

O'REILLY®

Practical Linear Algebra for Data Science

From Core Concepts
to Applications
Using Python



Mike X Cohen

Practical Linear Algebra for Data Science

If you want to work in any computational or technical field, you need to understand linear algebra. The study of matrices and the operations acting upon them, linear algebra is the mathematical basis of nearly all algorithms and analyses implemented in computers. But the way it's presented in decades-old textbooks is much different from the way professionals use linear algebra today to solve real-world problems.

This practical guide from Mike X Cohen teaches the core concepts of linear algebra as implemented in Python, including how they're used in data science, machine learning, deep learning, computational simulations, and biomedical data processing applications. Armed with knowledge from this book, you'll be able to understand, implement, and adapt myriad modern analysis methods and algorithms.

Ideal for practitioners and students using computer technology and algorithms, this book introduces you to:

- The interpretations and applications of vectors and matrices
- Matrix arithmetic (various multiplications and transformations)
- Independence, rank, and inverses
- Important decompositions used in applied linear algebra (including LU and QR)
- Eigendecomposition and singular value decomposition
- Applications including least squares model fitting and principal component analysis

"To newcomers, the abstract nature of linear algebra makes it hard to see its usefulness despite its universal applications. This book does a good job teaching not just the how but also the why with practical applications for linear algebra."

—Thomas Nield
Nield Consulting Group,
Author of *Essential Math for Data Science* and *Getting Started with SQL*

Mike X Cohen is an associate professor of neuroscience at the Donders Institute (Radboud University Medical Centre) in the Netherlands. He has more than 20 years of experience teaching scientific coding, data analysis, statistics, and related topics, and he has authored several online courses and textbooks. Mike has a suspiciously dry sense of humor and enjoys anything purple.

DATA

US \$69.99 CAN \$87.99

ISBN: 978-1-098-12061-0

Twitter: @oreillymedia
[linkedin.com/company/oreilly-media](https://www.linkedin.com/company/oreilly-media)
[youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)



5 6 9 9 9

9 7 8 1 0 9 8 1 2 0 6 1 0

Practical Linear Algebra for Data Science

*From Core Concepts to Applications
Using Python*

Mike X Cohen

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Practical Linear Algebra for Data Science

by Mike X Cohen

Copyright © 2022 Syncxpress BV. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Jessica Haberman

Indexer: Ellen Troutman

Development Editor: Shira Evans

Interior Designer: David Futato

Production Editor: Jonathon Owen

Cover Designer: Karen Montgomery

Copyeditor: Piper Editorial Consulting, LLC

Illustrator: Kate Dullea

Proofreader: Shannon Turlington

September 2022: First Edition

Revision History for the First Edition

2022-09-01: First Release

See <https://www.oreilly.com/catalog/errata.csp?isbn=0636920641025> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Practical Linear Algebra for Data Science*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-12061-0

[LSI]

Table of Contents

Preface.....	xi
1. Introduction.....	1
What Is Linear Algebra and Why Learn It?	1
About This Book	2
Prerequisites	2
Math	3
Attitude	3
Coding	3
Mathematical Proofs Versus Intuition from Coding	4
Code, Printed in the Book and Downloadable Online	5
Code Exercises	5
How to Use This Book (for Teachers and Self Learners)	6
2. Vectors, Part 1.....	7
Creating and Visualizing Vectors in NumPy	7
Geometry of Vectors	10
Operations on Vectors	11
Adding Two Vectors	11
Geometry of Vector Addition and Subtraction	12
Vector-Scalar Multiplication	13
Scalar-Vector Addition	14
Transpose	15
Vector Broadcasting in Python	16
Vector Magnitude and Unit Vectors	17
The Vector Dot Product	18
The Dot Product Is Distributive	20
Geometry of the Dot Product	21

Other Vector Multiplications	22
Hadamard Multiplication	22
Outer Product	23
Cross and Triple Products	24
Orthogonal Vector Decomposition	24
Summary	28
Code Exercises	29
3. Vectors, Part 2.....	33
Vector Sets	33
Linear Weighted Combination	34
Linear Independence	35
The Math of Linear Independence	37
Independence and the Zeros Vector	38
Subspace and Span	38
Basis	41
Definition of Basis	44
Summary	46
Code Exercises	46
4. Vector Applications.....	49
Correlation and Cosine Similarity	49
Time Series Filtering and Feature Detection	52
k-Means Clustering	53
Code Exercises	57
Correlation Exercises	57
Filtering and Feature Detection Exercises	58
k-Means Exercises	60
5. Matrices, Part 1.....	61
Creating and Visualizing Matrices in NumPy	61
Visualizing, Indexing, and Slicing Matrices	61
Special Matrices	63
Matrix Math: Addition, Scalar Multiplication, Hadamard Multiplication	65
Addition and Subtraction	65
“Shifting” a Matrix	66
Scalar and Hadamard Multiplications	67
Standard Matrix Multiplication	67
Rules for Matrix Multiplication Validity	68
Matrix Multiplication	69
Matrix-Vector Multiplication	70
Matrix Operations: Transpose	72

