2} & {}_k C_1 a^{k-1} & a^k \end{bmatrix}.$$

To confirm this, we compute \mathbf{J}^3 for a Jordan block \mathbf{J} of order 3 in SymPy.

```
In [1]: from sympy import Matrix, S
J = Matrix([[S('a'), 0, 0], [1, S('a'), 0], [0, 1, S('a')]])
```

```
Out[1]: Matrix([
[a, 0, 0],
[1, a, 0],
[0, 1, a]])
```

```
In [2]: J**2
```

```
Out[2]: Matrix([
[a**2, 0, 0],
[2*a, a**2, 0],
[1, 2*a, a**2]]))
```

In [3]: J^{**3} Out[3]: $\text{Matrix}([[a^{**3}, 0, 0], [3*a^{**2}, a^{**3}, 0], [3*a, 3*a^{**2}, a^{**3}]])$ In [4]: $J^{**S('k')}$ Out[4]: $\text{Matrix}([[a^{**k}, 0, 0], [a^{**(k-1)*k}, a^{**k}, 0], [a^{**(k-2)*k*(k-1)/2}, a^{**(k-1)*k}, a^{**k}]])$

For a general square matrix A of order n with the Jordan decomposition $A = B + C$, we have

$$A^k = (B + C)^k = \sum_{i=0}^k {}_k C_i B^i C^{k-i} = \sum_{i=0}^{n-1} {}_k C_i B^i C^{k-i}$$

for $k \geq n$.

8.4 Spectrum of a Matrix

A square matrix A of order n has exactly n eigenvalues with multiplicity in the complex plane. We call the set of all eigenvalues of A the *spectrum* of A .

Let us randomly generate a square matrix of order 100 and observe the spectrum of the matrix surrounded by the circle of the spectral radius. A is a matrix in which the real and imaginary parts of 100×100 elements are generated subject to the standard normal distribution, respectively. The spectrum of A is shown in Fig. 8.4 (left). The right figure shows the spectrum of the real matrix $A + \bar{A}$. The spectrum of a real matrix appears symmetric with respect to the real axis.

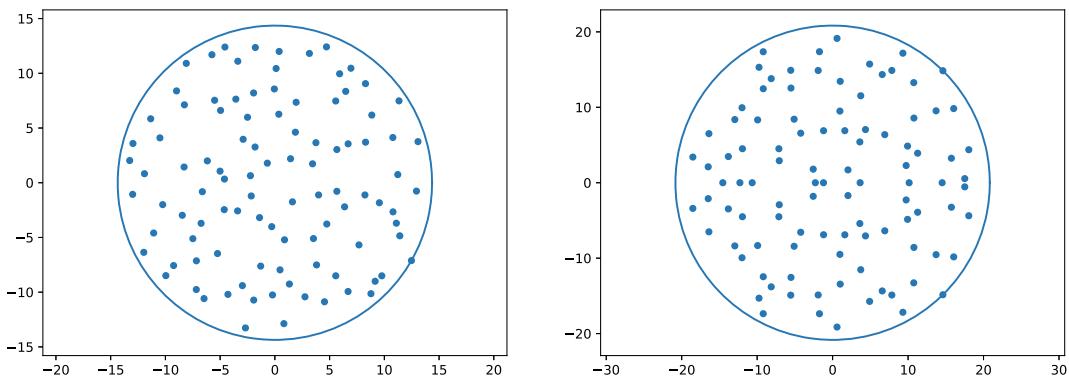


Fig. 8.4 Spectra of a complex matrix (left) and a real matrix (right)

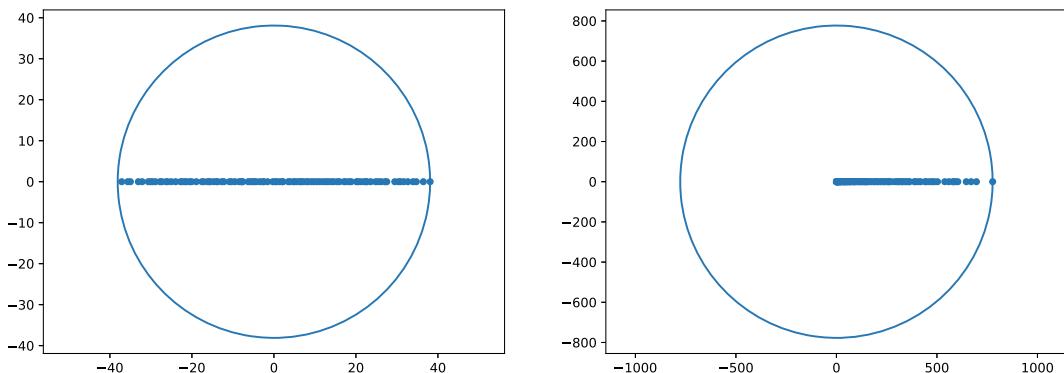


Fig. 8.5 Spectra of a Hermitian matrix (left) and a positive definite matrix (right)

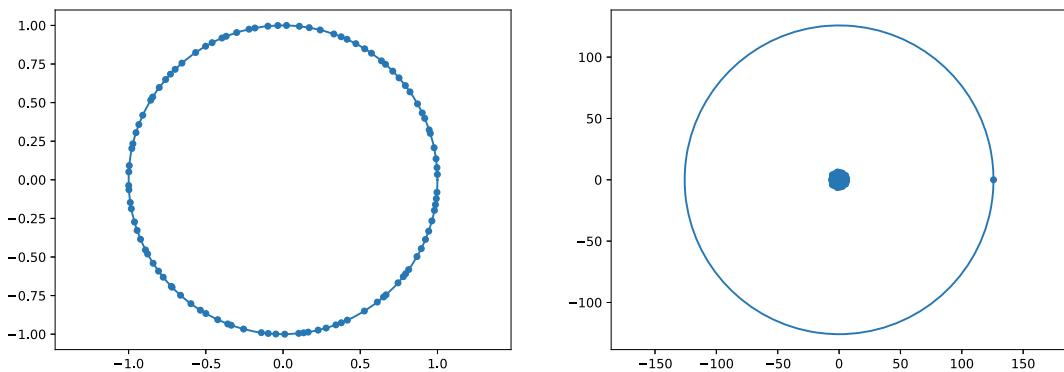


Fig. 8.6 Spectra of a unitary matrix (left) and a positive matrix (right)

The left part of Fig. 8.5 shows the spectrum of the Hermitian matrix $A + A^*$, which lies on the real axis. The right part shows the spectrum of the positive semi-definite matrix A^*A , which lies on the nonnegative half-line of the real axis.²

The spectrum of a unitary matrix is shown in Fig. 8.6 (left). This matrix is made by normalizing the eigenvectors of the Hermitian matrix. Its eigenvalues are complex numbers with absolute value 1.

A real square matrix is said to be a *positive matrix*³ (resp. *nonnegative matrix*) if all its elements are positive (resp. nonnegative). Figure 8.6 (right) shows the spectrum of the matrix⁴ whose elements are the absolute values of corresponding elements of A . For a positive matrix, only one positive eigenvalue appears on the circumference of the spectral radius. This phenomenon is explained by the Perron–Frobenius theorem, which will be stated later.

The following is the program which produces Figs. 8.4, 8.5, and 8.6.

² In general, A^*A is a positive semi-definite matrix, but in almost all cases it is a positive definite matrix because A is randomly generated. The spectrum of a positive definite matrix does not contain the origin.

³ Remark that a positive matrix is completely different from a positive definite matrix with a similar name. The positivity is not invariant under similarity nor under unitary equivalence of matrices. In this sense it is a peculiar property of matrices. Positive matrices are very important for applications.

⁴ In almost all cases this is a positive matrix.

Program: spectrum.py

```

In [1]: 1 from numpy import matrix, pi, sin, cos, linspace
2 from numpy.random import normal
3 from numpy.linalg import eig, eigh
4 import matplotlib.pyplot as plt
5
6 N = 100
7 B = normal(0, 1, (N, N, 2))
8 A = matrix(B[:, :, 0] + 1j * B[:, :, 1])
9 Real = A + A.conj()
10 Hermite = A + A.H
11 PositiveSemidefinite = A * A.H
12 PositiveComponents = abs(A)
13 Unitary = matrix(eigh(Hermite)[1])
14
15 X = PositiveComponents
16 Lmd = eig(X)[0]
17 r = max(abs(Lmd))
18 T = linspace(0, 2 * pi, 100)
19 plt.axis('equal')
20 plt.plot(r * cos(T), r * sin(T))
21 plt.scatter(Lmd.real, Lmd.imag, s=20)
22 plt.show()

```

Lines 9–13: Make and define real, Hermitian, positive definite, positive, and unitary matrices.

Line 15: If the definition of X changes, the figure of the spectrum displayed will change.

Let A be a square matrix of order n . For $\lambda \in \mathbb{K}$ with $|\lambda| > \rho(A)$, we shall show that

$$\frac{A^k}{\lambda^k} \rightarrow \mathbf{O} \quad (k \rightarrow \infty).$$

As discussed at the end of the previous section, we have the Jordan decomposition $A = B + C$ with a nilpotent matrix B and a diagonalizable matrix C which commute, and by performing the binomial expansion of $\left(\frac{B+C}{\lambda}\right)^k$ for $k \geq n$, we have

$$\frac{A^k}{\lambda^k} = \sum_{i=0}^{n-1} \frac{{}_k C_i B^i C^{k-i}}{\lambda^k}.$$

Since $\|C\| = \rho(A)$, we have

$$\left\| \frac{A^k}{\lambda^k} \right\| \leq \sum_{i=0}^{n-1} \frac{{}_k C_i \|B\|^i \rho(A)^{k-i}}{|\lambda|^k}.$$

Here, let $\alpha = \frac{\rho(A)}{|\lambda|}$ and $\beta = \max_{0 \leq i < n} \left(\frac{\|B\|}{\rho(A)} \right)^i$, then $\alpha < 1$ by assumption, β and n are constants and ${}_k C_i \leq k^i < k^n$ for $i < n$. Hence, the right-hand side above is bounded by $n\beta k^n \alpha^k$ which converges to 0 as $k \rightarrow \infty$. Hence, $\left\| \frac{A^k}{\lambda^k} \right\| \rightarrow 0$ as $k \rightarrow \infty$, and thus we obtain the desired result.

The following program randomly generates a square matrix A of order 3 and observes the convergence of each element and the matrix norm of $(A/\lambda)^k$ with λ a little larger than the spectral radius of A (see Fig. 8.7).

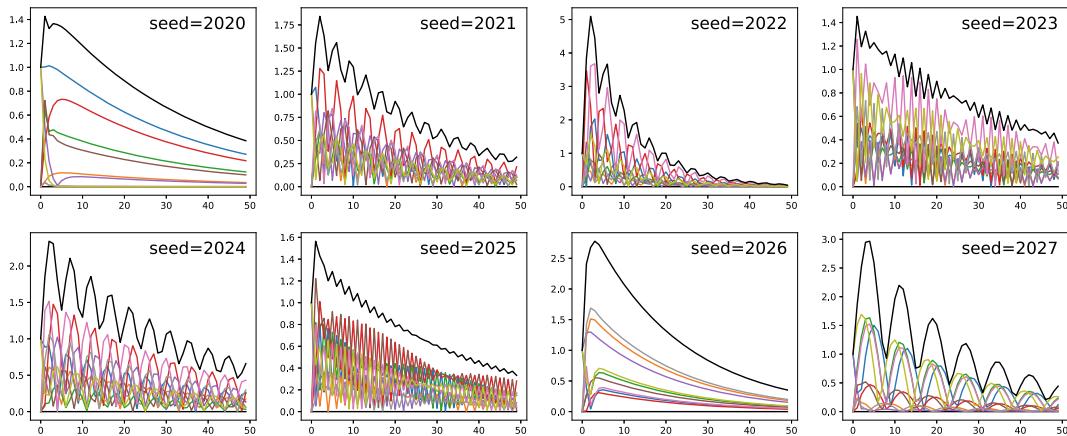


Fig. 8.7 Convergence of $(A/\lambda)^k$

Program: norm.py

```
In [1]: 1 from numpy import matrix
2 from numpy.linalg import eig, norm
3 from numpy.random import normal, seed
4 import matplotlib.pyplot as plt
5
6 def power(m, s):
7     seed(s)
8     A = matrix(normal(0, 2, (m, m)))
9     lmd = max(abs(eig(A)[0])) + 0.1
10    X = range(50)
11    P = [(A / lmd)**n for n in X]
12    Y = [norm(B, 2) for B in P]
13    plt.plot([X[0], X[-1]], [0, 0], c='k')
14    for i in range(m):
15        for j in range(m):
16            plt.plot(X, [abs(B[i, j]) for B in P])
17    plt.plot(X, Y, c='k')
18    plt.text(max(X), max(Y), f'seed={s}', size=18, ha='right', va='top')
19
20 power(3, 2021)
21
22 plt.show()
```

Using the above convergence, we show a formula indicating the relationship between the spectral radius and matrix norm, which will be used in the proof of the Perron–Frobenius theorem.

Theorem 8.1 (Gelfand's formula)

$$\rho(\mathbf{A}) = \lim_{k \rightarrow \infty} \|\mathbf{A}^k\|^{1/k}.$$

Proof Let λ be an eigenvalue of \mathbf{A} and \mathbf{x} the normalized eigenvector associated with it. For all $k \in \mathbb{N}$, we have

$$|\lambda|^k = \|\lambda^k \mathbf{x}\| = \|\mathbf{A}^k \mathbf{x}\| \leq \|\mathbf{A}^k\|,$$

and thus $|\lambda| \leq \|\mathbf{A}^k\|^{1/k}$, which yields $\rho(\mathbf{A}) \leq \|\mathbf{A}^k\|^{1/k}$. Let $\varepsilon > 0$ be arbitrary. From the convergence we have just proved with $\lambda = \varepsilon + \rho(\mathbf{A})$, we have

$$\lim_{k \rightarrow \infty} \frac{\|A^k\|}{(\varepsilon + \rho(A))^k} = \lim_{k \rightarrow \infty} \left\| \frac{A^k}{(\varepsilon + \rho(A))^k} \right\| = 0 < 1.$$

Hence, there exists an integer k_0 such that for all $k \geq k_0$

$$\|A^k\| < (\varepsilon + \rho(A))^k,$$

and therefore we have

$$\rho(A) \leq \|A^k\|^{1/k} < \varepsilon + \rho(A).$$

Since $\varepsilon > 0$ is arbitrary, it follows that $\lim_{k \rightarrow \infty} \|A^k\|^{1/k} = \rho(A)$.⁵ ■

The following program randomly generates a square matrix of order 3 and observes the convergence in Gelfand's formula (Fig. 8.8).

Program: gelfand.py

```
In [1]: 1 from numpy import matrix
2 from numpy.linalg import eig, norm
3 from numpy.random import normal, seed
4 import matplotlib.pyplot as plt
5
6 def gelfand(m, s):
7     seed(s)
8     A = matrix(normal(0, 1, (m, m)))
9     lmd = max(abs(eig(A)[0]))
10    X = range(1, 50)
11    P = [A**n for n in range(50)]
12    Y = [norm(P[n], 2)**(1 / n) for n in X]
13    plt.plot([X[0], X[-1]], [lmd, lmd], c='k')
14    for i in range(m):
15        for j in range(m):
16            plt.plot(X, [abs(P[n][i, j])***(1 / n) for n in X])
17    plt.plot(X, Y, c='k')
18    plt.text(max(X), max(Y), f'seed={s}', size=18, ha='right', va='top')
19
20
21 gelfand(3, 2021)
22 plt.show()
```

The appearances of convergences in Figs. 8.7 and 8.8 vary depending on the seeds selected.

Exercise 8.3 Experiment with changing seeds in Programs `norm.py` and `gelfand.py` and observe the convergences.

⁵ This argument is based on the so-called ε - δ definition of a limit.

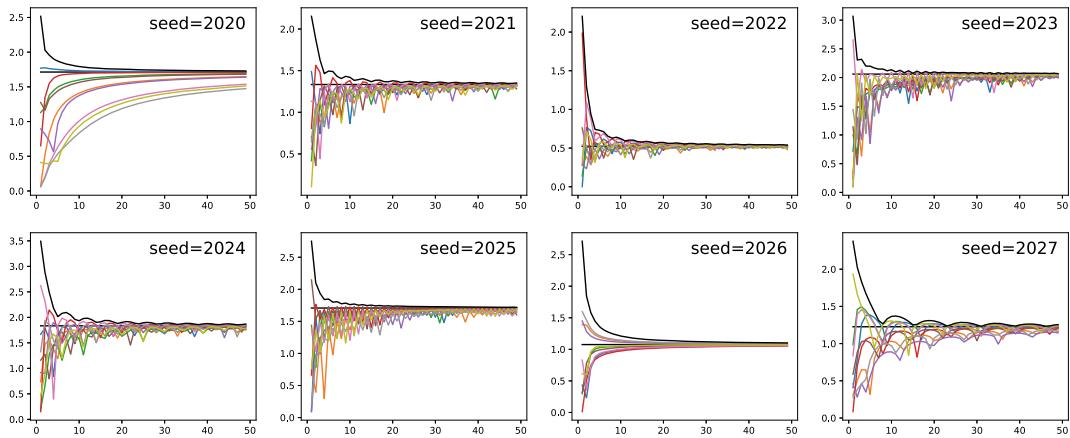


Fig. 8.8 Gelfand's formula

8.5 Perron–Frobenius Theorem

For real vectors $\mathbf{x} = (x_1, x_2, \dots, x_n)$, $\mathbf{y} = (y_1, y_2, \dots, y_n) \in \mathbb{R}^n$, we define the following inequalities:

$$\begin{aligned}\mathbf{x} &\geq \mathbf{y} \Leftrightarrow x_i \geq y_i \text{ for all } i, \\ \mathbf{x} &\geq \mathbf{y} \Leftrightarrow x_i \geq y_i \text{ for all } i \text{ and } x_i > y_i \text{ for some } i \Leftrightarrow \mathbf{x} \geq \mathbf{y} \text{ and } \mathbf{x} \neq \mathbf{y}, \\ \mathbf{x} &> \mathbf{y} \Leftrightarrow x_i > y_i \text{ for all } i.\end{aligned}$$

Lemma 8.1 *If $\mathbf{x} > \mathbf{y} \geq \mathbf{0}$, then there exists $\varepsilon > 0$ such that $\mathbf{x} > (1 + \varepsilon) \mathbf{y}$.*

Proof For each i , because $x_i > y_i = \lim_{\varepsilon \rightarrow 0} (1 + \varepsilon)y_i$, there is $\varepsilon_i > 0$ such that $x_i > (1 + \varepsilon_i)y_i$. Letting $\varepsilon = \min_{1 \leq i \leq n} \varepsilon_i$, we have the desired inequality. ■

For $\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{C}^n$, we define $|\mathbf{x}|$ by $(|x_1|, |x_2|, \dots, |x_n|)$.

Lemma 8.2 *For a complex vector $\mathbf{x} \neq \mathbf{0}$ and a real vector $\mathbf{y} > \mathbf{0}$, if $\langle \mathbf{y} \mid |\mathbf{x}| \rangle = |\langle \mathbf{y} \mid \mathbf{x} \rangle|$, then there exists $c \in \mathbb{C} \setminus \{0\}$ such that $c\mathbf{x} \geq \mathbf{0}$.*

Proof Let $\mathbf{x} = (x_1, x_2, \dots, x_n)$ and $\mathbf{y} = (y_1, y_2, \dots, y_n)$. By assumption, y_1, y_2, \dots, y_n are all positive and

$$\sum_{i=1}^n y_i |x_i| = \left| \sum_{i=1}^n y_i x_i \right|.$$

Squaring both sides, we have

$$\sum_{i=1}^n \sum_{j=1}^n y_i y_j |x_i| |x_j| = \sum_{i=1}^n \sum_{j=1}^n y_i y_j x_i \overline{x_j},$$

or

$$\sum_{i=1}^n \sum_{j=1}^n y_i y_j (|x_i \bar{x}_j| - x_i \bar{x}_j) = 0.$$

Because $y_i y_j > 0$ and $\operatorname{Re}(|x_i \bar{x}_j| - x_i \bar{x}_j) \geq 0$, we find $\operatorname{Re}(|x_i \bar{x}_j| - x_i \bar{x}_j) = 0$ for each i, j . In general, for $z \in \mathbb{C}$, $\operatorname{Re}(|z| - z) = 0$ if and only if z is a nonnegative real number. Thus, we have $x_i \bar{x}_j \geq 0$ for all i, j . Choose a nonzero x_j and put $c = \bar{x}_j$, then we obtain $c\mathbf{x} \geq \mathbf{0}$. ■

Theorem 8.2 (The Perron–Frobenius theorem) *For a positive matrix A , the following assertions hold:*

1. *The spectral radius $\rho(A)$ of A is positive and is an eigenvalue of A , and it is possible to take a positive eigenvector $\mathbf{x} > 0$ associated with it.*
2. *The absolute values of all eigenvalues of A except $\rho(A)$ are less than $\rho(A)$.*
3. *The multiplicity of the eigenvalue $\rho(A)$ is 1.*

Proof Let λ be an eigenvalue of A with $|\lambda| = \rho(A)$ and $\mathbf{x} = (x_1, x_2, \dots, x_n)$ be an eigenvector associated with it. Let a_{ij} be the (i, j) -element of A . For each i , we have

$$\begin{aligned} \text{the } i\text{-th element of } A|\mathbf{x}| &= \sum_{j=1}^n a_{ij} |x_j| = \sum_{j=1}^n |a_{ij} x_j| \geq \left| \sum_{j=1}^n a_{ij} x_j \right| \\ &= |\text{the } i\text{-th element of } A\mathbf{x}| = |\lambda x_i| = \rho(A) |x_i| \\ &= \text{the } i\text{-th element of } \rho(A)|\mathbf{x}|. \quad \cdots \quad (***) \end{aligned}$$

Hence we have the inequality $A|\mathbf{x}| \geq \rho(A)|\mathbf{x}|$. To lead to a contradiction assume that the equality does not hold, that is,

$$A|\mathbf{x}| > \rho(A)|\mathbf{x}|.$$

Operating the positive matrix A on both sides, we have

$$A^2|\mathbf{x}| > \rho(A)A|\mathbf{x}|.$$

Thus, by Lemma 8.1, there exists $\varepsilon > 0$ such that

$$A^2|\mathbf{x}| > (1 + \varepsilon)\rho(A)A|\mathbf{x}|.$$

Again, operating A on both sides, we have

$$A^3|\mathbf{x}| > (1 + \varepsilon)\rho(A)A^2|\mathbf{x}| > ((1 + \varepsilon)\rho(A))^2 A|\mathbf{x}|.$$

Repeating this operation yields

$$A^{k+1}|\mathbf{x}| > ((1 + \varepsilon)\rho(A))^k A|\mathbf{x}|$$

for all $k \geq 2$. Taking the norm on \mathbb{R}^n , we have

$$\begin{array}{ccc} \|A^{k+1}|\mathbf{x}|\| & > & \left\|((1+\varepsilon)\rho(A))^k A |\mathbf{x}|\right\| \\ \wedge \| & & \| \\ \|A^k\| \cdot \|A|\mathbf{x}|\| & > & ((1+\varepsilon)\rho(A))^k \cdot \|A|\mathbf{x}|\|. \end{array}$$

Since $A|\mathbf{x}| \neq \mathbf{0}$, we obtain

$$\|A^k\|^{1/k} > (1+\varepsilon)\rho(A).$$

As $k \rightarrow \infty$, the left-hand side converges to $\rho(A)$ by Gelfand's formula. This is a contradiction.

Therefore, we obtain $A|\mathbf{x}| = \rho(A)|\mathbf{x}|$, and $\rho(A)$ is an eigenvalue of A with associated eigenvector $|\mathbf{x}|$. Moreover, since A is positive and $|\mathbf{x}| \geq \mathbf{0}$, we see $A|\mathbf{x}| > \mathbf{0}$. It follows that $\rho(A) > 0$ and $|\mathbf{x}| > \mathbf{0}$. Therefore, assertion 1 has been proved.

Since both sides of (**) are equal, all the terms are connected by equal signs and we have

$$\langle \mathbf{a}_i \mid |\mathbf{x}|\rangle = \sum_{j=1}^n a_{ij} |x_j| = \left| \sum_{j=1}^n a_{ij} x_j \right| = |\langle \mathbf{a}_i \mid \mathbf{x} \rangle|,$$

where \mathbf{a}_i is the i -th row vector of A . Hence, by Lemma 8.2, there exists $c \in \mathbb{C} \setminus \{0\}$ such that $c\mathbf{x} \geq \mathbf{0}$. By the positivity of A , we have

$$\lambda(c\mathbf{x}) = A(c\mathbf{x}) > 0,$$

and so $\lambda > 0$. Since $|\lambda| = \rho(A)$, it follows that $\lambda = \rho(A)$. Namely, $\rho(A)$ is the only eigenvalue of A whose absolute value equals $\rho(A)$, and assertion 2 has been proved.

Finally, we shall prove assertion 3. First, let us show that the eigenspace W_λ for $\lambda = \rho(A)$ is one dimensional. By 1 we have $\mathbf{x} \in W_\lambda$ such that $\mathbf{x} > \mathbf{0}$. Assume that there exists another $\mathbf{y} \in W_\lambda$, which is linearly independent from \mathbf{x} . Then, by the same discussion as in 2, there exists $c \in \mathbb{C} \setminus \{0\}$ such that $c\mathbf{y} \geq \mathbf{0}$. For a real number $t \geq 0$ put $\mathbf{z}(t) = \mathbf{x} - tcy$. Since $\mathbf{z}(0) = \mathbf{x} > \mathbf{0}$ and \mathbf{x} is not a scalar multiple of $c\mathbf{y}$, we can choose t such that $\mathbf{z}(t) \geq \mathbf{0}$ and at least one component of $\mathbf{z}(t)$ is 0. Because $\mathbf{z}(t) \in W_\lambda$ and A is positive, we have $\lambda\mathbf{z}(t) = A\mathbf{z}(t) > 0$. Thus, we get $\mathbf{z}(t) > \mathbf{0}$, but this contradicts the choice of t , and we have proved that the eigenspace is one dimensional. Next, assume that the dimension of the generalized eigenspace was greater than 1, then we would have

$$\{\mathbf{0}\} \subsetneq W_\lambda = \text{kernel}(A - \lambda I) \subsetneq \text{kernel}((A - \lambda I)^2).$$

Therefore, there are \mathbf{v} with $\mathbf{0} \neq \mathbf{v} \in W_\lambda$ and $\mathbf{u} \in \text{kernel}((A - \lambda I)^2)$ such that $(A - \lambda I)\mathbf{u} = \mathbf{v}$. We may suppose $\mathbf{v} > \mathbf{0}$ because W_λ is one dimensional. On the other hand, since A^T is also a positive matrix, it has the eigenvalue $\mu = \rho(A^T)$ and an eigenvector $\mathbf{w} > 0$ associated with it. We have

$$\lambda \langle \mathbf{v} \mid \mathbf{w} \rangle = \langle \lambda \mathbf{v} \mid \mathbf{w} \rangle = \langle A\mathbf{v} \mid \mathbf{w} \rangle = \langle \mathbf{v} \mid A^T \mathbf{w} \rangle = \langle \mathbf{v} \mid \mu \mathbf{w} \rangle = \mu \langle \mathbf{v} \mid \mathbf{w} \rangle.$$

Here $\langle \mathbf{v} \mid \mathbf{w} \rangle > 0$ because $\mathbf{v} > \mathbf{0}$ and $\mathbf{w} > \mathbf{0}$, and so we find $\mu = \lambda$. Since $A\mathbf{u} = \lambda\mathbf{u} + \mathbf{v}$, taking inner product of both sides with \mathbf{w} , we have

$$\langle A\mathbf{u} \mid \mathbf{w} \rangle = \langle \lambda \mathbf{u} + \mathbf{v} \mid \mathbf{w} \rangle = \lambda \langle \mathbf{u} \mid \mathbf{w} \rangle + \langle \mathbf{v} \mid \mathbf{w} \rangle.$$

Further, since

$$\langle \mathbf{A}\mathbf{u} \mid \mathbf{w} \rangle = \langle \mathbf{u} \mid \mathbf{A}^T \mathbf{w} \rangle = \langle \mathbf{u} \mid \lambda \mathbf{w} \rangle = \lambda \langle \mathbf{u} \mid \mathbf{w} \rangle,$$

we obtain $\langle \mathbf{v} \mid \mathbf{w} \rangle = 0$, which is a contradiction. ■

We call $\lambda = \rho(\mathbf{A})$ in the theorem the *Perron–Frobenius eigenvalue* of \mathbf{A} .

Exercise 8.4 Let \mathbf{A} be the square matrix of order n whose elements are all 1. Find the Perron–Frobenius eigenvalue of \mathbf{A} and a positive eigenvector associated with it.



Dynamical Systems

9

Dynamical systems are mathematical models concerning variables that change depending on time. In this chapter, we consider a system represented by a linear differential equation in which a variable changes deterministically over continuously changing time, and a Markov chain in which a variable changes stochastically with discretely changing time. In studying these systems, the theory of linear algebra, which we have learned so far, will play a major role.

Visualization by computer simulation is an important tool to elucidate the behavior of a dynamical system. There are many interesting examples treated as Python problems and we cover some of them.

Chapters 9 and 10 focus on applications. The readers are recommended to experiment with how the execution results change by changing the parameters of the Python codes presented in this book. Moreover, try to improve the codes and remake the programs to simulate a more complex model or to analyze their own data.

Let us start the applied practice of linear algebra.

9.1 Differentiation of Vector-(Matrix-) Valued Functions

Let $I \subseteq \mathbb{R}$ be an interval and let $\mathbf{x}(t) = (x_1(t), x_2(t), \dots, x_n(t))$ be an n -fold tuple whose components are functions of $t \in I$ valued in \mathbb{K} , that is, $\mathbf{x}(t)$ is a vector in \mathbb{K}^n for each $t \in I$. The vector-valued function $\mathbf{x}(t)$ is said to be *differentiable* at $t = t_0$, if there exists $\mathbf{y} = (y_1, y_2, \dots, y_n) \in \mathbb{K}^n$ such that

$$\lim_{\Delta t \rightarrow 0} \left\| \frac{\mathbf{x}(t_0 + \Delta t) - \mathbf{x}(t_0)}{\Delta t} - \mathbf{y} \right\| = 0. \quad \dots \quad (*)$$

The limit \mathbf{y} is called the *differential* of $\mathbf{x}(t)$ at t_0 and denoted by $\frac{d}{dt}\mathbf{x}(t_0)$. If $\mathbf{x}(t)$ is differentiable at every $t \in I$, it is called differentiable on I , and $\frac{d}{dt}\mathbf{x}(t)$ is also a vector-valued function. It is called the *derivative (derived function)* of $\mathbf{x}(t)$ and denoted by $\frac{d}{dt}\mathbf{x}$ or simply \mathbf{x}' .

By the discussion in Sect. 5.6, (*) holds if and only if

$$\lim_{\Delta t \rightarrow 0} \left| \frac{x_i(t_0 + \Delta t) - x_i(t_0)}{\Delta t} - y_i \right| = 0$$

holds for each $i = 1, 2, \dots, n$. Hence, \mathbf{x} is differentiable at t_0 (on I) if and only if each $x_i(t)$ is differentiable at t_0 (on I), and the derivative \mathbf{x}' is given by

$$\mathbf{x}'(t) = (x'_1(t), x'_2(t), \dots, x'_n(t)),$$

where x'_i are the ordinary derivatives of x_i ($i = 1, 2, \dots, n$).

The mapping sending \mathbf{x} to \mathbf{x}' is a linear mapping from the space of all differentiable vector-valued functions on I to the space of all vector-valued functions on I . In fact,

$$\frac{d}{dt}(a\mathbf{x} + b\mathbf{y}) = a\frac{d}{dt}\mathbf{x} + b\frac{d}{dt}\mathbf{y}$$

holds for differentiable functions \mathbf{x}, \mathbf{y} on I and $a, b \in \mathbb{K}$. We call this mapping the *differential operator* and denote it by $\frac{d}{dt}$. Further, $\frac{d}{dt}$ commutes with an $m \times n$ real matrix A , that is,

$$\frac{d}{dt}A\mathbf{x} = A\frac{d}{dt}\mathbf{x}$$

holds for any differentiable vector-valued function \mathbf{x} ¹ on I . In particular, for any vector $\mathbf{a} \in \mathbb{K}^n$, we have

$$\frac{d}{dt}\langle \mathbf{a} | \mathbf{x}(t) \rangle = \langle \mathbf{a} | \mathbf{x}'(t) \rangle \quad \text{and} \quad \frac{d}{dt}\langle \mathbf{x}(t) | \mathbf{a} \rangle = \langle \mathbf{x}'(t) | \mathbf{a} \rangle.$$

Also, for a matrix-valued function

$$\mathbf{X}(t) = \begin{bmatrix} x_{11}(t) & x_{12}(t) & \cdots & x_{1n}(t) \\ x_{21}(t) & x_{22}(t) & \cdots & x_{2n}(t) \\ \vdots & \vdots & & \vdots \\ x_{m1}(t) & x_{m2}(t) & \cdots & x_{mn}(t) \end{bmatrix}$$

on I , we can define the differential through the limit in terms of the matrix norm in the same way as above, and we have

$$\frac{d}{dt}\mathbf{X}(t) = \mathbf{X}'(t) = \begin{bmatrix} x'_{11}(t) & x'_{12}(t) & \cdots & x'_{1n}(t) \\ x'_{21}(t) & x'_{22}(t) & \cdots & x'_{2n}(t) \\ \vdots & \vdots & & \vdots \\ x'_{m1}(t) & x'_{m2}(t) & \cdots & x'_{mn}(t) \end{bmatrix}.$$

For differentiable matrix-valued functions, $\mathbf{X}(t)$ and $\mathbf{Y}(t)$ of shapes (m, n) and (n, l) , respectively, we have the product formula of differential:

¹ Function \mathbf{x} is considered to be a column vector.

$$\frac{d}{dt}(\mathbf{X}(t) \mathbf{Y}(t)) = \mathbf{X}'(t) \mathbf{Y}(t) + \mathbf{X}(t) \mathbf{Y}'(t).$$

As a special case, we have

$$\frac{d}{dt} \langle \mathbf{x}(t) | \mathbf{y}(t) \rangle = \langle \mathbf{x}'(t) | \mathbf{y}(t) \rangle + \langle \mathbf{x}(t) | \mathbf{y}'(t) \rangle$$

for differentiable vector-valued functions \mathbf{x} and \mathbf{y} .

9.2 Newton's Equation of Motion

Newton's equation of motion is expressed as "Force = Mass \times Acceleration." Here, we consider a motion of an object attached to one end of a spring (*a simple harmonic motion*). If the spring is compressed, the force is generated in the direction of extension, and if it is extended, the force is generated in the direction of compression. Setting the equilibrium position as the origin, the object with mass m located in $x(t) \in \mathbb{R}$ at time t gains the force $-kx(t)$ (k is the spring constant). The velocity $v(t)$ at time t is given by the derivative $x'(t)$ of $x(t)$, and the acceleration is the derivative of $v(t)$, so the equation of motion becomes

$$-kx(t) = mv'(t).$$

For simplicity, let $k = m = 1$. Then we have the simultaneous differential equations:

$$\begin{cases} x'(t) = v(t) \\ v'(t) = -x(t). \end{cases}$$

First, let us solve it numerically with the help of a computer. We will utilize a method called the *Euler method*, which gives numerical solutions of differential equations.² By definition,

$$\begin{cases} x'(t) = \lim_{\Delta t \rightarrow 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} \\ v'(t) = \lim_{\Delta t \rightarrow 0} \frac{v(t + \Delta t) - v(t)}{\Delta t}. \end{cases}$$

If Δt is sufficiently small, we can consider that the equalities hold approximately even without $\lim_{\Delta t \rightarrow 0}$ in the expressions on the right-hand sides. So, by canceling the denominators, we have

$$\begin{cases} x(t + \Delta t) \doteq x(t) + v(t) \Delta t \\ v(t + \Delta t) \doteq v(t) - x(t) \Delta t, \end{cases}$$

² There is also a method called the *Runge–Kutta method*, which gives solutions with better approximation accuracy than the Euler method. See the textbook on numerical analysis [15] in the Bibliography.



Fig. 9.1 Simulation of simple harmonic motion

which is considered to be the first-order approximation of the Taylor expansion. The position and the velocity at time 0 are given as

$$\begin{cases} x(0) = x_0 \\ v(0) = v_0, \end{cases}$$

which are *initial values* for the differential equation. The sequences $\{x_k\}$ and $\{v_k\}$ determined by the recurrence formula

$$\begin{cases} x_k = x_{k-1} + v_{k-1} \Delta t \\ v_k = v_{k-1} - x_{k-1} \Delta t \end{cases}$$

approximate $x(t)$ and $v(t)$ sampled at $t = 0, \Delta t, 2\Delta t, \dots$

Let us make a 3D-animation with VPython.

Program: newton.py

```
In [1]: 1 from vpython import *
2
3 Ball = sphere(color=color.red)
4 Wall = box(pos=vec(-10, 0, 0), length=1, width=10, height=10)
5 Spring = helix(pos=vec(-10, 0, 0), length=10)
6 dt, x, v = 0.01, 2.0, 0.0
7 while True:
8     rate(1 / dt)
9     dx, dv = v * dt, -x * dt
10    x, v = x + dx, v + dv
11    Ball.pos.x, Spring.length = x, 10 + x
```

Line 6: Set the value of `dt` representing Δt and give the initial values of x and v .

Lines 7–11: Infinite loop executing the animation. On Line 8, `rate(1 / dt)` adjusts the speed so that the loop is executed $1/\Delta t$ times per second.

If we look closely at the animation (Fig. 9.1), the amplitude gradually increases over time, which violates the law of conservation of energy in physics. Figure 9.2 (left, center to right) is made by plotting the points $(x(t), v(t))$ over time t with Δt from 0.1, 0.01 to 0.001. When the value of Δt approaches 0, the points trace near a circle.

If we use the second-order approximation

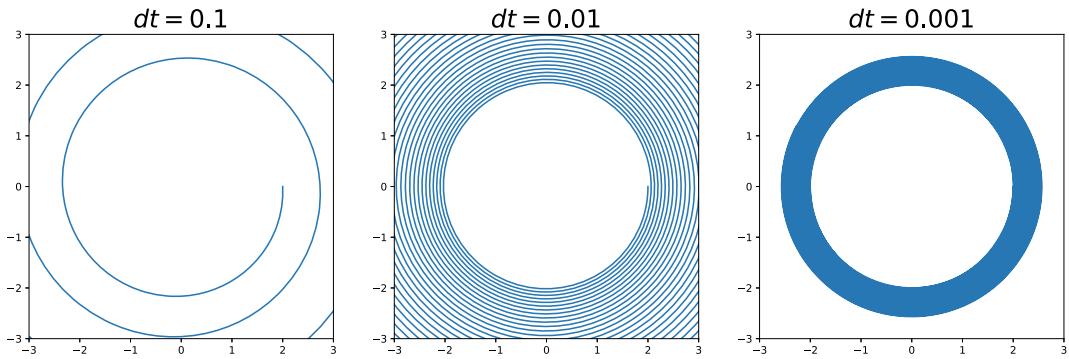


Fig. 9.2 Locus of position and velocity (first-order approximation)

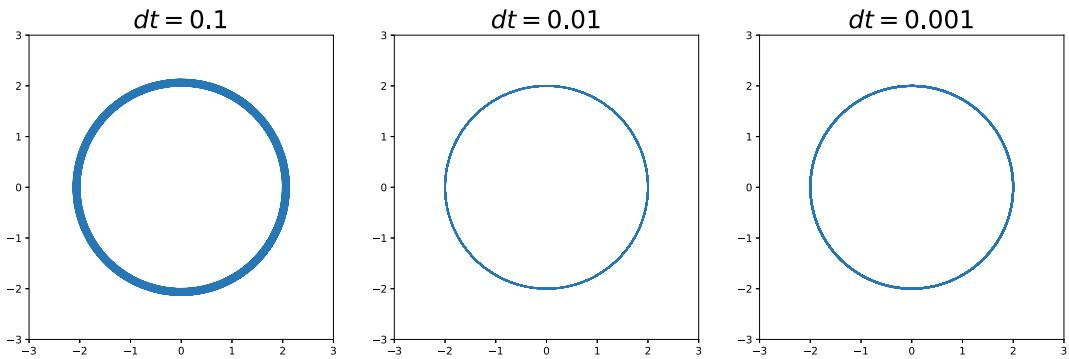


Fig. 9.3 Locus of position and velocity (second-order approximation)

$$\begin{aligned}x(t + \Delta t) &\doteq x(t) + \frac{x'(t)}{1!}\Delta t + \frac{x''(t)}{2!}\Delta t^2 \\&= x(t) + v(t)\Delta t - \frac{x(t)}{2}\Delta t^2, \\v(t + \Delta t) &\doteq v(t) + \frac{v'(t)}{1!}\Delta t + \frac{v''(t)}{2!}\Delta t^2 \\&= v(t) - x(t)\Delta t - \frac{v(t)}{2}\Delta t^2,\end{aligned}$$

we get Fig. 9.3 ($\Delta t = 0.1, 0.01, 0.001$ from left to right).

The following program draws Figs. 9.2 and 9.3.

Program: newton2.py

```
In [1]: 1 | from numpy import arange
2 | import matplotlib.pyplot as plt
3 |
4 | def taylor_1st(x, v, dt):
5 |     dx = v * dt
6 |     dv = -x * dt
7 |     return x + dx, v + dv
8 |
9 | def taylor_2nd(x, v, dt):
10 |     dx = v * dt - x / 2 * dt ** 2
11 |     dv = -x * dt - v / 2 * dt ** 2
12 |     return x + dx, v + dv
```

In [1]:

```

13 update = taylor_1st # taylor_2nd
14 dt = 0.1 # 0.01, 0.001
15 path = [(2.0, 0.0)]
16 for t in arange(0, 500, dt):
17     x, v = path[-1]
18     path.append(update(x, v, dt))
19 plt.axis('scaled'), plt.xlim(-3.0, 3.0), plt.ylim(-3.0, 3.0)
20 plt.plot(*zip(*path))
21 plt.show()
22

```

Lines 4–7: Update position x and velocity v with first-order approximation of the Taylor expansion.

Lines 9–12: Update x and v with second-order approximation of the Taylor expansion.

Line 14: Choose first-order approximation or second-order approximation for `update`.

Line 15: Set a value in `dt` representing Δt .

Line 16: Record the change of position and velocity in list `path`. $(2.0, 0.0)$ is a pair of the initial values of position and velocity.

Lines 17–19: Append updated (x, v) to `path` for t from 0 to below 500 in d increments.

Theoretically, the point $(x(t), v(t))$ draws an exact circle. Let us see this by solving the equation mathematically. Our equation is written as

$$\frac{d}{dt} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix} = A \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

with matrix $A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$. Diagonalizing A as

```

In [1]: from sympy import *
A = Matrix([[0, 1], [-1, 0]])
A.diagonalize()

```

```

Out[1]: (Matrix([
[I, -I],
[1, 1]]), Matrix([
[-I, 0],
[0, I]]))

```

we have

$$V = \begin{bmatrix} i & -i \\ 1 & 1 \end{bmatrix}, \quad A = V \begin{bmatrix} -i & 0 \\ 0 & i \end{bmatrix} V^{-1}.$$

Hence,

$$\frac{d}{dt} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix} = V \begin{bmatrix} -i & 0 \\ 0 & i \end{bmatrix} V^{-1} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}.$$

Operating V^{-1} from the left, we have

$$\frac{d}{dt} V^{-1} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} -i & 0 \\ 0 & i \end{bmatrix} V^{-1} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

because V^{-1} and $\frac{d}{dt}$ commute. Let

$$\begin{bmatrix} y(t) \\ w(t) \end{bmatrix} \stackrel{\text{def}}{=} V^{-1} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix},$$

then we obtain

$$\frac{d}{dt} \begin{bmatrix} y(t) \\ w(t) \end{bmatrix} = \begin{bmatrix} -i & 0 \\ 0 & i \end{bmatrix} \begin{bmatrix} y(t) \\ w(t) \end{bmatrix},$$

that is,

$$\begin{cases} \frac{d}{dt}y(t) = -iy(t) \\ \frac{d}{dt}w(t) = iw(t). \end{cases}$$

Thus, we have two independent differential equations, which can be solved as

$$\begin{cases} y(t) = C_1 e^{-it} \\ w(t) = C_2 e^{it}, \end{cases}$$

where C_1 and C_2 are integration constants. Because

$$V \begin{bmatrix} y(t) \\ w(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ v(t) \end{bmatrix},$$

we have

$$\begin{bmatrix} x(t) \\ v(t) \end{bmatrix} = V \begin{bmatrix} C_1 e^{-it} \\ C_2 e^{it} \end{bmatrix} = \begin{bmatrix} i & -i \\ 1 & 1 \end{bmatrix} \begin{bmatrix} C_1 e^{-it} \\ C_2 e^{it} \end{bmatrix} = \begin{bmatrix} C_1 i e^{-it} - C_2 i e^{it} \\ C_1 e^{-it} + C_2 e^{it} \end{bmatrix}.$$

Let us find the integration constants for the initial value $(x_0, v_0) = (2, 0)$. From

$$\begin{cases} x(0) = iC_1 - iC_2 = 2 \\ v(0) = C_1 + C_2 = 0, \end{cases}$$

we get

$$\begin{cases} C_1 = -i \\ C_2 = i. \end{cases}$$

Therefore, we obtain

$$\begin{cases} x(t) = e^{-it} + e^{it} = 2 \cos t \\ v(t) = -ie^{-it} + ie^{it} = -2 \sin t. \end{cases}$$

This is the equation of the circle of radius 2.

9.3 Linear Differential Equations

In the previous section, we considered a special differential equation, where the locus of the solution changes according to the change of the initial value. Let us look at this phenomenon with a little more general differential equation.

Let A be a square matrix of order n . We consider the differential equation

$$\mathbf{x}'(t) = A\mathbf{x}(t)$$

for a vector-valued function $\mathbf{x} : \mathbb{R} \rightarrow \mathbb{R}^n$. This can be expressed simply as $\mathbf{x}' = A\mathbf{x}$, and is called a *linear differential equation*. The differential equation in the previous section is given by letting $\mathbf{x} : \mathbb{R} \rightarrow \mathbb{R}^2$ and $A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$.

As with the usual exponentials, for the matrix exponentials, we have the formula

$$\frac{d}{dt} e^{tA} = Ae^{tA}.$$

In fact, we see

$$\begin{aligned} \left\| \frac{e^{(t+h)A} - e^{tA}}{h} - Ae^{tA} \right\| &\leq \|e^{tA}\| \left\| \frac{e^{hA} - I}{h} - A \right\| = \|e^{tA}\| \frac{1}{|h|} \left\| \sum_{k=2}^{\infty} \frac{(hA)^k}{k!} \right\| \\ &\leq \|e^{tA}\| \frac{1}{|h|} \sum_{k=2}^{\infty} \frac{|h|^k \|A\|^k}{k!} = \|e^{tA}\| \left(\frac{e^{|h|\|A\|} - 1}{|h|} - \|A\| \right) \rightarrow 0 \quad (h \rightarrow 0). \end{aligned}$$

Let $\mathbf{y}(t) \stackrel{\text{def}}{=} e^{-tA}\mathbf{x}(t)$, then by the product formula we have

$$\mathbf{y}'(t) = \left(\frac{d}{dt} e^{-tA} \right) \mathbf{x}(t) + e^{-tA} \frac{d}{dt} \mathbf{x}(t) = -Ae^{-tA}\mathbf{x}(t) + e^{-tA}A\mathbf{x}(t) = \mathbf{0}.$$

Hence, $\mathbf{y}(t)$ is a constant function; $\mathbf{y}(t) = \mathbf{C}$, and so $\mathbf{x}(t) = e^{tA}\mathbf{C}$. If the initial value of $\mathbf{x}(t)$ is \mathbf{x}_0 , then $\mathbf{x}_0 = \mathbf{x}(0) = e^{0A}\mathbf{C} = \mathbf{C}$. Thus, the solution of the differential equation is given by

$$\mathbf{x}(t) = e^{tA}\mathbf{x}_0.$$

For initial value $\mathbf{x}_0 \in \mathbb{R}^n$, the subset $\{e^{tA}\mathbf{x}_0 \mid t \in \mathbb{R}\}$ of \mathbb{R}^n is called the *trajectory* or *orbit* of solution of the differential equation $\mathbf{x}' = A\mathbf{x}$ in *phase space* \mathbb{R}^n . We shall observe the trajectory in \mathbb{R}^2 of solution of $\mathbf{x}' = A\mathbf{x}$ for a real matrix A of order 2. The characteristic equation of A which has real coefficients has two distinct real roots, a real double root or two complex roots conjugate with each other. Accordingly, the Jordan normal form of A falls in one of the following forms:

$$\mathbf{J}_1 = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} (\lambda_1, \lambda_2: \text{real}), \quad \mathbf{J}_2 = \begin{bmatrix} \lambda & 0 \\ 1 & \lambda \end{bmatrix} (\lambda: \text{real}), \quad \mathbf{J}_3 = \begin{bmatrix} \lambda & 0 \\ 0 & \bar{\lambda} \end{bmatrix} (\lambda: \text{complex}).$$

When A is equal or similar to one of these matrices, we generate 100 initial values $\mathbf{x}_0 = (x, y)$ with uniform random numbers $x, y \in [-1, 1]$.

(1) Case $A = \mathbf{J}_1$: We have

$$e^{tA} = \begin{bmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{bmatrix}.$$

The trajectories are shown in Fig. 9.4 for $(\lambda_1, \lambda_2) = (1, 0), (1, 1), (1, 2)$ and $(1, -1)$ from the top left to bottom right, respectively.

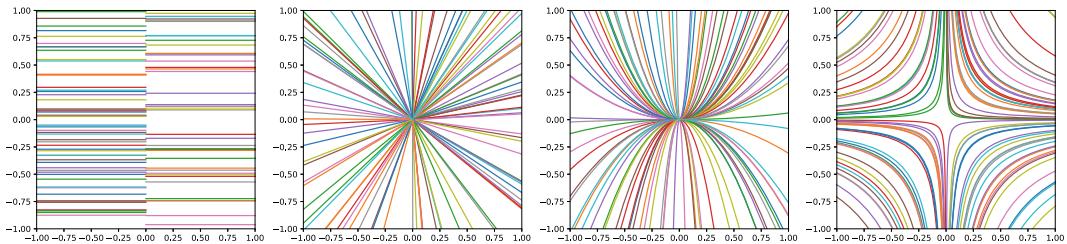
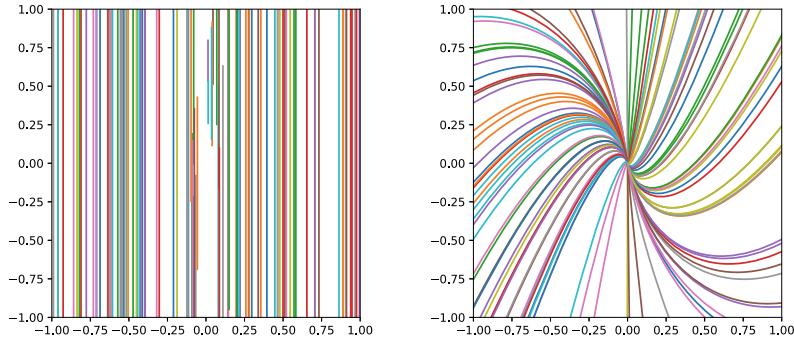


Fig. 9.4 Trajectories for A with two real eigenvalues

Fig. 9.5 Trajectories for A not diagonalizable and for A with complex eigenvalues



(2) Case $A = J_2$: For a Jordan cell $X = \begin{bmatrix} x & 0 \\ 1 & x \end{bmatrix}$ with $x \in \mathbb{R}$, we have $X^n = \begin{bmatrix} x^n & 0 \\ nx^{n-1} & x^n \end{bmatrix}$. So, we see

$$e^X = \sum_{n=0}^{\infty} \frac{X^n}{n!} = \sum_{n=0}^{\infty} \frac{1}{n!} \begin{bmatrix} x^n & 0 \\ nx^{n-1} & x^n \end{bmatrix} = \begin{bmatrix} \sum_{n=0}^{\infty} \frac{x^n}{n!} & 0 \\ \sum_{n=1}^{\infty} \frac{x^{(n-1)}}{(n-1)!} \sum_{n=0}^{\infty} \frac{x^n}{n!} & \end{bmatrix} = \begin{bmatrix} e^x & 0 \\ e^x & e^x \end{bmatrix}.$$

The Jordan normal form of tA is $\begin{bmatrix} t\lambda & 0 \\ 1 & t\lambda \end{bmatrix}$, for

$$\begin{bmatrix} t\lambda & 0 \\ t & t\lambda \end{bmatrix} = \frac{1}{t} \begin{bmatrix} 1 & 0 \\ 0 & t \end{bmatrix} \begin{bmatrix} t\lambda & 0 \\ 1 & t\lambda \end{bmatrix} \begin{bmatrix} t & 0 \\ 0 & 1 \end{bmatrix}.$$

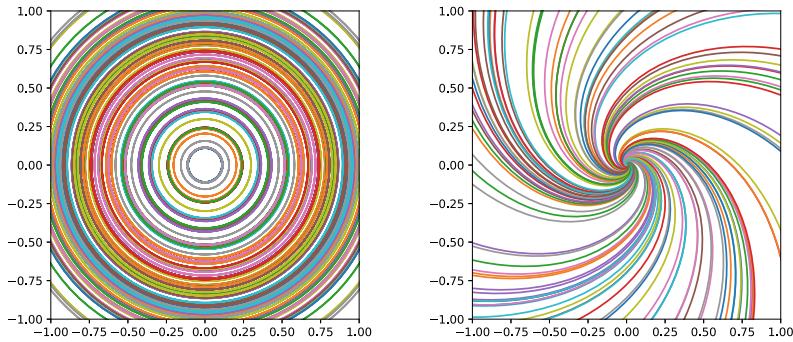
Hence, using the equality in Exercise 7.13, we have

$$e^{tA} = \frac{1}{t} \begin{bmatrix} 1 & 0 \\ 0 & t \end{bmatrix} \begin{bmatrix} e^{t\lambda} & 0 \\ e^{t\lambda} & e^{t\lambda} \end{bmatrix} \begin{bmatrix} t & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} e^{t\lambda} & 0 \\ te^{t\lambda} & e^{t\lambda} \end{bmatrix}.$$

Figure 9.5 shows the trajectories for $\lambda = 0$ (left) and for $\lambda = 1$ (right).

(3) Case where A is similar to J_3 : Let $A = \begin{bmatrix} a & b \\ -b & a \end{bmatrix}$ ($a, b \in \mathbb{R}, b \neq 0$). Then, $A = V^{-1}J_3V$ with $V = \begin{bmatrix} 1 & -i \\ 1 & i \end{bmatrix}$, where $\lambda = a + bi$. We have

Fig. 9.6 Trajectories for A diagonalizable and for A with complex eigenvalues



$$\begin{aligned} e^{tA} &= V^{-1} e^{tJ_3} V = V^{-1} \begin{bmatrix} e^{t\lambda} & 0 \\ 0 & e^{t\bar{\lambda}} \end{bmatrix} V \\ &= e^{at} V^{-1} \begin{bmatrix} e^{ibt} & 0 \\ 0 & e^{-ibt} \end{bmatrix} V = e^{at} \begin{bmatrix} \cos(bt) & \sin(bt) \\ -\sin(bt) & \cos(bt) \end{bmatrix}. \end{aligned}$$

Figure 9.6 shows the trajectories for $a = 0, b = 1$ (left) and for $a = b = 1$ (right).

The next program draws the trajectories in the phase space.

Program: phasesp.py

```

1  from numpy import array, arange, exp, sin, cos
2  from numpy.random import uniform
3  import matplotlib.pyplot as plt
4
5  def B1(lmd1, lmd2):
6      return lambda t: array([[exp(lmd1 * t), 0],
7                             [0, exp(lmd2 * t)]])
8
9  def B2(lmd):
10     return lambda t: exp(lmd * t) * array([[1, 0], [t, 1]])
11
12 def B3(a, b):
13     return lambda t: exp(a * t) * array([
14         [cos(b * t), sin(b * t)], [-sin(b * t), cos(b * t)]])
15
16 B = B1(1, -1) # B1(lmd1, lmd2), B2(lmd), B3(a, b)
17 V = uniform(-1, 1, (100, 2))
18 T = arange(-10, 10, 0.01)
19 plt.axis('scaled'), plt.xlim(-1, 1), plt.ylim(-1, 1)
20 [plt.plot(*zip(*[B(t).dot(v) for t in T])) for v in V]
21 plt.show()

```

Lines 5–7: Define the function with argument λ_1, λ_2 returning the matrix-valued function $t \mapsto e^{tJ_1}$.

Lines 9, 10: Define the function with argument λ returning the matrix-valued function $t \mapsto e^{tJ_2}$.

Lines 12–14: Define the function with argument a, b returning the matrix-valued function $t \mapsto e^{tA}$.

Line 16: Choose matrix $B(t)$.

Line 17: Choose 100 points in $[-1, 1] \times [-1, 1]$ randomly.

Line 18: T is the list of the terms less than 10 of the arithmetic progression with common difference 0.01 starting with -10 .

Line 20: Plot the trajectory of $B(t)v$ with t moving in T for each $v \in V$.

As stated in Sect. 7.4, SymPy calculates the exponential of a matrix as we wish, but NumPy does not because it executes componentwise calculation if we apply a function to a matrix. The power of

the `matrix` class in NumPy is calculated with the matrix product, while the power of an array is componentwise.

```
In [1]: import sympy as sp
import numpy as np
A = [[1, 2], [2, 1]]
sp.exp(sp.Matrix(A))

Out[1]: Matrix([
 [exp(-1)/2 + exp(3)/2, -exp(-1)/2 + exp(3)/2],
 [-exp(-1)/2 + exp(3)/2, exp(-1)/2 + exp(3)/2]])

In [2]: sp.exp(sp.Matrix(A)).evalf()

Out[2]: Matrix([
 [10.2267081821796, 9.85882874100811],
 [9.85882874100811, 10.2267081821796]])

In [3]: np.exp(np.matrix(A))

Out[3]: matrix([[2.71828183, 7.3890561 ],
 [7.3890561 , 2.71828183]])

In [4]: sp.Matrix(A)**2

Out[4]: Matrix([
 [5, 4],
 [4, 5]])

In [5]: np.matrix(A)**2

Out[5]: matrix([[5, 4],
 [4, 5]])
```

Exercise 9.1 Let $A = \begin{bmatrix} 1 & -2 \\ 3 & 1 \end{bmatrix}$. Solve the differential equation $x'(t) = A x(t)$ and draw the trajectories of the solutions for several initial values.

Let us consider a movement of an object in $V = \mathbb{R}^2$ with mass 1 attached to the ends of two springs S_1 and S_2 whose other ends are fixed. Suppose that S_1 is fixed at $p = (0, 0)$ and S_2 is fixed at $q = (2, 0)$. The object located in $x(t) = (x_1(t), x_2(t)) \in V$ at time t gains the forces $-x(t)$ from S_1 and $q - x(t)$ from S_2 assuming both of the spring constants are 1. Hence, in total, it gains force $q - 2x(t)$ and we have

$$\begin{cases} x'(t) = v(t) \\ v'(t) = q - 2x(t). \end{cases}$$

Let $y(t) \stackrel{\text{def}}{=} q - 2x(t)$, then

$$\begin{cases} y'(t) = -2x'(t) = -2v(t) \\ v'(t) = y(t). \end{cases}$$

This is expressed as

$$\begin{bmatrix} \mathbf{y}'(t) \\ \mathbf{v}'(t) \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathbf{y}(t) \\ \mathbf{v}(t) \end{bmatrix} \quad \text{with } \mathbf{A} = \begin{bmatrix} 0 & -2 \\ 1 & 0 \end{bmatrix},$$

which is a linear differential equation of dimension 4 because $\mathbf{y}(t)$ and $\mathbf{v}(t)$ are two-dimensional vectors. \mathbf{A} has two eigenvalues $i\sqrt{2}$ and $-i\sqrt{2}$ with corresponding eigenvectors $(i\sqrt{2}, 1)$ and $(-i\sqrt{2}, 1)$ respectively. So, \mathbf{A} is diagonalized as

$$\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \text{diag}(i\sqrt{2}, -i\sqrt{2})$$

with transformation matrix

$$\mathbf{V} = \begin{bmatrix} i\sqrt{2} & -i\sqrt{2} \\ 1 & 1 \end{bmatrix}.$$

Let

$$\begin{bmatrix} \mathbf{z}(t) \\ \mathbf{w}(t) \end{bmatrix} = \mathbf{V}^{-1} \begin{bmatrix} \mathbf{y}(t) \\ \mathbf{v}(t) \end{bmatrix} = \frac{1}{2i\sqrt{2}} \begin{bmatrix} 1 & i\sqrt{2} \\ -1 & i\sqrt{2} \end{bmatrix} \begin{bmatrix} \mathbf{y}(t) \\ \mathbf{v}(t) \end{bmatrix} = \frac{1}{2i\sqrt{2}} \begin{bmatrix} \mathbf{y}(t) + i\sqrt{2}\mathbf{v}(t) \\ -\mathbf{y}(t) + i\sqrt{2}\mathbf{v}(t) \end{bmatrix},$$

then

$$\begin{bmatrix} \mathbf{z}'(t) \\ \mathbf{w}'(t) \end{bmatrix} = \mathbf{V}^{-1} \begin{bmatrix} \mathbf{y}'(t) \\ \mathbf{v}'(t) \end{bmatrix} = \mathbf{V}^{-1} \mathbf{A} \mathbf{V} \mathbf{V}^{-1} \begin{bmatrix} \mathbf{y}(t) \\ \mathbf{v}(t) \end{bmatrix} = \begin{bmatrix} i\sqrt{2} & 0 \\ 0 & -i\sqrt{2} \end{bmatrix} \begin{bmatrix} \mathbf{z}(t) \\ \mathbf{w}(t) \end{bmatrix} = \begin{bmatrix} i\sqrt{2}\mathbf{z}(t) \\ -i\sqrt{2}\mathbf{w}(t) \end{bmatrix}.$$

This is solved as

$$\begin{cases} \mathbf{z}(t) = e^{i\sqrt{2}t} \mathbf{z}(0) \\ \mathbf{w}(t) = e^{-i\sqrt{2}t} \mathbf{w}(0). \end{cases}$$

Hence,

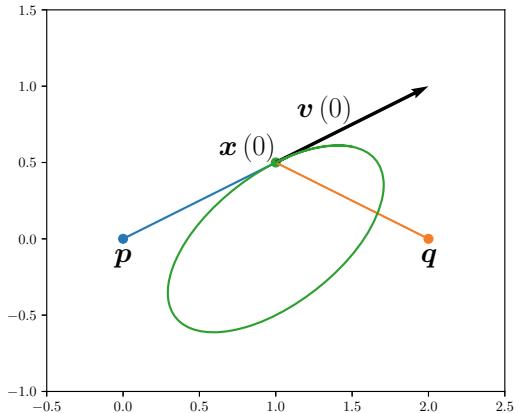
$$\begin{bmatrix} \mathbf{y}(t) \\ \mathbf{v}(t) \end{bmatrix} = \mathbf{V} \begin{bmatrix} \mathbf{z}(t) \\ \mathbf{w}(t) \end{bmatrix} = \begin{bmatrix} i\sqrt{2} (e^{i\sqrt{2}t} \mathbf{z}(0) - e^{-i\sqrt{2}t} \mathbf{w}(0)) \\ e^{i\sqrt{2}t} \mathbf{z}(0) + e^{-i\sqrt{2}t} \mathbf{w}(0) \end{bmatrix}.$$

Choose, for example, initial values $\mathbf{x}(0) = \left(1, \frac{1}{2}\right)$ and $\mathbf{v}(0) = \left(1, \frac{1}{2}\right)$, then we have

$$\begin{cases} \mathbf{y}(0) = \mathbf{q} - 2\mathbf{x}(0) = (0, -1), \\ \mathbf{z}(0) = \frac{1}{2i\sqrt{2}} (\mathbf{y}(0) + i\sqrt{2}\mathbf{v}(0)) = \frac{1}{4} (2, 1 + i\sqrt{2}), \\ \mathbf{w}(0) = \frac{1}{2i\sqrt{2}} (-\mathbf{y}(0) + i\sqrt{2}\mathbf{v}(0)) = \frac{1}{4} (2, 1 - i\sqrt{2}). \end{cases}$$

Thus, we obtain

Fig. 9.7 Trajectory of an object attached to two springs



$$\begin{aligned} \mathbf{x}(t) = \frac{\mathbf{q} - \mathbf{y}(t)}{2} &= \left(1 + \frac{i e^{-i\sqrt{2}t} - i e^{i\sqrt{2}t}}{2\sqrt{2}}, \frac{(i + \sqrt{2}) e^{-i\sqrt{2}t} + (-i + \sqrt{2}) e^{i\sqrt{2}t}}{4\sqrt{2}} \right) \\ &= \left(1 + \frac{\sin \sqrt{2}t}{\sqrt{2}}, \frac{\sqrt{2} \cos \sqrt{2}t + \sin \sqrt{2}t}{2\sqrt{2}} \right). \end{aligned}$$

The last equality is derived using Euler's formula (Sect. 1.2). The trajectory of $\mathbf{x}(t)$ forms an ellipse (Fig. 9.7).

Exercise 9.2

- (1) Solve the same problem above by Euler's method and compare the approximate solution with the exact solution above.
- (2) Investigate the movement of an object in \mathbb{R}^3 attached to three or more springs.

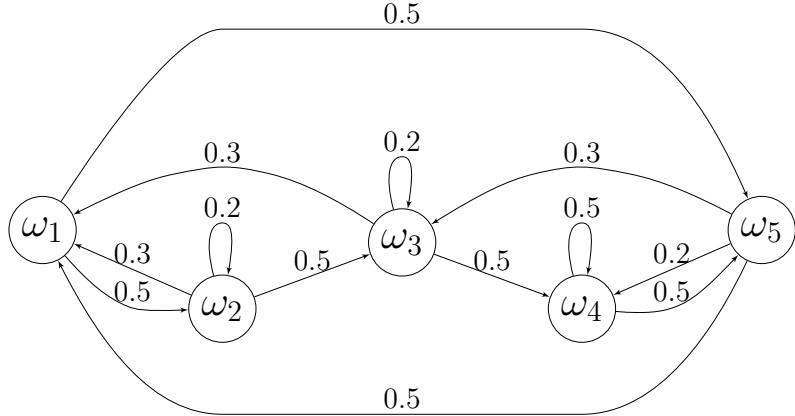
9.4 Stationary States of Markov Chain

The motion of celestial bodies can be accurately known for both the past and the future, because it is governed by differential equations. On the other hand, the course of a typhoon can only be predicted inaccurately. We can only know stochastically where it likely will be. Let us consider the simplest model for a system dominated by such probabilities. Let $\Omega = \{\omega_1, \omega_2, \dots, \omega_n\}$ be a finite set whose elements are called *states*. We consider a system in which state ω_i changes stochastically to $\omega_{i_0}, \omega_{i_1}, \omega_{i_2}, \dots$ with time $t = 0, 1, 2, \dots$. The state ω_j at time t changes to ω_i at the next time $t+1$ with the probability $P(\omega_i | \omega_j)$ determined only by the current state ω_j and the next state ω_i , regardless of time t and regardless of what state changes were followed before t . Here, P is a *state transition probability* satisfying the following conditions:

- M1. $0 \leq P(\omega_i | \omega_j) \leq 1$ ($i, j = 1, 2, \dots, n$),
- M2. $\sum_{i=1}^n P(\omega_i | \omega_j) = 1$ ($j = 1, 2, \dots, n$).

We call a stochastic system like this a *stationary Markov chain*.

Fig. 9.8 State transition diagram



The diagram in Fig. 9.8 is an example of a *state transition diagram*. If $P(\omega_i \mid \omega_j) > 0$, then ω_j is connected to ω_i with an arrow to which the value $P(\omega_i \mid \omega_j)$ is given. We call each circled ω_i a *node* (or a *vertex*) and the arrow an *edge* (or a *branch*). In a state transition diagram, the sum of the values of the arrows coming out of each node is 1, while the sum of the values of the incoming arrows is not always 1.

A function $x : \Omega \rightarrow [0, 1]$ satisfying

$$\sum_{i=1}^n x(\omega_i) = 1$$

is called a *probability distribution* on Ω . In particular, for $i = 1, 2, \dots, n$ the function $x_i : \Omega \rightarrow [0, 1]$ defined by

$$x_i(\omega_j) = \delta_{ij} \quad (j = 1, 2, \dots, n)$$

is a probability distribution on Ω . This x_i corresponds to the state $\omega_i \in \Omega$, that is, we consider that $\omega_i \in \Omega$ is the state x_i in ω_i at probability 1. From now on, we call a probability distribution on Ω a state. For a state x , define

$$(Px)(\omega_i) \stackrel{\text{def}}{=} \sum_{j=1}^n P(\omega_i \mid \omega_j) x(\omega_j) \quad (i = 1, 2, \dots, n),$$

then Px is also a state, because

$$\begin{aligned} \sum_{i=1}^n (Px)(\omega_i) &= \sum_{i=1}^n \left(\sum_{j=1}^n P(\omega_i \mid \omega_j) x(\omega_j) \right) = \sum_{j=1}^n \left(\sum_{i=1}^n P(\omega_i \mid \omega_j) \right) x(\omega_j) \\ &= \sum_{j=1}^n x(\omega_j) = 1. \end{aligned}$$

For two state transition probabilities P and Q , define

$$(PQ)(\omega_i \mid \omega_j) \stackrel{\text{def}}{=} \sum_{k=1}^n P(\omega_i \mid \omega_k) Q(\omega_k \mid \omega_j) \quad (i, j = 1, 2, \dots, n),$$

then PQ is a state transition probability. A state x and a state transition probability P can be expressed with an n -dimensional vector and a square matrix as

$$\mathbf{x} = \begin{bmatrix} x(\omega_1) \\ x(\omega_2) \\ \vdots \\ x(\omega_n) \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} P(\omega_1 \mid \omega_1) & P(\omega_1 \mid \omega_2) & \cdots & P(\omega_1 \mid \omega_n) \\ P(\omega_2 \mid \omega_1) & P(\omega_2 \mid \omega_2) & \cdots & P(\omega_2 \mid \omega_n) \\ \vdots & \vdots & \ddots & \vdots \\ P(\omega_n \mid \omega_1) & P(\omega_n \mid \omega_2) & \cdots & P(\omega_n \mid \omega_n) \end{bmatrix},$$

respectively. We call \mathbf{x} a *state (or probability) vector* and \mathbf{P} a *state transition matrix*. In general, a nonnegative matrix in which the sum of elements of each column is 1 is called a *stochastic matrix*. A state transition matrix is a stochastic matrix. Through the above matrix expressions, state transition probability PQ corresponds to the matrix product $\mathbf{P}\mathbf{Q}$ and the state Px corresponds to \mathbf{Px} .

For a state transition matrix \mathbf{P} and for $k \in \mathbb{N}$, the power \mathbf{P}^k is a state transition matrix. \mathbf{P} is called *irreducible* if for any $i, j = 1, 2, \dots, n$, there exists $k_{ij} \in \mathbb{N}$ such that the (i, j) -element of $\mathbf{P}^{k_{ij}}$ is positive. Also, \mathbf{P} is said to be *aperiodic*, if for any $i = 1, 2, \dots, n$, there exists k_i such that the (i, i) -component of \mathbf{P}^{k_i} is positive and the greatest common divisor of k_1, k_2, \dots, k_n is 1. In terms of a state transition diagram, it is irreducible if there is a path³ between any two (or the same) nodes. It is aperiodic if there is a cycle⁴ for each node and the greatest common divisor of the lengths of the cycles is 1.

A nonempty subset X of \mathbb{N} is called a *numerical semigroup* if it is closed under addition, that is, $x, y \in X \Rightarrow x + y \in X$.

Lemma 9.1 Let X be a numerical semigroup. If the greatest common divisor of X is 1, then all sufficiently large natural numbers are elements of X .

Proof Let $X = \{x_1, x_2, \dots\}$, where $x_1 < x_2 < \dots$ and set

$$k \stackrel{\text{def}}{=} \min \{x_{i+1} - x_i \mid i = 1, 2, \dots\}.$$

Take $x \in X$ such that $x + k \in X$. Assume that $k > 1$, then, because k is not a common divisor of X , there is $y \in X$ that is not a multiple of k . We can write $y = mk + n$ with $m, n \in \mathbb{Z}$ such that $m \geq 0$ and $0 < n < k$. Because X is closed under addition, we see $z = (m+1)(x+k)$ and $w = (m+1)x + y$ are contained in X . However,

$$0 < z - w = (m+1)k - y = k - n < k.$$

This contradicts the minimality of k , and we find that $k = 1$. Hence, $x, x+1 \in X$. Because X is closed under addition, we see $2x, 2x+1, 2x+2 \in X$. Similarly, we see $3x, 3x+1, 3x+2, 3x+3 \in X$, and inductively, we have

$$xx, xx+1, xx+2, \dots, xx+x \in X.$$

³Consecutive edges from the start node to the end node.

⁴A path coming back to the start node.

Further, adding $x, 2x, 3x, \dots \in X$ to these consecutive numbers in X , we can conclude that all numbers greater than or equal to xx belongs to X . ■

Exercise 9.3 Let X be a numerical semigroup containing p and q whose greatest common divisor is 1. Show that all numbers greater than or equal to $(p-1)(q-1)$ belong to X .⁵ (Hint: prove that the remainders of σq for $\sigma = 0, 1, \dots, p-1$ divided by p are all different, and for any $m \geq (p-1)(q-1)$, $m - \sigma q$ is a multiple of p for some σ .)

If \mathbf{P} is a positive matrix, that is, every element of \mathbf{P} is positive, then it is irreducible and aperiodic. Conversely, if \mathbf{P} is irreducible and aperiodic, \mathbf{P}^m becomes a positive matrix for a sufficiently large m .

Exercise 9.4 Prove the above results. (Hint: use Lemma 9.1.)

For a state transition matrix \mathbf{P} , a state vector \mathbf{x} such that $\mathbf{P}\mathbf{x} = \mathbf{x}$ is called a *stationary state* of \mathbf{P} .

Theorem 9.1 An irreducible aperiodic state transition matrix \mathbf{P} has a stationary state \mathbf{x}_1 . Moreover, for any state vector \mathbf{x} , it holds that

$$\mathbf{P}^m \mathbf{x} \rightarrow \mathbf{x}_1 \quad (m \rightarrow \infty).$$

Proof Let $\lambda_1, \lambda_2, \dots, \lambda_n$ be the eigenvalues of \mathbf{P} including multiplicity. Take m such that $\mathbf{Q} = \mathbf{P}^m$ is positive. Then, $\lambda_1^m, \lambda_2^m, \dots, \lambda_n^m$ are the eigenvalues of \mathbf{Q} and $\rho(\mathbf{Q}) = \rho(\mathbf{P})^m$ holds. We may suppose $\lambda_1^m = \rho(\mathbf{P})^m$. Let \mathbf{x}_1 be an eigenvector of \mathbf{P} corresponding to λ_1 . Then, \mathbf{x}_1 is also an eigenvector of \mathbf{Q} corresponding to λ_1^m . By the Perron–Frobenius theorem, the eigenspace for this eigenvalue is one-dimensional and contains a positive vector, so by taking a scalar multiple of \mathbf{x}_1 if necessary, we may suppose that \mathbf{x}_1 is a state vector. Then, since $\lambda_1 \mathbf{x}_1 = \mathbf{P} \mathbf{x}_1$ is also a state vector, we see $\lambda_1 = 1$. Hence, \mathbf{x}_1 a stationary state of \mathbf{P} . Moreover, we have $\rho(\mathbf{P}) = \lambda_1 = 1$, and so by the Perron–Frobenius theorem, the absolute values of all other eigenvalues of \mathbf{Q} than $\rho(\mathbf{Q})$ are less than $\rho(\mathbf{Q})$, that is, $|\lambda_i|^m = |\lambda_i^m| < \rho(\mathbf{P})^m = 1$ for $i \geq 2$. It follows that $|\lambda_i| < 1$ for $i \geq 2$.

Now, let $\mathbf{J} = \mathbf{V}^{-1} \mathbf{P} \mathbf{V}$ be a Jordan normal form of \mathbf{P} . Then, the Jordan cell corresponding to the eigenvalue 1 is a matrix of shape (1,1) and the absolute values of all other eigenvalues are less than 1. Hence,

$$\lim_{m \rightarrow \infty} \mathbf{J}^m = \mathbf{J}_\infty = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix},$$

and thus, $\mathbf{P}^m = \mathbf{V} \mathbf{J}^m \mathbf{V}^{-1}$ converges to $\mathbf{P}_\infty = \mathbf{V} \mathbf{J}_\infty \mathbf{V}^{-1}$ as $m \rightarrow \infty$. We have

$$\mathbf{P} \mathbf{P}_\infty = \mathbf{P} \left(\lim_{m \rightarrow \infty} \mathbf{P}^m \right) = \lim_{m \rightarrow \infty} \mathbf{P}^{m+1} = \mathbf{P}_\infty,$$

⁵ The number $f(p, q) = pq - p - q = (p-1)(q-1) - 1$ is called the *Frobenius number* of X . Actually, $f(p, q)$ is the largest number not contained in X .

and hence, for any state vector \mathbf{x} , we have $\mathbf{P}_\infty \mathbf{x} = \mathbf{P}(\mathbf{P}_\infty \mathbf{x})$. So, $\mathbf{P}_\infty \mathbf{x}$ is a state vector invariant under \mathbf{P} . Since the eigenspace for the eigenvalue 1 of \mathbf{P} is one-dimensional, \mathbf{x}_1 is the only state vector in it, and consequently, we have

$$\lim_{m \rightarrow \infty} \mathbf{P}^m \mathbf{x} = \mathbf{P}_\infty \mathbf{x} = \mathbf{x}_1. \blacksquare$$

We can think of the state transition diagram of an irreducible and aperiodic stationary Markov chain as a sugoroku game.⁶ First place a piece on an arbitrary node, then throw the dice according to the state transition probability and change the location of the piece according to the value of the dice, and repeat this play. Theorem 9.1 is called the *ergodic theorem* of stationary Markov chains. When we play the game sufficiently many times, the ratio (*time average*) of the frequency of the node where the piece was placed to the number of times we played is close to the probability (*phase average*) in the stationary state of the Markov chain. It is good to simulate a system over a long time to achieve many independent realizations of the same system. In physics, this law is called the *ergodic hypothesis*.

9.5 Markov Random Fields

In a Markov chain, the probability that an event occurs next depends only on the current state. A *Markov random field* is an analogous stochastic model in which the probability state of a point on a plane (or space) is determined by the state of points adjacent to it, namely, it satisfies a *spatial Markov property*. There are two reasons to take up Markov random fields here. One is that a Markov random field is an interesting example of visualizing the stationary state of a Markov chain. Another reason is that the state transition probability of a Markov random field is expressed by a very large matrix but most of the matrix components are 0 due to the spatial Markov property, and we actually solve the problem without large matrix calculations.

Let X be a nonempty finite set. We call X a *screen* and an element of X a *pixel*. We consider a subset A of X as a binary (black or white) *image* on the screen, where $x \in A$ is colored black and $x \notin A$ is colored white for every $x \in X$. Figure 9.9 shows all the images on $X = \{x_1, x_2, x_3\}$.

We consider a stochastic model where the power set $\Omega = 2^X$ of X is a set of states. We suppose that every image $A \in \Omega$ appears with positive probability $p(A)$, that is, p is a probability on Ω such that $\sum_{A \in \Omega} p(A) = 1$ and $p(A) > 0$ for every $A \in \Omega$.

A function $f : (0, \infty) \rightarrow (0, 1]$ satisfying

$$f\left(\frac{1}{t}\right) = \frac{1}{t} f(t)$$

is called an *acceptance function*. As examples of acceptance functions (Fig. 9.10), we have

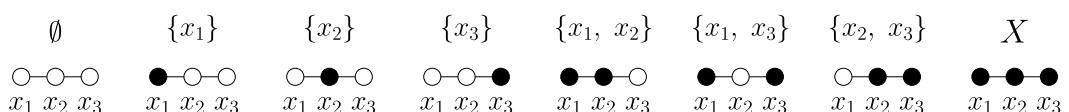
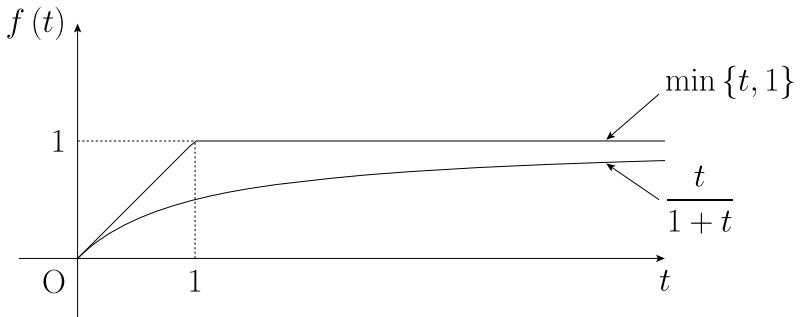


Fig. 9.9 All images on the screen with 3 pixels

⁶ A Japanese traditional board game played with dice similar to backgammon.