Dot and Outer Product Notation	73
Matrix Operations: LIVE EVIL (Order of Operations)	73
Symmetric Matrices	74
Creating Symmetric Matrices from Nonsymmetric Matrices	74
Summary	75
Code Exercises	76
6. Matrices, Part 2.....	81
Matrix Norms	82
Matrix Trace and Frobenius Norm	83
Matrix Spaces (Column, Row, Nulls)	84
Column Space	84
Row Space	88
Null Spaces	88
Rank	91
Ranks of Special Matrices	94
Rank of Added and Multiplied Matrices	96
Rank of Shifted Matrices	97
Theory and Practice	98
Rank Applications	99
In the Column Space?	99
Linear Independence of a Vector Set	100
Determinant	101
Computing the Determinant	102
Determinant with Linear Dependencies	103
The Characteristic Polynomial	104
Summary	106
Code Exercises	107
7. Matrix Applications.....	113
Multivariate Data Covariance Matrices	113
Geometric Transformations via Matrix-Vector Multiplication	116
Image Feature Detection	120
Summary	124
Code Exercises	124
Covariance and Correlation Matrices Exercises	124
Geometric Transformations Exercises	126
Image Feature Detection Exercises	127
8. Matrix Inverse.....	129
The Matrix Inverse	129
Types of Inverses and Conditions for Invertibility	130

Computing the Inverse	131
Inverse of a 2×2 Matrix	131
Inverse of a Diagonal Matrix	133
Inverting Any Square Full-Rank Matrix	134
One-Sided Inverses	136
The Inverse Is Unique	138
Moore-Penrose Pseudoinverse	138
Numerical Stability of the Inverse	139
Geometric Interpretation of the Inverse	141
Summary	142
Code Exercises	143
9. Orthogonal Matrices and QR Decomposition.....	147
Orthogonal Matrices	147
Gram-Schmidt	149
QR Decomposition	150
Sizes of Q and R	151
QR and Inverses	154
Summary	154
Code Exercises	155
10. Row Reduction and LU Decomposition.....	159
Systems of Equations	159
Converting Equations into Matrices	160
Working with Matrix Equations	161
Row Reduction	163
Gaussian Elimination	165
Gauss-Jordan Elimination	166
Matrix Inverse via Gauss-Jordan Elimination	167
LU Decomposition	169
Row Swaps via Permutation Matrices	170
Summary	171
Code Exercises	172
11. General Linear Models and Least Squares.....	175
General Linear Models	176
Terminology	176
Setting Up a General Linear Model	176
Solving GLMs	178
Is the Solution Exact?	179
A Geometric Perspective on Least Squares	180
Why Does Least Squares Work?	181

GLM in a Simple Example	183
Least Squares via QR	187
Summary	188
Code Exercises	188
12. Least Squares Applications.....	193
Predicting Bike Rentals Based on Weather	193
Regression Table Using statsmodels	198
Multicollinearity	199
Regularization	199
Polynomial Regression	200
Grid Search to Find Model Parameters	204
Summary	206
Code Exercises	206
Bike Rental Exercises	206
Multicollinearity Exercise	207
Regularization Exercise	208
Polynomial Regression Exercise	210
Grid Search Exercises	210
13. Eigendecomposition.....	213
Interpretations of Eigenvalues and Eigenvectors	214
Geometry	214
Statistics (Principal Components Analysis)	215
Noise Reduction	216
Dimension Reduction (Data Compression)	217
Finding Eigenvalues	217
Finding Eigenvectors	220
Sign and Scale Indeterminacy of Eigenvectors	221
Diagonalizing a Square Matrix	222
The Special Awesomeness of Symmetric Matrices	224
Orthogonal Eigenvectors	224
Real-Valued Eigenvalues	226
Eigendecomposition of Singular Matrices	227
Quadratic Form, Definiteness, and Eigenvalues	228
The Quadratic Form of a Matrix	228
Definiteness	230
$\mathbf{A}^T \mathbf{A}$ Is Positive (Semi)definite	231
Generalized Eigendecomposition	232
Summary	233
Code Exercises	234

14. Singular Value Decomposition.....	241
The Big Picture of the SVD	241
Singular Values and Matrix Rank	243
SVD in Python	243
SVD and Rank-1 “Layers” of a Matrix	244
SVD from EIG	246
SVD of $\mathbf{A}^T \mathbf{A}$	247
Converting Singular Values to Variance, Explained	247
Condition Number	248
SVD and the MP Pseudoinverse	249
Summary	250
Code Exercises	251
15. Eigendecomposition and SVD Applications.....	255
PCA Using Eigendecomposition and SVD	255
The Math of PCA	256
The Steps to Perform a PCA	259
PCA via SVD	259
Linear Discriminant Analysis	260
Low-Rank Approximations via SVD	262
SVD for Denoising	263
Summary	263
Exercises	264
PCA	264
Linear Discriminant Analyses	269
SVD for Low-Rank Approximations	272
SVD for Image Denoising	275
16. Python Tutorial.....	279
Why Python, and What Are the Alternatives?	279
IDEs (Interactive Development Environments)	280
Using Python Locally and Online	280
Working with Code Files in Google Colab	281
Variables	282
Data Types	283
Indexing	284
Functions	285
Methods as Functions	286
Writing Your Own Functions	287
Libraries	288
NumPy	289
Indexing and Slicing in NumPy	289

Visualization	290
Translating Formulas to Code	293
Print Formatting and F-Strings	296
Control Flow	297
Comparators	297
If Statements	297
For Loops	299
Nested Control Statements	300
Measuring Computation Time	301
Getting Help and Learning More	301
What to Do When Things Go Awry	301
Summary	302
Index.....	303

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/mikexcohen/LinAlg4DataScience>.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Practical Linear Algebra for Data Science* by Mike X. Cohen (O'Reilly). Copyright 2022 Syncxpress BV, 978-1-098-12061-0.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/practical-linear-algebra>.

Email bookquestions@oreilly.com with comments or technical questions about this book.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>

Follow us on Twitter: <https://twitter.com/oreillymedia>

Watch us on YouTube: <https://youtube.com/oreillymedia>

Acknowledgments

I have a confession: I really dislike writing acknowledgments sections. It's not because I lack gratitude or believe that I have no one to thank—quite the opposite: I have too many people to thank, and I don't know where to begin, who to list by name, and who to leave out. Shall I thank my parents for their role in shaping me into the kind of person who wrote this book? Perhaps their parents for shaping my parents? I remember my fourth-grade teacher telling me I should be a writer when I grow up. (I don't remember her name and I'm not sure when I will grow up, but perhaps she had some influence on this book.) I wrote most of this book during remote-working trips to the Canary Islands; perhaps I should thank the pilots who flew me there? Or the electricians who installed the wiring at the coworking spaces? Perhaps I should be grateful to Özdemir Pasha for his role in popularizing coffee, which both facilitated and distracted me from writing. And let's not forget the farmers who grew the delicious food that sustained me and kept me happy.

You can see where this is going: my fingers did the typing, but it took the entirety and history of human civilization to create me and the environment that allowed me to write this book—and that allowed you to read this book. So, thanks humanity!

But OK, I can also devote one paragraph to a more traditional acknowledgments section. Most importantly, I am grateful to all my students in my live-taught university and summer-school courses, and my Udemy online courses, for trusting me with their education and for motivating me to continue improving my explanations of applied math and other technical topics. I am also grateful for Jess Haberman, the acquisitions editor at O'Reilly who made “first contact” to ask if I was interested in writing this book. Shira Evans (development editor), Jonathon Owen (production editor), Elizabeth Oliver (copy editor), Kristen Brown (manager of content services), and two expert technical reviewers were directly instrumental in transforming my keystrokes into the book you're now reading. I'm sure this list is incomplete because other people who helped publish this book are unknown to me or because I've forgotten them due to memory loss at my extreme old age.¹ To anyone reading this who feels they made even an infinitesimal contribution to this book: thank you.

¹ LOL, I was 42 when I wrote this book.

Introduction

What Is Linear Algebra and Why Learn It?

Linear algebra has an interesting history in mathematics, dating back to the 17th century in the West and much earlier in China. Matrices—the spreadsheets of numbers at the heart of linear algebra—were used to provide a compact notation for storing sets of numbers like geometric coordinates (this was Descartes's original use of matrices) and systems of equations (pioneered by Gauss). In the 20th century, matrices and vectors were used for multivariate mathematics including calculus, differential equations, physics, and economics.

But most people didn't need to care about matrices until fairly recently. Here's the thing: computers are extremely efficient at working with matrices. And so, modern computing gave rise to modern linear algebra. Modern linear algebra is computational, whereas traditional linear algebra is abstract. Modern linear algebra is best learned through code and applications in graphics, statistics, data science, AI, and numerical simulations, whereas traditional linear algebra is learned through proofs and pondering infinite-dimensional vector spaces. Modern linear algebra provides the structural beams that support nearly every algorithm implemented on computers, whereas traditional linear algebra is often intellectual fodder for advanced mathematics university students.

Welcome to modern linear algebra.

Should you learn linear algebra? That depends on whether you want to understand algorithms and procedures, or simply apply methods that others have developed. I don't mean to disparage the latter—there is nothing intrinsically wrong with using tools you don't understand (I am writing this on a laptop that I can use but could not build from scratch). But given that you are reading a book with this title in the O'Reilly book collection, I guess you either (1) want to know how algorithms work or

(2) want to develop or adapt computational methods. So yes, you should learn linear algebra, and you should learn the modern version of it.

About This Book

The purpose of this book is to teach you modern linear algebra. But this is not about memorizing some key equations and slugging through abstract proofs; the purpose is to teach you how to *think* about matrices, vectors, and operations acting upon them. You will develop a geometric intuition for why linear algebra is the way it is. And you will understand how to implement linear algebra concepts in Python code, with a focus on applications in machine learning and data science.

Many traditional linear algebra textbooks avoid numerical examples in the interest of generalizations, expect you to derive difficult proofs on your own, and teach myriad concepts that have little or no relevance to application or implementation in computers. I do not write these as criticisms—abstract linear algebra is beautiful and elegant. But if your goal is to use linear algebra (and mathematics more generally) as a tool for understanding data, statistics, deep learning, image processing, etc., then traditional linear algebra textbooks may seem like a frustrating waste of time that leave you confused and concerned about your potential in a technical field.

This book is written with self-studying learners in mind. Perhaps you have a degree in math, engineering, or physics, but need to learn how to implement linear algebra in code. Or perhaps you didn't study math at university and now realize the importance of linear algebra for your studies or work. Either way, this book is a self-contained resource; it is not solely a supplement for a lecture-based course (though it could be used for that purpose).

If you were nodding your head in agreement while reading the past three paragraphs, then this book is definitely for you.

If you would like to take a deeper dive into linear algebra, with more proofs and explorations, then there are several excellent texts that you can consider, including my own *Linear Algebra: Theory, Intuition, Code* (Sincxpress BV).¹

Prerequisites

I have tried to write this book for enthusiastic learners with minimal formal background. That said, nothing is ever learned truly from scratch.