Fig. 9.10 Examples of acceptance functions



$$f(t) = \frac{t}{1+t} \quad \text{and} \quad f(t) = \min\{t, 1\}.$$

Let us use a computer to simulate such a mode for a given positive probability p . We choose an acceptance function f . For every image $A \subseteq X$, we denote by A_x the image whose color at $x \in X$ is inverted, that is, if $x \in A$, then $A_x = A \setminus \{x\}$, otherwise $A_x = A \cup \{x\}$. For a given pixel x , the state transition probability P_x from image A to image B is defined by

$$P_x(B | A) \stackrel{\text{def}}{=} \begin{cases} f\left(\frac{p(A_x)}{p(A)}\right) & \text{if } B = A_x, \\ 1 - f\left(\frac{p(A_x)}{p(A)}\right) & \text{if } B = A, \\ 0 & \text{otherwise.} \end{cases}$$

We call P_x the *inversion probability* of the color of pixel x . Since

$$\begin{aligned} (P_x p)(A) &= \sum_{B \in \Omega} P_x(A | B) p(B) \\ &= P_x(A | A_x) p(A_x) + P_x(A | A) p(A) \\ &= f\left(\frac{p(A)}{p(A_x)}\right) p(A_x) + \left(1 - f\left(\frac{p(A_x)}{p(A)}\right)\right) p(A) \\ &= \frac{p(A)}{p(A_x)} f\left(\frac{p(A_x)}{p(A)}\right) p(A_x) + \left(1 - f\left(\frac{p(A_x)}{p(A)}\right)\right) p(A) \\ &= p(A), \end{aligned}$$

p is a stationary state of the state transition probability P_x . This means that if an image B , which occurred in probability $p(B)$, transits to an image A with the state transition probability $P_x(A | B)$, then A can be also regarded as an image occurred in probability $p(A)$. Here, the stationary state p of transition probability P_x does not depend on $x \in X$.

Let $X = \{x_1, x_2, \dots, x_n\}$ be a screen. We propose here two methods of getting images with stationary state p . The first method scans the pixels of X in order and inverts the color of x_i with inversion probability P_{x_i} . We need to assume that the acceptance function satisfies $f < 1$. One scanning of the full screen causes a transition according to the state transition probability Q corresponding to the stochastic matrix Q defined by the matrix product

$$\mathbf{Q} \stackrel{\text{def}}{=} \mathbf{P}_{x_1} \mathbf{P}_{x_2} \cdots \mathbf{P}_{x_n},$$

where \mathbf{P}_{x_1} , \mathbf{P}_{x_2} , ..., \mathbf{P}_{x_n} are the state transition matrices corresponding to the state transition probabilities P_{x_1} , P_{x_2} , ..., P_{x_n} , respectively. It is possible to transit from any image A to any image B (including A itself) with a positive probability in one scan, by inverting the color for the pixels $x_i \in A \Delta B$ ⁷ and preserving the color for the pixels $x_i \in A \cap B$ or $x_i \in (A \cup B)^C$. Namely, we have $\mathbf{Q} > \mathbf{0}$. For this, we need the condition $f < 1$. In fact, if we adopt the acceptance function $f(x) = \min\{1, x\}$, for example, and consider the system for $X = \{x_1, x_2\}$ in which we assign each of four images on X the same probability of $\frac{1}{4}$. Then $P_x(A_x | A) = 1$ for any pixel $x = x_1, x_2$ and for any image $A = \emptyset, \{x_1\}, \{x_2\}, X$, and A transits to A^C with probability 1 by state transition probability \mathbf{Q} and the probabilities of other transitions are 0, and $\mathbf{Q} > \mathbf{0}$ fails. Let us give a different method effective for such an acceptance function. In this second method, we scan the screen randomly rather than in order. We select one of n pixels on the screen or select nothing in probability $\frac{1}{n+1}$. When x_i is selected, we convert its color according to inversion probability P_{x_i} , otherwise do nothing. Applying this operation to an image causes a transition of the image with the transition probability \mathbf{Q} corresponding to the stochastic matrix

$$\mathbf{Q} \stackrel{\text{def}}{=} \frac{1}{n+1} (\mathbf{I} + \mathbf{P}_{x_1} + \mathbf{P}_{x_2} + \cdots + \mathbf{P}_{x_n}),$$

where \mathbf{I} is the identity matrix. Since all of the diagonal components of \mathbf{Q} are positive, \mathbf{Q} is an aperiodic stochastic matrix. Furthermore, by the same discussion showing $\mathbf{Q} > \mathbf{0}$ in the first method, we can conclude $\mathbf{Q}^n > \mathbf{0}$. Hence \mathbf{Q} is irreducible.

With both the first and second methods, we get $\mathbf{Q}^k \mathbf{q} \rightarrow \mathbf{p}$ as $k \rightarrow \infty$ for any state vector \mathbf{q} , since \mathbf{p} is a stationary state of \mathbf{Q} . Hence, if the state transition probability \mathbf{Q} is repeatedly applied to a given image, the image after sufficiently many times of repetitions is getting close to the image generated by probability distribution \mathbf{p} .

Consider an undirected graph structure on X in which pixels of the screen are vertices. We do not allow multiple edges or loops. When there is an edge between x and y for $x, y \in X$, x and y are said to be *adjacent*. For $x \in X$, the set of all pixels adjacent to x is called the *neighborhood* of x and represented by ∂x . For $Y \subseteq X$, the set of all pixels of Y^C adjacent to at least one pixel in Y is called the *boundary* of Y and denoted by ∂Y . A nonempty subset $S \subseteq X$ is called a *simplex*, if either it consists of only one pixel or any two different pixels in it are adjacent to each other. For a subset $A \subseteq X$ we define $\mathcal{S}_A \stackrel{\text{def}}{=} \{S \mid S \subseteq A \text{ and } S \text{ is a simplex}\}$. We consider a function $J : \mathcal{S}_X \rightarrow \mathbb{R}$ called an *interaction* on X . We define the *potential* $V(A)$ of image $A \subseteq X$ by

$$V(A) \stackrel{\text{def}}{=} \sum_{B \in \mathcal{S}_A} J(B)$$

when $A \neq \emptyset$ and by $V(\emptyset) \stackrel{\text{def}}{=} 0$. For $\beta \in \mathbb{R}$, define a probability density p by

$$p(A) \stackrel{\text{def}}{=} \frac{e^{\beta V(A)}}{\sum_{A \subseteq X} e^{\beta V(A)}}$$

⁷ The *symmetric difference* defined by $A \Delta B \stackrel{\text{def}}{=} (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$.

for $A \subseteq X$. We say that p is a *Gibbs state* or an *equilibrium state* with interaction J and β . Let us see the role of parameter β . If $\beta = 0$, all images on the screen X occur in the equal probability $\frac{1}{2^n}$. Divide the denominator and numerator of the definition expression of p by $e^{\beta v_1}$, where $v_1 = \max_{A \subseteq X} V(A)$. Then, the numerator becomes $e^{\beta(V(A)-v_1)} = 1$ for any β , if A attains $V(A) = v_1$, otherwise it converges to 0 as $\beta \rightarrow \infty$. This means that the larger the parameter $\beta > 0$, the closer to 1 the probability with which the image with the maximum potential appears. Dividing the denominator and numerator by $e^{\beta v_0}$ with $v_0 = \min_{A \subseteq X} V(A)$, we can also conclude that the smaller the parameter $\beta < 0$, the closer to 1 the probability with which the image with the minimum potential appears. *Simulated annealing* is a method of gradually increasing or decreasing the parameter β while realizing p as a stationary state of a Markov chain, and making the potential of the image appear according to the probability p closer to the maximum or the minimum.⁸

Let $Y \subseteq X$, where we think of Y as an area on the screen, not an image. Let $A \subseteq Y$ and $B \subseteq Y^C$. We denote by $p_Y(A | B)$ the conditional probability that A appears inside Y under the condition that B appeared outside Y . The appearance of B outside Y means that an image belonging to the family $\{A' \cup B \mid A' \subseteq Y\}$ appears, and its probability is $\sum_{A' \subseteq Y} p(A' \cup B)$. Therefore, we have

$$p_Y(A | B) = \frac{p(A \cup B)}{\sum_{A' \subseteq Y} p(A' \cup B)}.$$

We shall show that

$$p_Y(A | B) = p_Y(A | B')$$

holds for any $A \subseteq Y$ and $B, B' \subseteq Y^C$ such that $B \cap \partial Y = B' \cap \partial Y$. From this property, p is said to have a spatial Markov property. It suffices to show the equality for $B' = B \cap \partial Y$. Since any simplex does not intersect both Y and $(Y \cup \partial Y)^C$, for any $A' \subseteq Y$ and $B \subseteq Y^C$, the set $\mathcal{S}_{A' \cup B}$ of simplexes is decomposed into 5 disjoint sets of simplexes as

$$\mathcal{S}_{A' \cup B} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3 \cup \mathcal{S}_4 \cup \mathcal{S}_5,$$

where $S_1 \in \mathcal{S}_1$ is contained in Y , $S_2 \in \mathcal{S}_2$ intersects both Y and ∂Y , $S_3 \in \mathcal{S}_3$ is contained in ∂Y , $S_4 \in \mathcal{S}_4$ intersects both ∂Y and $(Y \cup \partial Y)^C$, and $S_5 \in \mathcal{S}_5$ is contained in $(Y \cup \partial Y)^C$ (see Fig. 9.11).

Note that

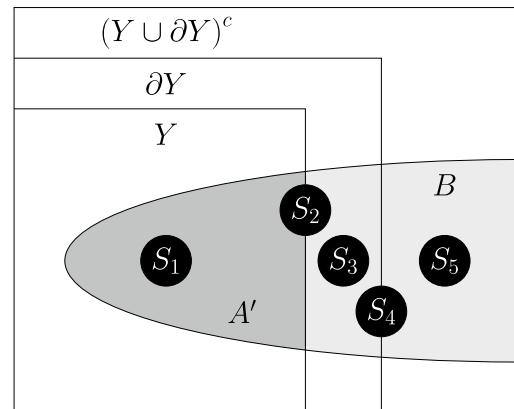
$$\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3 = \mathcal{S}_{A' \cup (B \cap \partial Y)}$$

holds and a simplex in $\mathcal{S}_4 \cup \mathcal{S}_5$ does not intersect with $A' \subseteq Y$. Therefore, since

$$V(A' \cup B) = \sum_{C \in \mathcal{S}_{A' \cup B}} J(C) = V(A' \cup (B \cap \partial Y)) + \sum_{C \in \mathcal{S}_4 \cup \mathcal{S}_5} J(C),$$

⁸ This name is due to the method of gradually decreasing the temperature of a substance in a liquid or gas state to form an aligned crystalline state. In order to get out of a local minimum (maximum) and reach the global minimum (maximum) with probability 1, proper control of the temperature parameter β is required. On the other hand, the idea of *quantum annealing*, which uses a quantum computer to perform annealing using quantum fluctuations while superimposing states, has also been proposed recently.

Fig. 9.11 The spatial Markov property



we have

$$p_Y(A \mid B) = \frac{e^{\beta V(A \cup B)}}{\sum_{A' \subseteq Y} e^{\beta V(A' \cup B)}} = \frac{e^{\beta V(A \cup (B \cap \partial Y))}}{\sum_{A' \subseteq Y} e^{\beta V(A' \cup (B \cap \partial Y))}} = p_Y(A \mid B \cap \partial Y).$$

A stochastic model like this is a typical Markov random field, where it is relatively easy to make the state transition probability of a Markov chain using an acceptance function, because the value

$$\frac{p(A_x)}{p(A)} = \frac{e^{\beta V(A_x)}}{e^{\beta V(A)}} = e^{\beta(V(A_x) - V(A))},$$

which is needed to calculate the inversion probability, depends only on $A \cap \partial x$.

Let us generate an image according to a Gibbs state. Consider, as a screen, the grid graph X with $N \times N$ pixels. In the program below, both the width and height of the screen are set to $N = 100$. In order to avoid specializing the pixels on the sides, the pixels on the upper (resp. left) side and lower (resp. right) side are adjacent so that every pixel has 8 neighbors in a torus form (Fig. 9.12).

There are 10 types of simplexes as shown in Fig. 9.13. We divide them into five groups as follows: one type of simplexes consisting of 1 pixel, two types of 2 pixels aligned vertically or horizontally, two types of 2 pixels arranged diagonally, four types of 3 pixels, and one type of 4 pixels. We give simplexes in the same group the same interaction.

Let us use the method studied in this section in a Python program to see how the shapes of images in the stationary state change by changing values of interaction J and parameter β .

Program: gibbs.py

```
In [1]: 1 from numpy import random, exp, dot
2 from tkinter import Tk, Canvas
3
4 class Screen:
5     def __init__(self, N, size=600):
6         self.N = N
7         self.unit = size // self.N
8         tk = Tk()
9         self.canvas = Canvas(tk, width=size, height=size)
10        self.canvas.pack()
11        self.pallet = ['white', 'black']
12        self.matrix = [[self.pixel(i, j) for j in range(N)]
```

In [1]:

```

13             for i in range(N)]
14
15     def pixel(self, i, j):
16         rect = self.canvas.create_rectangle
17         x0, x1 = i * self.unit, (i + 1) * self.unit
18         y0, y1 = j * self.unit, (j + 1) * self.unit
19         return rect(x0, y0, x1, y1)
20
21     def update(self, X):
22         config = self.canvas.itemconfigure
23         for i in range(self.N):
24             for j in range(self.N):
25                 c = self.pallet[X[i, j]]
26                 ij = self.matrix[i][j]
27                 config(ij, outline=c, fill=c)
28         self.canvas.update()
29
30     def reverse(X, i, j):
31         i0, i1 = i - 1, i + 1
32         j0, j1 = j - 1, j + 1
33         n, s, w, e = [X[i0, j], X[i1, j], X[i, j0], X[i, j1]]
34         nw, ne, sw, se = [X[i0, j0], X[i0, j1], X[i1, j0], X[i1, j1]]
35
36         a = X[i, j]
37         b = 1 - 2 * a
38         intr1 = b
39         intr20 = b * sum([n, s, w, e])
40         intr21 = b * sum([nw, ne, sw, se])
41         intr3 = b * sum([n * ne, ne * e, e * n, e * se,
42                         se * s, s * e, s * sw, sw * w,
43                         w * s, w * nw, nw * n, n * w])
44         intr4 = b * sum([n * ne * e, e * se * s,
45                         s * sw * w, w * nw * n])
46         return intr1, intr20, intr21, intr3, intr4
47

```

Fig. 9.12 The eight-neighbor grid graph

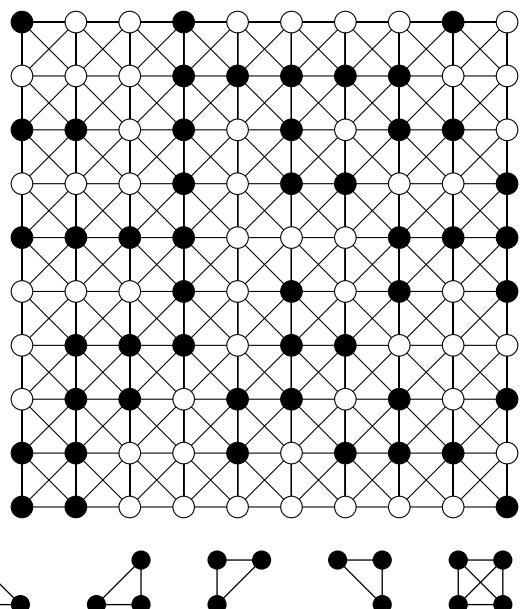
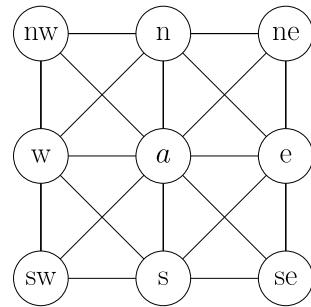


Fig. 9.13 Ten types of simplexes

Fig. 9.14 A pixel a and its 8 neighbors



```
In [1]: 48 N = 100
49 beta, J = 1.0, [-4.0, 1.0, 1.0, 0.0, 0.0]
50 scrn = Screen(N)
51 X = random.randint(0, 2, (N, N))
52 while True:
53     for i in range(-1, N - 1):
54         for j in range(-1, N - 1):
55             S = reverse(X, i, j)
56             p = exp(beta * dot(J, S))
57             if random.uniform(0.0, 1.0) < p / (1 + p):
58                 X[i, j] = 1 - X[i, j]
59             scrn.update(X)
```

Line 2: Use the module `tkinter` as the GUI tool.

Lines 4–28: Define class `Screen` to represent the state of X on the computer display. Use a `Canvas` class object of `tkinter` on which pixels of X are drawn as small squares filled with black or white. The method `pixel` defined on Lines 15–19 draws such small squares on `Canvas`. The class variable `matrix` defined on Lines 12 and 13 is a nested list whose elements are the objects of $N \times N$ small squares drawn by `pixel`. The method `update` refers to X and reflects the state of pixels of X to `Canvas` (Lines 21–28).

Lines 30–46: Define a function to calculate the increase/decrease in the number of single elements that affect the increase/decrease in the potential when $X[i, j]$ (the color of the pixel at (i, j) of X) is inverted. For simplicity, $X[i, j]$ (black is 1, white is 0) is denoted by α , and will refer to the state in the neighborhood by the variable whose name is regarded as north, south, east or west (Fig. 9.14).

Line 48: Define the width and height of the screen to be N .

Line 49: Define the parameter `beta` and the interaction J .

Line 50: Generate an object `scrn` of `Screen` class.

Line 51: The initial color of each pixel on screen X is randomly determined.

Lines 52–59: The main loop of the program. Calculation is performed on array X , and to scan correctly the pixels on the screen as a torus, $X[i, j]$ runs for i and j on `range(-1, N - 1)`. The amount `dot(J, S)` is the difference of potential V due to the inversion of $X[i, j]$. We adopt the acceptance function $f(t) = t / (1 + t)$ (Line 57). Update an image on the computer display with method `scrn.update(X)` after each screen scan.

We perform experiments for the Gibbs states with the following four pairs of parameters and interactions:

```
beta, J = 1.0, [-4.0, 1.0, 1.0, 0.0, 0.0],
beta, J = 2.0, [0.0, 1.0, -1.0, 0.0, 0.0],
beta, J = 4.0, [-2.0, 2.0, 0.0, -1.0, 2.0],
beta, J = 1.5, [-2.0, -1.0, 1.0, 1.0, -2.0].
```

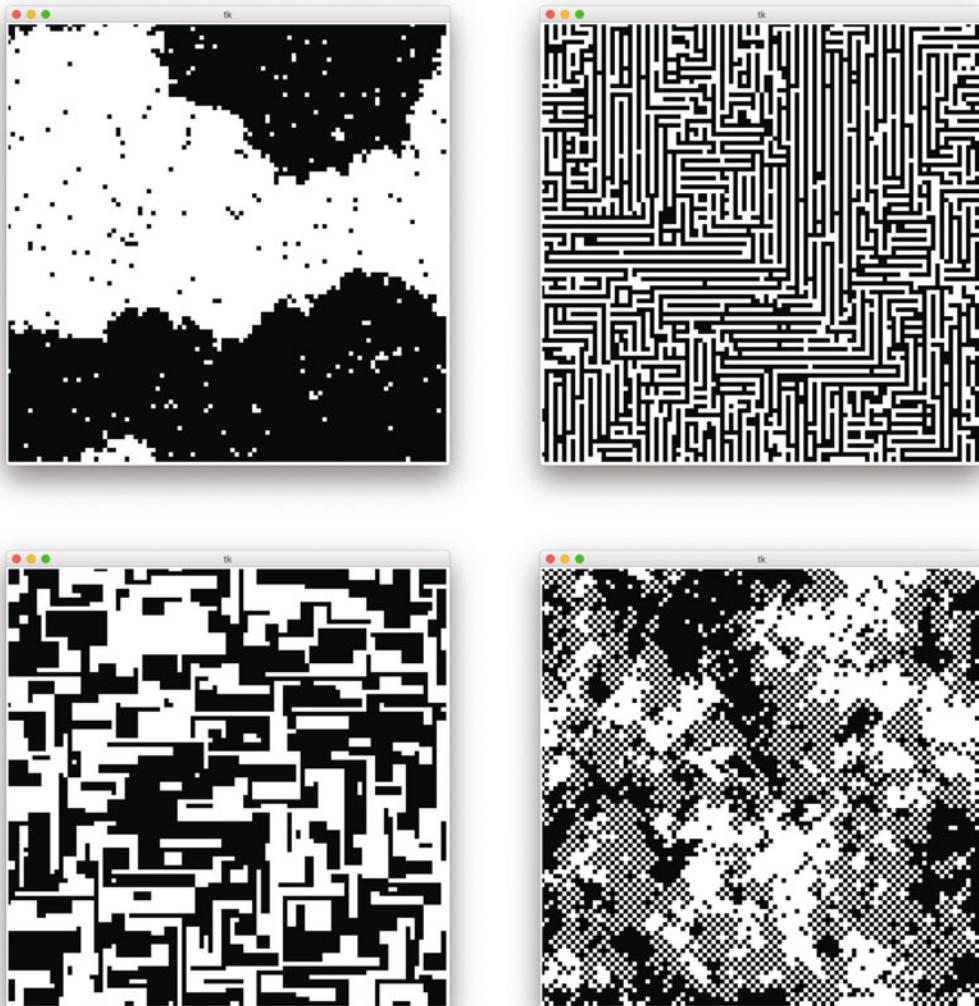


Fig. 9.15 Images in equilibrium states

We get images in Fig. 9.15 after repeating the main loop sufficiently many times. We can consider that the images change with a probability near the stationary state of the state transition probabilities, and they are close to the images generated in the equilibrium state for the given parameter β and interaction J .

9.6 One-Parameter Semigroups

The situation in which a machine goes back and forth between normal operation and failure, or the change in the number of people in a queue at a counter is an example of a stochastic system that takes discrete states as time progresses continuously. Let us consider a model that idealizes such a system.

Let \mathbf{G} be a square matrix of order n , where $\mathbf{G}(i, j)$ denotes its (i, j) -element. Suppose that $\mathbf{G}(i, j) \geq 0$ for any $i \neq j$, and $\sum_{i=1}^n \mathbf{G}(i, j) = 0$ for any fixed j . So, $\mathbf{G}(i, i) \leq 0$ for every i . Define

$$\mathbf{P}_t \stackrel{\text{def}}{=} e^{t\mathbf{G}} \quad (t \geq 0),$$

then the following (1)–(4) hold:

- (1) \mathbf{P}_t is a stochastic matrix.
- (2) $\mathbf{P}_{s+t} = \mathbf{P}_s \mathbf{P}_t$.
- (3) $\mathbf{P}_t \rightarrow \mathbf{P}_0 = \mathbf{I}$ ($t \downarrow 0$).
- (4) $\frac{\mathbf{P}_t - \mathbf{I}}{t} \rightarrow \mathbf{G}$ ($t \downarrow 0$).

To prove (1), we first show that \mathbf{P}_t is a positive matrix under the condition $\mathbf{G}(i, j) > 0$ for $i \neq j$. For a sufficiently small $\Delta t > 0$, we can write

$$\mathbf{P}_{\Delta t} = \mathbf{I} + \mathbf{G} \cdot \Delta t + \mathcal{O}(\Delta t^2),$$

where $\mathcal{O}(\Delta t^2)$ is a matrix whose elements are all bounded by a constant multiple of Δt^2 .⁹ Here, $\mathcal{O}(\Delta t^2)$ is negligible compared to \mathbf{I} and $\mathbf{G} \cdot \Delta t$. Because the diagonal elements of $\mathbf{G} \cdot \Delta t$ can be sufficiently small, the diagonal elements of $\mathbf{I} + \mathbf{G} \cdot \Delta t$ become positive. In addition, all the non-diagonal elements of $\mathbf{I} + \mathbf{G} \cdot \Delta t$ are positive. Thus, $\mathbf{I} + \mathbf{G} \cdot \Delta t$ and also $\mathbf{P}_{\Delta t}$ becomes a positive matrix. Now, for any $t > 0$, take k large enough so that $\mathbf{P}_{\Delta t}$ becomes positive with $\Delta t = t/k$, then we see that $\mathbf{P}_t = (\mathbf{P}_{\Delta t})^k$ is a positive matrix. When \mathbf{G} does not satisfy the above condition, let \mathbf{G}_ϵ be the matrix obtained from \mathbf{G} by adding $\epsilon > 0$ to each element of \mathbf{G} . Then, $\mathbf{P}_{t,\epsilon} \stackrel{\text{def}}{=} e^{t\mathbf{G}_\epsilon}$ is positive and $\lim_{\epsilon \rightarrow 0} \mathbf{P}_{t,\epsilon} = \mathbf{P}_t$. It follows that \mathbf{P}_t is nonnegative. Let $\mathbf{1} = [1 \ 1 \ \cdots \ 1]$ be the matrix of shape $(1, n)$ whose elements are all 1. Because the sum of the elements of each column of \mathbf{G} is 0, we have $\mathbf{1}\mathbf{G} = \mathbf{0}$, and hence $\mathbf{1}\mathbf{G}^k = \mathbf{0}$ for any $k > 0$. Thus, we have

$$\mathbf{1}\mathbf{P}_t = \mathbf{1}e^{t\mathbf{G}} = \sum_{k=0}^{\infty} \frac{t^k}{k!} \mathbf{1}\mathbf{G}^k = \mathbf{1}\mathbf{I} + \sum_{k=1}^{\infty} \frac{t^k}{k!} \mathbf{1}\mathbf{G}^k = \mathbf{1}.$$

Therefore, \mathbf{P}_t is a nonnegative matrix such that the sum of the elements of each column is 1, that is, \mathbf{P}_t is a stochastic matrix.

Exercise 9.5 Prove the above properties (2)–(4).

We call $\{\mathbf{P}_t\}_{t \geq 0}$ a *one-parameter semigroup* and \mathbf{G} its *generator matrix*. Let $\mathbf{P}_t(i, j)$ denote the (i, j) -element of \mathbf{P}_t . We consider the following stochastic system on the set $\{0, 1, \dots, n-1\}$ of states. The probability that state j changes to state i after t seconds is given by $\mathbf{P}_t(i, j)$. We can consider that for an infinitesimal time Δt , it is the limit of the stationary Markov chain for $\mathbf{P}_{\Delta t}$ approximated by $\mathbf{I} + \mathbf{G} \cdot \Delta t$ as $\Delta t \rightarrow 0$.

\mathbf{G} is called *irreducible*¹⁰ if for any $i \neq j$, there exists a sequence $i = i_0, i_1, \dots, i_k = j$ such that

⁹ This is called a Landau notation. Precisely, for functions f and g , $f(t) = O(g(t))$ as $t \rightarrow 0$ means that there is a constant $C > 0$ such that $|f(t)| \leq C|g(t)|$ for all sufficiently small t .

¹⁰ If \mathbf{G} is irreducible, then $e^{t\mathbf{G}}$ is an irreducible state transition matrix as seen below.

$$\mathbf{G}(i_k, i_{k-1}) \cdots \mathbf{G}(i_2, i_1) \mathbf{G}(i_1, i_0) > 0.$$

Then, for sufficiently small $\Delta t > 0$, we have

$$\mathbf{P}_{\frac{\Delta t}{k}} = \mathbf{I} + \mathbf{G} \cdot \frac{\Delta t}{k} + O\left(\frac{\Delta t^2}{k^2}\right),$$

and so the (i, j) -elements of both sides for $i \neq j$ satisfy

$$\mathbf{P}_{\frac{\Delta t}{k}}(i, j) = \mathbf{G}(i, j) \cdot \frac{\Delta t}{k} + O\left(\frac{\Delta t^2}{k^2}\right).$$

Hence,

$$\begin{aligned} \mathbf{P}_{\Delta t}(i, j) &\geq \mathbf{P}_{\frac{\Delta t}{k}}(i_k, i_{k-1}) \cdots \mathbf{P}_{\frac{\Delta t}{k}}(i_2, i_1) \mathbf{P}_{\frac{\Delta t}{k}}(i_1, i_0) \\ &= \mathbf{G}(i_k, i_{k-1}) \cdots \mathbf{G}(i_2, i_1) \mathbf{G}(i_1, i_0) \cdot \frac{\Delta t^k}{k^k} + O\left(\frac{\Delta t^{k+1}}{k^{k+1}}\right). \end{aligned}$$

Because, compared to the first term, the second term on the rightmost side above is negligible, we can conclude that the leftmost side $\mathbf{P}_{\Delta t}(i, j)$ is positive. Moreover, we have $\mathbf{P}_{\Delta t}(i, i) \geq \mathbf{P}_{\Delta t/2}(i, j) \mathbf{P}_{\Delta t/2}(j, i) > 0$ for any i . In this way, we find that $\mathbf{P}_{\Delta t}$ is a positive matrix for sufficiently small $\Delta t > 0$ if \mathbf{G} is irreducible. For any $t > 0$ setting $k = t/\Delta t$, $\mathbf{P}_t = \mathbf{P}_{\Delta t}^k$ is also positive. By the Perron–Frobenius Theorem, \mathbf{P}_t has the eigenvalue 1 and the unique corresponding eigenvector \mathbf{x}_1 that is a state vector. This \mathbf{x}_1 is unique and is the unique stationary state for the state transition matrix \mathbf{P}_t by Theorem 9.1. This also corresponds to the (maximal) eigenvalue 0 of \mathbf{G} .

The following program simulates the one-parameter semigroup with the state set $\{0, 1, 2\}$ and generator matrix

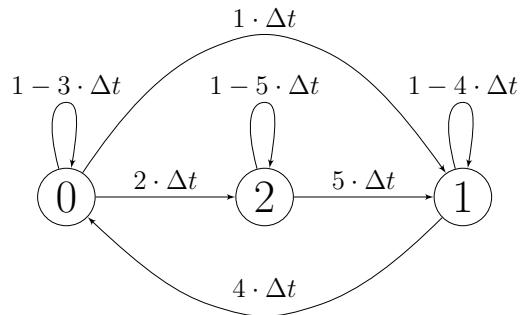
$$\mathbf{G} = \begin{bmatrix} -3 & 4 & 0 \\ 1 & -4 & 5 \\ 2 & 0 & -5 \end{bmatrix}.$$

Figure 9.16 is the state transition diagram for the state transition matrix $\mathbf{P}_{\Delta t} \doteq \mathbf{I} + \mathbf{G} \cdot \Delta t$ showing the transition after a small time Δt .

Program: semigroup.py

```
In [1]: 1 from numpy import arange, array, eye, exp
2 from numpy.random import choice, seed
3 from numpy.linalg import eig
4 import matplotlib.pyplot as plt
5
6 seed(2021)
7 dt, tmax = 0.01, 1000
8 T = arange(0.0, tmax, dt)
9 G = array([[ -3,   4,   0], [  1,  -4,   5], [  2,   0, -5]])
10 v = eig(G)[1][:, 0]
11 print(v / sum(v))
12 dP = eye(3) + G * dt
13
14 X = [0]
15 S = [[dt], [], []]
16 for t in T:
17     x = X[-1]
18     y = choice(3, p=dP[:, x])
19     if x == y:
```

Fig. 9.16 State transition diagram



```
In [1]: 20     S[x][-1] += dt
21     else:
22         S[y].append(dt)
23     x.append(y)
24
25 plt.figure(figsize=(15, 5))
26 plt.plot(T[:2000], x[:2000])
27 plt.yticks(range(3))
28 fig, axs = plt.subplots(1, 3, figsize=(20, 5))
29 for x in range(3):
30     s, n = sum(S[x]), len(S[x])
31     print(s / tmax)
32     m = s / n
33     axs[x].hist(S[x], bins=10)
34     t = arange(0, 3, 0.01)
35     axs[x].plot(t, exp(-t / m) / m * s)
36     axs[x].set_xlim(0, 3), axs[x].set_ylim(0, 600)
37 plt.show()
```

Lines 7,8: dt represents Δt .

Line 9: Define generator matrix G .

Lines 10,11: Get an eigenvector v corresponding to the maximal eigenvalue 0 of G and print the normalized v .

Line 12: dP is expression $I + G\Delta t$ approximating $P_{\Delta t}$.

Line 14: x is a list to record the state change with initial state 0.

Line 15: Counter to measure the time staying at each state.

Lines 16–23: When the current state is x , find the state after Δt seconds by generating y with probability $P_{\Delta t}(y, x)$.

Lines 25–27: Draw the graph of the state changes for the first 2000 steps (20 s) (Fig. 9.17). This is called the *sampling function* of G .

Lines 28–36: Draw the histogram of the frequency distribution of the staying time (the horizontal length of the sampling function) (Fig. 9.18). These follow the so-called *exponential distribution*. Also, find the ratio of staying time for each state.

```
[0.46511628-0.j 0.34883721-0.j 0.18604651-0.j]
0.469059999999994
0.346999999999995
0.1839500000000028
```

The first line of the outputs is the eigenvector for the eigenvalue 0 of G (expressed by a complex number with positive real part and 0 imaginary part), which is the stationary state. The next three numbers are the ratios of staying times of the states 0, 1, and 2 downwards and are close to the values of the stationary state. Some differences may occur depending on the sample function.

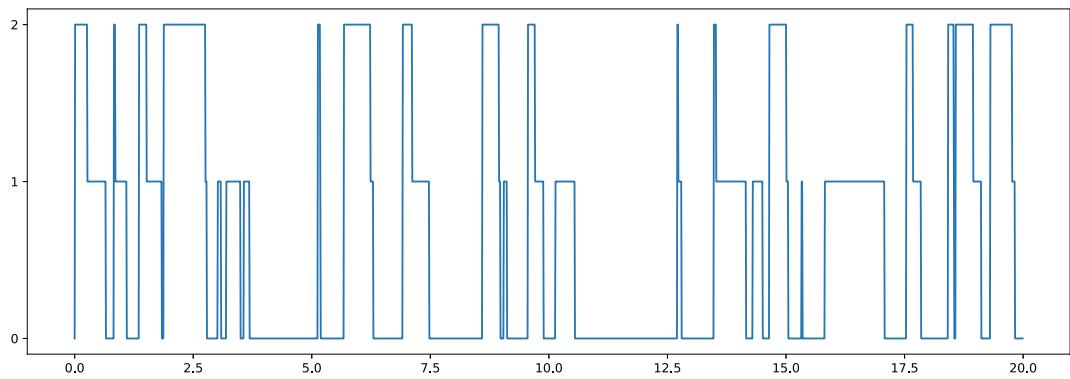


Fig. 9.17 State change

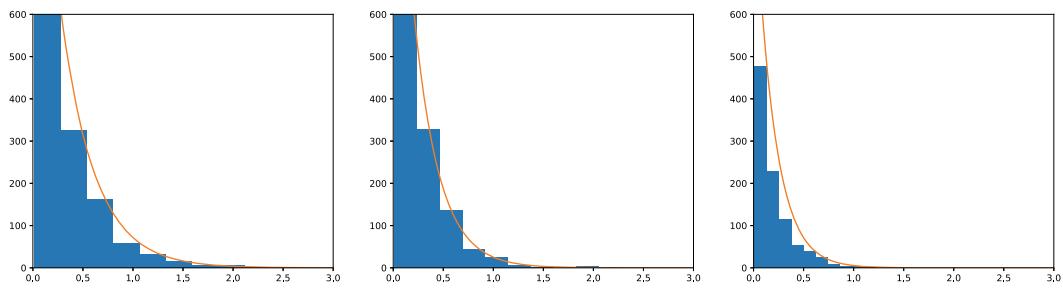


Fig. 9.18 Distribution of staying times for states 0, 1, and 2 from left to right



Applications and Development of Linear Algebra

10

In this chapter, we integrate what we have learned so far into the theory of singular value decompositions and generalized inverse matrices, and discuss related topics. These theories can be considered as generalizations of Fourier analysis for orthonormal systems and as the theory of linear equations for non-regular matrices. They are closely linked to the eigenvalue problem, and because only Hermitian matrices (or real symmetric matrices) appear, we can handle them relatively easily both theoretically and computationally.

First, we reconsider the least squares method discussed in Chap. 6 from the viewpoint of solving linear equations. We introduce a generalized inverse matrix and a singular value decomposition as generic methods for solving the problems. Further, we introduce the concept of a tensor product and see that Fourier expansion, diagonalization of a matrix, and singular value decomposition all have expressions of tensor decomposition.

This is also related to the mathematical model of observables and states in quantum mechanics. Tensor decompositions of random vector-valued variables are realized as methods of data analysis and data compression called principal component analysis and KL expansion, tools frequently used in applications. Estimating or predicting the true value through linear transformations from time-series data of observed values that are disturbed or missed due to noise, etc., is called linear regression. It is used in various fields such as weather forecasting, stock price forecasting, robotics, etc. Based on mathematical theory, we will learn programming in Python to solve these problems.

10.1 Linear Equations and Least Squares

As before, $M_{\mathbb{K}}(m, n)$ denotes the linear space of all matrices of shape (m, n) . For a matrix $A \in M_{\mathbb{K}}(m, n)$ the following equalities hold:

- | | |
|---|---|
| 1. $\text{kernel}(A) = \text{range}(A^*)^\perp$ | 2. $\text{kernel}(A)^\perp = \text{range}(A^*)$ |
| 3. $\text{range}(A) = \text{kernel}(A^*)^\perp$ | 4. $\text{range}(A)^\perp = \text{kernel}(A^*)$. |

We can show 1 as follows:

$$\begin{aligned} \mathbf{x} \in \text{kernel}(A) &\Leftrightarrow A\mathbf{x} = \mathbf{0} \\ &\Leftrightarrow \langle \mathbf{y} \mid A\mathbf{x} \rangle = 0 \text{ for any } \mathbf{y} \in \mathbb{K}^m \\ &\Leftrightarrow \langle A^*\mathbf{y} \mid \mathbf{x} \rangle = 0 \text{ for any } \mathbf{y} \in \mathbb{K}^m \\ &\Leftrightarrow \mathbf{x} \in \text{range}(A^*)^\perp. \end{aligned}$$

The equalities 2–4 can be obtained from 1 using the fact that the operations $*$ and $^\perp$ are involutive, that is, they are restored when repeated twice.

From 1 (or 2), we see $\mathbb{K}^n = \text{kernel}(A) \oplus \text{range}(A^*)$, and any $\mathbf{x} \in \mathbb{K}^n$ is uniquely written as $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ with $\mathbf{x}_1 \in \text{kernel}(A)$ and $\mathbf{x}_2 \in \text{range}(A^*)$. Hence, we have $A\mathbf{x} = A\mathbf{x}_2$ and the mapping $\mathbf{x} \mapsto A\mathbf{x}$ is a linear isomorphism from $\text{range}(A^*)$ to $\text{range}(A)$. In particular, $\text{rank}(A) = \text{rank}(A^*)$.

Let us consider the following problem: for given vectors,

$$\mathbf{a}_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}, \quad \mathbf{a}_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}, \quad \dots, \quad \mathbf{a}_n = \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

in \mathbb{K}^m , express \mathbf{b} as a linear combination of $A = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$.

A necessary and sufficient condition for this problem to have a solution is that \mathbf{b} belongs to the subspace W generated by A . Moreover, when a solution exists, it is unique if and only if A is linearly independent. When there is no solution, our goal is to find an element in W that is the best approximation of \mathbf{b} in some sense, and when solutions are not unique, the goal is to find a solution that is standard in some sense. If A is an orthonormal basis of \mathbb{K}^m (so $m = n$), then this problem is nothing but obtaining the Fourier expansion of \mathbf{b} . If A is an orthonormal system ($n \leq m$), the solution is the orthogonal projection $\langle \mathbf{a}_1 \mid \mathbf{b} \rangle \mathbf{a}_1 + \langle \mathbf{a}_2 \mid \mathbf{b} \rangle \mathbf{a}_2 + \dots + \langle \mathbf{a}_n \mid \mathbf{b} \rangle \mathbf{a}_n$ of \mathbf{b} onto W .

In the general case, letting

$$A = [\mathbf{a}_1 \ \mathbf{a}_2 \ \dots \ \mathbf{a}_n] = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

the problem is to find an exact solution or an approximate solution of $A\mathbf{x} = \mathbf{b}$, and is reduced to the following problems equivalent to each other:

- P1. Find $\mathbf{x} \in \mathbb{K}^n$ minimizing $\|\mathbf{b} - A\mathbf{x}\|$,
- P2. Find $\mathbf{x} \in \mathbb{K}^n$ satisfying $A^*A\mathbf{x} = A^*\mathbf{b}$.

The equivalence of P1 and P2 is shown as follows. Let $\mathbf{y}_0 = \text{proj}_{\text{range}(A)}(\mathbf{b})$, then

$$\|\mathbf{b} - \mathbf{y}_0\| = \min_{\mathbf{y} \in \text{range}(A)} \|\mathbf{b} - \mathbf{y}\|.$$

Choose $\mathbf{x}_0 \in \mathbb{K}^n$ such that $\mathbf{y}_0 = A\mathbf{x}_0$, then \mathbf{x}_0 is a solution for P1 and $\mathbf{b} - A\mathbf{x}_0$ is orthogonal to $\text{range}(A)$. Hence, for any $\mathbf{x} \in \mathbb{K}^n$, we have

$$0 = \langle \mathbf{b} - \mathbf{A}\mathbf{x}_0 \mid \mathbf{A}\mathbf{x} \rangle = \langle \mathbf{A}^*(\mathbf{b} - \mathbf{A}\mathbf{x}_0) \mid \mathbf{x} \rangle$$

It follows that $\mathbf{A}^*\mathbf{b} - \mathbf{A}^*\mathbf{A}\mathbf{x}_0 = \mathbf{0}$ and \mathbf{x}_0 is a solution for P2. If we follow this argument in reverse, we can show that a solution for P2 is also a solution for P1.