¹ Apologies for the shameless self-promotion; I promise that's the only time in this book I'll subject you to such an indulgence.

Math

You need to be comfortable with high-school math. Just basic algebra and geometry; nothing fancy.

Absolutely zero calculus is required for this book (though differential calculus is important for applications where linear algebra is often used, such as deep learning and optimization).

But most importantly, you need to be comfortable thinking about math, looking at equations and graphics, and embracing the intellectual challenge that comes with studying math.

Attitude

Linear algebra is a branch of mathematics, ergo this is a mathematics book. Learning math, especially as an adult, requires some patience, dedication, and an assertive attitude. Get a cup of coffee, take a deep breath, put your phone in a different room, and dive in.

There will be a voice in the back of your head telling you that you are too old or too stupid to learn advanced mathematics. Sometimes that voice is louder and sometimes softer, but it's always there. And it's not just you—everyone has it. You cannot suppress or destroy that voice; don't even bother trying. Just accept that a bit of insecurity and self-doubt is part of being human. Each time that voice speaks up is a challenge for you to prove it wrong.

Coding

This book is focused on linear algebra applications in code. I wrote this book for Python, because Python is currently the most widely used language in data science, machine learning, and related fields. If you prefer other languages like MATLAB, R, C, or Julia, then I hope you find it straightforward to translate the Python code.

I've tried to make the Python code as simple as possible, while still being relevant for applications. [Chapter 16](#) provides a basic introduction to Python programming. Should you go through that chapter? That depends on your level of Python skills:

Intermediate/advanced (>1 year coding experience)

Skip [Chapter 16](#) entirely, or perhaps skim it to get a sense of the kind of code that will appear in the rest of the book.

Some knowledge (<1 year experience)

Please work through the chapter in case there is material that is new or that you need to refresh. But you should be able to get through it rather briskly.

Total beginner

Go through the chapter in detail. Please understand that this book is not a complete Python tutorial, so if you find yourself struggling with the code in the content chapters, you might want to put this book down, work through a dedicated Python course or book, then come back to this book.

Mathematical Proofs Versus Intuition from Coding

The purpose of studying math is, well, to understand math. How do you understand math? Let us count the ways:

Rigorous proofs

A proof in mathematics is a sequence of statements showing that a set of assumptions leads to a logical conclusion. Proofs are unquestionably important in pure mathematics.

Visualizations and examples

Clearly written explanations, diagrams, and numerical examples help you gain intuition for concepts and operations in linear algebra. Most examples are done in 2D or 3D for easy visualization, but the principles also apply to higher dimensions.

The difference between these is that formal mathematical proofs provide rigor but rarely intuition, whereas visualizations and examples provide lasting intuition through hands-on experience but can risk inaccuracies based on specific examples that do not generalize.

Proofs of important claims are included, but I focus more on building intuition through explanations, visualizations, and code examples.

And this brings me to mathematical intuition from coding (what I sometimes call “soft proofs”). Here’s the idea: you assume that Python (and libraries such as NumPy and SciPy) correctly implements the low-level number crunching, while you focus on the principles by exploring many numerical examples in code.

A quick example: we will “soft-prove” the commutivity principle of multiplication, which states that $a \times b = b \times a$:

```
a = np.random.randn()  
b = np.random.randn()  
a*b - b*a
```

This code generates two random numbers and tests the hypothesis that swapping the order of multiplication has no impact on the result. The third line of would print out `0.0` if the commutivity principle is true. If you run this code multiple times and always get `0.0`, then you have gained intuition for commutivity by seeing the same result in many different numerical examples.

To be clear: intuition from code is no substitute for a rigorous mathematical proof. The point is that “soft proofs” allow you to understand mathematical concepts without having to worry about the details of abstract mathematical syntax and arguments. This is particularly advantageous to coders who lack an advanced mathematics background.

The bottom line is that *you can learn a lot of math with a bit of coding*.

Code, Printed in the Book and Downloadable Online

You can read this book without looking at code or solving code exercises. That’s fine, and you will certainly learn something. But don’t be disappointed if your knowledge is superficial and fleeting. If you really want to *understand* linear algebra, you need to solve problems. That’s why this book comes with code demonstrations and exercises for each mathematical concept.

Important code is printed directly in the book. I want you to read the text and equations, look at the graphs, and *see the code* at the same time. That will allow you to link concepts and equations to code.

But printing code in a book can take up a lot of space, and hand-copying code on your computer is tedious. Therefore, only the key code lines are printed in the book pages; the online code contains additional code, comments, graphics embellishments, and so on. The online code also contains solutions to the coding exercises (all of them, not only the odd-numbered problems!). You should definitely download the code and go through it while working through the book.

All the code can be obtained from the GitHub site <https://github.com/mikexcohen/LinAlg4DataScience>. You can clone this repository or simply download the entire repository as a ZIP file (you do not need to register, log in, or pay to download the code).

I wrote the code using Jupyter notebook in Google’s Colab environment. I chose to use Jupyter because it’s a friendly and easy-to-use environment. That said, I encourage you to use whichever Python IDE you prefer. The online code is also provided as raw `.py` files for convenience.

Code Exercises

Math is not a spectator sport. Most math books have countless paper-and-pencil problems to work through (and let’s be honest: no one does all of them). But this book is all about *applied* linear algebra, and no one applies linear algebra on paper! Instead, you apply linear algebra in code. Therefore, in lieu of hand-worked problems and tedious proofs “left as an exercise to the reader” (as math textbook authors love to write), this book has lots of code exercises.

The code exercises vary in difficulty. If you are new to Python and to linear algebra, you might find some exercises really challenging. If you get stuck, here's a suggestion: have a quick glance at my solution for inspiration, then put it away so you can't see my code, and continue working on your own code.

When comparing your solution to mine, keep in mind that there are many ways to solve problems in Python. Arriving at the correct answer is important; the steps you take to get there are often a matter of personal coding style.

How to Use This Book (for Teachers and Self Learners)

There are three environments in which this book is useful:

Self-learner

I have tried to make this book accessible to readers who want to learn linear algebra on their own, outside a formal classroom environment. No additional resources or online lectures are necessary, although of course there are myriad other books, websites, YouTube videos, and online courses that students might find helpful.

Primary textbook in a data science class

This book can be used as a primary textbook in a course on the math underlying data science, machine learning, AI, and related topics. There are 14 content chapters (excluding this introduction and the Python appendix), and students could be expected to work through one to two chapters per week. Because students have access to the solutions to all exercises, instructors may wish to supplement the book exercises with additional problem sets.

Secondary textbook in a math-focused linear algebra course

This book could also be used as a supplement in a mathematics course with a strong focus on proofs. In this case, the lectures would focus on theory and rigorous proofs while this book could be referenced for translating the concepts into code with an eye towards applications in data science and machine learning. As I wrote above, instructors may wish to provide supplementary exercises because the solutions to all the book exercises are available online.

CHAPTER 2

Vectors, Part 1

Vectors provide the foundations upon which all of linear algebra (and therefore, the rest of this book) is built.

By the end of this chapter, you will know all about vectors: what they are, what they do, how to interpret them, and how to create and work with them in Python. You will understand the most important operations acting on vectors, including vector algebra and the dot product. Finally, you will learn about vector decompositions, which is one of the main goals of linear algebra.

Creating and Visualizing Vectors in NumPy

In linear algebra, a *vector* is an ordered list of numbers. (In abstract linear algebra, vectors may contain other mathematical objects including functions; however, because this book is focused on applications, we will only consider vectors comprising numbers.)

Vectors have several important characteristics. The first two we will start with are:

Dimensionality

The number of numbers in the vector

Orientation

Whether the vector is in *column orientation* (standing up tall) or *row orientation* (laying flat and wide)

Dimensionality is often indicated using a fancy-looking \mathbb{R}^N , where the \mathbb{R} indicates real-valued numbers (cf. \mathbb{C} for complex-valued numbers) and the N indicates the dimensionality. For example, a vector with two elements is said to be a member of \mathbb{R}^2 . That special \mathbb{R} character is made using latex code, but you can also write R², R2, or R^2.

Equation 2-1 shows a few examples of vectors; please determine their dimensionality and orientation before reading the subsequent paragraph.

Equation 2-1. Examples of column vectors and row vectors

$$\mathbf{x} = \begin{bmatrix} 1 \\ 4 \\ 5 \\ 6 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} .3 \\ -7 \end{bmatrix}, \quad \mathbf{z} = [1 \ 4 \ 5 \ 6]$$

Here are the answers: \mathbf{x} is a 4D column vector, \mathbf{y} is a 2D column vector, and \mathbf{z} is a 4D row vector. You can also write, e.g., $\mathbf{x} \in \mathbb{R}^4$, where the \in symbol means “is contained in the set of.”

Are \mathbf{x} and \mathbf{z} the same vector? Technically they are different, even though they have the same elements in the same order. See “[Does Vector Orientation Matter?](#)” on page 9 for more discussion.

You will learn, in this book and throughout your adventures integrating math and coding, that there are differences between math “on the chalkboard” versus implemented in code. Some discrepancies are minor and inconsequential, while others cause confusion and errors. Let me now introduce you to a terminological difference between math and coding.

I wrote earlier that the *dimensionality* of a vector is the number of elements in that vector. However, in Python, the dimensionality of a vector or matrix is the number of geometric dimensions used to print out a numerical object. For example, all of the vectors shown above are considered “two-dimensional arrays” in Python, regardless of the number of elements contained in the vectors (which is the mathematical dimensionality). A list of numbers without a particular orientation is considered a 1D array in Python, regardless of the number of elements (that array will be printed out as a row, but, as you’ll see later, it is treated differently from row vectors). The mathematical dimensionality—the number of elements in the vector—is called the *length* or the *shape* of the vector in Python.

This inconsistent and sometimes conflicting terminology can be confusing. Indeed, terminology is often a sticky issue at the intersection of different disciplines (in this case, mathematics and computer science). But don’t worry, you’ll get the hang of it with some experience.

When referring to vectors, it is common to use lowercase bolded Roman letters, like \mathbf{v} for “vector v .” Some texts use italics (v) or print an arrow on top (\vec{v}).

Linear algebra convention is to assume that vectors are in column orientation unless otherwise specified. Row vectors are written as \mathbf{w}^T . The T indicates the *transpose*

operation, which you'll learn more about later; for now, suffice it to say that the transpose operation transforms a column vector into a row vector.



Does Vector Orientation Matter?

Do you really need to worry about whether vectors are column- or row-oriented, or orientationless 1D arrays? Sometimes yes, sometimes no. When using vectors to store data, orientation usually doesn't matter. But some operations in Python can give errors or unexpected results if the orientation is wrong. Therefore, vector orientation is important to understand, because spending 30 minutes debugging code only to realize that a row vector needs to be a column vector is guaranteed to give you a headache.