In general, we call the set of all solutions to a linear equation $\mathbf{A}\mathbf{x} = \mathbf{b}$, the *solution set* or *solution space*. The solution set may be empty. If it is not empty, any solution \mathbf{x}_0 is uniquely written as $\mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2$ with $\mathbf{x}_1 \in \text{kernel}(\mathbf{A})$ and $\mathbf{x}_2 \in \text{kernel}(\mathbf{A})^\perp$. Thus, we see $\mathbf{A}\mathbf{x}_0 = \mathbf{A}\mathbf{x}_2 = \mathbf{b}$ and $\|\mathbf{x}_0\| \geq \|\mathbf{x}_2\|$, that is, \mathbf{x}_2 is the solution with minimum norm. Therefore, we can conclude that the linear equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a unique solution \mathbf{x}_2 in $\text{kernel}(\mathbf{A})^\perp = \text{range}(\mathbf{A}^*)$, and the solution set is expressed as $\mathbf{x}_2 + \text{kernel}(\mathbf{A})$.¹

For a set $\{\varphi_1, \varphi_2, \dots, \varphi_n\}$ of functions and another function f defined on a set X valued in \mathbb{K} , consider the expression

$$f(x) = a_1\varphi_1(x) + a_2\varphi_2(x) + \cdots + a_n\varphi_n(x) \quad \dots \quad (*)$$

with $a_1, a_2, \dots, a_n \in \mathbb{K}$. For *sampling points* $x_1, x_2, \dots, x_m \in X$, we call $f(x_1), f(x_2), \dots, f(x_m)$ the *sampling values*. We consider the problem to find solutions a_1, a_2, \dots, a_n satisfying $(*)$ at every sampling point. This is equivalent to solving the linear equation

$$\begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \cdots & \varphi_n(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \cdots & \varphi_n(x_2) \\ \vdots & \vdots & & \vdots \\ \varphi_1(x_m) & \varphi_2(x_m) & \cdots & \varphi_n(x_m) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{bmatrix}. \quad \dots \quad (**)$$

This is further equivalent to solving the equation

$$\sum_{i=1}^m \left| f(x_i) - \sum_{j=1}^n a_j \varphi_j(x_i) \right|^2 = 0. \quad \dots \quad (***)$$

When $(**)$ has no solution, we try to find a solution minimizing the *squared error* in the left-hand side of $(***)$. To find this approximate solution, we use the least squares method discussed in Sect. 6.4. NumPy provides a function for the least squares method, so we compare the results by solving the linear equation and by using the function.

Program: `lstsq2.py`

```
In [1]: 1 from numpy import array, linspace, sqrt, random, linalg
2 import matplotlib.pyplot as plt
3
4 n, m = 30, 1000
5 random.seed(2021)
6 x = linspace(0.0, 1.0, m)
7 w = random.normal(0.0, sqrt(1.0/m), m)
8 y = w.cumsum()
9 tA = array([x**j for j in range(n + 1)])
10 A = tA.T
11 S = linalg.solve(tA.dot(A), tA.dot(y))
12 L = linalg.lstsq(A, y, rcond=None)[0]
13
```

¹ A subset $\mathbf{x} + W = \{\mathbf{x} + \mathbf{w} \mid \mathbf{w} \in W\}$ of a linear space V obtained by translating a subspace W by $\mathbf{x} \in V$ is called an *affine space*.

```
In [1]: 14 | fig, axs = plt.subplots(1, 2, figsize=(15, 5))
15 | for ax, B, title in zip(axs, [S, L], ['solve', 'lstsq']):
16 |     z = B.dot(tA)
17 |     ax.plot(x, y), ax.plot(x, z), ax.set_ylim(-0.7, 1)
18 |     ax.text(0, -0.6, f'linslg.{title}', fontsize=16)
19 | plt.show()
```

Lines 4–9: The sample function of the one-dimensional Brownian motion will be approximated by the least squares method with polynomial functions of degree 30 using 1000 sampling points obtained by dividing equally the interval $[0, 1]$.

Line 11: Solve the equation $A^*Ax = A^*y$ using `linalg.solve`.

Line 12: Find the least squares approximation using `linalg.lstsq`. Computation of high-order exponentiation likely causes errors from information loss. Since `linalg.lstsq` takes that into account, it gives a little better approximation accuracy (Fig. 10.1). To further improve the approximation accuracy, we need to use orthogonal polynomials such as the Chebyshev polynomials and Legendre polynomials.²

A point moving in a two-dimensional plane can be regarded as a function $f : [a, b] \rightarrow \mathbb{R}^2$. Identifying \mathbb{R}^2 with \mathbb{C} , we consider f as a complex-valued function, on which we perform a least squares approximation. Here, we use the data file `tablet.txt` obtained by program `tablet.py` in Sect. 1.8, and try to approximate the handwritten characters on \mathbb{C} with $f : [0, 1] \rightarrow \mathbb{C}$. The function f represents the stroke order (Fig. 10.2) starting with $f(0)$ and ending with $f(1)$. We try to find an approximation by changing the function system and changing the number n of functions. The sample points divide $[0, 1]$ into m equal parts.

Program: `moji.py`

```
In [1]: 1 | from numpy import array, linspace, identity, exp, pi, linalg
2 | from numpy.polynomial.legendre import Legendre
3 | import matplotlib.pyplot as plt
4 |
5 | with open('tablet.txt', 'r') as fd:
6 |     y = eval(fd.read())
7 | m = len(y)
8 | x = linspace(0.0, 1.0, m)
```

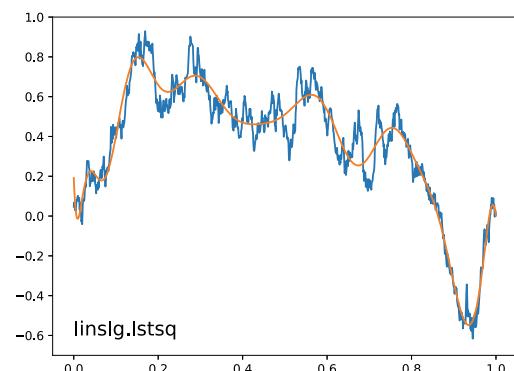
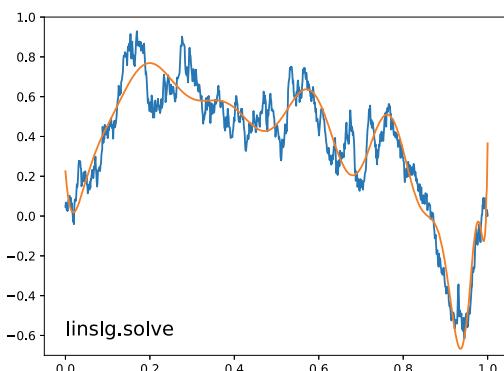


Fig. 10.1 The least squares approximations with polynomials

² The monomial x^n is close to 0 on $-1 < x < 1$ for large n . On the other hand, the Chebyshev polynomials and Legendre polynomials vibrate moderately between -1 and 1 like trigonometric functions.

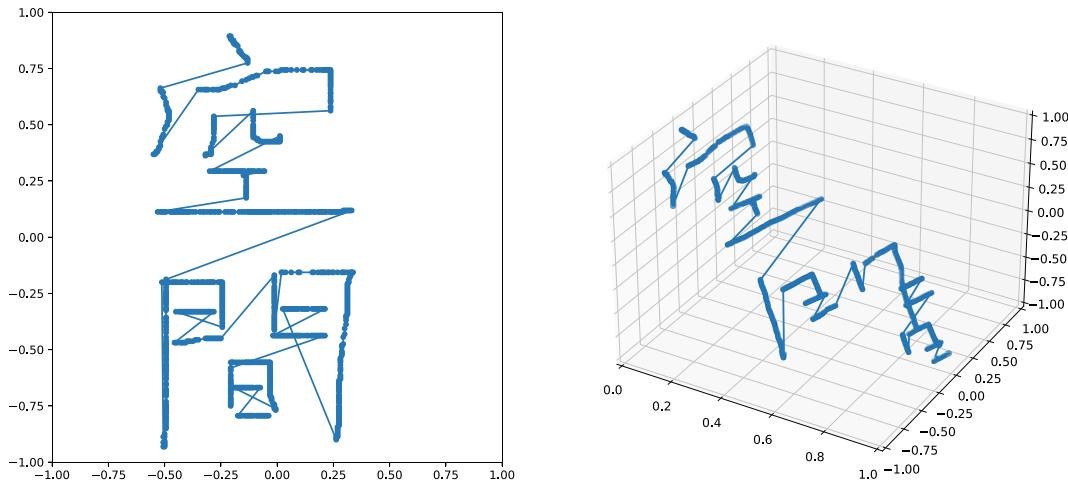


Fig. 10.2 Regard points moving on the complex plane as a function valued in \mathbb{C}

```
In [1]: 9
10 def phi1(n):
11     return array([(x0 >= x).astype('int')
12                  for x0 in linspace(0, 1, n)]).T
13
14 def phi2(n):
15     return array([exp(2 * pi * k * 1j * x)
16                  for k in range(-n // 2, n // 2 + 1)]).T
17
18 def phi3(n):
19     return array([Legendre.basis(j, domain=[0, 1])(x)
20                  for j in range(n)]).T
21
22 fig, axs = plt.subplots(3, 5, figsize=(15, 8))
23 for i, f in enumerate([phi1, phi2, phi3]):
24     for j, n in enumerate([8, 16, 32, 64, 128]):
25         ax = axs[i, j]
26         c = linalg.lstsq(f(n), y, rcond=None)[0]
27         z = f(n).dot(c)
28         ax.scatter(z.real, z.imag, s=5), ax.plot(z.real, z.imag)
29         ax.axis('scaled'), ax.set_xlim(-1, 1), ax.set_ylim(-1, 1)
30         ax.tick_params(labelbottom=False, labelleft=False,
31                         color='white')
32         ax.text(-0.9, 0.8, f'n={n}', fontsize=12)
33
34 plt.subplots_adjust(left=0.2, right=0.8, bottom=0.1, top=0.9,
35                     hspace=0.01, wspace=0.02)
36 plt.show()
```

Lines 5–8: Read a complex number sequence from `tablet.txt` as the sample values. Place the corresponding sample points x_1, x_2, \dots, x_m evenly on $[0, 1]$, where m is the number of terms in the sequence.

Lines 10–20: Define the following three systems of functions:

- **Binary functions:** The system $\{1_{x_0}, 1_{x_1}, \dots, 1_{x_n}\}$ of functions defined by

$$1_{x_k}(x) = \begin{cases} 0 & \text{if } x < x_k \\ 1 & \text{if } x_k \leq x \end{cases}$$

for the points $0 = x_0 < x_1 < \dots < x_{n-1} < x_n = 1$ dividing $[0, 1]$ evenly.

- **Fourier series:** The system $\{e_{-n/2}, \dots, e_{-1}, e_0, e_1, \dots, e_{n/2}\}$ with $e_k(x) = e^{2\pi i k x}$.

- **Polynomials:** The system $\{p_0, p_1, \dots, p_n\}$ consisting of the Legendre polynomials $p_k(x)$ of degree k .³

Functions phi1, phi2, and phi3 return the matrix

$$\begin{bmatrix} \varphi_0(x_1) & \varphi_1(x_1) & \cdots & \varphi_n(x_1) \\ \varphi_0(x_2) & \varphi_1(x_2) & \cdots & \varphi_n(x_2) \\ \vdots & \vdots & & \vdots \\ \varphi_0(x_m) & \varphi_1(x_m) & \cdots & \varphi_n(x_m) \end{bmatrix}$$

of shape $(m, n + 1)$ for systems $\{\varphi_1, \varphi_2, \dots, \varphi_n\}$ of three kinds, respectively.

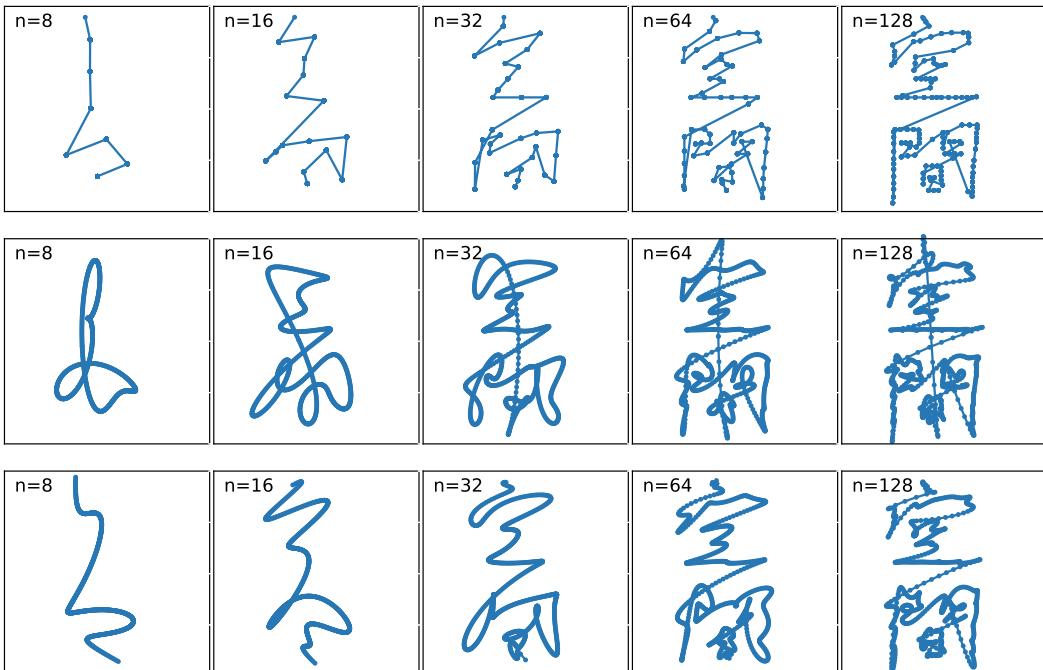


Fig. 10.3 Least squares approximation

³ We can compute it also by Fourier expansion using the orthogonality of the Legendre polynomial, as we did for the low-pass filter using Fourier series or trigonometric series in Chap. 6. Because the orthogonality is not used in this program, we can obtain the same results if we use other orthogonal polynomials such as the Chebyshev polynomial with different inner products. If we use the monomial system $\{1, x, x^2, \dots, x^n, \dots\}$, the calculation error may become large as n increases, as we mention in Program `1stsqr2.py`.

Lines 22–32: Draw the graphs of the functions estimated by binary functions (top), Fourier series (middle), and polynomials (bottom) for $n = 8, 16, 32, 64, 128$ (Fig. 10.3).

10.2 Generalized Inverse and Singular Value Decomposition

Let A be a matrix of shape (m, n) over \mathbb{K} , and let P and Q be the orthogonal projections on range (A^*) and on range (A) , respectively (Fig. 10.4). As stated in the beginning of the last section, we have $\mathbb{K}^n = \text{kernel}(A) \oplus \text{range}(A^*)$ and the restriction f on range (A^*) of the linear mapping represented by A is an isomorphism from range (A^*) to range (A) . Since $x - Px \in \text{range}(A^*)^\perp = \text{kernel}(A)$ for any $x \in \mathbb{K}^n$, we have

$$\mathbf{0} = A(x - Px) = Ax - APx.$$

It follows that $A = AP$. Similarly, we have $A = QA$ and $A^* = A^*Q = PA^*$.

Using the inverse linear mapping $f^{-1} : \text{range}(A) \rightarrow \text{range}(A^*)$, we define a linear mapping $g : \mathbb{K}^m \rightarrow \mathbb{K}^n$ by

$$g(y) \stackrel{\text{def}}{=} f^{-1}(Qy) \quad (y \in \mathbb{K}^m).$$

The representation matrix of g denoted by A^\dagger is called the *Moore-Penrose generalized inverse matrix* of A . The following properties 1–5 are satisfied:

$$\begin{array}{lll} 1. A^\dagger = A^\dagger Q = PA^\dagger, & 2. A^\dagger A = P, & 3. AA^\dagger = Q, \\ 4. A^{\dagger\dagger} = A, & & 5. (A^\dagger)^* = (A^*)^\dagger. \end{array}$$

Since $\text{range}(A^\dagger) = \text{range}(A^*)$, $PA^\dagger = A^\dagger$ follows. Further, for any $y \in \mathbb{K}^m$, we have

$$A^\dagger Qy = f^{-1}(QQy) = f^{-1}(Qy) = A^\dagger y,$$

and so $A^\dagger Q = A^\dagger$. Because

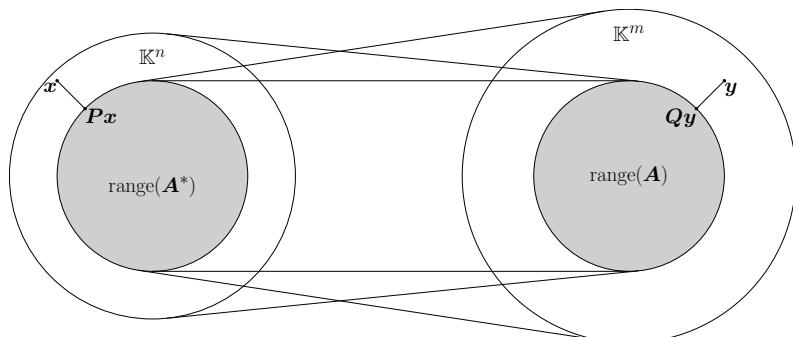
$$A^\dagger Ax = f^{-1}(QAx) = f^{-1}(Ax) = f^{-1}(APx) = f^{-1}(f(Px)) = Px$$

holds for any $x \in \mathbb{K}^n$, we have $A^\dagger A = P$. Because

$$AA^\dagger y = f(f^{-1}(Qy)) = Qy$$

for any $y \in \mathbb{K}^m$, we have $AA^\dagger = Q$.

Fig. 10.4 Relationship between P and Q



Conversely, the properties $A'Q = A^\dagger$ and $A'A = P$ characterize the generalized inverse A^\dagger . In fact, let A' be a matrix of shape (n, m) satisfying $A'Q = A'$ and $A'A = P$. For an arbitrary $y \in \mathbb{K}^m$ choose $x \in \mathbb{K}^n$ such that $Qy = Ax$. Then, we have

$$A'y = A'Qy = A'Ax = Px.$$

This holds if we replace A' by A^\dagger , that is, $A^\dagger y = Px$. Therefore, we have $A'y = A^\dagger y$ for arbitrary y . It follows that $A' = A^\dagger$.

Now, because $AP = A$ and $AA^\dagger = Q$, the uniqueness of the generalized inverse matrix we have just proved implies that the generalized inverse of A^\dagger must be A , that is, $A^{\dagger\dagger} = A$.

Lastly, we shall prove 5. Taking adjoint matrices of both sides of $A^\dagger = PA^\dagger$ and $AA^\dagger = Q$ in 1 and 3, we have $(A^\dagger)^* = (A^\dagger)^*P^* = (A^\dagger)^*P$ and $(A^\dagger)^*A^* = Q^* = Q$. Hence the generalized inverse of A^* is $(A^\dagger)^*$ again by its uniqueness, that is, $(A^*)^\dagger = (A^\dagger)^*$.

The generalized inverse matrix gives, in some sense, the unique solution of P2 in the previous section. In fact, since

$$A^*AA^\dagger b = A^*Qb = A^*b,$$

$x = A^\dagger b$ is a solution to the equation $A^*Ax = A^*b$ for $v \in \mathbb{K}^m$. In addition, it is the unique solution in range (A^*) . The solution space of $A^*Ax = A^*b$ is expressed as $A^\dagger b + \text{kernel}(A)$, and $A^\dagger b$ is one with the minimum norm in the solution space.

Because A^*A is a nonnegative Hermitian matrix, its eigenvalues are all nonnegative. Suppose that its nonzero eigenvalues are $\sigma_1^2, \sigma_2^2, \dots, \sigma_k^2$ including multiplicity, where σ_i are positive real numbers. We can take the corresponding eigenvectors v_1, v_2, \dots, v_k so that they form an orthonormal basis of $\text{range}(A^*) = \text{kernel}(A)^\perp$. Let

$$\mathbf{w}_i \stackrel{\text{def}}{=} \frac{Av_i}{\sigma_i} \quad (i = 1, 2, \dots, k),$$

then $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_k\}$ forms an orthonormal basis of $\text{range}(A) = \text{kernel}(A^*)^\perp$, because

$$\langle \mathbf{w}_i | \mathbf{w}_j \rangle = \left\langle \frac{Av_i}{\sigma_i} \mid \frac{Av_j}{\sigma_j} \right\rangle = \frac{\langle A^*Av_i \mid v_j \rangle}{\sigma_i \sigma_j} = \frac{\langle \sigma_i^2 v_i \mid v_j \rangle}{\sigma_i \sigma_j} = \frac{\sigma_i^2}{\sigma_i \sigma_j} \langle v_i \mid v_j \rangle = \delta_{ij}.$$

Since $Av_i = \sigma_i \mathbf{w}_i$ for $i = 1, 2, \dots, k$, the linear mapping $f : \text{range}(A^*) \rightarrow \text{range}(A)$ and its inverse $f^{-1} : \text{range}(A) \rightarrow \text{range}(A^*)$ are represented by

$$\begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_k} \end{bmatrix}$$

respectively. Let

$$\mathbf{V} \stackrel{\text{def}}{=} [v_1 \ v_2 \ \cdots \ v_k] \quad \text{and} \quad \mathbf{W} \stackrel{\text{def}}{=} [\mathbf{w}_1 \ \mathbf{w}_2 \ \cdots \ \mathbf{w}_k]$$

be the $n \times k$ matrix \mathbf{V} and $m \times k$ matrix \mathbf{W} consisting of the column vectors $\{v_1, v_2, \dots, v_k\}$ and $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_k\}$, respectively. Let $\{e_1, e_2, \dots, e_n\}$ be the standard basis of \mathbb{K}^n . Because

$$\mathbf{P} \mathbf{e}_j = \sum_{i=1}^k \langle \mathbf{v}_i \mid \mathbf{e}_j \rangle \mathbf{v}_i = \sum_{i=1}^k \overline{v_{ji}} \mathbf{v}_i,$$

\mathbf{V}^* is the representation matrix of the orthogonal projection onto $\text{range}(\mathbf{A}^*)$ on the standard basis of \mathbb{K}^n and on the basis $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ of $\text{range}(\mathbf{A}^*)$. On the other hand, the embedding mapping of $\text{range}(\mathbf{A})$ into \mathbb{K}^m is represented by \mathbf{W} on the basis $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_k\}$ of $\text{range}(\mathbf{A})$ and on the standard basis $\{\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_m\}$ of \mathbb{K}^m , because

$$\mathbf{w}_j = \sum_{i=1}^m \langle \mathbf{f}_i \mid \mathbf{w}_j \rangle \mathbf{f}_i = \sum_{i=1}^m w_{ij} \mathbf{f}_i.$$

By these results, we have

$$\mathbf{A} = \mathbf{W} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} \mathbf{V}^*.$$

Similarly, we have

$$\mathbf{A}^\dagger = \mathbf{V} \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_k} \end{bmatrix} \mathbf{W}^*.$$

Theorem 10.1 (Singular value decomposition) *Let \mathbf{A} be a matrix of shape (m, n) and let*

$$\sigma_1^2 \geq \sigma_2^2 \geq \cdots \geq \sigma_k^2 > 0 \quad (\sigma_i \text{ are positive real numbers})$$

be the nonzero eigenvalues of $\mathbf{A}^ \mathbf{A}$. Then, there exist a unitary matrix \mathbf{U}_1 of order m , a unitary matrix \mathbf{U}_2 of order n and a matrix Σ of shape (m, n) such that*

$$\mathbf{A} = \mathbf{U}_1 \Sigma \mathbf{U}_2, \quad \Sigma = \left[\begin{array}{ccc|cc} \sigma_1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & \sigma_k & 0 & \cdots & 0 \\ \hline 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \cdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 \end{array} \right].$$

We call $\sigma_1, \sigma_2, \dots, \sigma_k$ the singular values of \mathbf{A} .

Proof Let $\{\mathbf{w}_{k+1}, \mathbf{w}_{k+2}, \dots, \mathbf{w}_m\}$ be an orthonormal basis of $\text{kernel}(\mathbf{A}^*)$. From this, together with the orthonormal basis $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_k\}$ of $\text{range}(\mathbf{A})$, we have an orthonormal basis of \mathbb{K}^m and a unitary matrix

$$\mathbf{U}_1 = [\mathbf{w}_1 \ \mathbf{w}_2 \ \cdots \ \mathbf{w}_k \ \mathbf{w}_{k+1} \ \cdots \ \mathbf{w}_m].$$

On the other hand, from the orthonormal basis of \mathbb{K}^n obtained by joining the orthonormal basis $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ of range(A^*) and an orthonormal basis $\{\mathbf{v}_{k+1}, \mathbf{v}_{k+2}, \dots, \mathbf{v}_n\}$ of kernel(A), we have the unitary matrix

$$\mathbf{U}_2^* = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_k \ \mathbf{v}_{k+1} \ \cdots \ \mathbf{v}_n]$$

With these \mathbf{U}_1 and \mathbf{U}_2 , we have the desired equality above. ■

From the singular value decomposition $\mathbf{A} = \mathbf{U}_1 \boldsymbol{\Sigma} \mathbf{U}_2$ of \mathbf{A} , we have the singular value decomposition $\mathbf{A}^* = \mathbf{U}_2^* \boldsymbol{\Sigma}^* \mathbf{U}_1^*$ of \mathbf{A}^* . Hence, we have

$$\mathbf{A}^* \mathbf{A} = \mathbf{U}_2^* (\boldsymbol{\Sigma}^* \boldsymbol{\Sigma}) \mathbf{U}_2, \quad \mathbf{A} \mathbf{A}^* = \mathbf{U}_1 (\boldsymbol{\Sigma} \boldsymbol{\Sigma}^*) \mathbf{U}_1^*.$$

Because the positive diagonal elements of $\boldsymbol{\Sigma}^* \boldsymbol{\Sigma}$ and $\boldsymbol{\Sigma} \boldsymbol{\Sigma}^*$ coincide, so do the singular values of \mathbf{A} and \mathbf{A}^* . The transpose of the matrix obtained from $\boldsymbol{\Sigma}$ by replacing each singular values σ_i by σ_i^{-1} is the generalized inverse $\boldsymbol{\Sigma}^\dagger$ of $\boldsymbol{\Sigma}$, and we have the singular value decomposition

$$\mathbf{A}^\dagger = \mathbf{U}_2^* \boldsymbol{\Sigma}^\dagger \mathbf{U}_1^*$$

of \mathbf{A}^\dagger .

Suppose that \mathbf{A} is a square matrix. With the singular value decomposition $\mathbf{A} = \mathbf{U}_1 \boldsymbol{\Sigma} \mathbf{U}_2$ of \mathbf{A} , define

$$[\mathbf{A}] \stackrel{\text{def}}{=} \mathbf{U}_2^* \boldsymbol{\Sigma} \mathbf{U}_2.$$

Then $[\mathbf{A}]$ (absolute value matrix) is a positive semi-definite matrix and because $\boldsymbol{\Sigma}^* = \boldsymbol{\Sigma}$, we have

$$[\mathbf{A}]^2 = (\mathbf{U}_2^* \boldsymbol{\Sigma} \mathbf{U}_2)^2 = \mathbf{U}_2^* \boldsymbol{\Sigma}^2 \mathbf{U}_2 = \mathbf{A}^* \mathbf{A}.$$

Hence, $[\mathbf{A}] = \sqrt{\mathbf{A}^* \mathbf{A}}$. Letting $\mathbf{V} \stackrel{\text{def}}{=} \mathbf{U}_1 \mathbf{U}_2$, we have the following equalities:

$$1. \mathbf{A} = \mathbf{V} [\mathbf{A}], \quad 2. [\mathbf{A}] = \mathbf{V}^* \mathbf{A}, \quad 3. [\mathbf{A}^*] = \mathbf{V} [\mathbf{A}] \mathbf{V}^*, \quad 4. \mathbf{A}^* = \mathbf{V}^* [\mathbf{A}^*].$$

Formula 1 is considered to be a generalization of the polar expression $z = e^{i\theta} |z|$ of a complex number z to a matrix \mathbf{A} .

Exercise 10.1 Prove the above equalities 1–4. (Hint: use the singular decomposition $\mathbf{A}^* = \mathbf{U}_2^* \boldsymbol{\Sigma} \mathbf{U}_1^*$.)

NumPy provides functions for finding the Moore–Penrose generalized inverse and the singular value decomposition.

Program: svd1.py

```
In [1]: 1 from numpy import array, diag, zeros
2 from numpy.linalg import pinv, svd
3
4 A = array([[1, 2], [3, 4], [5, 6], [7, 8]])
5 print(pinv(A))
6 U1, S, U2 = svd(A)
7 Z = zeros((4, 2))
8 Z[:, :2] = diag(S)
9 print(U1.dot(Z.dot(U2)))
```

Line 1: Given a one-dimensional array, the function `diag` returns the diagonal matrix whose diagonal elements are the components of an array.

Line 2: The Moore–Penrose generalized inverse `pinv` and the singular value decomposition `svd` are defined in the module `linalg`.

Line 4: Consider a matrix A in $M_{\mathbb{R}}(4, 2)$.

Line 5: Calculate A^\dagger .

Line 6: `svd(A)` returns the tuple of three arrays, the array for the unitary matrix U_1 , one-dimensional array of the singular values and the array for the unitary matrix U_2 .

Lines 7, 8: Make the matrix with the singular values on the diagonal corresponding to Σ .

Line 9: Calculate $U_1 \Sigma U_2$ to confirm that it is equal to A .

```
[[-1.0000000e+00 -5.0000000e-01  1.54390389e-15  5.0000000e-01]
 [ 8.5000000e-01  4.5000000e-01  5.0000000e-02 -3.5000000e-01]]
 [[1.  2.]
 [3.  4.]
 [5.  6.]
 [7.  8.]]]
```

For a square matrix A , we define

$$\|A\|_{\text{Tr}} \stackrel{\text{def}}{=} \text{Tr}(\|A\|).$$

$\|A\|_{\text{Tr}}$ is called the *trace norm* of A .

Exercise 10.2 Prove the following properties:

1. $\|A\|_{\text{Tr}} = \sum_{i=1}^k \sigma_i$, where $\sigma_1, \sigma_2, \dots, \sigma_k$ are all of the singular values of A .
2. $\|A\|_{\text{Tr}} \geq 0$ and the equality holds if and only if $A = \mathbf{0}$
3. $\|cA\|_{\text{Tr}} = |c| \|A\|_{\text{Tr}}$.
4. $\|A + B\|_{\text{Tr}} \leq \|A\|_{\text{Tr}} + \|B\|_{\text{Tr}}$.

For square matrices A and B of order n , we define

$$\langle A \mid B \rangle_{\text{HS}} \stackrel{\text{def}}{=} \text{Tr}(A^* B)$$

and

$$\|A\|_{\text{HS}} \stackrel{\text{def}}{=} \sqrt{\text{Tr}(A^* A)}.$$

Exercise 10.3 Prove the following properties:

1. $\langle \cdot \mid \cdot \rangle_{\text{HS}}$ is an inner product on the space of all square matrices of order n .
2. $\|A\|_{\text{HS}} = \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}$, where a_{ij} is the (i, j) -element of A .
3. $\|A\|_{\text{HS}} = \left(\sum_{i=1}^k |\sigma_i|^2 \right)^{1/2}$, where $\sigma_1, \sigma_2, \dots, \sigma_k$ are all of the singular values of A . $\|A\|_{\text{HS}}$ is called the *Hilbert–Schmidt norm* of A .

Exercise 10.4 Prove that the maximum singular value of square matrix A equals the spectral radius (the matrix norm) of A . (Hint: use $\|A^*A\| = \|A\|^2$ shown in Sect. 7.4.)

The module `linalg` of NumPy has a function that computes various types of matrix norms, including the ones mentioned above, so let us use them.

Program: svd2.py

```
In [1]: 1 from numpy import array, sqrt, trace, diag, linalg
2
3 A = array([[1, 2], [3, 4]])
4 U, S, V = linalg.svd(A)
5 A1 = V.T.dot(diag(S).dot(V))
6
7 print(trace(A1))
8 print(S.sum())
9 print(linalg.norm(A, ord='nuc'))
10
11 print(sqrt(trace(A.T.dot(A))))
12 print(sqrt((A**2).sum()))
13 print(linalg.norm(A, ord='fro'))
14
15 print(S.max() / S.min())
16 print(linalg.cond(A))
17 B = linalg.inv(A)
18 print(linalg.norm(A, ord=2) * linalg.norm(B, ord=2))
```

Line 5: Calculates $\|A\|$.

Lines 7–9: Calculate the trace norm of A in three different ways. The norm is also called the *nuclear norm*.

Lines 11–13: Calculate the Hilbert–Schmidt norm of A in three different ways. The norm is also called the *Frobenius norm*.

Lines 15–18: Calculate the *condition number*⁴ of A in three different ways. The condition number is used in numerical analysis as one of the parameters that indicate the difficulty of matrix calculation. `linalg.norm(A, ord=2)` calculates the maximum singular value of A .

```
5.830951894845301
5.8309518948453
5.8309518948453
5.477225575051661
5.477225575051661
5.477225575051661
14.933034373659265
14.933034373659265
14.93303437365925
```

10.3 Tensor Products

In this section, we introduce the notion of a tensor product of vectors and of linear spaces. For that purpose, we need some preparation.

⁴ About the condition number, see [15] or [30] of the Bibliography.

Let V be a linear space over \mathbb{K} and W a subspace of V . For x and y in V , we write $x \simeq_W y$, if the difference $x - y$ belongs to W . Then, \simeq_W is an *equivalence relation* on V in the sense that it satisfies the following properties:

1. **reflexive:** $x \simeq_W x$ for all $x \in V$,
2. **symmetric:** $x \simeq_W y \Rightarrow y \simeq_W x$ for all $x, y \in V$,
3. **transitive:** $x \simeq_W y$ and $y \simeq_W z \Rightarrow x \simeq_W z$ for all $x, y, z \in V$.

Exercise 10.5 Prove the above properties 1–3 of \simeq_W .

For $x \in V$ we define

$$[x]_W \stackrel{\text{def}}{=} \{y \in V \mid x \simeq_W y\},$$

and call this subset of V the *equivalence class* of x with respect to the equivalence relation \simeq_W . Note that $[x]_W = [y]_W$ if and only if $x \simeq_W y$ if and only if $y \in [x]_W$. The element x is called a *representative* of the class $[x]_W$. Any vector in $[x]_W$ can be a representative. The family

$$V/W \stackrel{\text{def}}{=} \{[x]_W \mid x \in V\}.$$

of all different equivalence classes is called the *quotient set* of V by W with respect to \simeq_W .

The quotient set V/W has a linear space structure in a natural way, namely, the addition is given by

$$[x]_W + [y]_W \stackrel{\text{def}}{=} [x + y]_W \quad (x, y \in V)$$

and the scalar multiplication is given by

$$a[x]_W \stackrel{\text{def}}{=} [ax]_W \quad (a \in \mathbb{K}, x \in V).$$

We call V/W the *quotient linear space* of the linear space V by its linear subspace W .

Exercise 10.6 Show that the addition and scalar multiplication above are well-defined, that is, they do not depend on the choice of representatives x and y of the classes.

Now, we will introduce the tensor product of linear spaces. Let V and W be linear spaces over \mathbb{K} , and let \mathcal{L} be the set of all merely symbolical linear combinations of elements of $V \times W$:

$$\sum_{i=1}^k a_i (x_i, y_i) = a_1 (x_1, y_1) + a_2 (x_2, y_2) + \cdots + a_k (x_k, y_k),$$

with $a_1, a_2, \dots, a_k \in \mathbb{K}$ and $(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k) \in V \times W$. \mathcal{L} is a very large infinite dimensional linear space over \mathbb{K} . Indeed, any distinct elements $(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)$ of $V \times W$ are linearly independent. Next, let \mathcal{K} be the linear subspace of \mathcal{L} generated by all vectors

$$\begin{aligned} (\mathbf{x}_1, \mathbf{y}) + (\mathbf{x}_2, \mathbf{y}) &= (\mathbf{x}_1 + \mathbf{x}_2, \mathbf{y}), \\ (\mathbf{x}, \mathbf{y}_1) + (\mathbf{x}, \mathbf{y}_2) &= (\mathbf{x}, \mathbf{y}_1 + \mathbf{y}_2), \\ a(\mathbf{x}, \mathbf{y}) &= (a\mathbf{x}, \mathbf{y}), \\ a(\mathbf{x}, \mathbf{y}) &= (\mathbf{x}, a\mathbf{y}) \end{aligned}$$

for $a \in \mathbb{K}$, $\mathbf{x}, \mathbf{x}_1, \mathbf{x}_2 \in V$ and $\mathbf{y}, \mathbf{y}_1, \mathbf{y}_2 \in W$. We define the *tensor product linear space* $V \otimes_{\mathbb{K}} W$ (or simply $V \otimes W$) of V and W to be the quotient space \mathcal{L}/\mathcal{K} of \mathcal{L} by \mathcal{K} . The equivalence class of $(\mathbf{x}, \mathbf{y}) \in V \times W$ is an element of $V \otimes W$ and is denoted by $\mathbf{x} \otimes \mathbf{y}$. By the way of construction, the following equalities are satisfied in $V \otimes W$:

$$\begin{aligned} (\mathbf{x}_1 \otimes \mathbf{y}) + (\mathbf{x}_2 \otimes \mathbf{y}) &= (\mathbf{x}_1 + \mathbf{x}_2) \otimes \mathbf{y}, \\ (\mathbf{x} \otimes \mathbf{y}_1) + (\mathbf{x} \otimes \mathbf{y}_2) &= \mathbf{x} \otimes (\mathbf{y}_1 + \mathbf{y}_2), \\ a(\mathbf{x} \otimes \mathbf{y}) &= (a\mathbf{x}) \otimes \mathbf{y}, \\ a(\mathbf{x} \otimes \mathbf{y}) &= \mathbf{x} \otimes (a\mathbf{y}). \end{aligned}$$

Assume that V and W are finite-dimensional spaces of dimension m and n with bases $X = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m\}$ and $Y = \{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_n\}$, respectively. Then, using the above equalities, for any elements $\mathbf{x} = \sum_{i=1}^m a_i \mathbf{v}_i \in V$ ($a_i \in \mathbb{K}$) and $\mathbf{y} = \sum_{j=1}^n b_j \mathbf{w}_j \in W$ ($b_j \in \mathbb{K}$), we have

$$\mathbf{x} \otimes \mathbf{y} = \sum_{ij} a_i b_j (\mathbf{v}_i \otimes \mathbf{w}_j).$$

Thus, $Z = \{\mathbf{v}_i \otimes \mathbf{w}_j | i = 1, 2, \dots, m, j = 1, 2, \dots, n\}$ generates $V \otimes W$. Suppose that V and W are equipped with inner products $\langle \cdot | \cdot \rangle_V$ and $\langle \cdot | \cdot \rangle_W$, respectively, and that X and Y are orthonormal bases with respect to these inner products. We define a function $\varphi : \mathcal{L} \times \mathcal{L} \rightarrow \mathbb{K}$ by

$$\varphi(\xi, \eta) = \sum_{i=1}^k \sum_{j=1}^l \bar{a}_i b_j \langle \mathbf{x}_i | \mathbf{x}'_j \rangle_V \langle \mathbf{y}_i | \mathbf{y}'_j \rangle_W$$

for $\xi = \sum_{i=1}^k a_i (\mathbf{x}_i, \mathbf{y}_i)$, $\eta = \sum_{j=1}^l b_j (\mathbf{x}'_j, \mathbf{y}'_j) \in \mathcal{L}$ ($a_i, b_j \in \mathbb{K}$, $\mathbf{x}_i, \mathbf{x}'_j \in V$, $\mathbf{y}_i, \mathbf{y}'_j \in W$). Then we can easily verify that it is a Hermitian homogeneous linear form, that is,

$$\varphi(\xi, a\eta + b\xi) = a\varphi(\xi, \eta) + b\varphi(\xi, \xi) \quad \text{and} \quad \varphi(\eta, \xi) = \overline{\varphi(\xi, \eta)}$$

for all $a, b \in \mathbb{K}$ and $\xi, \eta, \zeta \in \mathcal{L}$. Moreover, for any generator element ξ of \mathcal{K} and for any $\eta \in \mathcal{L}$, we see $\varphi(\xi, \eta) = \varphi(\eta, \xi) = 0$. For example, for $\xi = (\mathbf{x}_1, \mathbf{y}) + (\mathbf{x}_2, \mathbf{y}) - (\mathbf{x}_1 + \mathbf{x}_2, \mathbf{y})$ we see

$$\varphi(\xi, \eta) = \varphi((\mathbf{x}_1, \mathbf{y}), \eta) + \varphi((\mathbf{x}_2, \mathbf{y}), \eta) - \varphi((\mathbf{x}_1 + \mathbf{x}_2, \mathbf{y}), \eta) = 0.$$

It follows that $\varphi(\mathcal{K} \times \mathcal{L}) = \varphi(\mathcal{L} \times \mathcal{K}) = \{\mathbf{0}\}$. Hence, φ induces a function $\langle \cdot | \cdot \rangle$ on $V \otimes W$ defined by

$$\langle \xi | \eta \rangle = \sum_{i=1}^k \sum_{j=1}^l \bar{a}_i b_j \langle \mathbf{x}_i | \mathbf{x}'_j \rangle_V \langle \mathbf{y}_i | \mathbf{y}'_j \rangle_W$$

for $\xi = \sum_{i=1}^k a_i (\mathbf{x}_i \otimes \mathbf{y}_i)$, $\eta = \sum_{j=1}^l b_j (\mathbf{x}'_j \otimes \mathbf{y}'_j) \in V \otimes W$.

Exercise 10.7 Prove the results discussed above in detail.

We shall show $\langle \cdot | \cdot \rangle$ is actually positive. For any element $\xi = \sum_{i=1}^m \sum_{j=1}^n c_{ij} (\mathbf{v}_i \otimes \mathbf{w}_j)$ of $V \otimes W$, we have

$$\begin{aligned}\langle \xi | \xi \rangle &= \sum_{i=1}^m \sum_{j=1}^n \sum_{i'=1}^m \sum_{j'=1}^n \overline{c_{ij}} c_{i'j'} \langle \mathbf{v}_i | \mathbf{v}_{i'} \rangle_V \langle \mathbf{w}_j | \mathbf{w}_{j'} \rangle_W \\ &= \sum_{i=1}^m \sum_{j=1}^n \overline{c_{ij}} c_{ij} \langle \mathbf{v}_i | \mathbf{v}_i \rangle_V \langle \mathbf{w}_j | \mathbf{w}_j \rangle_W \\ &= \sum_{i=1}^m \sum_{j=1}^n |c_{ij}|^2 \geq 0.\end{aligned}$$

Here, $\langle \xi | \xi \rangle = 0$ if and only if $c_{ij} = 0$ for all i, j if and only if $\xi = \mathbf{0}$. Thus, we find that $\langle \cdot | \cdot \rangle$ is positive and it becomes an inner product on $V \otimes W$. Because

$$\langle \mathbf{v}_i \otimes \mathbf{w}_j | \mathbf{v}_k \otimes \mathbf{w}_l \rangle = \langle \mathbf{v}_i | \mathbf{v}_k \rangle_V \langle \mathbf{w}_j | \mathbf{w}_l \rangle_W = \delta_{ik} \delta_{jl},$$

$Z = \{\mathbf{v}_i \otimes \mathbf{w}_j\}_{ij}$ forms an orthonormal basis of $V \otimes W$. Therefore, $V \otimes W$ is an mn -dimensional space and is linearly isomorphic to \mathbb{K}^{mn} . Let $\xi = \sum_{i=1}^m \sum_{j=1}^n c_{ij} (\mathbf{v}_i \otimes \mathbf{w}_j) \subseteq V \otimes W$ correspond to the

matrix $\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{bmatrix}$, then this correspondence gives a linear isomorphism of $V \otimes W$ to the

space $M_{\mathbb{K}}(m, n)$ of all $m \times n$ matrices over \mathbb{K} . Thus, $V \otimes_{\mathbb{K}} W \cong M_{\mathbb{K}}(m, n) \cong \mathbb{K}^{mn}$. These isomorphisms preserve the inner products and so $V \otimes W$ is unitarily isomorphic to \mathbb{K}^{mn} (with the standard inner product).

Let U be another linear space over \mathbb{K} . A mapping $f : V \times W \rightarrow U$ is called *bilinear*, if it satisfies

$$\begin{aligned}f(\mathbf{x} + \mathbf{x}', \mathbf{y}) &= f(\mathbf{x}, \mathbf{y}) + f(\mathbf{x}', \mathbf{y}), \\ f(\mathbf{x}, \mathbf{y} + \mathbf{y}') &= f(\mathbf{x}, \mathbf{y}) + f(\mathbf{x}, \mathbf{y}'), \\ f(a\mathbf{x}, \mathbf{y}) &= f(\mathbf{x}, a\mathbf{y}) = af(\mathbf{x}, \mathbf{y})\end{aligned}$$

for any $\mathbf{x}, \mathbf{x}' \in V$, $\mathbf{y}, \mathbf{y}' \in W$ and $a \in \mathbb{K}$. Define a linear mapping $\bar{f} : V \otimes W \rightarrow U$ by $\bar{f}(\mathbf{v}_i \otimes \mathbf{w}_j) = f(\mathbf{v}_i, \mathbf{w}_j)$ for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$. Then, for any $\mathbf{x} = \sum_{i=1}^m a_i \mathbf{v}_i \in V$ ($a_i \in \mathbb{K}$) and

$\mathbf{y} = \sum_{j=1}^n b_j \mathbf{w}_j \in W$ we have

$$\overline{f}(\mathbf{x} \otimes \mathbf{y}) = \overline{f}\left(\sum_{ij} a_i b_j (\mathbf{v}_i \otimes \mathbf{w}_j)\right) = \sum_{ij} a_i b_j f(\mathbf{v}_i, \mathbf{w}_j) = f(\mathbf{x}, \mathbf{y}).$$

In this way, from a bilinear mapping f , the linear mapping \overline{f} on $V \otimes W$ satisfying $\overline{f}(\mathbf{x} \otimes \mathbf{y}) = f(\mathbf{x}, \mathbf{y})$ is induced.

In the rest of this section, we only discuss the tensor product $\mathbb{K}^m \otimes \mathbb{K}^n$. We consider the inner product on $\mathbb{K}^m \otimes \mathbb{K}^n$ defined above with the standard inner products on \mathbb{R}^m and \mathbb{R}^n . Then, $\{e_i \otimes f_j \mid i = 1, 2, \dots, m, j = 1, 2, \dots, n\}$ is an orthonormal basis of $\mathbb{K}^m \otimes \mathbb{K}^n$, where $\{e_i \mid i = 1, 2, \dots, m\}$ and $\{f_j \mid j = 1, 2, \dots, n\}$ are standard bases of \mathbb{K}^m and \mathbb{K}^n , respectively. We call it the *standard basis* of $\mathbb{K}^m \otimes \mathbb{K}^n$. In this situation, the unitary isomorphism from $\mathbb{K}^m \otimes \mathbb{K}^n$ to $M_{\mathbb{K}}(m, n)$ is given by

$$\mathbf{x} \otimes \mathbf{y} \mapsto \mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \cdots & x_m y_n \end{bmatrix}.$$

for column vectors $\mathbf{x} \in \mathbb{K}^m$ and $\mathbf{y} \in \mathbb{K}^n$. We usually identify the tensor product $\mathbf{x} \otimes \mathbf{y}$ with the matrix in the left-hand side above, and we write $\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}^T$.

The following dialogs show how to handle the tensor product of vectors in Python.

```
In [1]: x = [1, 2]
y = [3, 4, 5]
[[a * b for b in y] for a in x]
```

```
Out[1]: [[3, 4, 5], [6, 8, 10]]
```

```
In [2]: from numpy import dot, reshape, outer, tensordot
dot(reshape(x, (2, 1)), reshape(y, (1, 3)))
```

```
Out[2]: array([[ 3,  4,  5],
   [ 6,  8, 10]])
```

```
In [3]: outer(x, y)
```

```
Out[3]: array([[ 3,  4,  5],
   [ 6,  8, 10]])
```

```
In [4]: tensordot(x, y, axes=0)
```

```
Out[4]: array([[ 3,  4,  5],
   [ 6,  8, 10]])
```

In [1] is an example of using list comprehension. In NumPy, we have a couple of ways. One is a way of using the `dot` function. In this case, in order to consider tensor product of vectors as a matrix product, we must represent \mathbf{x} as a column vector and \mathbf{y} as a row vector. Alternatively, NumPy prepares the functions `outer`⁵ and `tensordot`⁶ for calculating tensor product of vectors.

In SymPy, it can be expressed literally as $\mathbf{x}\mathbf{y}^T$.

⁵Note that the operation of tensor product of vectors is also called an *outer product*.

⁶For a tensor product of vectors, we set the name argument `axes=0`.

```
In [1]: from sympy import Matrix
x = Matrix([1, 2])
y = Matrix([3, 4, 5])
x * y.T
```

```
Out[1]: Matrix([
[3, 4, 5],
[6, 8, 10]])
```

Let \mathbf{A} and \mathbf{B} be square matrices of order m and n , respectively. Then the mapping $f : \mathbb{K}^m \times \mathbb{K}^n \rightarrow \mathbb{K}^m \otimes \mathbb{K}^n$ defined by

$$f(\mathbf{x}, \mathbf{y}) = (\mathbf{Ax}) \otimes (\mathbf{By})$$

for $(\mathbf{x}, \mathbf{y}) \in \mathbb{K}^m \times \mathbb{K}^n$ is bilinear as easily seen. Hence, f induces the linear mapping $\bar{f} : \mathbb{K}^m \otimes \mathbb{K}^n \rightarrow \mathbb{K}^m \otimes \mathbb{K}^n$ with $\bar{f}(\mathbf{x} \otimes \mathbf{y}) = \mathbf{Ax} \otimes \mathbf{By}$. Its representation matrix on the standard basis is the square matrix

$$\begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1m}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mm}\mathbf{B} \end{bmatrix}$$

of order mn . We denote this matrix by $\mathbf{A} \otimes \mathbf{B}$ and call it the *Kronecker product* of \mathbf{A} and \mathbf{B} . We have

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{x} \otimes \mathbf{y}) = \mathbf{Ax} \otimes \mathbf{By} = \mathbf{Ax}\mathbf{y}^T\mathbf{B}^T$$

for any $\mathbf{x} \in \mathbb{K}^m$ and $\mathbf{y} \in \mathbb{K}^n$.

For column vectors $\mathbf{x} \in \mathbb{K}^m$ and $\mathbf{y} \in \mathbb{K}^n$, we consider the $m \times n$ matrix

$$\mathbf{x} \otimes \bar{\mathbf{y}} = \mathbf{x}\mathbf{y}^* = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \begin{bmatrix} \bar{y}_1 & \bar{y}_2 & \cdots & \bar{y}_n \end{bmatrix} = \begin{bmatrix} x_1\bar{y}_1 & x_1\bar{y}_2 & \cdots & x_1\bar{y}_n \\ x_2\bar{y}_1 & x_2\bar{y}_2 & \cdots & x_2\bar{y}_n \\ \vdots & \vdots & & \vdots \\ x_m\bar{y}_1 & x_m\bar{y}_2 & \cdots & x_m\bar{y}_n \end{bmatrix}.$$

We call $\mathbf{x} \otimes \bar{\mathbf{y}}$ the *Schatten product* of \mathbf{x} and \mathbf{y} . For any $\mathbf{v} \in \mathbb{K}^n$, we have

$$(\mathbf{x} \otimes \bar{\mathbf{y}})\mathbf{v} = \mathbf{x}\mathbf{y}^*\mathbf{v} = \langle \mathbf{y} \mid \mathbf{v} \rangle \mathbf{x}.$$

In particular, if $m = n$ and $\mathbf{e} \in \mathbb{K}^n$ is a norm-one vector, then

$$(\mathbf{e} \otimes \bar{\mathbf{e}})\mathbf{v} = \langle \mathbf{e} \mid \mathbf{v} \rangle \mathbf{e}$$

for any $\mathbf{v} \in \mathbb{K}^n$. Hence, $\mathbf{e} \otimes \bar{\mathbf{e}}$ is nothing but the orthogonal projection onto the one-dimensional subspace $\langle \mathbf{e} \rangle$. More generally, the Fourier expansion of $\mathbf{v} \in \mathbb{K}^n$ can be expressed as

$$\mathbf{v} = \sum_{i=1}^n (\mathbf{e}_i \otimes \bar{\mathbf{e}}_i) \mathbf{v}$$

for an orthonormal basis $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$ of \mathbb{K}^n .

Let A be a normal matrix of order n . As we proved in Sect. 7.3, there is an orthonormal basis $\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n$ such that A is diagonalized as $\mathbf{U}^* \mathbf{A} \mathbf{U} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ with the unitary matrix $\mathbf{U} = [\mathbf{e}_1 \ \mathbf{e}_2 \ \cdots \ \mathbf{e}_n]$. Hence, A can be expressed as

$$\mathbf{A} = \sum_{i=1}^n \lambda_i \mathbf{e}_i \otimes \overline{\mathbf{e}_i},$$

which is called the *spectral decomposition* of A .

For a general $m \times n$ matrix A , we also have an equivalent expression of the singular value decomposition in Theorem 10.1 as follows:

$$\mathbf{A} = \sum_{i=1}^k \sigma_i \mathbf{w}_i \otimes \overline{\mathbf{v}_i},$$

where $\{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_k\}$ and $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ are orthonormal bases of $\text{range}(A)$ and $\text{range}(A^*)$, respectively.

Exercise 10.8 Find the spectral decomposition of $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ and the singular value decomposition of $B = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$. Use NumPy or SymPy if necessary.

For three linear spaces U , V , and W over \mathbb{K} , we can construct the *three-fold tensor product* $U \otimes V \otimes W$ in the same way as the tensor product $U \otimes V$ (the details are omitted). An element (u, v, w) of $U \times V \times W$ corresponds to an element of $U \otimes V \otimes W$ expressed by $\mathbf{u} \otimes \mathbf{v} \otimes \mathbf{w}$. If $\{\mathbf{u}_i\}$, $\{\mathbf{v}_j\}$ and $\{\mathbf{w}_k\}$ are bases of U , V , and W respectively, then $\{\mathbf{u}_i \otimes \mathbf{v}_j \otimes \mathbf{w}_k\}$ is a basis of $U \otimes V \otimes W$. For any elements $\mathbf{x} = \sum_i x_i \mathbf{u}_i \in U$, $\mathbf{y} = \sum_j y_j \mathbf{v}_j \in V$ and $\mathbf{z} = \sum_k z_k \mathbf{w}_k \in W$, we have

$$\mathbf{x} \otimes \mathbf{y} \otimes \mathbf{z} = \sum_{i,j,k} x_i y_j z_k (\mathbf{u}_i \otimes \mathbf{v}_j \otimes \mathbf{w}_k).$$

There are the natural isomorphisms

$$U \otimes V \otimes W \cong (U \otimes V) \otimes W \cong U \otimes (V \otimes W).$$

In the context of quantum mechanics, we express a row vector and a column vector that are adjoint with each other as

$$\langle i | = [\overline{x_1} \ \overline{x_2} \ \cdots \ \overline{x_n}] \quad \text{and} \quad | i \rangle = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

and we call them a *bra-vector* and a *ket-vector* respectively.⁷ We consider a finite quantum system in which a Hermitian matrix \mathbf{A} of order n called an *observable* is given. Let $\{|1\rangle, |2\rangle, \dots, |n\rangle\}$ be an orthonormal basis of \mathbb{K}^n . The spectral decomposition of \mathbf{A} can be expressed as

$$\mathbf{A} = \sum_{i=1}^n \lambda_i |i\rangle\langle i|$$

with $\lambda_1, \lambda_2, \dots, \lambda_n \in \mathbb{R}$, where $|i\rangle\langle i| = |i\rangle \otimes \overline{|i\rangle}$ are the Schatten products. When λ_i is the outcome of observable \mathbf{A} , we know that the system is in state $|i\rangle$ called a *pure state* for \mathbf{A} . A norm-one vector $|\rho\rangle \in \mathbb{C}^n$ is called a *mixed state*. Its Fourier expansion

$$|\rho\rangle = \rho_1 |1\rangle + \rho_2 |2\rangle + \dots + \rho_n |n\rangle,$$

is called a *quantum superposition* of pure states for \mathbf{A} . By the Riesz–Fischer identity, we have $\sum_{i=1}^n |\rho_i|^2 = 1$, so the power spectrum $(|\rho_1|^2, |\rho_2|^2, \dots, |\rho_n|^2)$ of $|\rho\rangle$ is a probability distribution. When the system is in the mixed state, we consider that the outcome λ_i is obtained with probability $|\rho_i|^2$. The tuple $(\rho_1, \rho_2, \dots, \rho_n)$ of Fourier coefficients is called a *probability amplitude*, distinguishing it from the probability distribution. When the system is in the mixed state $|\rho\rangle$, the expectation of the outcome obtained by observing \mathbf{A} is calculated by

$$\langle \rho | \mathbf{A} | \rho \rangle = \langle \rho | \left(\sum_{i=1}^n \lambda_i |i\rangle\langle i| \right) | \rho \rangle = \sum_{i=1}^n \lambda_i \langle \rho | |i\rangle\langle i| | \rho \rangle = \sum_{i=1}^n \lambda_i |\rho_i|^2.$$

Let $\{|1\rangle, |2\rangle, \dots, |n\rangle\}$ be the standard basis of \mathbb{C}^n and $\{|1'\rangle, |2'\rangle, \dots, |n'\rangle\}$ the orthonormal basis of \mathbb{C}^n defined by

$$|1'\rangle = \begin{bmatrix} u_{11} \\ u_{21} \\ \vdots \\ u_{n1} \end{bmatrix}, \quad |2'\rangle = \begin{bmatrix} u_{12} \\ u_{22} \\ \vdots \\ u_{n2} \end{bmatrix}, \quad \dots, \quad |n'\rangle = \begin{bmatrix} u_{1n} \\ u_{2n} \\ \vdots \\ u_{nn} \end{bmatrix},$$

where

$$u_{ij} = \frac{1}{\sqrt{n}} e^{2\pi\sqrt{-1}(i-1)(j-1)/n} \quad (i, j = 1, 2, \dots, n).$$

We consider two observables \mathbf{P} and \mathbf{Q} defined by

$$\mathbf{P} \stackrel{\text{def}}{=} \sum_{i=1}^n \lambda_i |i\rangle\langle i| \quad \text{and} \quad \mathbf{Q} \stackrel{\text{def}}{=} \sum_{j=1}^n \mu_j |j'\rangle\langle j'|$$

for some $\lambda_1, \lambda_2, \dots, \lambda_n, \mu_1, \mu_2, \dots, \mu_n \in \mathbb{R}$. A pure state for observable \mathbf{P} is a mixed state for observable \mathbf{Q} as

$$|i\rangle = \sum_{j=1}^n \langle j' | |i\rangle |j'\rangle = \sum_{j=1}^n \overline{u_{ij}} |j'\rangle.$$

⁷ These are invented by physicists to express the inner product and the Schatten product.

When we observe \mathbf{P} and the outcome is λ_i , the system is in the state $|i\rangle$, and in this state, the expectation by observing \mathbf{Q} is calculated by

$$\langle i | \mathbf{Q} | i \rangle = \sum_{j=1}^n \mu_j |\overline{u_{ij}}|^2 = \frac{1}{n} \sum_{j=1}^n \mu_j.$$

Hence, all μ_i are observed with same probability. This says that, even if we exactly know that the system is in state $|i\rangle$ by observing \mathbf{P} , we cannot exactly know which state of $|j\rangle$'s the system is in by observing \mathbf{Q} , and vice versa. This fact is called the *uncertainty principle* in quantum mechanics. A more authentic theory of quantum measurement is based on the theory of Hilbert spaces and operator algebras on them, whereas probability theory is based on measure theory and Lebesgue integrals, but both of them exceed the scope of this book.

The two observables \mathbf{P} and \mathbf{Q} do not necessarily commute, unlike random variables dealt with in the next section.

Exercise 10.9 Calculate $\mathbf{P}\mathbf{Q} - \mathbf{Q}\mathbf{P}$ in the case of $n = 2$, $\lambda_1 = \mu_1$, $\lambda_2 = \mu_2$.

10.4 Tensor Product Representation of Vector-Valued Random Variables

In order to formulate the theory of probability in a mathematically rigorous way in the Kolmogorov style, we need to discuss it within the framework of measure theory. On a general measure space, we need the Lebesgue integration to compute the expectation $E(X)$ of a random variable X , for example. Because we have not prepared for that here, we only explain it in a finite probability space $\{\omega_1, \omega_2, \dots, \omega_n\}$, where the expectation is the finite summation $\sum_{i=1}^n X(\omega_i) p(\omega_i)$. However, the algebraic structures of random variables are basically not different in both settings.

Let us consider a *finite (discrete) probability space* (Ω, p) , where Ω is a non-empty finite set and p is a function on Ω valued in $[0, 1]$ satisfying $\sum_{\omega \in \Omega} p(\omega) = 1$. We call Ω a *sample space* and p a *probability distribution*. A subset A of Ω is called an *event*, and the *probability of occurrence* of A is defined by

$$P(A) \stackrel{\text{def}}{=} \sum_{\omega \in A} p(\omega).$$

Then, P is a function from 2^Ω to $[0, 1]$ and is called a *probability measure* on (Ω, p) . A real function X on Ω is called a *random variable* on (Ω, p) . The value $X(\omega)$ for $\omega \in \Omega$ is called the *observed value* of X when ω occurs. The *expectation (expected value)* $E(X)$ of X is calculated by

$$E(X) \stackrel{\text{def}}{=} \sum_{\omega \in \Omega} X(\omega) p(\omega).$$

We denote by $L(\Omega, p)$ the set of all random variables on (Ω, p) . Then, $L(\Omega, p) = \mathbb{R}^\Omega$ is a real linear space in the usual way (Sect. 2.1), and the expectation E is a linear mapping of $L(\Omega, p)$ into \mathbb{R} .

We consider a product XY of random variables X and Y defined by pointwise product $\omega \mapsto X(\omega)Y(\omega)$. Furthermore, we equip $L(\Omega, p)$ with an inner product defined by

$$\langle X \mid Y \rangle \stackrel{\text{def}}{=} E(XY)$$

for $X, Y \in L(\Omega, p)$ and a norm defined by

$$\|X\| = \sqrt{\langle X \mid X \rangle} = \sqrt{E(X^2)}$$

for $X \in L(\Omega, p)$.⁸

For a linear space V over \mathbb{R} , a mapping $X : \Omega \rightarrow V$ is called a *vector-valued (V-valued) random variable*, and its expectation is defined by

$$E(X) \stackrel{\text{def}}{=} \sum_{\omega \in \Omega} X(\omega) p(\omega)$$

in the same way as for real-valued random variables.

We consider an experiment with rolling two dice. Consider the sample space $\Omega = \{1, \dots, 6\} \times \{1, \dots, 6\}$ and the probability distribution p given by

$$p((\omega_1, \omega_2)) = \frac{1}{36}$$

for $(\omega_1, \omega_2) \in \Omega$. Let X be an \mathbb{R}^2 -valued random variable defined by

$$X((\omega_1, \omega_2)) = \begin{bmatrix} \omega_1 + \omega_2 \\ \omega_1 - \omega_2 \end{bmatrix}$$

for $(\omega_1, \omega_2) \in \Omega$. This X represents the pair of the sum and the difference of the numbers that appear on the top faces of the dice.

We express this stochastic model in Python as faithfully as possible in the following program.

Program: probab1.py

```
In [1]: 1 from numpy import array
2 from numpy.random import randint, seed
3
4 seed(2021)
5 N = [1, 2, 3, 4, 5, 6]
6 Omega = [(w1, w2) for w1 in N for w2 in N]
7
8 def omega():
9     return Omega[randint(len(Omega))]
10
11 def P(w):
12     return 1 / len(Omega)
13
14 def X(w):
15     return array([w[0] + w[1], w[0] - w[1]])
16
17 def E(X):
18     return sum([X(w) * P(w) for w in Omega])
19
20 for n in range(5):
21     w = omega()
```

⁸This inner product is essentially the same as the inner product for functions associated with weight p discussed in Sect. 6.5.

```
In [1]: 22 |     print(X(w), end=' ')
23 | print(f'\nE(X)={E(X)}')
```

```
[8 0] [2 0] [7 5] [ 9 -1] [9 1]
E(X)=[ 7.0000000e+00 -8.32667268e-17]
```

The theoretical value of expectation of X is calculated as

$$\begin{aligned} E(X) &= \frac{1}{36} \sum_{i=1}^6 \sum_{j=1}^6 \begin{bmatrix} i+j \\ i-j \end{bmatrix} = \frac{1}{36} \left[\begin{array}{c} \sum_{i=1}^6 i \sum_{j=1}^6 1 + \sum_{i=1}^6 1 \sum_{j=1}^6 j \\ \sum_{i=1}^6 i \sum_{j=1}^6 1 - \sum_{i=1}^6 1 \sum_{j=1}^6 j \end{array} \right] \\ &= \frac{1}{36} \begin{bmatrix} 21 \times 6 + 6 \times 21 \\ 21 \times 6 - 6 \times 21 \end{bmatrix} = \begin{bmatrix} 7 \\ 0 \end{bmatrix}. \end{aligned}$$

If we consider more complicated problems such as rolling many times or using more dice, Ω must be a much larger set.⁹

Let $L(\Omega, p; V)$ denote the set of V -valued random variables on a probability space (Ω, p) . Then, $L(\Omega, p; V)$ is a real linear space in the same way as $L(\Omega, p)$. For $X \in L(\Omega, p)$ and $v \in V$, consider the mapping $Xv : \omega \mapsto X(\omega)v$. Then, Xv is an element of $L(\Omega, p; V)$, and we have the mapping $f : L(\Omega, p) \times V \rightarrow L(\Omega, p; V)$ defined by $f(X, v) = Xv$. It is easy to see that f is bilinear. So, it induces a linear mapping $\bar{f} : L(\Omega, p) \otimes V \rightarrow L(\Omega, p; V)$ satisfying $\bar{f}(X \otimes v) = Xv$. Actually, \bar{f} is an isomorphism and so $L(\Omega, p; V) \cong L(\Omega, p) \otimes V$.

Exercise 10.10 Prove that the above mapping \bar{f} is bijective.

For two probability spaces (Ω_1, p_1) and (Ω_2, p_2) , we define a probability space $(\Omega_1 \times \Omega_2, p)$ with probability distribution p defined by

$$p((\omega_1, \omega_2)) \stackrel{\text{def}}{=} p_1(\omega_1)p_2(\omega_2)$$

for $(\omega_1, \omega_2) \in \Omega_1 \times \Omega_2$. Let X be a random variable on $(\Omega_1 \times \Omega_2, p)$. For a fixed $\omega_1 \in \Omega_1$ let $X_1(\omega_1)$ be a random variable on (Ω_2, p_2) defined by

$$X_1(\omega_1) : \omega_2 \mapsto X((\omega_1, \omega_2))$$

for $\omega_2 \in \Omega_2$. Thus, X_1 is an $L(\Omega_2, p_2)$ -valued random variable on (Ω_1, p_1) , and the mapping $X \mapsto X_1$ is a linear isomorphism from $L(\Omega_1 \times \Omega_2, p)$ to $L(\Omega_1, p_1; L(\Omega_2, p_2))$. Therefore, we have a linear isomorphism

$$L(\Omega_1 \times \Omega_2, p_1 \otimes p_2) \cong L(\Omega_1, p_1) \otimes L(\Omega_2, p_2),$$

where we write $p_1 \otimes p_2$ for p .

⁹ Random numbers generated by computer are also considered to be the values of some random variable X , and the sample space Ω is very large, as large as the number in all possible states of computer memory. We cannot know which $\omega \in \Omega$ happens, and we need not know how the random numbers are defined. All we know is the realized value $X(\omega)$ of random numbers and what probabilistic properties X has.

The following program is an example of correspondence of X and X_1 above expressed in Python.

Program: probab2.py

```
In [1]: 1 from numpy.random import choice, seed
2
3 s = 2021
4 W1 = W2 = [1, 2, 3, 4, 5, 6]
5
6 def X(w):
7     return w[0] + w[1]
8
9 def X1(w1):
10    return X((w1, choice(W2)))
11
12 seed(s)
13 for n in range(20):
14    w1 = choice(W1)
15    print(X1(w1), end=' ')
16 print()
17 seed(s)
18 for n in range(20):
19    w = choice(W1), choice(W2)
20    print(X(w), end=' ')
```

```
11 3 11 10 12 6 8 4 11 6 5 3 7 11 7 5 8 8 6 11
11 3 11 10 12 6 8 4 11 6 5 3 7 11 7 5 8 8 6 11
```

We get the same sequence of random numbers X and X_1 from the same seed $s = 2021$.

More generally, for a vector space V , $L(\Omega_1 \times \Omega_2, p_1 \otimes p_2; V)$ is linear isomorphic to $L(\Omega_1, p_1) \otimes L(\Omega_2, p_2) \otimes V$. In the example of rolling the two dice mentioned above, the set of all \mathbb{R}^2 -valued random variables is linear isomorphic to the three-fold tensor space $\mathbb{R}^6 \otimes \mathbb{R}^6 \otimes \mathbb{R}^2 \cong \mathbb{R}^{72}$ because the space of all real-valued random variables rolling one dice is linear isomorphic to \mathbb{R}^6 .

Let V be a finite-dimensional linear space over \mathbb{R} with basis $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m\}$. Let X be a V -valued random variable on (Ω, p) . For any $\omega \in \Omega$, $X(\omega)$ has representation

$$(X_1(\omega), X_2(\omega), \dots, X_m(\omega)) \in \mathbb{R}^m$$

on this basis, where X_1, X_2, \dots, X_m are real-valued random variables. Then, the expectation $E(X)$ is represented by

$$(E(X_1), E(X_2), \dots, E(X_m)) \in \mathbb{R}^m.$$

When $V = M_{\mathbb{R}}(m, n)$, for a V -valued random variable

$$X(\omega) = \begin{bmatrix} X_{11}(\omega) & X_{12}(\omega) & \cdots & X_{1n}(\omega) \\ X_{21}(\omega) & X_{22}(\omega) & \cdots & X_{2n}(\omega) \\ \vdots & \vdots & & \vdots \\ X_{m1}(\omega) & X_{m2}(\omega) & \cdots & X_{mn}(\omega) \end{bmatrix},$$

we have the expectation

$$E(\mathbf{X}) = \begin{bmatrix} E(X_{11}) & E(X_{12}) & \cdots & E(X_{1n}) \\ E(X_{21}) & E(X_{22}) & \cdots & E(X_{2n}) \\ \vdots & \vdots & & \vdots \\ E(X_{m1}) & E(X_{m2}) & \cdots & E(X_{mn}) \end{bmatrix}.$$

For any matrices $\mathbf{A} \in M_{\mathbb{R}}(l, m)$ and $\mathbf{B} \in M_{\mathbb{R}}(n, k)$, we have a useful formula

$$E(\mathbf{AXB}) = \mathbf{AE}(X)\mathbf{B}$$

called the *linearity of expectation*.

Exercise 10.11 Prove the linearity of expectation above. (Hint: first prove it for the case where both \mathbf{A} and \mathbf{B} are matrix units.)

10.5 Principal Component Analysis and KL Expansion

For an \mathbb{R}^n -valued random variable X , let

$$\mathbf{X}(\omega) = \begin{bmatrix} X_1(\omega) \\ X_2(\omega) \\ \vdots \\ X_n(\omega) \end{bmatrix} \quad (\omega \in \Omega), \quad E(\mathbf{X}) = \mathbf{m} = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_n \end{bmatrix}.$$

We consider the random variable

$$(\mathbf{X} - \mathbf{m})(\mathbf{X} - \mathbf{m})^T(\omega) \stackrel{\text{def}}{=} (\mathbf{X}(\omega) - \mathbf{m})(\mathbf{X}(\omega) - \mathbf{m})^T \quad (\omega \in \Omega)$$

valued in square matrices of order n . We call its expectation

$$E((\mathbf{X} - \mathbf{m})(\mathbf{X} - \mathbf{m})^T)$$

the *variance–covariance* (or simply *variance* or *covariance*) *matrix*. For $1 \leq i, j \leq n$ let s_{ij} be the (i, j) -element $E((X_i - m_i)(X_j - m_j))$ of this matrix. The diagonal element s_{ii} is called the *variance* of the random variable X_i , and off-diagonal element s_{ij} ($i \neq j$) is called the covariance of X_i and X_j . Let

$$r_{ij} = \frac{s_{ij}}{\sqrt{s_{ii}}\sqrt{s_{jj}}},$$

then, because $|s_{ij}| \leq \sqrt{s_{ii}}\sqrt{s_{jj}}$ holds by the Schwartz inequality, we see $-1 \leq r_{ij} \leq 1$. We call this value r_{ij} the *correlation constant* of X_i and X_j . If $r_{ij} > 0$, we say that X_i and X_j have a *positive correlation*, and if $r_{ij} < 0$, we say that X_i and X_j has a *negative correlation*. In the case $r_{ij} = 0$, we say that X_i and X_j are *uncorrelated*.

In the first part of this section, we discuss the problem of basis change in \mathbb{R}^n so that the random variables of different components of \mathbf{X} become uncorrelated.¹⁰

¹⁰ There is also the idea of making random variables of different components independent, which is called *independent component analysis*. It requires a different mathematical technique.

Table 10.1 Grade data

English	Math A	Math B
95	92	81
94.5	98	56
84	87	84
:	:	:

The variance–covariance matrix is symmetric and positive semi-definite. Moreover, we have

$$\begin{aligned} E((X - \mathbf{m})(X - \mathbf{m})^T) &= E(XX^T) - E(X)\mathbf{m}^T - \mathbf{m}E(X)^T + \mathbf{m}\mathbf{m}^T \\ &= E(XX^T) - \mathbf{m}\mathbf{m}^T. \end{aligned}$$

Hence, it is diagonalizable as

$$\mathbf{U}^T(E(XX^T) - \mathbf{m}\mathbf{m}^T)\mathbf{U} = \text{diag}(\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2)$$

with orthogonal matrix \mathbf{U} as observed in Sect. 7.3. By the linearity of the expectation, the left-hand side is transformed into

$$E(\mathbf{U}^TXX^T\mathbf{U}) - \mathbf{U}^T\mathbf{m}(\mathbf{m}^T\mathbf{U}).$$

The basis change matrix \mathbf{U} is written as

$$\mathbf{U} = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n]$$

with an orthonormal basis $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ of \mathbb{R}^n , and $\mathbf{U}^T\mathbf{X}$ and $\mathbf{U}^T\mathbf{m}$ are representations of \mathbf{X} and \mathbf{m} on this basis, respectively. The variance–covariance matrix of the vector-valued random variable $\mathbf{U}^T\mathbf{X}$ with the expectation $\mathbf{U}^T\mathbf{m}$ is a diagonal matrix $\text{diag}(\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2)$. That is, the different components of $\mathbf{U}^T\mathbf{X}$ are not correlated. The idea of diagonalizing the covariance matrix and extracting more important information hidden in the directions of the eigenvectors for larger eigenvalues is called a *principal component analysis*.

Suppose that the grade data of some test (Table 10.1) is given in a csv file. Let us execute the principal component analysis using this data.

Program: scatter.py

```
In [1]: 1 import numpy as np
2 import vpython as vp
3 import matplotlib.pyplot as plt
4
5 with open('data.csv', 'r') as fd:
6     lines = fd.readlines()
7 data = np.array([eval(line) for line in lines[1:]])
8
9 def scatter3d(data):
10    o = vp.vec(0, 0, 0)
11    vp.curve(pos=[o, vp.vec(100, 0, 0)], color=vp.color.red)
12    vp.curve(pos=[o, vp.vec(0, 100, 0)], color=vp.color.green)
13    vp.curve(pos=[o, vp.vec(0, 0, 100)], color=vp.color.blue)
14    vp.points(pos=[vp.vec(*a) for a in data], radius=3)
```

In [1]:

```

16 def scatter2d(data):
17     A = data.T
18     fig, axs = plt.subplots(1, 3, figsize=(15, 5))
19     for n, B in enumerate([A[[0, 1]], A[[0, 2]], A[[1, 2]]]):
20         s = B.dot(B.T)
21         cor = s[0, 1] / np.sqrt(s[0, 0]) / np.sqrt(s[1, 1])
22         print(f'{cor:.3f}')
23         axs[n].scatter(B[0], B[1])
24     plt.show()
25
26 if __name__ == '__main__':
27     scatter3d(data)
28     scatter2d(data)

```

Lines 5–7: Read a file of csv-format as a text file. Each row except the title row has three numbers separated with commas, so the data is converted to a matrix consisting of three-dimensional vectors aligned for the number of students.

Lines 9–14: Define a function to draw scatter plots in 3D-space.

Lines 16–24: Define a function that displays the correlation coefficient for each of the two subjects and the 2D scatter plot (correlation diagram).

Lines 26–28: Draw 3D scatter plots (Fig. 10.5, left) and 2D scatter plots (Fig. 10.6). The data used here shows that the correlation coefficient between Math A and Math B is the highest at 0.972 (Fig. 10.6, right), followed by English and Math A at 0.966 (Fig. 10.6, left), and lastly English and Math B at 0.954 (Fig. 10.6, center).

Suppose that there are n students and (x_i, y_i, z_i) expresses the grade of the i -th student. Let (Ω, p) be a probability space with $\Omega = \{1, 2, \dots, n\}$ and $p(i) = \frac{1}{n}$ ($i = 1, 2, \dots, n$). We consider $X(i) = (x_i, y_i, z_i)$ as a probability variable on (Ω, p) . Then, we have

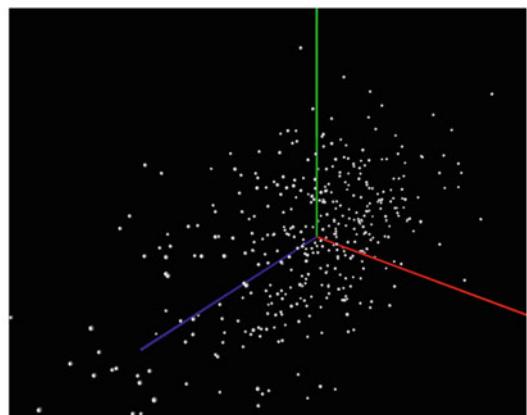
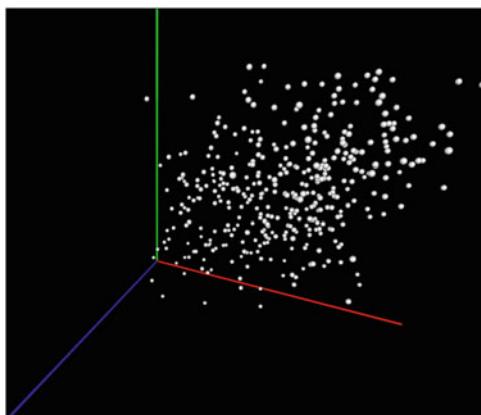


Fig. 10.5 Scatter plots in 3D space

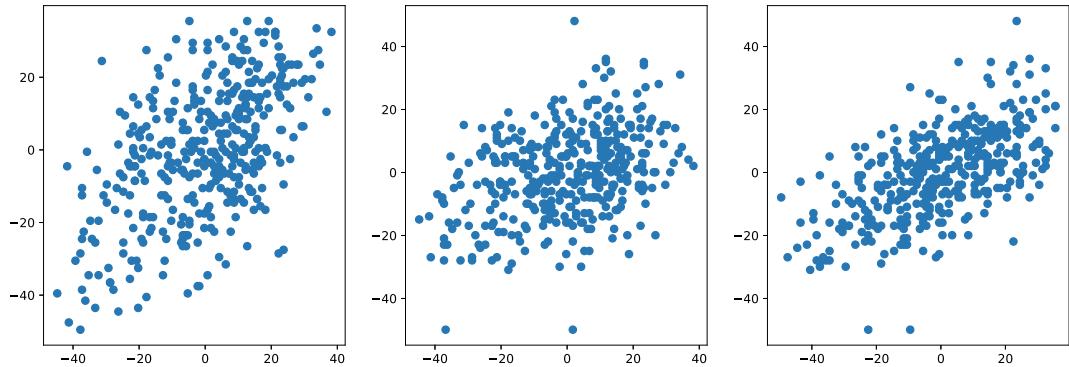


Fig. 10.6 Scatter plots in a plane

$$E(\mathbf{X}) = \frac{1}{n} \sum_{i=1}^n \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} = \begin{bmatrix} \frac{1}{n} \sum_{i=1}^n x_i \\ \frac{1}{n} \sum_{i=1}^n y_i \\ \frac{1}{n} \sum_{i=1}^n z_i \end{bmatrix}.$$

Let $m_x = \frac{1}{n} \sum_{i=1}^n x_i$, $m_y = \frac{1}{n} \sum_{i=1}^n y_i$ and $m_z = \frac{1}{n} \sum_{i=1}^n z_i$, and $\mathbf{m} = \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}$. Since

$$\begin{aligned} (\mathbf{X} - \mathbf{m})(\mathbf{X} - \mathbf{m})^T(i) &= (\mathbf{X}(i) - \mathbf{m})(\mathbf{X}(i) - \mathbf{m})^T \\ &= \begin{bmatrix} (x_i - m_x)(x_i - m_x) & (x_i - m_x)(y_i - m_y) & (x_i - m_x)(z_i - m_z) \\ (y_i - m_y)(x_i - m_x) & (y_i - m_y)(y_i - m_y) & (y_i - m_y)(z_i - m_z) \\ (z_i - m_z)(x_i - m_x) & (z_i - m_z)(y_i - m_y) & (z_i - m_z)(z_i - m_z) \end{bmatrix}, \end{aligned}$$

the variance-covariance matrix of \mathbf{X} is given by

$$E((\mathbf{X} - \mathbf{m})(\mathbf{X} - \mathbf{m})^T) = \begin{bmatrix} s_{xx} & s_{xy} & s_{xz} \\ s_{yx} & s_{yy} & s_{yz} \\ s_{zx} & s_{zy} & s_{zz} \end{bmatrix},$$

where

$$\begin{aligned} s_{xx} &= \frac{1}{n} \sum_{i=1}^n (x_i - m_x)^2, \\ s_{xy} &= \frac{1}{n} \sum_{i=1}^n (x_i - m_x)(y_i - m_y), \end{aligned}$$

and s_{xz} , s_{yx} , s_{yy} , s_{yz} , s_{zx} , s_{zy} , s_{zz} are given similarly.¹¹

¹¹ If we want to estimate the variance or covariance of the population from sample data drawn from some population, we divide by $n - 1$ instead of n to get an unbiased estimator. However, because we are only interested in the arrangement

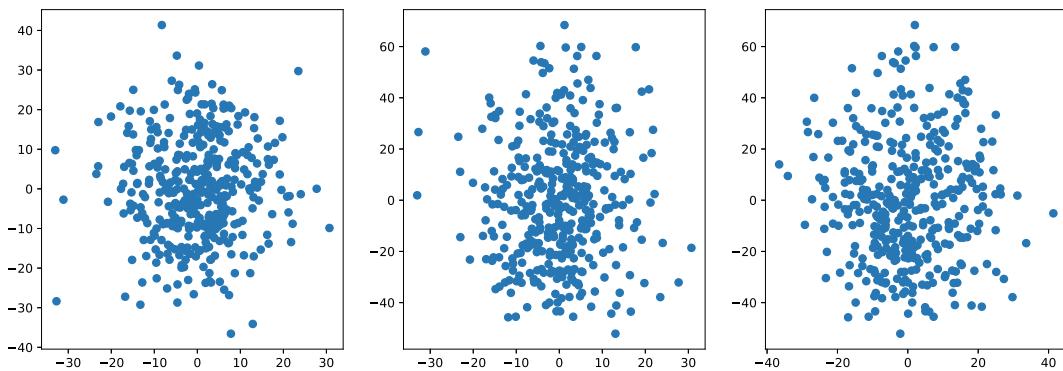


Fig. 10.7 Correlation diagrams after principal component analysis

Program: principal.py

```
In [1]: 1 from numpy.linalg import eigh
2 from scatter import data, scatter2d, scatter3d
3
4 n = len(data)
5 mean = sum(data) / n
6 C = data - mean
7 A = C.T
8 AAt = A.dot(C) / n
9 E, U = eigh(AAt)
10 print(E)
11 scatter3d(C.dot(U))
12 scatter2d(C.dot(U))
```

Line 1: Use the function `eigh` to obtain the eigenvalues and the eigenvectors. Use it instead of `eig` so that the eigenvectors form an orthonormal basis.