Vectors in Python can be represented using several data types. The `list` type may seem like the simplest way to represent a vector—and it is for some applications. But many linear algebra operations won't work on Python lists. Therefore, most of the time it's best to create vectors as NumPy arrays. The following code shows four ways of creating a vector:

```
asList  = [1,2,3]
asArray = np.array([1,2,3]) # 1D array
rowVec = np.array([[1,2,3]]) # row
colVec = np.array([[1],[2],[3]]) # column
```

The variable `asArray` is an *orientationless* array, meaning it is neither a row nor a column vector but simply a 1D list of numbers in NumPy. Orientation in NumPy is given by brackets: the outermost brackets group all of the numbers together into one object. Then, each additional set of brackets indicates a row: a row vector (variable `rowVec`) has all numbers in one row, while a column vector (variable `colVec`) has multiple rows, with each row containing one number.

We can explore these orientations by examining the shapes of the variables (inspecting variable shapes is often very useful while coding):

```
print(f'asList: {np.shape(asList)}')
print(f'asArray: {asArray.shape}')
print(f'rowVec: {rowVec.shape}')
print(f'colVec: {colVec.shape}')
```

Here's what the output looks like:

```
asList: (3,)
asArray: (3,)
rowVec: (1, 3)
colVec: (3, 1)
```

The output shows that the 1D array `asArray` is of size (3), whereas the orientation-endowed vectors are 2D arrays and are stored as size (1,3) or (3,1) depending on the orientation. Dimensions are always listed as (rows,columns).

Geometry of Vectors

Ordered list of numbers is the algebraic interpretation of a vector; the geometric interpretation of a vector is a straight line with a specific length (also called *magnitude*) and direction (also called *angle*; it is computed relative to the positive x -axis). The two points of a vector are called the tail (where it starts) and the head (where it ends); the head often has an arrow tip to disambiguate from the tail.

You may think that a vector encodes a geometric coordinate, but vectors and coordinates are actually different things. They are, however, concordant when the vector starts at the origin. This is called the *standard position* and is illustrated in [Figure 2-1](#).

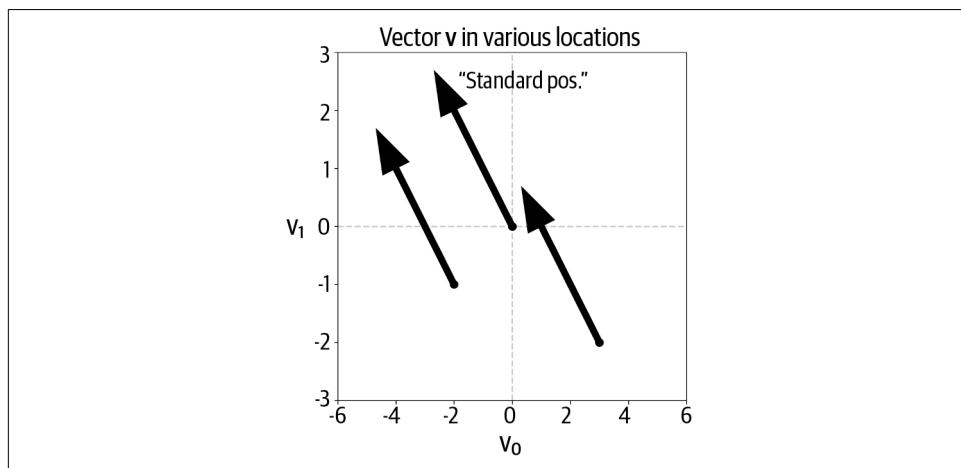


Figure 2-1. All arrows express the same vector. A vector in standard position has its tail at the origin and its head at the concordant geometric coordinate.

Conceptualizing vectors either geometrically or algebraically facilitates intuition in different applications, but these are simply two sides of the same coin. For example, the geometric interpretation of a vector is useful in physics and engineering (e.g., representing physical forces), and the algebraic interpretation of a vector is useful in data science (e.g., storing sales data over time). Oftentimes, linear algebra concepts are learned geometrically in 2D graphs, and then are expanded to higher dimensions using algebra.

Operations on Vectors

Vectors are like nouns; they are the characters in our linear algebra story. The fun in linear algebra comes from the verbs—the actions that breathe life into the characters. Those actions are called *operations*.

Some linear algebra operations are simple and intuitive and work exactly how you'd expect (e.g., addition), whereas others are more involved and require entire chapters to explain (e.g., singular value decomposition). Let's begin with simple operations.

Adding Two Vectors

To add two vectors, simply add each corresponding element. [Equation 2-2](#) shows an example:

Equation 2-2. Adding two vectors

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 14 \\ 25 \\ 36 \end{bmatrix}$$

As you might have guessed, vector addition is defined only for two vectors that have the same dimensionality; it is not possible to add, e.g., a vector in \mathbb{R}^3 with a vector in \mathbb{R}^5 .

Vector subtraction is also what you'd expect: subtract the two vectors element-wise. [Equation 2-3](#) shows an example:

Equation 2-3. Subtracting two vectors

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} - \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} -6 \\ -15 \\ -24 \end{bmatrix}$$

Adding vectors is straightforward in Python:

```
v = np.array([4,5,6])
w = np.array([10,20,30])
u = np.array([0,3,6,9])
vPlusW = v+w
uPlusW = u+w # error! dimensions mismatched!
```

Does vector orientation matter for addition? Consider [Equation 2-4](#):

Equation 2-4. Can you add a row vector to a column vector?