Line 5: `mean` represents the mean \mathbf{m} .

Lines 6–9: Calculate the variance–covariance matrix.

Line 10: Print the eigenvalues of the variance–covariance matrix.

Lines 11, 12: After the coordinate transformation by unitary matrix \mathbf{U} , draw the 3D scatter plots (Fig. 10.5, right) and the correlation diagram (Fig. 10.7).

Let $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \in \mathbb{R}^m$ be m -dimensional data changing over time $t = 1, \dots, n$. Let $\mathbf{A} = [\mathbf{a}_1 \mathbf{a}_2 \dots \mathbf{a}_n] \in M_{\mathbb{R}}(m, n)$, and consider its singular value decomposition

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V},$$

where Σ has the singular values $s_1 \geq s_2 \geq \dots \geq s_k > 0$ at (i, i) ($i = 1, 2, \dots, k$). Suppose that $\mathbf{U} = [\mathbf{e}_1 \mathbf{e}_2 \dots \mathbf{e}_m]$ with column vectors $\mathbf{e}_i \in \mathbb{R}^m$ ($i = 1, 2, \dots, m$) and \mathbf{V} consists of the (i, j) -elements φ_{ij} ($i, j = 1, 2, \dots, n$). Then, we have

$$\mathbf{a}_j = \sum_{i=1}^k s_i \mathbf{e}_i \varphi_{ij} \quad (j = 1, 2, \dots, n),$$

of eigenvalues (the direction and magnitude of the eigenvectors) of the covariance matrix, we need not care which one we divide by.

which we call the *KL (Karhunen–Loëve) expansion*. This implies that the movement $t \mapsto \mathbf{a}_t$ which varies over time t can be expressed as a sum of the independently moving motions $t \mapsto s_i \mathbf{e}_i \varphi_{it}$ ($i = 1, 2, \dots, k$) on the orthogonal axes of some subspace of \mathbb{R}^m . The singular values represent the magnitudes of the contributions of these independently moving motions.

We artificially create a model in which there are four variables that change over time, and there are two hidden variables that contribute significantly to the movement. Let us find them by an experiment with Python.

Program: KL2.py

```
In [1]: 1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 tmax, N = 100, 1000
5 dt = tmax / N
6
7 np.random.seed(2021)
8 W = np.random.normal(0, dt, (2, N))
9 Noise = np.random.normal(0, 0.25, (4, N))
10 B = W.cumsum(axis=1)
11 P = np.array([[1, 2], [1, -2], [2, 1], [-2, 1]])
12 A0 = P.dot(B)
13 A = A0 + Noise
14 U, S, V = np.linalg.svd(A)
15 print(f'singular values = {S}')
16
17 C = U[:, :2].dot(np.diag(S[:2]).dot(V[:2, :]))
18 plt.figure(figsize=(20, 5))
19 T = np.linspace(0, tmax, N)
20 plt.subplot(131)
21 for i in range(4):
22     plt.plot(T, A[i])
23 plt.subplot(132)
24 for i in range(2):
25     plt.plot(T, V[i])
26 plt.subplot(133)
27 for i in range(4):
28     plt.plot(T, C[i])
29 error0 = np.sum((A0 - A)**2, axis=1) / N
30 error1 = np.sum((A0 - C)**2, axis=1) / N
31 print(f'error0 = {error0}')
32 print(f'error1 = {error1}')
33 plt.show()
```

Lines 7–13: Create independent Brownian motions $b_1(t)$ and $b_2(t)$, and let

$$\begin{aligned} a_1(t) &= b_1(t) + 2 b_2(t), \\ a_2(t) &= b_1(t) - 2 b_2(t), \\ a_3(t) &= 2 b_1(t) + b_2(t), \\ a_4(t) &= -2 b_1(t) + b_2(t) \end{aligned}$$

for $t = 0.0, 0.1, \dots, 999.9$. Then, add noise to them to make A . The hidden variables are $b_1(t)$ and $b_2(t)$, and we can observe four variables $a_i(t)$ ($i = 1, 2, 3, 4$).

Line 14: Singular value decomposition of A . The obtained singular values are approximately 235.1, 155.2, 8.1, and 7.8, two large values and two small values.

Line 17: Calculate $U\Sigma'V$, where Σ' is made from Σ by replacing the two small singular values with 0.

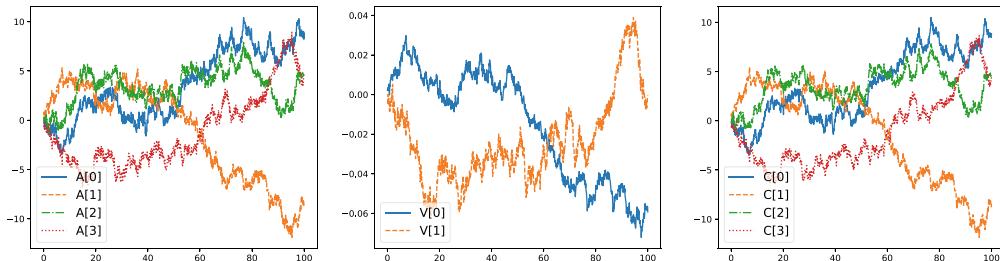


Fig. 10.8 The observed data, two principal components, and reconstructed data

Lines 18–28: Draw the movement of the observed data (Fig. 10.8 left), the movements of principal components (hidden variables) obtained by the KL expansion and the reconstructed data from the principal components (Fig. 10.8 center).

Lines 29, 30: Compute the squared errors between the original data (without noise) and the observed data and between the original data and the reconstructed data (Fig. 10.8 right).

```
singular values = [235.13538383 155.21177657 8.09565099 7.8359052]
error0 = [0.0614098 0.06428303 0.06365725 0.06560834]
error1 = [0.032575 0.03152683 0.03245869 0.03145176]
```

Let d be an integer with $0 \leq d \leq k$. For the singular value decomposition $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}$ of \mathbf{A} , let Σ_d be the matrix obtained from Σ by replacing all diagonal elements by 0 except for d elements from the largest. We can write $\Sigma_d = \mathbf{P}\Sigma\mathbf{Q}$ with square matrices \mathbf{P} and \mathbf{Q} of order m and n , respectively such that only the diagonal elements from $(1, 1)$ to (d, d) are 1 and all other components are 0. Then, we have

$$\mathbf{U}\Sigma_d\mathbf{V} = \mathbf{U}\mathbf{P}\Sigma\mathbf{Q}\mathbf{V} = (\mathbf{U}\mathbf{P}\mathbf{U}^*)\mathbf{A}(\mathbf{V}^*\mathbf{Q}\mathbf{V}).$$

Here, $\mathbf{U}\mathbf{P}\mathbf{U}^*$ and $\mathbf{V}^*\mathbf{Q}\mathbf{V}$ are orthogonal projections defined on \mathbb{K}^m and on \mathbb{K}^n , respectively.

For $m \times n$ matrices \mathbf{X} and \mathbf{Y} define

$$\langle \mathbf{X} \mid \mathbf{Y} \rangle \stackrel{\text{def}}{=} \text{Tr}(\mathbf{X}^*\mathbf{Y}),$$

then $\langle \cdot \mid \cdot \rangle$ is an inner product on $M_{\mathbb{K}}(m, n)$, and $M_{\mathbb{K}}(m, n)$ becomes an inner product space.

Exercise 10.12 Prove that the above $\langle \cdot \mid \cdot \rangle$ satisfies axioms of the inner product. Moreover, the mapping sending a matrix $\mathbf{A} \in M_{\mathbb{K}}(m, n)$ to the vector $\mathbf{v} \in \mathbb{K}^{mn}$ consisting of the elements of \mathbf{A} is a linear isomorphism preserving the inner product.

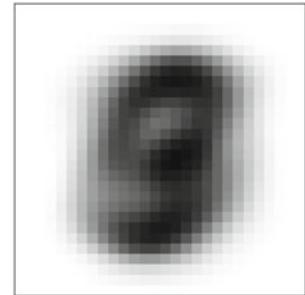
Define

$$\mathfrak{P}_d(\mathbf{X}) \stackrel{\text{def}}{=} (\mathbf{U}\mathbf{P}\mathbf{U}^*)\mathbf{X}(\mathbf{V}^*\mathbf{Q}\mathbf{V}),$$

then because $(\mathbf{U}\mathbf{P}\mathbf{U}^*)^2 = \mathbf{U}\mathbf{P}\mathbf{U}^*$ and $(\mathbf{V}^*\mathbf{Q}\mathbf{V})^2 = \mathbf{V}^*\mathbf{Q}\mathbf{V}$, we see $\mathfrak{P}_d^2 = \mathfrak{P}_d$. Moreover, since

$$\begin{aligned} \langle \mathfrak{P}_d(\mathbf{X}) \mid \mathbf{Y} \rangle &= \text{Tr}(((\mathbf{U}\mathbf{P}\mathbf{U}^*)\mathbf{X}(\mathbf{V}^*\mathbf{Q}\mathbf{V}))^*\mathbf{Y}) = \text{Tr}(\mathbf{X}^*(\mathbf{U}\mathbf{P}\mathbf{U}^*)\mathbf{Y}(\mathbf{V}^*\mathbf{Q}\mathbf{V})) \\ &= \langle \mathbf{X} \mid \mathfrak{P}_d(\mathbf{Y}) \rangle, \end{aligned}$$

Fig. 10.9 Average of all patterns



we have $\mathfrak{P}_d^* = \mathfrak{P}_d$. Thus, \mathfrak{P}_d is an orthogonal projection defined on the mn -dimensional space $M_{\mathbb{K}}(m, n)$ onto a d^2 -dimensional subspace.

In the MNIST character data discussed in Sect. 1.8, each image pattern corresponds to a 28×28 matrix A_i ($i = 1, 2, \dots, N$). Calculating the average A of all A_i as vectors of dimension $28 \times 28 = 784$, we get the image in Fig. 10.9. Let us experiment with Python to see what happens if we compute \mathfrak{P}_d from this average A and apply it to each pattern by changing d .

Program: mnist_KL2.py

```
In [1]: 1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 cutoff = 28
5 N = 60000
6 with open('train-images.bin', 'rb') as fd:
7     X = np.fromfile(fd, 'uint8', -1)[16:]
8 X = X.reshape((N, 28, 28))
9 with open('train-labels.bin', 'rb') as fd:
10    Y = np.fromfile(fd, 'uint8', -1)[8:]
11 D = {y: [] for y in set(Y)}
12 for x, y in zip(X, Y):
13     D[y].append(x)
14
15 A = sum([x.astype('float') for x in X]) / N
16 U, Sigma, V = np.linalg.svd(A)
17 print(Sigma)
18
19 def proj(X, U, V, k):
20     U1, V1 = U[:, :k], V[:k, :]
21     P, Q = U1.dot(U1.T), V1.T.dot(V1)
22     return P.dot(X.dot(Q))
23
24 fig, axs = plt.subplots(10, 10)
25 for y in D:
26     for k in range(10):
27         ax = axs[y][k]
28         A = D[y][k]
29         B = proj(A, U, V, cutoff)
30         ax.imshow(255 - B, 'gray')
31         ax.tick_params(labelbottom=False, labelleft=False,
32                         color='white')
33 plt.show()
```

Line 4: cutoff is d and the maximum is 28 in the case of MNIST.

Lines 5–13: Read all 60000 patterns of train data of MNIST.

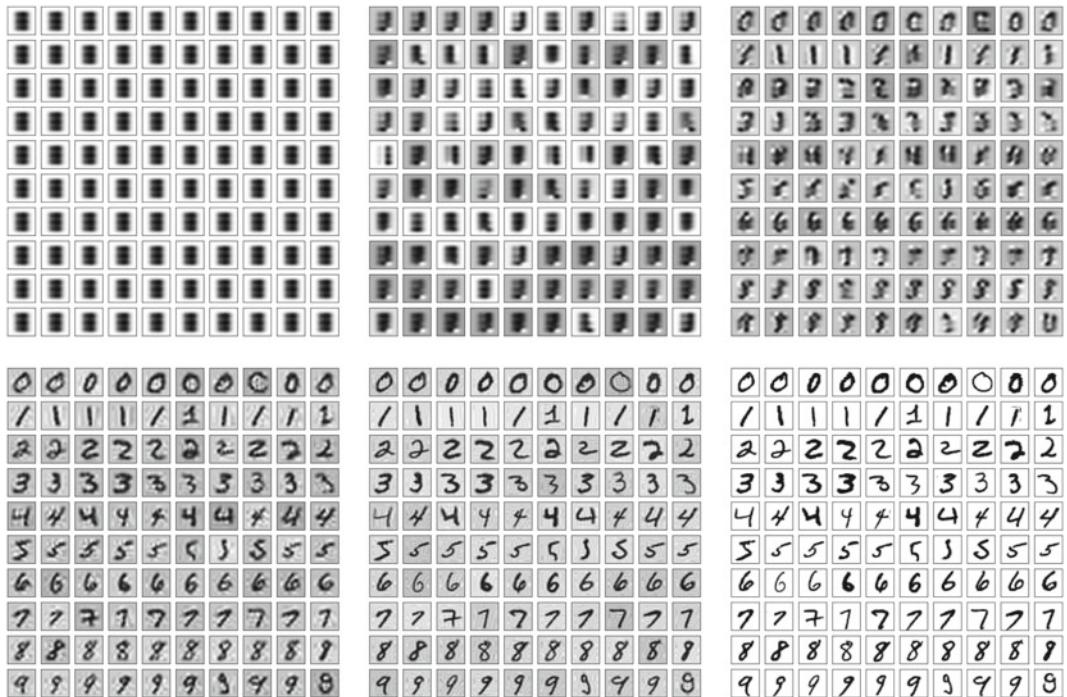


Fig. 10.10 Pattern projected by \mathcal{P}_d ($d = 1, 2, 4, 7, 14, 28$)

Lines 15–17: Apply the singular value decomposition to the average of all patterns. The average is calculated after converting an array of 28×28 bit integers to real type.

Lines 19–22: Define a function to compute \mathfrak{P}_d for given d .

Lines 24–33: Pick up 10 patterns from each character, apply \mathfrak{P}_d to them and display the obtained images (Fig. 10.10). Because \mathfrak{P}_{28} is the identity mapping on $M_{\mathbb{K}}(m, n)$, the images on the bottom right are the same as the original images.

The character data of MNIST represents one character as a point in 784-dimensional space. The purpose of this experiment is to observe if how small d can be for the characters still to be read, when orthogonally projected onto a d^2 -dimensional subspace. It can be said that the direction of the eigenvector corresponding to a particularly large singular value is an important direction for the readability of characters. This gives an idea of information compression and character recognition technologies.

10.6 Estimation of Random Variables by Linear Regression Models

For a sequence X_1, X_2, \dots, X_n of random variables whose realized values we cannot directly know, we want to estimate them from the realized values of random variables Y_1, Y_2, \dots, Y_m which are linear combinations of X_1, X_2, \dots, X_n . We assume that X_1, X_2, \dots, X_n satisfy

$$E(X_i X_j) = \delta_{ij} \quad (i = 1, 2, \dots, n, j = 1, 2, \dots, m),$$

and Y_1, Y_2, \dots, Y_m are determined from X_1, X_2, \dots, X_n through the linear relation

$$\mathbf{Y} = \mathbf{AX},$$

where

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_m \end{bmatrix} \quad \text{and} \quad \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

is a known matrix. If \mathbf{A} is a regular matrix, \mathbf{X} can be obtained by multiplying the inverse matrix of \mathbf{A} on both sides above. So we suppose that \mathbf{A} is not necessarily regular. Let H and K be the spaces generated by X_1, X_2, \dots, X_n and Y_1, Y_2, \dots, Y_m , respectively, then K is a subspace of H .

We define an inner product on H by

$$\langle U | V \rangle \stackrel{\text{def}}{=} E(UV)$$

for $U, V \in H$. By assumption, $\{X_1, X_2, \dots, X_n\}$ is an orthonormal basis of H . Let Z_1, Z_2, \dots, Z_n be the orthogonal projections of X_1, X_2, \dots, X_n on K , respectively, then for each i ,

$$\|X_i - Z_i\| = E((X_i - Z_i)^2)^{1/2} \quad (i = 1, 2, \dots, n)$$

is the shortest difference from X_i to K . Because Z_i is a linear combination of Y_1, Y_2, \dots, Y_m , we can say that Z_i is the best estimate of X_i with \mathbf{Y} .

We represent a random variable in H with the orthonormal basis $\{X_1, X_2, \dots, X_n\}$. Then, X_1, X_2, \dots, X_n are represented by vectors $\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n$ that forms the standard basis of \mathbb{R}^n . On the other hand, the representations $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_m$ of Y_1, Y_2, \dots, Y_m are determined by

$$[\mathbf{y}_1 \ \mathbf{y}_2 \ \dots \ \mathbf{y}_m]^T = \mathbf{A} [\mathbf{e}_1 \ \mathbf{e}_2 \ \dots \ \mathbf{e}_n]^T = \mathbf{A}.$$

Hence, $[\mathbf{y}_1 \ \mathbf{y}_2 \ \dots \ \mathbf{y}_m] = \mathbf{A}^T$ and we see

$$K = \langle \mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_m \rangle = \text{range}(\mathbf{A}^T).$$

Because $\mathbf{A}^\dagger \mathbf{A}$ is the orthogonal projection onto $\text{range}(\mathbf{A}^T)$, we have the representations

$$z_1 = (\mathbf{A}^\dagger \mathbf{A}) \mathbf{e}_1, \quad z_2 = (\mathbf{A}^\dagger \mathbf{A}) \mathbf{e}_2, \quad \dots, \quad z_n = (\mathbf{A}^\dagger \mathbf{A}) \mathbf{e}_n$$

of Z_1, Z_2, \dots, Z_n respectively. Therefore, we obtain

$$\begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_n \end{bmatrix} = \mathbf{A}^\dagger \mathbf{A} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix} = \mathbf{A}^\dagger \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_m \end{bmatrix}.$$

Example 10.1 Let U_1 and U_2 , be random variables with mean μ and standard deviation $\sigma > 0$. Let Error_1 and Error_2 be random variables with mean 0 and standard deviation $\tau > 0$. We suppose that U_1, U_2, Error_1 and Error_2 are independent. Put

$$X_1 \stackrel{\text{def}}{=} \frac{U_1 - \mu}{\sigma}, \quad X_2 \stackrel{\text{def}}{=} \frac{U_2 - \mu}{\sigma}, \quad X_3 \stackrel{\text{def}}{=} \frac{\text{Error}_1}{\tau}, \quad X_4 \stackrel{\text{def}}{=} \frac{\text{Error}_2}{\tau},$$

then

$$E(X_i X_j) = \delta_{ij}$$

holds. Let $Y_1 = \sigma X_1 + \tau X_3$ and $Y_2 = \sigma X_2 + \tau X_4$, that is,

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} \sigma & 0 & \tau & 0 \\ 0 & \sigma & 0 & \tau \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}.$$

We would like to estimate $U_1 = \sigma X_1 + \mu$ and $U_2 = \sigma X_2 + \mu$ by observing $V_1 \stackrel{\text{def}}{=} U_1 + \text{Error}_1 = Y_1 + \mu$ and $V_2 \stackrel{\text{def}}{=} U_2 + \text{Error}_2 = Y_2 + \mu$. We have

$$\begin{bmatrix} \sigma & 0 & \tau & 0 \\ 0 & \sigma & 0 & \tau \end{bmatrix}^\dagger = \frac{1}{\sigma^2 + \tau^2} \begin{bmatrix} \sigma & 0 \\ 0 & \sigma \\ \tau & 0 \\ 0 & \tau \end{bmatrix}.$$

We can confirm this by the following program.

Program: ginv.py

```
In [1]: 1 from sympy import *
2 s = Symbol(r'\sigma', positive=True)
3 t = Symbol(r'\tau', positive=True)
4 A = Matrix([[s, 0, t, 0], [0, s, 0, t]])
5
6 B = A.pinv()
7 print(latex(simplify(B)))
```

```
\left[\begin{matrix} \frac{\sigma}{\sigma^2 + \tau^2} & 0 & \frac{\tau}{\sigma^2 + \tau^2} & 0 \\ 0 & \frac{\sigma}{\sigma^2 + \tau^2} & 0 & \frac{\tau}{\sigma^2 + \tau^2} \end{matrix}\right]
```

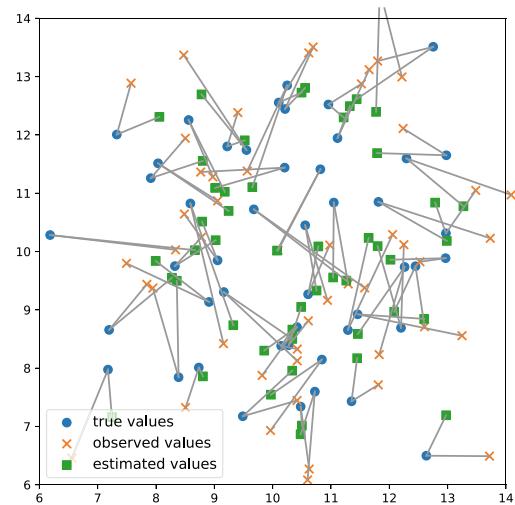
Thus, letting $\rho = \frac{\sigma}{\sigma^2 + \tau^2}$, we have $Z_1 = \rho Y_1$ and $Z_2 = \rho Y_2$ as the best estimates of X_1 and X_2 respectively. Consequently,

$$W_i = \sigma Z_i + \mu = \sigma \rho Y_i + \mu = \sigma \rho (V_i - \mu) + \mu = \frac{\sigma^2 V_i + \tau^2 \mu}{\sigma^2 + \tau^2}$$

is the best estimate for U_i ($i = 1, 2$).

When the variables U_1, U_2, Error_1 , and Error_2 are subject to a normal distribution, we calculate with NumPy using the above result and illustrate the relationship between the true value, the observed value, and the estimated value. We can observe that the estimated values are often closer to the true values than the observed values (Fig. 10.11).

Fig. 10.11 Estimation in Example 1



Program: estimate1.py

```

In [1]: 1  from numpy import array, random, linalg, sqrt
2  import matplotlib.pyplot as plt
3
4  random.seed(2021)
5  n = 50
6  mu, sigma, tau = 10, 2, 1
7
8  U1, U2 = random.normal(mu, sigma, (2, n))
9  Error1, Error2 = random.normal(0, tau, (2, n))
10 V1, V2 = U1 + Error1, U2 + Error2
11 W1 = (sigma**2 * V1 + tau**2 * mu) / (sigma**2 + tau**2)
12 W2 = (sigma**2 * V2 + tau**2 * mu) / (sigma**2 + tau**2)
13
14 plt.figure(figsize=(7, 7))
15 plt.xlim(mu-sigma*2, mu+sigma*2), plt.ylim(mu-sigma*2, mu+sigma*2)
16 plt.scatter(U1, U2, s=50, marker='o')
17 plt.scatter(V1, V2, s=50, marker='x')
18 plt.scatter(W1, W2, s=50, marker='s')
19
20 UV = UW = 0
21 for u1, u2, v1, v2, w1, w2 in zip(U1, U2, V1, V2, W1, W2):
22     plt.plot([u1, v1], [u2, v2], color='gray')
23     plt.plot([u1, w1], [u2, w2], color='gray')
24     UV += sqrt((u1 - v1)**2 + (u2 - v2)**2)
25     UW += sqrt((u1 - w1)**2 + (u2 - w2)**2)
26
27 print(f'U-V: {UV / n}')
28 print(f'U-W: {UW / n}')
29 plt.show()

```

Lines 8, 9: U_1 (resp. U_2 , $Error1$, and $Error2$) is an array consisting of $n = 50$ samplings of random variable U_1 (resp. U_2 , $Error_1$, and $Error_2$). Though every random variable is assumed to follow a normal distribution here, we may adopt any distribution with known mean and variance.

Line 10: Array V_1 (resp. V_2) consists of n samplings of random variable V_1 (resp. V_2).

Lines 11, 12: Calculate the best estimates W_1 and W_2 of U_1 and U_2 by the observations V_1 and V_2 , respectively, using the above formula.

Lines 16–18: Mark with ● the n points (plane vectors) made by coupling the components of U_1 and U_2 in order. Also mark with × for V_1 and V_2 , and with ■ for W_1 and W_2 .

Lines 22, 23: Connect each ● (true value) and × (observed value), and ● and ■ (estimated value) with line segments.

Lines 24,25: Calculate the mean length (= mean error) of each of two line segments.



U-V: 1.2742691622021318
U-W: 1.1379182601867972

Example 10.2 Let $U_1, U_2, \dots, U_n, V_1, V_2, \dots, V_n$ be independent probability variables subject to the standard normal distribution. For $\tau, \rho, \sigma > 0$ let

$$X_1 = \sigma U_1, \quad X_i = \rho X_{i-1} + \sigma U_i \quad (i = 2, 3, \dots, n)$$

and

$$Y_i = X_i + \tau V_i \quad (i = 1, 2, \dots, n).$$

We want to estimate X_1, X_2, \dots, X_n by observing Y_1, Y_2, \dots, Y_n . Eliminating X_i , the above situation is expressed by

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} \sigma & 0 & 0 & \cdots & 0 & \tau & 0 & 0 & \cdots & 0 \\ \rho\sigma & \sigma & 0 & \cdots & 0 & 0 & \tau & 0 & \cdots & 0 \\ \rho^2\sigma & \rho\sigma & \sigma & \cdots & 0 & 0 & 0 & \tau & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \rho^{n-1}\sigma & \rho^{n-2}\sigma & \rho^{n-3}\sigma & \cdots & \sigma & 0 & 0 & 0 & \cdots & \tau \end{bmatrix} \begin{bmatrix} U_1 \\ \vdots \\ U_n \\ V_1 \\ \vdots \\ V_n \end{bmatrix}.$$

We can estimate U_1, U_2, \dots, U_n from Y_1, Y_2, \dots, Y_n by calculating the generalized inverse of the matrix on the right-hand side. Let these estimates be W_1, W_2, \dots, W_n , then Z_1, Z_2, \dots, Z_n obtained by

$$Z_1 = \sigma W_1, \quad Z_i = \rho Z_{i-1} + \sigma W_i \quad (i = 2, 3, \dots, n)$$

are considered to be the best estimates of X_1, X_2, \dots, X_n .

Doing this calculation in NumPy, we draw the graphs of the true, observed, and estimated values that change over time. We can observe that the change in the estimated value is closer to the change in the true value than the change in the observed value (Fig. 10.12).

Program: estimate2.py

```
In [1]: 1 from numpy import zeros, arange, random, linalg
2 import matplotlib.pyplot as plt
3
4 N, rho, sigma, tau = 100, 1.0, 0.1, 0.1
5 random.seed(2021)
6
7 x, y = zeros(N), zeros(N)
8 for i in range(N):
9     x[i] = rho*x[i - 1] + sigma*random.normal(0, 1)
10    y[i] = x[i] + tau*random.normal(0, 1)
11
12 A = zeros((N, 2 * N))
```

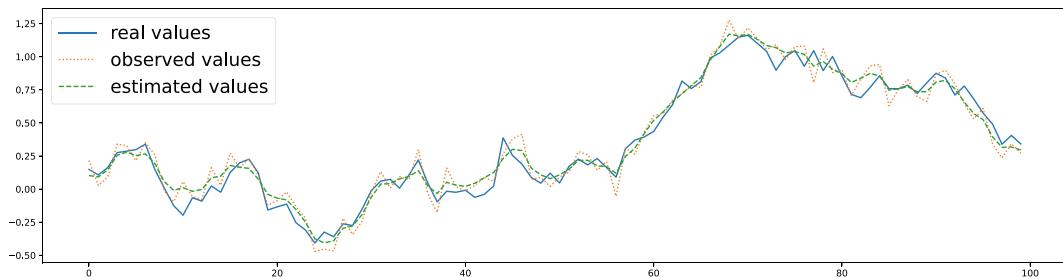


Fig. 10.12 Estimation in Example 2

```
In [1]:  
13 for i in range(N):  
14     for j in range(i + 1):  
15         A[i, j] = rho**(i - j) * sigma  
16     A[i, N + i] = tau  
17 B = linalg.pinv(A)  
18  
19 v = B.dot(y)  
20 z = zeros(N)  
21 for i in range(N):  
22     z[i] = rho*z[i - 1] + sigma*v[i]  
23 print(f'(y-x)^2 = {sum((y-x) ** 2)}')  
24 print(f'(z-x)^2 = {sum((z-x) ** 2)}')  
25  
26 plt.figure(figsize=(20, 5))  
27 T = arange(N)  
28 plt.plot(T, x, color='black', linestyle = 'solid')  
29 plt.plot(T, y, color='black', linestyle = 'dotted')  
30 plt.plot(T, z, color='black', linestyle = 'dashed')  
31 plt.show()
```

Line 4: Set the parameters.

Line 5: Explicitly set the seed of the random number for comparison with the estimation for the Kalman filter described in the next section.

Lines 7–10: Generate the realized values of X_1, X_2, \dots, X_n and Y_1, Y_2, \dots, Y_n .

Lines 12–17: Find the generalized inverse for the estimation.

Lines 19–31: Get the estimates and calculate the squared error. Draw the graphs showing true, observed, and estimated values.

```
(y-x)^2 = 0.9032093121921972  
(z-x)^2 = 0.48559541437198783
```

The squared error between the true and observed values is about 0.9, and the squared error between the true and estimated values is about 0.5.

10.7 Kalman Filter

Consider a robot that automatically travels on a bumpy road surface. The robot tracks its position using sensors, and corrects the movement if it deviates from the desired direction under the influence of the road surface. However, the sensors (for example, cameras) also vibrate under the influence of the road surface and do not always tell the correct state. In this case, it is necessary to remove the error component added to the sensors in real time.

In the estimation problem in Example 10.2 in the previous section, knowing the realized values of Y_1, Y_2, \dots, Y_n , we estimate the realized values of X_1, X_2, \dots, X_n . Here, for the same probability model, with the passage of time $k = 1, 2, \dots, n$, we estimate X_k from the observed value Y_k and the value Z_{k-1} which estimates X_{k-1} immediately before. Let random variables $U_1, U_2, \dots, U_n, V_1, V_2, \dots, V_n$ be the same as before, and let H be the linear space generated by all of them.

All of $X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_n$ belong to H , because they are linear combinations of $U_1, U_2, \dots, U_n, V_1, V_2, \dots, V_n$. For $X, Y \in H$, we define an inner product $\langle X | Y \rangle = E(XY)$ and the norm with associated it is called the *root mean square norm*. Since all the random variables in H have an average of 0, two independent random variables belonging to H are orthogonal.¹²

For $k \geq 0$, let \mathbf{P}_k be the orthogonal projection onto the subspace H_k generated by Y_1, Y_2, \dots, Y_k . In particular, $H_0 = \{\mathbf{0}\}$ and \mathbf{P}_0 is the zero mapping. What we want is $Z_k \stackrel{\text{def}}{=} \mathbf{P}_k(X_k)$ for each k , which is the element Z of H_k minimizing $E((X_k - Z)^2)$. Let $Z'_k \stackrel{\text{def}}{=} \mathbf{P}_{k-1}(X_k)$. Then, Z_k is the best estimate of X_k by Y_1, Y_2, \dots, Y_k , and Z'_k is the best estimate of X_k by Y_1, Y_2, \dots, Y_{k-1} . Let

$$F_k \stackrel{\text{def}}{=} Y_k - Z'_k, \quad a_k \stackrel{\text{def}}{=} \|X_k - Z'_k\|^2 \quad (k = 1, 2, \dots).$$

Note that $a_1 = \sigma^2$, for $Z'_1 = \mathbf{0}$. By the Pythagorean theorem, we have

$$\|F_k\|^2 = \|X_k - Z'_k + \tau V_k\|^2 = \|X_k - Z'_k\|^2 + \|\tau V_k\|^2 = a_k + \tau^2.$$

We see $Z_k - Z'_k$ belongs to H_k and is orthogonal to H_{k-1} . On the other hand, because

$$\mathbf{P}_{k-1}(Y_k) = \mathbf{P}_{k-1}(X_k + \tau V_k) = \mathbf{P}_{k-1}(X_k) + \tau \mathbf{P}_{k-1}(V_k) = Z'_k,$$

$F_k = Y_k - Z'_k = Y_k - \mathbf{P}_{k-1}(Y_k) \in H_k$ is also orthogonal to H_{k-1} . Since F_k and H_{k-1} generate H_k , we see $Z_k - Z'_k = b_k F_k$ for some $b_k \in \mathbb{R}$. Because

$$\begin{aligned} b_k \|F_k\|^2 &= \langle Z_k - Z'_k | F_k \rangle = \langle Z_k | F_k \rangle = \langle X_k | F_k \rangle = \langle X_k - Z'_k | F_k \rangle \\ &= \langle X_k - Z'_k | Y_k - Z'_k \rangle = \langle X_k - Z'_k | X_k - Z'_k \rangle = \|X_k - Z'_k\|^2 = a_k, \end{aligned}$$

we obtain

$$b_k = \frac{a_k}{\|F_k\|^2} = \frac{a_k}{\tau^2 + a_k}.$$

Consequently, we get

$$Z_k = Z'_k + b_k F_k = Z'_k + \frac{a_k}{\tau^2 + a_k} (Y_k - Z'_k).$$

Thus, we can calculate Z_k if we can find a_k and Z'_k . Applying \mathbf{P}_{k-1} on both sides of

$$X_k = \rho X_{k-1} + \sigma U_k, \quad \cdots (\dagger)$$

we have

¹² We need to discuss the definition of independence of random variables, but here we only focus on the inner product, the orthogonality, and the root mean square norm.

$$Z_k' = \mathbf{P}_{k-1}(X_k) = \rho \mathbf{P}_{k-1}(X_{k-1}) = \rho Z_{k-1}, \quad \dots (\dagger\dagger)$$

noting $\mathbf{P}_{k-1}(U_k) = 0$. In this way, we obtain the difference equation

$$Z_k = \rho Z_{k-1} + \frac{a_k}{\tau^2 + a_k} (Y_k - \rho Z_{k-1}).$$

From (\dagger) and $(\dagger\dagger)$, we have

$$X_k - Z_k' = \rho (X_{k-1} - Z_{k-1}) + \sigma U_k,$$

and the two terms on the right-hand side above are orthogonal to each other. Hence, by the Pythagorean theorem, we have

$$\|X_k - Z_k'\|^2 = \rho^2 \|X_{k-1} - Z_{k-1}\|^2 + \sigma^2.$$

Let $c_k \stackrel{\text{def}}{=} \|X_k - Z_k\|^2$ for $k > 0$ and $c_0 = 0$. Then the above equality becomes

$$a_k = \rho^2 c_{k-1} + \sigma^2.$$

On the other hand, from

$$\langle X_k - Z_k \mid Z_k - Z_k' \rangle = \langle \mathbf{P}_k(X_k - Z_k) \mid Z_k - Z_k' \rangle = \langle Z_k - Z_k \mid Z_k - Z_k' \rangle = 0,$$

$X_k - Z_k$ and $Z_k - Z_k'$ are orthogonal, so again by the Pythagorean theorem, we have

$$\|X_k - Z_k'\|^2 = \|X_k - Z_k\|^2 + \|Z_k - Z_k'\|^2,$$

that is,

$$a_k = c_k + \|b_k F_k\|^2 = c_k + b_k^2 (\tau^2 + a_k) = c_k + \frac{a_k^2}{\tau^2 + a_k}$$

holds. To summarize, starting with $Z_0 = 0$ and $a_1 = \sigma^2$, we can calculate Z_k sequentially by

$$\begin{aligned} Z_k &= \rho Z_{k-1} + \frac{a_k}{\tau^2 + a_k} (Y_k - \rho Z_{k-1}), \\ c_k &= a_k - \frac{a_k^2}{\tau^2 + a_k}, \\ a_{k+1} &= \rho^2 c_k + \sigma^2. \end{aligned}$$

We call this Z_k the *Kalman filter* of X_k by the observed values Y_1, Y_2, \dots, Y_k .

Remark that due to the relationship in $(\dagger\dagger)$, to obtain the prediction (extrapolation) of X_{k+1} by Y_1, Y_2, \dots, Y_k , we only need to multiply the filter Z_k of Y_1, Y_2, \dots, Y_k by ρ .

Here, σ , τ , and ρ are constants (steady) regardless of time, but even if they change (are unsteady), like σ_k , τ_k and ρ_k , depending on time, the difference equation remains valid by simply replacing σ , τ , and ρ with σ_k , τ_k , and ρ_k , respectively. However, these values must be known all the time.

Let us find the estimated values by the Kalman filter for the same data of the true values and observed values as in Program `estimate2.py`.

Program: `kalman.py`

```
In [1]: 1 from numpy import *
2 import matplotlib.pyplot as plt
3
4 random.seed(2021)
5 N, r, s, t = 100, 1.0, 0.1, 0.1
6 T = range(N)
7
8 x, y, z = zeros(N), zeros(N), zeros(N)
9 a = s**2
10 for i in range(N):
11     x[i] = r * x[i - 1] + s * random.normal(0, 1)
12     y[i] = x[i] + t * random.normal(0, 1)
13     z[i] = r * z[i - 1] + a / (t**2 + a) * (y[i] - r * z[i - 1])
14     c = a - a**2 / (t**2 + a)
15     a = r * c + s**2
16 print(f'(y-x)^2 = {(sum((y-x)**2))}')
17 print(f'(z-x)^2 = {(sum((z-x)**2))}')
18
19 plt.figure(figsize=(20, 5))
20 plt.plot(T, x, color='black', linestyle = 'solid')
21 plt.plot(T, y, color='black', linestyle = 'dotted')
22 plt.plot(T, z, color='black', linestyle = 'dashed')
23 plt.show()
```

Line 4: Using the same seed as `estimate2.py`, we have the same graphs of x and y (Fig. 10.13).