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} + [10 \ 20 \ 30] = ?$$

You might think that there is no difference between this example and the one shown earlier—after all, both vectors have three elements. Let's see what Python does:

```
v = np.array([[4,5,6]]) # row vector
w = np.array([[10,20,30]]).T # column vector
v+w

>> array([[14, 15, 16],
           [24, 25, 26],
           [34, 35, 36]])
```

The result may seem confusing and inconsistent with the definition of vector addition given earlier. In fact, Python is implementing an operation called *broadcasting*. You will learn more about broadcasting later in this chapter, but I encourage you to spend a moment pondering the result and thinking about how it arose from adding a row and a column vector. Regardless, this example shows that orientation is indeed important: *two vectors can be added together only if they have the same dimensionality and the same orientation*.

Geometry of Vector Addition and Subtraction

To add two vectors geometrically, place the vectors such that the tail of one vector is at the head of the other vector. The summed vector traverses from the tail of the first vector to the head of the second (graph A in [Figure 2-2](#)). You can extend this procedure to sum any number of vectors: simply stack all the vectors tail-to-head, and then the sum is the line that goes from the first tail to the final head.

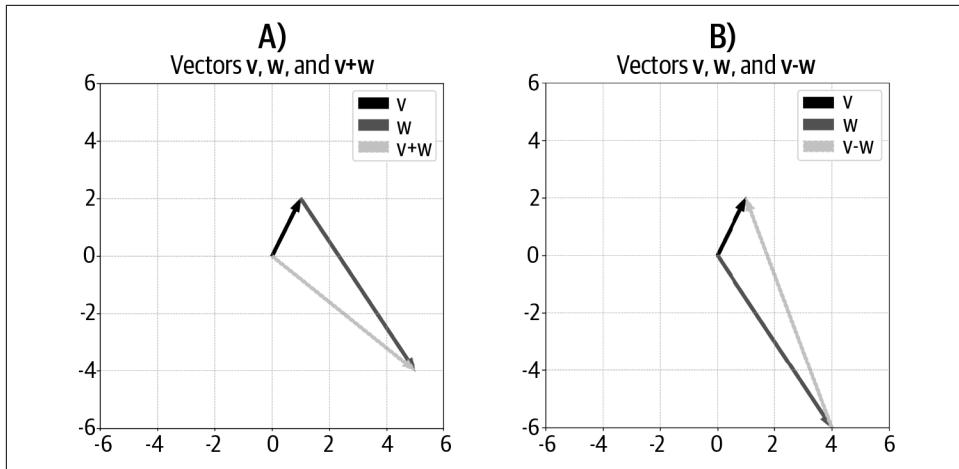


Figure 2-2. The sum and difference of two vectors

Subtracting vectors geometrically is slightly different but equally straightforward: line up the two vectors such that their tails are at the same coordinate (this is easily accomplished by having both vectors in standard position); the difference vector is the line that goes from the head of the “negative” vector to the head of the “positive” vector (graph B in Figure 2-2).

Do not underestimate the importance of the geometry of vector subtraction: it is the basis for orthogonal vector decomposition, which in turn is the basis for linear least squares, which is one of the most important applications of linear algebra in science and engineering.

Vector-Scalar Multiplication

A *scalar* in linear algebra is a number on its own, not embedded in a vector or matrix. Scalars are typically indicated using lowercase Greek letters such as α or λ . Therefore, vector-scalar multiplication is indicated as, for example, $\beta\mathbf{u}$.

Vector-scalar multiplication is very simple: multiply each vector element by the scalar. One numerical example (Equation 2-5) will suffice for understanding:

Equation 2-5. Vector-scalar multiplication (or: scalar-vector multiplication)

$$\lambda = 4, \mathbf{w} = \begin{bmatrix} 9 \\ 4 \\ 1 \end{bmatrix}, \quad \lambda\mathbf{w} = \begin{bmatrix} 36 \\ 16 \\ 4 \end{bmatrix}$$

The Zeros Vector

A vector of all zeros is called the *zeros vector*, indicated using a boldfaced zero, **0**, and is a special vector in linear algebra. In fact, using the zeros vector to solve a problem is often called the *trivial solution* and is excluded. Linear algebra is full of statements like “find a nonzeros vector that can solve...” or “find a nontrivial solution to...”

I wrote earlier that the data type of a variable storing a vector is sometimes important and sometimes unimportant. Vector-scalar multiplication is an example where data type matters:

```
s = 2
a = [3,4,5] # as list
b = np.array(a) # as np array
print(a*s)
print(b*s)

>> [ 3, 4, 5, 3, 4, 5 ]
>> [ 6 8 10 ]
```

The code creates a scalar (variable `s`) and a vector as a list (variable `a`), then converts that into a NumPy array (variable `b`). The asterisk is overloaded in Python, meaning its behavior depends on the variable type: scalar multiplying a list repeats the list `s` times (in this case, twice), which is definitely *not* the linear algebra operation of scalar-vector multiplication. When the vector is stored as a NumPy array, however, the asterisk is interpreted as element-wise multiplication. (Here’s a small exercise for you: what happens if you set `s = 2.0`, and why?) Both of these operations (list repetition and vector-scalar multiplication) are used in real-world coding, so be mindful of the distinction.