Lines 8–15: While updating the true value x and the observed value y , calculate the estimated value z by Kalman's method each time.



```
(y-x)^2 = 0.9032093121921977
(z-x)^2 = 0.6551470144698649
```

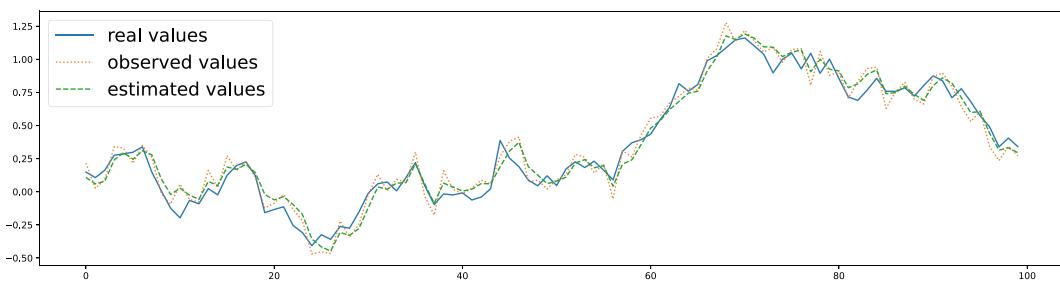


Fig. 10.13 Estimation by the Kalman filter

The squared error between the true and estimated values is about 0.7. This is larger than the one occurring in `estimate2.py`, which calculates the estimated values from the observed values of the entire section. While the Kalman filter uses only the observed values up to every current time (future information cannot be used).

Appendix

In this appendix, we first describe how to build an environment in which all the program codes in this book can be performed on our computer. The platforms covered here are Windows, macOS, and Raspberry Pi OS. We introduce the following two environments for using Python:

- IDLE—A classical integrated development environment of Python;
- Jupyter—An educational environment for using Python.

We need the following external libraries of Python:

- PIL—For reading image files;
- NumPy—For mathematical functions, numeric vector and matrix calculation and random numbers;
- Matplotlib—For drawing 2D/3D figures and graphs of functions;
- SciPy—For reading and writing audio data;
- SymPy—For symbolic calculation;
- VPython—For 3D rendering and animation.

These are very powerful widely used libraries, but the uses listed above are just a few. This book is not an introduction to these libraries. We use Python and these libraries as an aid for mathematical formulas and natural language in order to explain the idea of linear algebra.

Second, we will roughly explain uses of Python and the libraries listed above, which the reader must know before starting Chap. 1. However, the actual understanding of Python programming will be obtained by studying the many code examples found in this book. We also recommend that the reader refers to Internet pages such as <https://docs.python.org/3/reference/index.html> when necessary.

Finally, we introduce other Python libraries and non-Python applications, which were used by the authors on a Raspberry Pi to write this book.

A.1 Python Environment Used in This Book

A.1.1 Windows

We recommend to install a system called Anaconda, because it includes Python3 and all the libraries we need except VPython. Get Anaconda from <https://www.anaconda.com> and install it. There are two versions for 32-bit and 64-bit Windows, but if unsure which to choose, try to install the latter first.

When we start the installation, we will be asked if it is our machine or just our own account, but the latter is a good choice.

A.1.2 macOS

Anaconda is also available; get it from <https://www.anaconda.com> and install it. Here, we explain another way to install Python using the latest version of Python3 obtained from <https://www.python.org/>. When the installation of Python3 is complete, a new folder named Python3.* will be created in the Application folder. Open a terminal window from the Utility folder and execute the following commands line by line.

```

1 python3 -m pip install --upgrade pip
2 python3 -m pip install jupyterlab
3 python3 -m pip install pillow
4 python3 -m pip install numpy
5 python3 -m pip install matplotlib
6 python3 -m pip install scipy
7 python3 -m pip install sympy
8 python3 -m pip install vpython

```

If the installed version is Python 3.9, executable commands will be found in /Library/Frameworks/Python.framework/Versions/3.9/bin, and a path to this directory is automatically created.

A.1.3 Raspberry Pi OS

Before installing Python and the libraries on a Raspberry Pi, even if the reader already has a micro-SD card with OS installed, we highly recommend that the reader prepares another card, downloads the latest OS image file from <http://raspberrypi.org>, and installs it. This is not only to be able to use the latest OS, but also to prevent any damage to the card. Here, we will start with a micro-SD card with a newly installed Raspberry Pi OS. We have several choices for installing libraries.

If the reader has Raspberry Pi 4 or later (recommended), open LXTerminal from the task bar on the desktop, and execute the following commands line by line.

For Raspberry Pi 4/400

```

1 sudo apt install idle3 -y
2 python3 -m pip install --upgrade --user pip
3 ~/.local/bin/pip install --user jupyterlab
4 ~/.local/bin/pip install --user vpython
5 sudo apt install python3-matplotlib -y
6 sudo apt install python3-scipy -y
7 sudo apt install python3-sympy -y

```

Jupyter, VPython, and related libraries are installed in the folder /home/pi/.local/.

If using a Raspberry Pi Zero to 3 model, omit lines 2–4. This is because using Jupyter Notebook or VPython on these models is not practical due to inadequate CPU performance. However, it is worth noting that much of the code in this book that does not use them can also be run on these models, allowing linear computations to be performed locally as an embedded system.

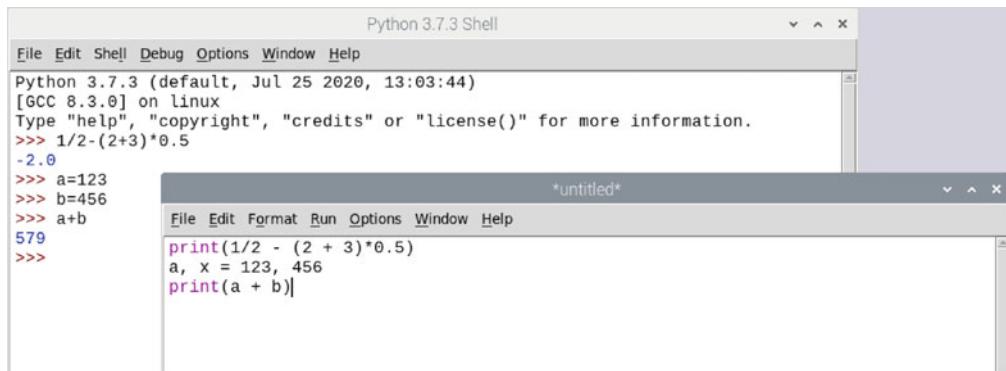


Fig. A.1 Shell window (left back) and Program editor (right front) of IDLE

There are also ways to create a virtual environment and install multiple different Python environments, but since Raspberry Pi can create different OS environments by easily replacing the micro-SD card, we will not explain the Python virtual environment here.

A.2 Launching Python

To work with Python, it is convenient to use an *IDE* (integrated development environment). The IDE integrates an editor that writes program code and an environment that runs it interactively. With the settings up to this point, an IDE named IDLE can be used on all platforms, so here we will proceed using IDLE (Fig. A.1).¹

- **Windows**

On Windows with Anaconda installed, open Anaconda PowerShell Prompt and execute the `idle` command to launch it. Anaconda ships with a newer Python IDE called Spyder (Fig. A.2, left), which we can also use.

- **macOS**

When using Python obtained from python.org, we can find a folder called Python3 in the Application folder, so launch it from the icon named IDLE in the folder. Alternatively, open a terminal window and execute the `idle3` command to launch it.

- **Raspberry Pi OS**

Launch it from Python 3 (IDLE) in Programming of the main menu, or open LXTerminal and execute the `idle` command. Raspberry Pi's default Python IDE named Thonny Python IDE (Fig. A.2, right) found in the same menu is also available.

When we launch IDLE, we see a message containing the Python version, etc. This window is called a *shell window*, the place where we interact with Python. The first >>> is the prompt for input.

Here we have a simple dialog.

¹ IDLE has been the standard for Python's IDE, but recently, other more sophisticated IDEs have appeared, and IDLE is gradually being superseded. In this book, we introduce some different Python environments, but we cannot find a common new IDE to recommend. Although they are of higher performance than IDLE, we need to explain the functions one by one and it may be a high threshold for beginners. Python installed from python.org ships with IDLE, which still has merits.

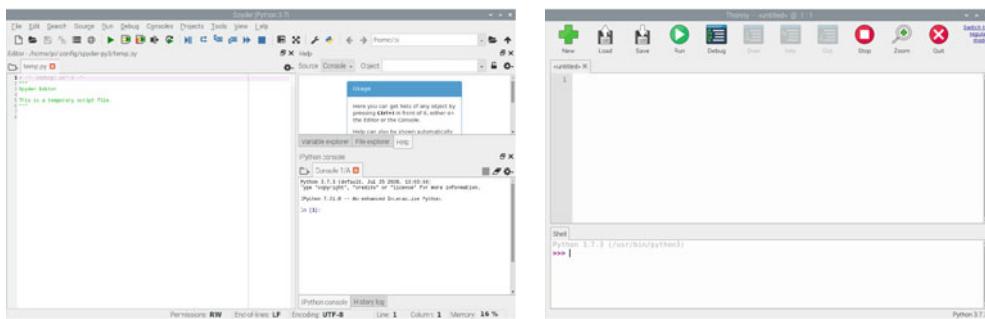


Fig. A.2 Spyder (left) and Thonny Python IDE (right)

Interaction in the shell window of IDLE

```
>>> 1 / 2 - (2 + 3) * 0.5
-2.0
>>> a = 123
>>> x = 456
>>> a + x
579
>>> x
456
>>> x = x + a
>>> x
579
```

Expression $x=x+a$ is called an assignment statement and it overwrites x .²

In some newer types of IDE, for example, Spyder, the window is called the console and its prompt is $\text{In } [*]$, where $*$ is a number. The dialog looks like this:

Interaction in the console of Spyder

```
In [1]: 1 / 2 - (2 + 3) * 0.5
Out [1]: -2.0

In [2]: a = 123

In [3]: x = 456

In [4]: a + x
Out [4]: 579
```

After the prompt $>>>$ or $\text{In } [*]$, we type the arithmetic formula on the keyboard and press the Enter/Return key, then the calculation result is returned. We can use variables as memory. This usage of Python is called an *interactive mode*, which is a sophisticated calculator. Do not enter a space at the beginning of the line following the shell prompt. A space at the beginning of a line has an important meaning in Python. This will be discussed later. It is allowed to put spaces before and after the operators $+$, $*$, $=$, etc. for readability. Before pressing Enter/Return, we can use the arrow keys, Backspace key, and Delete key to edit a line.

When we use IDLE, from the menu bar of the shell window,³ select “New File” in the “File” menu. Then another window called the *edit window* will open. We write the Python program code there. Let us program the same calculation done in the above interactive modes.

² $x=x+a$ is simply written as $x+=a$, which is called an *augmented assignment statement*. There are other *augmented assignment operators* $*=$, $-=$, and $/=$.

³ In the case of macOS, activate the shell window and select it from the menu bar displayed at the top of the screen.

Program in the Editor (example01.py)

```

1 print(1/2 - (2 + 3) * 0.5)
2 a, x = 123, 456
3 print(a + x)

```

As in interactive mode, do not put a space at the beginning of each line. In this example, spaces are placed on both sides of some operators. In Python code, the `print` function is used to print the calculation result. If we write only the formula in a program, it will not be displayed, though it will be calculated. If we want to see the value assigned in a variable, after running the program, enter an interactive mode and refer to the variable. When the program is complete, select “Run Module” from “Run” in the menu bar of the edit window,⁴ alternatively, simply press the function key F5. Because we will be asked if we want to save the file, click the “OK” button. Then, a dialog for specifying the save destination (file name and folder) is displayed. Let us name it `example01` and save it to the current folder. Click the “OK” button to save the file and display the calculation results in the shell window. Python program files are automatically given names with extension `.py`.

Results and interaction in the shell window

```

-2.0
579
>>> a, x
(123, 456)

```

If we modify the code and run it, the program file will be overwritten.

Let us quit IDLE once here by closing both the shell window and the edit window. Then, open a terminal, move to the folder where `example01.py` is saved, and execute the following command.

Windows

```
python example01.py
```

macOS, Raspberry Pi OS

```
python3 example01.py
```

The calculation result of the program will be displayed on the terminal.

A.3 Using Jupyter Notebook

We can use Jupyter Notebook, which is a Web application, instead of Python’s IDE.

- **Windows or macOS:** When Anaconda is installed, launch it from Anaconda Navigator.
- **Raspberry Pi 4/400:** Select item `Jupyter Notebook` in the menu of Programming or Education.



⁴ In the case of macOS, activate the edit window and select it from the menu bar displayed at the top of the screen.

In all environments we can launch it by executing the following command.

```
jupyter notebook
```

We may replace the above command by `jupyter-notebook`.

If we launch it, the Web browser starts up. Move to a suitable directory displayed on the tab if necessary. Next, create a new notebook of Python 3 (Fig. A.3). Then another tab called a notebook will open (Fig. A.4).

In [1]: `1 | 1/2 - (2+3) *0.5`

Out [1]: `-2.0`

In the notebook `In [n]` is a prompt, where `n` indicates the order of dialog with Python. The gray zone is called a *cell*, where we can write a question with Python code. To ask Python a question, hold down the shift key and press enter (Shift+Enter). Pressing only the enter key means a line break. We can write multiple lines of Python code in a cell.

In [2]: `a=123
x=456
a+x`

Out [2]: `579`

By Shift+Enter we get the answer.

We can also write multiple Python program codes in a cell. If we want to see the line numbers, press the escape key and then the L key. This operation is toggled.

In [3]: `1 | print(1/2 - (2 + 3)*0.5)
2 | a, x = 123, 456
3 | print(a + x)`

`-2.0
579`

In [4]: `a, x`

Out [4]: `(123, 456)`

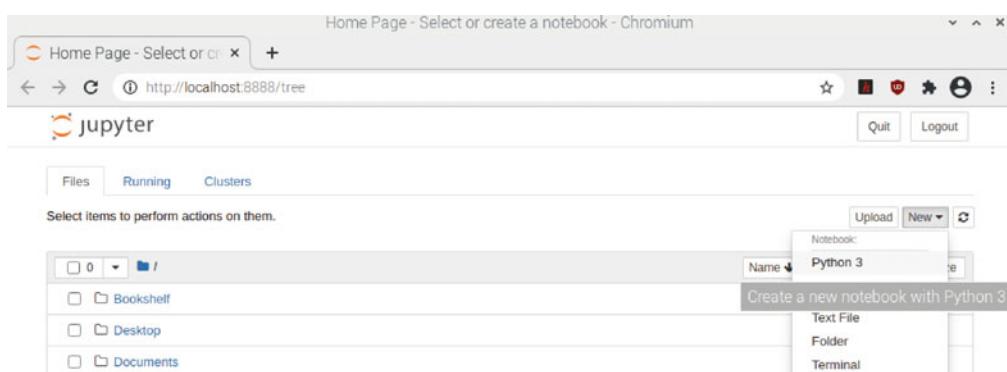


Fig. A.3 Launch of Jupyter Notebook

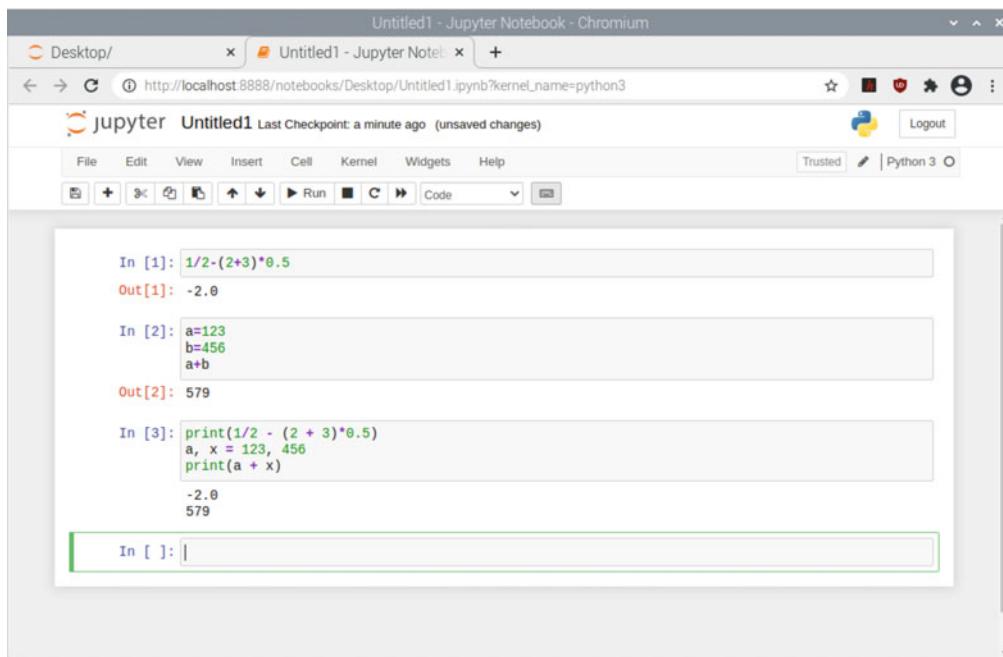


Fig. A.4 Using Jupyter Notebook

These interactions are automatically saved in a file with the file extension `.ipynb` on the current directory after a short time. If necessary, we may save it with another name or rename it later.

A.4 Using Libraries

In Python we can add new features by using *libraries*. In Python the terms *library* and *module* are interchangeable and there is no clear difference in definition. It seems that relatively large ones are often called libraries, and relatively small ones, such as libraries included in libraries, and self-made libraries are called modules.

To use a library, we need to import it both in an interactive mode and in a program. Importing will be explained again in the next section about Python syntax. Let us use `math`, which is a standard library for mathematical functions.

Interaction in the shell window of IDLE

```
>>> from math import pi, sin, cos
>>> pi
3.141592653589793
>>> sin(pi)
1.2246467991473532e-16
>>> sin(pi/2)
1.0
>>> cos(pi/4)**2
0.5000000000000001
```

We will explain the above interactions line by line in the following Jupyter Notebook style.

```
In [1]: from math import pi, sin, cos
```

This is called an *import statement*, and it allows us to use the names `pi`, `sin`, and `cos` defined in the libraries `math`, which represent π , sin, and cos in mathematics, respectively.

In [2]: `pi`

Out [2]: `3.141592653589793`

The displayed value of π is rounded to the specified number of effective digits after the decimal point because it continues infinitely without repetition.

In [3]: `sin(pi)`

Out [3]: `1.2246467991473532e-16`

The calculated answer of $\sin(\pi)$ is not exactly 0 because `pi` is rounded and contains an error. `1.2246467991473532e-16` means $1.2246467991473532 \times 10^{-16}$.

In [4]: `sin(pi/2)`

Out [4]: `1.0`

Calculate $\sin(\pi/2) = 1$, which is actually rounded to 1.

In [5]: `cos(pi/4)**2`

Out [5]: `0.5000000000000001`

Calculate $\cos(\pi/4)^2 = 0.5$ (`**` means the power operation in Python), in which we can see some error.

If the number `n` of the prompt `In[n]` starts with 1 as `In[1]`, it means that we restart the shell of IDLE (select “Shell>Restart Shell (Ctrl+F6)” from the menu bar) or restart the kernel of the Jupyter Notebook (click the **C** icon on the menu bar). If we run codes continuously in the same shell/kernel, `n` increases in order.

Let us use the external libraries NumPy and Matplotlib. NumPy is a library that supports numerical calculations of vectors and matrices, which are the main subject of this book. Matplotlib is a library used to display graphs of functions. Let us create a new file called `example02.py` in the same way as `example01.py` with IDLE, and run it.

Program in the edit window of IDLE (`example02.py`)

```

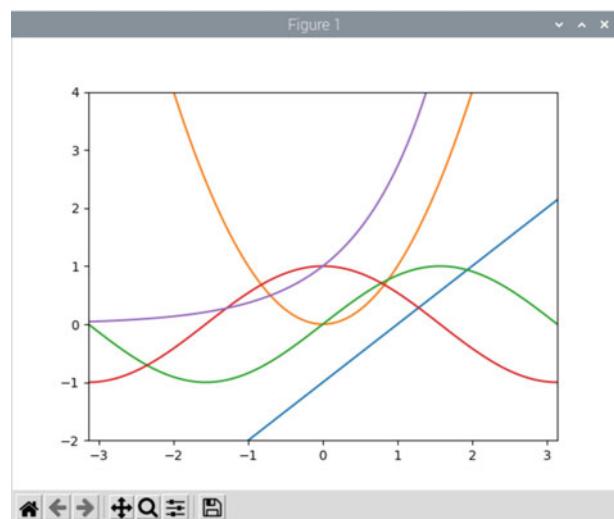
1 from numpy import linspace, pi, sin, cos, exp
2 import matplotlib.pyplot as plt
3
4 x = linspace(-pi, pi, 101)
5 plt.xlim(-pi, pi), plt.ylim(-2, 4)
6 for y in [x - 1, x**2, sin(x), cos(x), exp(x)]:
7     plt.plot(x, y)
8 plt.show()

```

When we run this program, another window will open and graphs displayed as shown in Fig. A.5. If we want to save this graph as an image file, replace `plt.show()` on Line 8 by `plt.savefig('example02.png')`, where the `example02` can be replaced by another file name of our choice, and `png` can be replaced by another file extension such as `jpg`, `pdf`, etc.

If using Jupyter Notebook, create a new notebook named `example02.ipynb`. We can write down this program in a cell as follows. The function `plt.show()` on Line 8 in `example02.py` above is not needed (though it causes no error) in the notebook.

Fig. A.5 Graphs of several functions



Program in the Jupyter Notebook (example02.ipynb)

```
In [1]: 1 from numpy import linspace, pi, sin, cos, exp
2 import matplotlib.pyplot as plt
3
4 x = linspace(-pi, pi, 101)
5 plt.xlim(-pi, pi), plt.ylim(-2, 4)
6 for y in [x - 1, x**2, sin(x), cos(x), exp(x)]:
7     plt.plot(x, y)
```

In the Jupyter Notebook, we can write the above program across multiple cells as shown below.

```
In [1]: from numpy import linspace, pi, sin, cos, exp
```

Import `linspace`, `pi`, `sin`, `cos`, and `exp` defined in NumPy. Mathematical functions and constants are also defined in the standard library `math`, but in this book we mainly use those defined in NumPy.

```
In [2]: import matplotlib.pyplot as plt
```

Import the library `pyplot` included in the library `Matplotlib`. The library may have a tree structure. Declaring in this way allows all names defined in `matplotlib.pyplot` to be referenced with prefix `plt`.

```
In [3]: x = linspace(-pi, pi, 101)
```

Python variable `x` refers to the arithmetic sequence consisting of 101 points

$$-\pi = x_0, x_1, \dots, x_{100} = \pi$$

including both ends of the real interval from $-\pi$ to π divided into 100 equal parts. Let us see what `x` looks like.

```
In [4]: x
```

```
Out[4]: array([-3.14159265e+00, -3.07876080e+00, -3.01592895e+00, -2.95309709e+00,
 -2.89026524e+00, -2.82743339e+00, -2.76460154e+00, -2.70176968e+00,
 -2.63893783e+00, -2.57610598e+00, -2.51327412e+00, -2.45044227e+00,
 -2.38761042e+00, -2.32477856e+00, -2.26194671e+00, -2.19911486e+00,
 -2.13628300e+00, -2.07345115e+00, -2.01061930e+00, -1.94778745e+00,
 -1.88495559e+00, -1.82212374e+00, -1.75929189e+00, -1.69646003e+00,
 -1.63362818e+00, -1.57079633e+00, -1.50796447e+00, -1.44513262e+00,
 -1.38230077e+00, -1.31946891e+00, -1.25663706e+00, -1.19380521e+00,
 -1.13097336e+00, -1.06814150e+00, -1.00530965e+00, -9.42477796e-01,
 -8.79645943e-01, -8.16814090e-01, -7.53982237e-01, -6.91150384e-01,
 -6.28318531e-01, -5.65486678e-01, -5.02654825e-01, -4.39822972e-01,
 -3.76991118e-01, -3.14159265e-01, -2.51327412e-01, -1.88495559e-01,
 -1.25663706e-01, -6.28318531e-02, 4.44089210e-16, 6.28318531e-02,
 1.25663706e-01, 1.88495559e-01, 2.51327412e-01, 3.14159265e-01,
 3.76991118e-01, 4.39822972e-01, 5.02654825e-01, 5.65486678e-01,
 6.28318531e-01, 6.91150384e-01, 7.53982237e-01, 8.16814090e-01,
 8.79645943e-01, 9.42477796e-01, 1.00530965e+00, 1.06814150e+00,
 1.13097336e+00, 1.19380521e+00, 1.25663706e+00, 1.31946891e+00,
 1.38230077e+00, 1.44513262e+00, 1.50796447e+00, 1.57079633e+00,
 1.63362818e+00, 1.69646003e+00, 1.75929189e+00, 1.82212374e+00,
 1.88495559e+00, 1.94778745e+00, 2.01061930e+00, 2.07345115e+00,
 2.13628300e+00, 2.19911486e+00, 2.26194671e+00, 2.32477856e+00,
 2.38761042e+00, 2.45044227e+00, 2.51327412e+00, 2.57610598e+00,
 2.63893783e+00, 2.70176968e+00, 2.76460154e+00, 2.82743339e+00,
 2.89026524e+00, 2.95309709e+00, 3.01592895e+00, 3.07876080e+00,
 3.14159265e+00])
```

These are not stored in the computer as they are. Each number is stored as a *floating-point binary number*.

```
In [5]: plt.xlim(-pi, pi), plt.ylim(-2, 4)
```

```
Out[5]: ((-3.141592653589793, 3.141592653589793), (-2.0, 4.0))
```

Functions `plt.xlim(-pi, pi)` and `plt.ylim(-2, 4)` specify the drawing range of the x -axis and y -axis of the graph. Functions can be written on one line, separated by commas.

```
In [6]: for y in [x - 1, x**2, sin(x), cos(x), exp(x)]:
    plt.plot(x, y)
```

Display the graphs of the four functions $y = x - 1$, $y = x^2$, $y = \sin x$, $y = \cos x$ and $y = \exp(x)$, where x is an object affiliated with the class *array* (precisely *ndarray*).⁵ Because x is an array, for example, $y = x^{**}2$ expresses the sequence $\{y_n\}$ with

$$y_0 = x_0^2, y_1 = x_1^2, \dots, y_{100} = x_{100}^2,$$

while `plt.plot(x, y)` expresses a line graph connecting 100 points,

$$(x_0, y_0), (x_1, y_1), \dots, (x_{100}, y_{100}).$$

Next, let us solve an equation using SymPy, an external library for symbolic calculation.

Program: `example03.py/example03.ipynb`

```
1 import sympy
2 from sympy.abc import x, y
3
4 ans1 = sympy.solve([x + 2*y - 1, 4*x + 5*y - 2])
5 print(ans1)
6 ans2 = sympy.solve([x**2 + x + 1])
7 print(ans2)
```

⁵ Python classes and objects are discussed in Sect. 1.6.

```
In [1]: 8 | ans3 = sympy.solve([x**2 + y**2 - 1, x - y])
9 | print(ans3)
```

Line 1: By importing in this way, all the names defined in SymPy can be used with prefix `sympy`. The name `solve` defined in SymPy is used as `sympy.solve`.

Line 2: Import and use symbols `x` and `y` for unknowns in the equation. These are different from Python variables.

Lines 4,5: Solve the simultaneous equations. Pass the left-hand side (LHS) of the equation $LHS = 0$ enclosing with braces `[]` to the function `solve`.

Lines 6,7: Solve the quadratic equation.

Lines 8,9: Solve the simultaneous quadratic equations. Here, we are looking for the intersection of a circle and a straight line.

When using IDLE, write this program in the program editor of IDLE, save it with the file name `example03.py` (the file extension `.py` is automatically added), and run it. When using Jupyter Notebook, create a new notebook, name it `example03.ipynb` (the file extension `.ipynb` is automatically added), write the program in the first cell, and press Shift+Enter to run it.



```
{x: -1/3, y: 2/3}
[{x: -1/2 - sqrt(3)*I/2}, {x: -1/2 + sqrt(3)*I/2}]
[{x: -sqrt(2)/2, y: -sqrt(2)/2}, {x: sqrt(2)/2, y: sqrt(2)/2}]
```

When using IDLE, results are displayed in the shell window. When using Jupyter Notebook, results are displayed right after the input cell. Solutions are given as expressions using fractions and radicals instead of numerical values, and are expressed in a format called a dictionary that uses variable symbols as keys. Also, when there are multiple solutions, they are expressed in a format called a list whose elements are dictionaries. Python lists and dictionaries are explained in detail in Chap. 1.

In the shell or in the notebook, after running the program, we can refer to names defined in it.

```
In [2]: ans1
```

```
Out[2]: {x: -1/3, y: 2/3}
```

Next, we use the external library PIL to directly manipulate the contents of image files. Let us convert a color image to grayscale and reduce the size for use in some of the experiments in this book. Here, we use the file `mypict.jpg` which is the cover picture of this book. You can use a photo image taken with your mobile phone or downloaded from the Internet. In that case, replace the file name with that of yours. Most image file formats such as `jpg/png` and others are available. If the image is too large or you want to use only part of it, use an application GIMP mentioned later in this appendix to reduce or crop it.

Program: `mypict.py/mypict.ipynb`

```
In [1]: 1 | import PIL.Image as Img
2 |
3 | im0 = Img.open('mypict.jpg')
4 | print(im0.size, im0.mode)
5 | im1 = im0.convert('L')
6 | im1.thumbnail((100, 100))
7 | print(im1.size, im1.mode)
8 | im1.save('mypict1.jpg')
```

Line 1: Import the module `Image` of the library `PIL`, and give it the name `Img`.

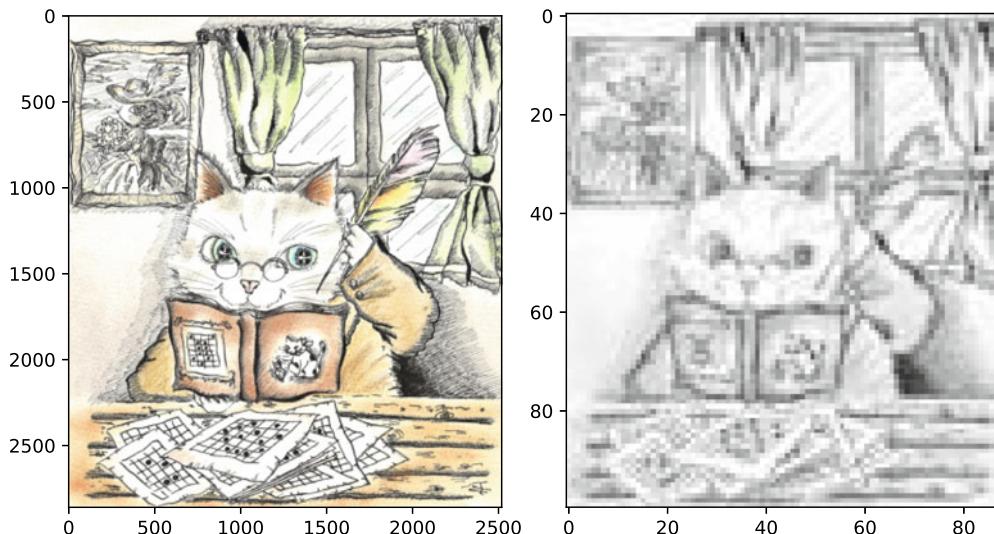


Fig. A.6 The original image (left) and converted image (right)

Line 3: Load the image file `mypict.jpg` (Fig. A.6, left) and name it `im0`. The format of the image file is automatically determined. The image file should be in the same folder as the program, but if it is in a different folder, specify the path, such as `photos/mypict.jpg`.

Line 4: Display the size and color information of image `im0`.

Line 5: From `im0`, create a new image `im1` with the color information converted to grayscale.

Line 6: Reduce the size of the image `im1` to 100×100 pixels. It is converted to fit in the specified size without changing the aspect ratio of the image.

Line 7: Display the size and color information of image `im1`.

Line 8: Save image `im1` in the specified format with a file extension. The converted image of `mypict1.jpg` is shown in Fig. A.6, right.



```
(2519, 2860) RGB
(88, 100) L
```

Figure A.6 is created by using the libraries NumPy, Matplotlib, and PIL.

Program: `mypict1_plot.py`

```
In [1]: 1 import numpy as np
2 import matplotlib.pyplot as plt
3 import PIL.Image as Img
4
5 im0 = Img.open('mypict1.jpg')
6 im1 = np.asarray(im0)
7 plt.imshow(im1, cmap='gray')
8 plt.show()
```

VPython is a great library for easily manipulating 3D images and animations. Anaconda users need to install VPython before using it. Type the following command from the Anaconda PowerShell Prompt of Windows or the terminal of macOS:

```
conda install -c vpython vpython
```

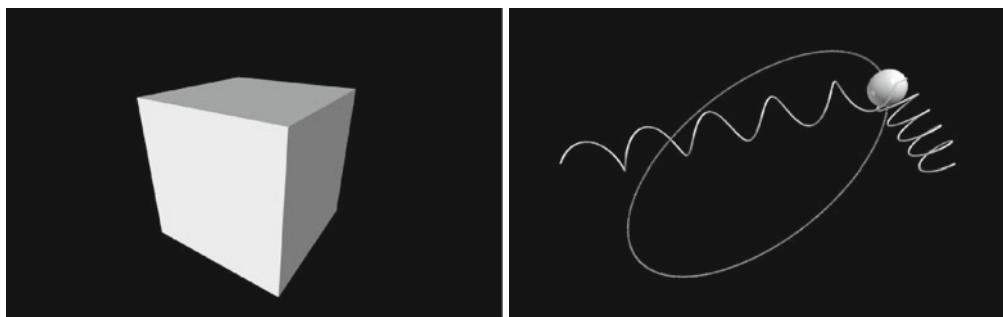


Fig. A.7 3D images with VPython

Let us use this in an interactive mode. While using IDLE, restart it, and while using a Jupyter Notebook, open a new notebook.

```
In [1]: from vpython import *
```

When using IDLE, replace In [*] with the IDLE prompt >>>.

```
In [2]: B=box()
```

When n of In [n] is a consecutive number, it means that the interaction continues in the same shell or notebook. When using IDLE, the browser will open and the cube in 3D is displayed. For notebooks, it will be displayed on the same tab. We can resize it by left-clicking and dragging the sides or corners of the screen, change the viewing direction by moving the mouse while right-clicking on the screen, and move the viewpoint back and forth by clicking left and right at the same time on the screen, or by moving the mouse while holding down the center button (Fig. A.7, left). The color of the cube changes to red.

```
In [3]: B.color=color.red
```

VPython also provides an environment for creating 3D animations (Fig. A.7, right). For example, see `newton.py` in Sect. 9.2.

A.5 Python Syntax

The function `f(N)` of the following program returns a list of prime numbers less than the given integer N. For each integer n with $2 \leq n < N$, we try to divide it in order by the numbers in the list P that stores the prime numbers found so far (the initial state of P is empty). If n is divided by some number, then it is not a prime number, otherwise, it is determined to be a prime number and appended to list P. This method of finding prime numbers is called the *sieve theory*.

Program: `prime.py`/`prime.ipynb`

```
In [1]: 1 | def f(N):
2 |     P = []
3 |     for n in range(2, N):
4 |         q = 1
5 |         for p in P:
6 |             q = n % p
7 |             if q == 0:
```

In [1]:

```

8         break
9     if q:
10        P.append(n)
11    return P
12
13 if __name__ == '__main__':
14     P = f(20)
15     print(P)

```

Line 1: Up to Line 11 is one block for defining `f(N)`. This block is called a *def-block* and the first line is called a *def-statement*.

Line 2: `P` is a list that stores the prime numbers found so far, and is initially empty.

Lines 3–10: A block called *for-block* iterates the contents from Line 4 to Line 10. Line 3 is called a *for-statement*, in which `n` is an integer called a *loop counter* moving between 1 and 10.

Line 4: Set `q` to 1 initially, and if this value becomes 0 by the subsequent operation, `n` is determined not to be a prime number.

Lines 5–8: A block iterates the contents from Line 6 to Line 8, where `p` is a loop counter moving the elements of list `P`.

Line 6: The expression `n % p` calculates the remainder of `n` divided by `p`. Let this remainder be `q`.

Lines 7,8: This is a block called an *if-block*. The expression to the right of “`if`” of the *if-statement* is called a *conditional clause*, and if this condition is true, it executes the content on Line 8 of the block. The conditional clause `q == 0` means an equality and is distinct from the assignment statement `q = 0`. We call `break` a *break statement* and it is an instruction to go out of the innermost loop block in which the `break` statement exists. That is, break out of the loop block Lines 5–8 when `q` that is the remainder of dividing `n` by `p` becomes 0.

Lines 9,10: The `if-block` here is at the same indentation level as the block starting with the `for-block` Lines 5–8. We reach this block when we get out of the previous block either by the `break` statement or by the loop counter moved through all the values. In the former case, the value of `q` is 0, and in the latter case, it is not zero. In Python, nonzero numbers have a Boolean value `True`. If the Boolean value of `q` is `True`, append the integer `n` as a new prime number to list `P`.

Line 11: This is called a *return statement*, and returns list `P` of prime numbers completed.

Line 12: We may include one or more blank lines to make the program code easier to read.

Lines 13–15: The function is not executed just by defining it. What this program actually executes is the block starting with the `if-statement` on Line 14. This `if-statement` is one of the idioms in Python. When this program is run as the main program rather than as a library (that is, instead of being imported into another program), the conditional clause of `__name__ == __main__` will be true and the contents of this block will be executed. There is no rule that the function definition part and the main program must be written separately in the program. If we remove Line 14 and also delete the indentation of both Line 15 and Line 16, these lines will be executed when we import this program as a module into another program. The expression `n` of `f(n)` in the `def-statement` is called a *parameter* (or *formal argument*) of the function `f`. A parameter is a variable with the scope only in a `def-block`. On the other hand, 20 inside `f(20)` at Line 15 is called a *argument* (or *actual argument*) of the function `f`. When `f(20)` is called, 20 is passed to the parameter `n` of `f`, and the contents of the `def-block` are executed for the first time. The value returned by the `return` statement is called a *return value*. The return value of `f(20)` is assigned to `P` on Line 15, and it is printed out on Line 16. We find prime numbers less than 20.

```
[2, 3, 5, 7, 11, 13, 17, 19]
```

Spaces at the beginning of a line are called *indentation*. Indentation in Python represents a nested block structure. The above program has a block structure as shown in Fig. A.8. IDLE and the Notebook



```

def f(N):
    P = []
    for n in range(3, N):
        q = 1
        for p in P:
            q = n % p
            if q == 0:
                break
        if q:
            P.append(n)
    return P

```

```

if __name__ == '__main__':
    P = f(20)
    print(P)

```

Fig. A.8 Block structure of prime.py

will guess where indentation is needed when a line break occurs and move the cursor to the appropriate position. When we press the Enter/Return key on Line 8 and move to Line 9, the cursor automatically comes to the same level as `break` on Line 7. Then we use the BackSpace key to erase the spaces and move the cursor to the desired position where `for` starts on Line 5.

Let us continue to find prime numbers less than 50 in interactive mode.

```

In [2]: f(50)
Out[2]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

```

Variable `P` on Line 15 is distinct from the variable `P` used in the `def`-block on Lines 1–11. The `P` on Line 15 has not changed even after `f(50)` was called.

```

In [3]: P
Out[3]: [2, 3, 5, 7, 11, 13, 17, 19]

```

In Python, telling the processor to use names defined in a library is called *importing*. Often the same name has different functionality depending on the library in which it is defined. There are several ways to import libraries.

The first is the following way which we used for Numpy in Program example02.

```
from numpy import linspace, pi, sin, cos, exp
```

Listing the names defined in the library `numpy`, we can use the names `linspace`, `pi`, `sin`, `cos`, and `exp` listed there as they are. If it is annoying to list each name defined in the library that we want to use, here is the way we used in the VPython example in the last section.

```
from vpython import *
```

While useful, this can lead to unexpected rewriting of previously defined names and name clashing when using several libraries at the same time. Here is a way that allows us to use all the names defined in the library, avoiding such a risk. This is used in `example03` in the SymPy example above.

```
import sympy
```

We can use all the names defined in SymPy by prefixing them with the prefix `sympy` like `sympy.solve`. If the name of the library is too long, we can use a shortcut as we did in `example02`.

```
import matplotlib.pyplot as plt
```

We can use `plt` for `matplotlib.pyplot`.

We are able to import the program `prime.py` in this section as a library. Try the following three ways explained above to import `prime` as a library.

1. `from <library> import <name>1, <name>2, ..., <name>n`

```
from prime import f
print(f(50))
```

2. `import <library>`

```
import prime
print(prime.f(50))
```

3. `import <library> as <alias>`

```
import prime as pr
print(pr.f(50))
```

When we want to use `prime.ipynb` made with Jupyter Notebook as a library, convert it to `prime.py` by the following command.

```
jupyter nbconvert --to python prime.ipynb
```

Alternatively, we can open `prime.ipynb` in Jupyter Notebook and convert it with “File>Download as>Python (.py)” from the menu bar. The converted file `prime.py` will be in the Downloads folder.

A.6 Other Tools (Supplementary)

- **LATEX**

LATEX is a widely used system for typesetting documents containing mathematical formulas. In this book, LATEX codes may be used to make it easier to see the calculation results of SymPy. We will also use it to typeset automatically generated exercises in some chapters (e.g., Exercises 4.2 and 4.7).

```
In [1]: from sympy import solve, latex
from sympy.abc import a, b, c, x
A = solve([a*x**2+b*x+c], [x]); A
```

```
Out[1]: [(-b/(2*a) - sqrt(-4*a*c + b**2)/(2*a),),
 (-b/(2*a) + sqrt(-4*a*c + b**2)/(2*a),)]
```

```
In [2]: print(latex(A[0][0]))
```

```
Out[2]: - \frac{b}{2\ a} - \frac{\sqrt{-4\ a\ c + b^{2}}}{2\ a}
```

When using Jupyter Notebook, change the type of a new cell from code to Markdown. Next, copy and paste the above LATEX code into the cell and enclose it with \$’s.



```
$ - \frac{b}{2 a} - \frac{\sqrt{-4 a c + b^2}}{2 a}
```

The icon represents the markdown cell being edited. After editing the cell, press Shift+Enter, then we have the typeset formula below in the cell:

$$\frac{b}{2a} - \frac{\sqrt{-4ac+b^2}}{2a}$$

When using IDLE, we need a L^AT_EX system installed.

Windows: To use the L^AT_EX system, install TeX Live. We can get the system from <https://www.tug.org/texlive/>.

macOS: To use the L^AT_EX system, install MacTex. For that we recommend to use the package manager called Homebrew commonly used by macOS users. For more information visit <https://brew.sh/>. After installing Homebrew, install MacTex with the following commands.

```
brew install --cask mactex
sudo tlmgr update --self --all
```

Raspberry Pi OS: We can install it using the apt command as below.

```
sudo apt install texlive-science -y
sudo apt install texlive-latex-extra -y
sudo apt install texlive-font-utils -y
sudo apt install texworks -y
```

If installed, we can use an application called TeXworks, which is an integrated environment for editing a L^AT_EX source code and typesetting it.

L^AT_EX template file: template.tex

```
\documentclass{standalone}
\usepackage{amssymb, amsmath}

\begin{document}

$ - \frac{b}{2 a} - \frac{\sqrt{-4 a c + b^2}}{2 a}$

\end{document}
```

Typesetting the above, we will have a pdf file of the cropped image of the formula
$$\frac{b}{2a} - \frac{\sqrt{-4ac+b^2}}{2a}$$
. Use this as a template by replacing the code between \$ and \$ with another L^AT_EX code.

If you want to upload a formula typeset in L^AT_EX to your Web home page, the easiest way is to convert the Jupyter Notebook page as it is to an HTML file. Alternatively, you can convert the pdf file into another image format using GIMP described below and include it directly into the HTML.

- **GIMP**

The output of L^AT_EX described above will be a pdf file, but if you would like to convert it to another image format for inserting into another document, you may use GIMP. It is a cross-platform editor

for image manipulation and converting the file formats. We can install it as follows depending on the environment.

Windows: Get the installer from <https://www.gimp.org/downloads/>.

macOS:

```
brew install --cask gimp
```

Raspberry Pi OS:

```
sudo apt install gimp -y
```

By using GIMP, you can retouch your favorite photos or any images obtained from the Internet. For example, we can get the same result through the GUI screen by grayscaling or reducing the image as done in Sect. A.4 (Fig. A.9).

- **PyX**

A Python library PyX allows us to draw a figure containing mathematical formulas typeset by L^AT_EX.

Windows:

```
pip install pyx
```

macOS:

```
python3 -m pip install pyx
```

Raspberry Pi OS:

```
sudo apt install python3-pyx -y
```

Alternatively, we can also install it by the pip command. Depending on our environment, add sudo before pip.

```
python3 -m pip install pyx
```

Some of the figures in this book are drawn using PyX. Here is the program to create Fig. 1.2 (right) of the complex plane in Sect. 1.3 for example.

Program: comp.py

```
In [1]: 1  from pyx import *
2
3  C = canvas.canvas()
4  text.set(text.LatexRunner)
5  text.preamble(r'\usepackage{amsmath}')
6  text.preamble(r'\usepackage{amsfonts}')
7  text.preamble(r'\usepackage{amssymb}')
8
9  C.stroke(path.circle(0, 0, 1),
10           [style.linewidth.thick, deco.filled([color.gray(0.75)])])
11 C.stroke(path.line(-5, 0, 5, 0),
12           [style.linewidth.THick, deco.earrow.Large])
```

```
In [1]: 13 C.stroke(path.line(0, -5, 0, 5),
14     [style.linewidth.THick, deco.earrow.Large])
15 C.stroke(path.line(1, -0.1, 1, 0.1))
16 C.stroke(path.line(-0.1, 1, 0.1, 1))
17 C.text(-0.1, -0.1, r"\huge 0",
18     [text.halign.right, text.valign.top])
19 C.text(1, -0.2, r"\huge 1",
20     [text.halign.left, text.valign.top])
21 C.text(-0.2, 1, r"\huge $i$",
22     [text.halign.right, text.valign.bottom])
23 C.stroke(path.circle(2, 3, 0.05),
24     [deco.filled([color.grey.black])])
25 C.text(2.1, 3.1, r"\huge $z=x+iy$",
26     [text.halign.left, text.valign.bottom])
27 C.stroke(path.line(2, 3, 2, 0),
28     [style.linewidth.thick, style.linestyle.dashed])
29 C.stroke(path.line(2, 3, 0, 3),
30     [style.linewidth.thick, style.linestyle.dashed])
31 C.text(2, -0.3, r"\huge $x$",
32     [text.halign.center, text.valign.top])
33 C.text(-0.1, 3, r"\huge $iy$",
34     [text.halign.right, text.valign.middle])
35 C.writePDFfile('comp.pdf')
```

• Graphviz

Graphs showing the dependencies in some items can be created with a Python wrapper for an application called Graphviz.

Windows:

```
conda install python-graphviz
```

macOS:

```
brew install graphviz
python3 -m pip install graphviz
```

Raspberry Pi OS:

```
sudo apt install python3-graphviz -y
```

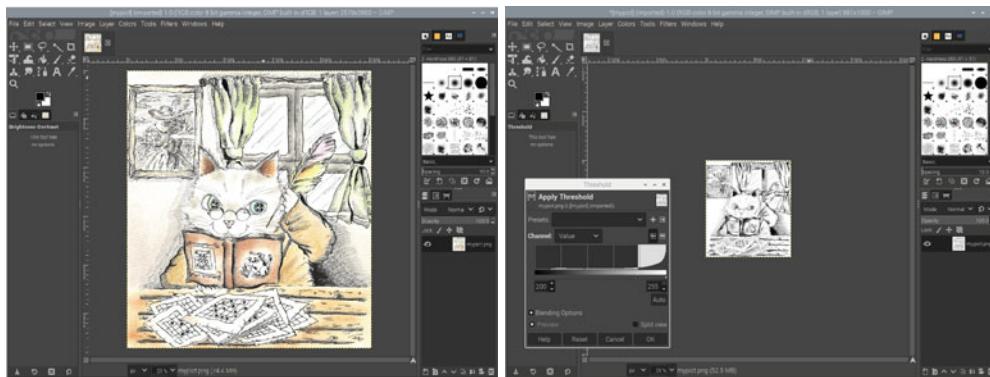
If we want to use the `pip` command, we must install Graphviz independently with the `apt` command. Depending on our environment, add `sudo` before `pip`.

```
sudo apt install graphviz -y
python3 -m pip install graphviz
```

This is the source code for drawing the diagram in the Preface.

Program: book_diagram.py

```
In [1]: 1 from graphviz import Digraph
2
3 G = Digraph(format='pdf')
4 G.attr('node', shape='box')
5
6 G.node('Apx', 'Appendix', style='dashed', shape='oval')
7 G.node('ch1', 'Chapter 1: Mathematics and Python')
8 G.node('ch2', 'Chapter 2: Linear Space and Linear Mapping')
9 G.node('ch3', 'Chapter 3: Basis and Dimension')
```

**Fig. A.9** GIMP

```
In [1]: 10 | G.node('ch4', 'Chapter 4: Matrix')
11 | G.node('ch5', 'Chapter 5: Elementary Operation and Matrix Invariants')
12 | G.node('ch6', 'Chapter 6: Inner Product and Fourier Expansion')
13 | G.node('ch7', 'Chapter 7: Eigenvalue and Eigenvector')
14 | G.node('ch8', 'Chapter 8: Jordan Normal Form and Spectrum')
15 | G.node('ch9', 'Chapter 9: Dynamical System')
16 | G.node('ch10', 'Chapter 10: Applications and Development of Linear Algebra')
17 |
18 | G.edge('Apx', 'ch1')
19 | G.edge('ch1', 'ch2')
20 | G.edge('ch2', 'ch3')
21 | G.edge('ch2', 'ch6')
22 | G.edge('ch3', 'ch4')
23 | G.edge('ch4', 'ch5')
24 | G.edge('ch4', 'ch6')
25 | G.edge('ch5', 'ch7')
26 | G.edge('ch5', 'ch10')
27 | G.edge('ch6', 'ch7')
28 | G.edge('ch6', 'ch8')
29 | G.edge('ch6', 'ch10')
30 | G.edge('ch7', 'ch8')
31 | G.edge('ch7', 'ch10')
32 | G.edge('ch8', 'ch9')
33 |
34 | G.render('diagram')
```

- **Audacity**

In Chaps. 2 and 6, we treat audio data. Audacity is a GUI tool which can be used as recorder, editor, and analyzer for audio data (Fig. A.10).

Windows: Get the installer from <https://www.audacityteam.org/download/>.

macOS:

```
brew install --cask audacity
```

Raspberry Pi OS:

```
sudo apt install audacity -y
```

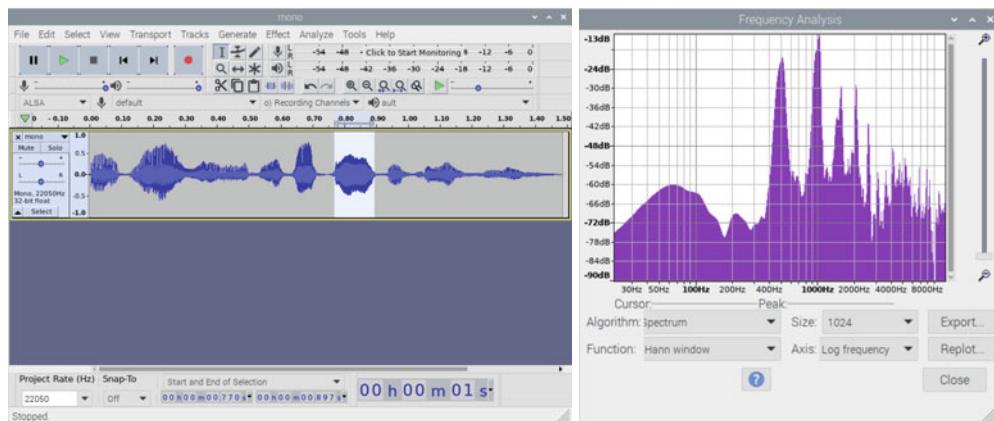


Fig. A.10 Audacity

Afterword and Bibliography

We wrote this book for three purposes; as a textbook of linear algebra, as an introduction to Python programming, and as a guide to the application of linear algebra.

First, it covers all the basics of linear algebra. It is possible to read only the mathematical aspects, skipping the Python codes and their descriptions. The reader who teaches in a linear algebra class can lecture on only the mathematical parts of this book and use the Python parts for exercises.

Second, we have explained a lot of Python codes but not the language itself. If the readers want to learn Python grammar, they should refer to other resources. Hopefully, while learning linear algebra in this book, the readers will naturally be able to write Python codes.

As a commentary on the application of linear algebra, this book covers several important topics. Still, these may be biased toward the tastes of the authors. The linear algebra applications are wide ranging, and some specialized books on each of these themes are called masterpieces. However, readers of these books may be overwhelmed and lose track of the essence. The third purpose will be achieved if this book provides the readers with some hints to select literature to read according to their own purposes.

Below are some references that will be useful to the readers in the future, including those we referred to when writing this book.

The following is a great work, written with a similar purpose to this book, and has many overlapping parts explained from some different perspectives. Reading it together with this book will help create a deeper understanding.

[1] Klein, P.N.: *Coding the Matrix: Linear Algebra Through Computer Science Applications*. Newtonian Press (2013).

There are many books on Python, but for those who are new to Python and want to get an overview, we introduce here just one book. However, Python 3, with which the book is compliant, is a little old. We recommend that the readers refer to the documentation on the Internet for a new Python version.

[2] Lubanovic, B.: *Introducing Python: Modern Computing in Simple Packages*. O'Reilly (2014).

See the following for more on Python.

[3] Guttag, J.V.: *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. MIT Press (2016).

There is not much literature on VPython.

[4] Russell, J., Cohn, R.: *VPython*. Book on Demand Ltd. (2012).

[5] Scherer, D., Dubois, P., Sherwood, B.: VPython: 3D Interactive Scientific Graphics for Students. *Computing in Science and Engineering*, 2 (2000).

Here are some standard linear algebra textbooks:

[6] Lang, S.: Linear Algebra (Undergraduate Texts in Mathematics), 3rd ed. Springer (1987).

[7] Nair, M.T., Singh, A.: Linear Algebra. Springer (2018).

[8] Strang, G.: Introduction to Linear Algebra, 5th ed. Wellesley-Cambridge Press (2021).

[9] Gantmacher, F.R.: Theory of Matrices. Chelsea, New York (1959).

[10] Anton, H., Busby, R.C.: Contemporary Linear Algebra. Wiley (2003).

[11] Bhatia, R.: Matrix Analysis, Graduate Texts in Mathematics. Springer (1996).

[12] Golub, G.H., Van Loan, C.F.: Matrix Computations, 4th ed. The Johns Hopkins University Press (2013).

[13] Hiai, F., Petz, D.: Introduction to Matrix Analysis and Applications. Springer (Hindustan Book Agency) (2014).

Our book details the Jordan decomposition and the singular value decomposition with some applications, but it does not touch on the Schur decomposition, the QR decomposition, and others. These are also useful in numerical calculations of matrices. The next book covers many topics that are not covered in this book.

[14] Hogben, L., Brualdi, R., Stewart, G.W.: Handbook of Linear Algebra, 2nd ed. CRC Press (2014).

We have included little discussion about the fast Fourier transform, Simpson's rule in numerical integration, and the Runge–Kutta method for differential equations. Numerical linear algebra with error analysis is also an important theme, but it was not discussed in detail in this book. We refer the reader to the next book on these themes.

[15] Stoer, J., Bulirsch, R.: Introduction to Numerical Analysis, 3rd ed. Text in Applied Mathematics. Springer (2010).

Matrix computation with floating-point approximation has the problem of zero judgment. We would like to recommend the following article, which shows a general theory for performing correct zero judgment.

[16] Shirayanagi, K., Sweedler, M.: A Theory of Stabilizing Algebraic Algorithms, Technical Report 95-28. Mathematical Sciences Institute, Cornell University (1995).

We started with logic and basics of mathematics in Chap. 1. The following books are already classics.

[17] Mendelson, E.: Introduction to Mathematical Logic, 5th ed. CRC Press (2010).

[18] Halmos, P.R.: Naive Set Theory. Springer (1998).

[19] Bourbaki, N.: Elements de Mathematique, Lible I, Theorie de Ensembles. Hermann.

[20] Kelly, J.L.: General Topology. Van Nostrand (1955).

[21] McCoy, N.H.: Introduction to Modern Algebra. Allyn and Bacon (1960).

[22] Kolmogorov, A.N., Fomin, S.V.: Introductory Real Analysis, 4th ed. Dover (1975).

[23] Simmons, G.S.: Introduction to Topology and Modern Analysis. Krieger Pub. Co. (2003) (McGraw-Hill, 1963).

We devoted a lot of space to inner products and norms. This is intended as an introduction to Fourier analysis and leads to the field of functional analysis.

[24] Halmos, P.R.: Finite-Dimensional Vector Spaces. Undergraduate Texts in Mathematics. Springer (1993).

- [25] Kato, T.: Perturbation Theory for Linear Operators, 2nd ed. Springer (1976).
- [26] Yosida, K.: Functional Analysis. Springer (1995).
- [27] Papoulis, A.: The Fourier Integral and Its Applications. McGraw-Hill (1962).
- [28] Körner, T.W.: Fourier Analysis. Cambridge (1988).
- [29] Stein, E.M., Shakarchi, R.: Fourier Analysis, An Introduction. Princeton University Press (2003).
- [30] Collatz, L.: Functional Analysis and Numerical Mathematics. Academic Press Inc., New York (1966).

Lastly, we list some references on the fields of application of linear algebra.

- [31] Rao, C.R., Mitra, S.K.: Generalized Inverse of Matrices and Its Applications. Wiley (1973).
- [32] Ben-Israel, A., Greville, T.N.E.: Generalized Inverses—Theory and Applications. Springer (2003).
- [33] Hirsch, M.W., Smale, S.: Differential Equations, Dynamical Systems, and Linear Algebra. Academic Press (1974).
- [34] Kemeny, J.G., Snell, J.L.: Finite Markov Chains. D. Van Nostrand Company (1960).
- [35] Preston, C.J.: Gibbs States on Countable Sets. Cambridge University Press (1974).
- [36] Kindermann, R., Snell, J.L.: Markov Random Fields and Their Applications. American Mathematical Society (1980).
- [37] Harville, A.: Matrix Algebra from a Statistician's Perspective. Springer (1997).
- [38] Rao, C.R.: Linear Statistical Inference and Its Applications, 2nd ed. (1977).
- [39] Anderson, T.W.: An Introduction to Multivariate Statistical Analysis, 3rd ed. Wiley (2003).
- [40] Kalman, R.E.: A new approach to linear filtering and prediction problems. In: Transactions of the American Society, Mechanical Engineers, vol. 82D, pp. 35–45. (1960).
- [41] Preston, C.J.: Gibbs States on Countable Sets. Cambridge University Press (1974).
- [42] Schatten, R.: Norm Ideals of Completely Continuous Operators. Springer (1960).
- [43] Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006).
- [44] Erkki, O.: Subspace Methods of Pattern Recognition. Research Studies Press (1983).

Symbol Index

Symbols

- 2^A (power set), 6
 $=$ (equality of sets), 6
 $C([a, b], \mathbb{K})$ (space of continuous functions), 133
 $N(A)$ (numerical radius), 173
 $P(x)$ (propositional function), 2
 Y^X (set of all mappings from X to Y), 12
 \Leftrightarrow (equivalence of propositions), 2
 \Rightarrow (implication), 2
 \perp (orthogonal), 123
 \cap (cap, intersection), 7
 \complement (complement), 8
 \cup (cup, union), 7
 $\int_a^b f(x) dx$ (definite integral), 133
 $\bigcap_{i \in I} W_i$ (intersection of subspaces), 38
 $\lim_{k \rightarrow \infty} x_k$ (limit of vector sequence), 144
 $\sum_{k=1}^{\infty} A_k$ (infinite summation of matrices), 175
 \emptyset (empty set), 6
 \in (belongs to), 6
 $(-\infty, \infty)$ (all real numbers), 10
 $(-\infty, a]$ (all real numbers less than a), 10
 $(-\infty, a]$ (all real numbers less than or equal to a), 10
 (a, ∞) (all real numbers greater than a), 10
 (a, b) (open interval), 10
 $(a, b]$ (open-closed interval), 10
 $[a, \infty)$ (all real numbers greater than or equal to a), 10
 $[a, b)$ (closed-open interval), 10
 $[a, b]$ (closed interval), 10
 $[a_{i_1 i_2 \dots i_n}]_{i_1=1}^{k_1},_{i_2=1}^{k_2}, \dots,_{i_n=1}^{k_n}$ (n -dimensional array), 18
 $[a_{ij}]_{i=1}^k,_{j=1}^l$ (matrix), 18
 $\langle A \rangle$ (span), 48
 $\langle a_1, a_2, \dots, a_n \rangle$ (span), 48
 $|i\rangle$ (bra-vector), 248
 $\langle AB_{HS} \rangle$ (Hilbert–Schmidt inner product), 241
 $\{x_n\}_{n=1}^{\infty}$ (sequence), 67
 $\{A_k\}_{k=1}^{\infty}$ (matrix sequence), 174
 $\{x_k\}_{k=1}^{\infty}$ (vector sequence), 144
 $|i\rangle$ (ket-vector), 248
 \Leftrightarrow (if and only if), 1
 $\llbracket A \rrbracket$, 240
 \mathbb{C} (complex number), 6
 \mathbb{K} (scalar field), 29
 \mathbb{N} (natural number), 6
 \mathbb{R} (real number), 6
 diag (diagonal matrix), 153
 proj_W (orthogonal projection), 156
 $|\det A|$ (determinant of matrix), 102
 $|z|$, 3
 \neg (not), 1
 \oplus (direct sum of linear mappings), 179
 \oplus (direct sum of linear matrices), 180
 \oplus (direct sum), 64
 \otimes (tensor product), 244
 A^\dagger (generalized inverse matrix), 237
 A' (cofactor matrix), 118
 A^* (adjoint matrix), 89
 A^T (transposed matrix), 89
 A^p (p -th power matrix), 88
 A^{-1} (inverse matrix), 86
 $E_1^{(i,j,c)}$ (elementary matrix), 93
 $E_2^{(i,j)}$ (elementary matrix), 93
 $E_3^{(i,c)}$ (elementary matrix), 93
 I (unit matrix), 81
 I_n (unit matrix of order n), 81
 O (zero matrix), 81
 O_{mn} (zero matrix of shape (m, n)), 81
 $\exp(A)$ (matrix exponential), 176
 \mapsto (maps to), 13
 Im (imaginary part), 3
 Re (real part), 3
 Tr (trace), 110
 \det (determinant), 102
 id_X (identity mapping), 14
 kernel (kernel), 41

-
- range (range), 13
 rank (rank), 99
 \neq (inequality of sets), 6
 \notin (does not belong to), 6
 \bar{z} (complex conjugate), 3
 $\sqrt[2]{A}$ (square root of matrix), 168
 $\sqrt[k]{A}$ (k -th root of matrix), 168
 \sqrt{A} (square root of matrix), 168
 $\{\}$ (empty set), 6
 $\rho(A)$ (spectral radius), 173
 \rightarrow (implies), 1
 \setminus (set minus), 7
 $\sigma(p_1, p_2, \dots, p_n)$ (signature of permutation), 102
 $\stackrel{\text{def}}{=}$ (definition), 3
 \subseteq (subset), 6
 \subsetneq (proper subset), 6
 \times (direct product, Cartesian product), 10
 Δ (symmetric difference), 221
 \restriction_W (restriction of domain onto W), 179
 \vee (or), 1
 \wedge (and), 1
 e^A (matrix exponential), 176
 $f : X \rightarrow Y$ (mapping), 12
 $f(A)$ (image), 13
 f^{-1} (inverse mapping), 14
 $f^{-1}(B)$ (inverse image), 13
 $g \circ f$ (composition), 14
 $p(A)$ (polynomial of matrix), 175
 $x > y$ (inequality of vectors), 198
 $x \geq y$ (inequality of vectors), 198
 $x \geqq y$ (inequality of vectors), 198

Python Index

Symbols

<=, 1
*, 61
**, 5, 278
*=, 274
+*, 274
-*, 274
/*, 274
;, 8
=, 31
==, 1
>>>, 273
#, 140
%, 9, 284
%s, 16
&, 9
_, 23
__init__, 150
'__main__', 283
__name__, 283
|, 9

A

abs, 5, 174
all, 9
and, 1

B

\, 130
break, 115, 285

C

class, 150, 223
complex
 1.0j, 5
 1j, 5
 .conjugate, 5

.imag, 5
.real, 5

D

def, 12, 21, 284
dict, 15
{}, 8
divmod, 138

E

elif, 43, 109
else, 13, 100
enumerate, 129
eval, 22

F

False, 1
float, 4
for ... in ..., 15, 280, 285
from ... import ..., 286
functools
 reduce, 102, 103, 183, 187

G

gram_schmidt, 129
graphviz, 290

I

if, 13, 284
... if ... else ..., 13
import ... as, 286
in, 9
int, 4

J`jpg, 278`**L**`lambda, 13``len, 26``list, 14``[], 18``.append, 15, 283``.pop, 23``.sort, 16`**M**`map, 36``math, 277``matplotlib.pyplot, 279``axis, 22``hist, 228``imshow, 26, 261``plot, 280``quiver, 31, 53, 171``scatter, 22, 66``show, 278``subplot, 137``subplots, 26``subplots_adjust, 26``text, 83``tick_params, 26, 234, 261``xlim, 44, 280``ylim, 280``yticks, 228``matplotlib.pyplot.figure, 36``add_subplot, 36, 139``set_xlim, 44, 139``set_ylim, 139``set_zlim, 139``max, 100``min, 100``mpl_toolkits.mplot3d``Axes3D, 139`**N**`None, 233``not, 1``numpy, 279``arange, 43``array, 19, 69``.all, 70``.any, 70``.astype, 26``.conj, 90``.conjugate, 90``.copy, 70``.cumsum, 233``.dot, 76, 124``.max, 19``.reshape, 26, 71``.shape, 19, 79``.sum, 19``.T, 89, 90``.tolist, 70``.transpose, 90``concatenate, 65``conj, 124``cos, 279``cumsum, 135``diag, 153, 241``dot, 76, 124``dtype, 20``e, 5``exp, 279``eye, 81``fromfile, 26``identity, 234``inf, 125``inner, 124``isreal, 171``matrix, 79, 177, 213``.H, 90``ones, 137``outer, 246``linspace, 279``pi, 5, 279``sin, 279``sort, 135``sqrt, 127``tan, 83``tensordot, 246``trace, 242``uint, 20``vdot, 124``zeros, 19, 81``numpy.fft, 147``fft, 147``ifft, 147``numpy.linalg``cond, 242``det, 108``eig, 161, 169, 258``eigh, 161, 169, 258``inv, 87``matrix_rank, 61, 100``norm, 125, 242``pinv, 240, 241``solve, 115``svd, 240, 241``numpy.polynomial, 142``legendre, 234``numpy.random``choice, 73``normal, 48``permutation, 100``randint, 73``seed, 57``shuffle, 115``uniform, 127`

O

or, 1

- .eigenvals, 159, 161
- .eigenvecs, 161, 162
- .evalf, 213
- .H, 90

P

- pdf, 278
- PIL.Image, 19, 281
- png, 278
- prime, 286
- print, 12, 70, 275
- pyx, 288

- .jordan_form, 190
- .norm, 169
- .rank, 100
- .row_join, 65, 165
- .simplify, 83
- .subs, 160
- .T, 90

- oo, 143
- pi, 135

- Rational, 80

- .evalf, 51

- S, 87

- simplify, 169

- sin, 82

- solve, 49, 281

- sqrt, 281

- Symbol, 163

- symbols, 108

- var, 96, 160

- eye, 81

- zeros, 81

- sympy.abc, 286

- a,b,c, 286

- theta, 82

- x,y, 281

- sympy.polys.orthopolys, 143

R

- range, 9, 283

- repr, 21

- return, 12, 21, 283

S

- scipy.io.wav, 43

- read, 43

- write, 44

- scipy.io.wavfile
- wav, 150

- self, 151

- set, 8, 9

- .intersection, 9

- .issubset, 9

- .union, 9

- set(), 8

- sorted, 16

- sound

- Sound, 148, 150

- str, 14, 21

- ", 14

- ', 14

- .format, 16

- split, 290

- sum, 26, 91, 102

- sympy, 281

- cos, 82

- det, 160, 163

- diag, 189

- E, 178

- eye, 86, 160

- factor, 160

- factor_list, 163

- I, 90, 281

- Integer, 80

- integrate, 135, 143

- Lambda, 160

- latex, 73, 84, 286

- Matrix, 65, 80

- .C, 90

- .charpoly, 163

- .col_join, 65

- .copy, 97

- .det, 108

- .diagonalize, 168

T

- time, 91

- time, 91

- tkinter, 23, 24, 223

- True, 1, 70, 284

- tuple, 10

- (), 18

V

- vpython, 283

- arrow, 31

- box, 94, 283

- canvas, 57, 94

- camera, 94

- capture, 94

- color, 57, 129, 283

- curve, 57, 94, 255

- hat, 130

- helix, 206

- label, 129

- mag, 31

- points, 48, 57

- proj, 129, 130

- rate, 206

- sphere, 206

- vec, 31, 48

- vector, 31

W

wav, [43](#)

while, [100, 115](#)

with open(...) as, [21, 26](#)

.readlines, [255](#)

.read, [22](#)

.write, [21](#)

Z

zip, [22, 77](#)

Index

A

Absolute value, 3
Acceptance function, 219
Additivity, 122
Affine space, 233
Algorithm, 126
Aperiodic, 217
Argument, 3, 284
 actual —, 284
 formal —, 284
 name —, 31
 position —, 31
Array, 18, 280
Audacity, 290
Augmented assignment
 — operator, 274
 — statement, 274
 —operator, 97
 —statement, 98
Axioms of
 — the inner product, 122
 — the norm, 121

B

Backward substitution, 114
Basis, 55
Belongs to, 6
Bijection, 14
Bijective, 14
Bilinear, 245
Binary functions, 236
Block, 284
Boolean value, 1
Boundary, 221
Branch, 216
Broadcast, 36

C

Cartesian product, 10
Cauchy sequence, 175
Cell, 276
Characteristic
 —equation, 157
 —polynomial, 157
Class, 11, 14
Class initialization method, 151
Closed under
 — scalar multiplication, 29, 38
 — vector inversion, 38
 — vector summation, 29, 38
Cofactor, 118
Column, 18
Commute, 176
Compiler, 19
Complement, 8
Complete, 175
Complex plane, 6
Component, 10, 18
Composition, 14, 40
Condition number, 242
Conditional clause, 284
Conjugate
 complex —, 3
Constructor, 151
Contains, 6
Converge, 144, 174
Correlation constant, 254
Cramer's formula, 118
Csv-format, 256
Cutoff frequency, 136

D

Data type, 11
Decomposes, 180
Definable, 79
Derivative, 203

- Derived function, 203
 Destructive, 17
 Determinant, 102
 Diagonal elements, 69
 Dictionary, 15
 Difference set, 7
 Differentiable, 203
 Differential, 203
 Differential operator, 204
 Dimension, 60
 Direct product, 10
 Direct sum, 63, 64
 - decomposition, 180
 - of linear mappings, 179
 - of matrices, 180
 Distributive law, 122
 Domain, 13
- E**
 Edge, 216
 Edit window, 274
 Eigen
 - equation, 157
 - polynomial, 157
 - space, 161
 - value, 156
 - vector, 156
 Element, 5, 10, 18, 69
 Elementary
 - column operation, 97
 - matrix, 94
 - matrix operation, 97
 - row operation, 97
 Embedding, 42
 Empty set, 6
 Equivalence
 - class, 243
 - relation, 243
 Equivalent, 2
 Ergodic
 - hypothesis, 219
 - theorem, 219
 Euler's formula, 4
 Event, 250
 Event-driven programming, 25
 Existence of
 - inverse, 29
 - zero, 29
 Expansion, 56
 Expectation, 250
 Expected value, 250
 Exponent law, 88
 Exponential distribution, 229
 Extensional definition, 5
 External, 64
- F**
 F-string, 16
- False, 1
 Family, 38
 Filter
 - Kalman —, 269
 - low-pass —, 136
 Finite set, 6
 Floating-point binary number, 280
 Forward elimination, 114
 Fourier
 - analysis, 146
 - coefficient, 128
 - expansion, 128
 - integral, 146
 - series, 236
 - series expansion, 139, 146
 - transform, 146
 - discrete — transform, 147
 - discrete inverse — transform, 147
 - fast — transform, 147
 Free variable, 6
 Frequency, 136
 Frobenius number, 218
 Function, 12
- G**
 Galois theory, 158
 Gaussian elimination, 114
 Generalized eigenspace, 182
 Generated by, 48
 GIMP, 287
 Gram–Schmidt orthogonalization, 126
 Graph, 13
- H**
 Hadamard product, 72
 Has the zero vector, 38
 Hermitian property, 122
 Homogeneity, 122
 - absolute—, 121
 - conjugate—, 122
- I**
 IDE, 273
 If and only if, 2
 Image, 13, 219
 - inverse —, 13
 Imaginary part, 3
 Immutable, 17
 Implies, 2
 Importing, 285
 Indentation, 284
 Index, 18
 Infinite, 6
 Initial value, 206
 Injective, 13
 Inline for sentence, 10
 Inner product, 122

- space, 122
- standard—, 122
- Instance, 151
- Intensional definition
 - (of sets in mathematics), 6
- Intentional definition
 - of lists (Python), 48
- Interaction, 221
- Interactive mode, 274
- Internal, 64
- Interpreter, 19
- Intersection, 7
- Interval, 10
 - closed —, 10
 - closed–open —, 10
 - finite —, 10
 - infinite —, 10
 - open —, 10
 - open–closed —, 10
- Invariant, 93, 168
- Irreducible, 217, 227
- Isomorphic, 42

- J**
- Jordan
 - block, 189
 - decomposition, 191
 - normal form, 183

- K**
- k -th root, 168
- Karhunen–Loëve expansion, 259
- Kernel, 41
- KL expansion, 259
- Kronecker product, 247
- Kronecker’s delta, 111

- L**
- Lambda expression, 13
- L^AT_EX, 286
- Least squares approximation, 135
- Length, 17
- Library, 277
- Limit, 144
- Linear
 - isomorphism, 40
 - map, 39
 - mapping, 39
 - space, 29
 - transformation, 39
 - combination, 47
 - system, 111
 - complex — space, 30
 - conjugate — mapping, 132
 - differential equation, 210
 - finite-dimensional—space, 48
 - infinite-dimensional—space, 60
- quotient — space, 243
- real — space, 30
- sum of — mappings, 40
- system of—equations, 111
- Linearity, 122
 - of expectation, 254
 - property, 133
 - conjugate—, 122
- Linearly
 - dependent, 52
 - independent, 52, 63
- List, 14
- Loop counter, 284

- M**
- MacOS, 272
- Mapping, 12
 - identity —, 14
 - inverse —, 14
- Markov
 - random field, 219
 - spatial — property, 219
 - stationary — chain, 215
- Mathematical induction, 3
- Matrix, 18
 - exponential, 176
 - polynomial, 175
 - power, 88
 - product, 78
 - representation, 74
 - unit, 111
 - adjoint—, 89
 - basis change—, 88
 - coefficient—, 115
 - cofactor—, 118
 - covariance —, 254
 - diagonal—, 69
 - diagonalizable—, 164
 - extended coefficient—, 115
 - generator —, 227
 - Hermitian—, 154
 - inverse—, 86
 - k -th root of—, 168
 - Moore–Penrose generalized inverse —, 237
 - nilpotent—, 190
 - nonnegative definite—, 154
 - nonnegative—, 194
 - normal—, 165
 - orthogonal—, 154
 - positive definite—, 154
 - positive semi-definite—, 154
 - positive—, 194
 - real—, 154
 - regular—, 86
 - representation—, 74
 - rotation—, 77
 - square root of—, 168
 - square—, 69
 - state transition —, 217

- stochastic —, 217
 symmetric—, 154
 transposed—, 89
 unit—, 81
 unitary—, 154
 upper triangular—, 99
 variance —, 254
 variance–covariance —, 254
 zero—, 72
 Method, 9, 17
 Midpoint formula, 134
 Minor, 118
 Module, 277
 Multiplicity, 161
 Mutable, 17
- N**
 n -dimensional array, 18
 Natural number, 6
 Ndarray, 19, 65, 280
 Negative correlation, 254
 Node, 216
 Norm, 121
 L^1 - —, 134
 L^2 - —, 134
 L^∞ - —, 134
 l^1 - —, 122
 l^2 - —, 124
 l^∞ - —, 122
 Euclidean —, 124
 Euclidean—, 171
 Frobenius —, 242
 Hilbert–Schmidt —, 241
 matrix—, 171
 nuclear —, 242
 root mean square —, 268
 trace —, 241
 Normalize, 126
 Normed space, 121
 Numerical semigroup, 217
- O**
 Object, 14
 Observable, 249
 Observed value, 250
 One-parameter semigroup, 227
 One-to-one, 13
 Onto, 13
 Orbit, 210
 Order, 69
 Ordered pair, 10
 Orthogonal, 123
 — complement, 132
 — polynomial, 140
 — projection, 126
 — system, 125
 —projection, 156
 normal — system, 125
- Orthonormal
 — basis, 126
 — system, 125
 Outer product, 246
- P**
 Parameter, 284
 Permutation, 101
 even—, 101
 identity—, 101
 odd—, 101
 Perron–Frobenius eigenvalue, 201
 Phase average, 219
 Phase space, 210
 Pivot, 100
 Pixel, 219
 Polar representation, 3
 Polynomials, 236
 Chebyshev —
 of the second kind, 140
 Hermite —, 140
 Laguerre —, 140
 Legendre —, 140
 Positive correlation, 254
 Positivity, 121, 122
 — property, 133
 Potential, 221
 Power set, 6
 Power spectrum, 147
 Preserve
 — inverse vector, 39
 — linear structure, 39
 — scalar multiple, 39
 — vector sum, 39
 — zero vector, 39
 Principal component analysis, 255
 Probability, 217
 — amplitude, 249
 — distribution, 216, 250
 — measure, 250
 — of occurrence, 250
 discrete — space, 250
 finite — space, 250
 Procedure, 12
 Product formula, 108
 Projection
 orthogonal—, 82
 orthographic—, 82
 Proper subset, 6
 Proposition, 1
 Propositional function, 2
 PyX, 288
- Q**
 QR decomposition, 294
 Quantum annealing, 222
 Quantum superposition, 249
 Quotes, 84

- Quotient set, 243
- R**
- Radius
- numerical—, 173
 - spectral—, 173
- Random variable, 250
- Range, 13
- Rank
- of a set of vectors, 60
- Rank
- of a matrix, 99
- Raspberry Pi, 272
- Real line, 6
- Real part, 3
- Recursive, 13
- Reflexive, 243
- Regression, 136
- Representation, 56
- Representative, 243
- Return value, 284
- Root, 4
- Row, 18
- Runge–Kutta method, 205
- S**
- Sample space, 250
- Sampling, 137
- function, 229
 - points, 233
 - values, 233
- Scalar, 29
- field, 29
 - multiple, 29, 40
- Schatten product, 247
- Schur decomposition, 294
- Schur product, 72
- Screen, 219
- Seed, 57
- Set, 5
- Set operations, 7
- Shape, 18
- Shell window, 273
- Sieve theory, 283
- Signature, 102
- Similar, 88
- Simple harmonic motion, 205
- Simplex, 221
- Simpson’s rule, 134
- Simulated annealing, 222
- Singular values, 239
- Size, 17
- Slice, 66
- Solution set, 233
- Solution space, 233
- Space complexity, 91
- Spanned by, 48
- Specification, 12
- Spectral decomposition, 248
- Spectrum, 193
- Squared error, 233
- Standard basis
- of $\mathbb{K}^m \otimes \mathbb{K}^n$, 246
 - of \mathbb{K}^n , 56
- State, 215, 217
- transition diagram, 216
 - transition probability, 215
 - equilibrium —, 222
 - Gibbs —, 222
 - mixed —, 249
 - pure —, 249
 - stationary —, 218
- String, 14
- Sub-additivity, 121
- Subprogram, 12
- Subroutine, 12
- Subscript, 18
- Subset, 6
- Subspace, 37
- invariant—, 179
 - orthocomplemented —, 132
 - sum of—s, 63
 - trivial —, 38
- Surjective, 13
- Sweeping method, 117
- Symbol, 96
- Symmetric, 243
- Symmetric difference, 221
- Symmetry, 122
- T**
- Tensor product
- linear space, 244
 - 3-fold —, 248
- Ternary operator, 13
- Theorem
- basis and dimension, 58
 - dimension—, 61
 - Fourier expansion, 128
 - fundamental—of algebra, 153, 157
 - Gelfand’s formula, 196
 - parallelogram law, 124
 - Parseval’s identity, 128
 - Perron–Frobenius—, 199
 - polarization identity, 124
 - Pythagorean—, 124
 - Riesz–Fischer identity, 128
 - Schwarz’s inequality, 123
 - singular value decomposition, 239
 - stationary state of Markov chain, 218
- Time average, 219
- Time complexity, 91
- Tkinter, 23
- Trace, 110
- Trajectory, 210
- Transitive, 243
- Transposition, 101

-
- Trapezoidal rule, 134
 Triangular inequality, 121
 Trigonometric series expansion, 136
 Triple quotation marks, 84
 True, 1
 Truth
 - table, 1
 - value, 1
 Tuple, 10
 Type casting, 11
- U**
 Uncertainty principle, 250
 Uncorrelated, 254
 Union, 7
 Uniqueness of
 - inverse vector, 33
 - zero vector, 33
 Unit
 - circle, 6
 - disk, 6
 Unitarily
 - equivalent, 168
 Unitary
 - invariant, 168
 - transformation, 154
- V**
 Variance, 254
 Vector, 29, 217
 - -valued random variable, 251
 - sequence, 144
 - space, 29
 - sum, 29
 - bra—, 249
 - column—, 71
 - inverse —, 29
 - ket- —, 249
 - row—, 71
 - zero —, 29
 Vertex, 216
- W**
 Wavelength, 136
 Weight function, 140
 Windows, 271
- Z**
 Zero judgment, 294