Scalar-Vector Addition

Adding a scalar to a vector is not formally defined in linear algebra: they are two separate kinds of mathematical objects and cannot be combined. However, numerical processing programs like Python will allow adding scalars to vectors, and the operation is comparable to scalar-vector multiplication: the scalar is added to each vector element. The following code illustrates the idea:

```
s = 2
v = np.array([3,6])
s+v
>> [5 8]
```

¹ `a*s` throws an error, because list repetition can only be done using integers; it’s not possible to repeat a list 2.72 times!

The geometry of vector-scalar multiplication

Why are scalars called “scalars”? That comes from the geometric interpretation. Scalars scale vectors without changing their direction. There are four effects of vector-scalar multiplication that depend on whether the scalar is greater than 1, between 0 and 1, exactly 0, or negative. [Figure 2-3](#) illustrates the concept.

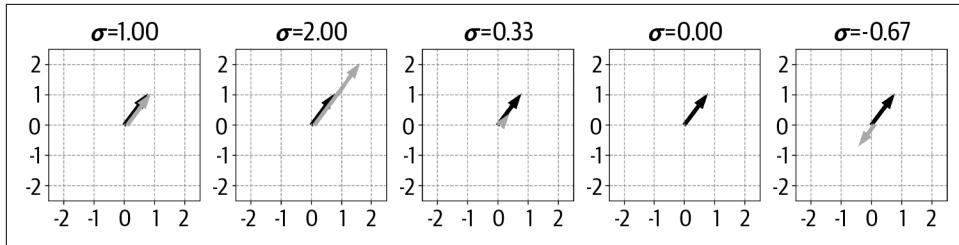


Figure 2-3. The same vector (black arrow) multiplied by different scalars σ (gray line; shifted slightly for visibility)

I wrote earlier that scalars do not change the direction of the vector. But the figure shows that the vector direction flips when the scalar is negative (that is, its angle rotates by 180°). That might seem a contradiction, but there is an interpretation of vectors as pointing along an infinitely long line that passes through the origin and goes to infinity in both directions (in the next chapter, I'll call this a “one-dimensional subspace”). In that sense, the “rotated” vector still points along the same infinite line and thus the negative scalar does not change the direction. This interpretation is important for matrix spaces, eigenvectors, and singular vectors, all of which are introduced in later chapters.

Vector-scalar multiplication in combination with vector addition leads directly to *vector averaging*. Averaging vectors is the same as averaging numbers: sum and divide by the number of numbers. So, to average two vectors, add them and then scalar multiply by .5. In general, to average N vectors, sum them and scalar multiply the result by $1/N$.

Transpose

You already learned about the transpose operation: it converts column vectors into row vectors, and vice versa. Let me provide a slightly more formal definition that will generalize to transposing matrices (a topic in [Chapter 5](#)).

A matrix has rows and columns; therefore, each matrix element has a $(row, column)$ index. The transpose operation simply swaps those indices. This is formalized in [Equation 2-6](#):

Equation 2-6. The transpose operation

$$\mathbf{m}_{i,j}^T = \mathbf{m}_{j,i}$$

Vectors have either one row or one column, depending on their orientation. For example, a 6D row vector has $i = 1$ and j indices from 1 to 6, whereas a 6D column vector has i indices from 1 to 6 and $j = 1$. So swapping the i,j indices swaps the rows and columns.

Here's an important rule: transposing twice returns the vector to its original orientation. In other words, $\mathbf{v}^{TT} = \mathbf{v}$. That may seem obvious and trivial, but it is the keystone of several important proofs in data science and machine learning, including creating symmetric covariance matrices as the data matrix times its transpose (which in turn is the reason why a principal components analysis is an orthogonal rotation of the data space...don't worry, that sentence will make sense later in the book!).

Vector Broadcasting in Python

Broadcasting is an operation that exists only in modern computer-based linear algebra; this is not a procedure you would find in a traditional linear algebra textbook.

Broadcasting essentially means to repeat an operation multiple times between one vector and each element of another vector. Consider the following series of equations:

$$\begin{aligned} [1 \ 1] + [10 \ 20] \\ [2 \ 2] + [10 \ 20] \\ [3 \ 3] + [10 \ 20] \end{aligned}$$

Notice the patterns in the vectors. We can implement this set of equations compactly by condensing those patterns into vectors $[1 \ 2 \ 3]$ and $[10 \ 20]$, and then broadcasting the addition. Here's how it looks in Python:

```
v = np.array([[1,2,3]]).T # col vector
w = np.array([[10,20]])    # row vector
v + w # addition with broadcasting

>> array([[11, 21],
           [12, 22],
           [13, 23]])
```

Here again you can see the importance of orientation in linear algebra operations: try running the code above, changing v into a row vector and w into a column vector.²

Because broadcasting allows for efficient and compact computations, it is used often in numerical coding. You'll see several examples of broadcasting in this book, including in the section on k -means clustering (Chapter 4).

Vector Magnitude and Unit Vectors

The *magnitude* of a vector—also called the *geometric length* or the *norm*—is the distance from tail to head of a vector, and is computed using the standard Euclidean distance formula: the square root of the sum of squared vector elements (see Equation 2-7). Vector magnitude is indicated using double-vertical bars around the vector: $\| v \|$.

Equation 2-7. The norm of a vector

$$\| v \| = \sqrt{\sum_{i=1}^n v_i^2}$$

Some applications use squared magnitudes (written $\| v \|^2$), in which case the square root term on the right-hand side drops out.

Before showing the Python code, let me explain some more terminological discrepancies between “chalkboard” linear algebra and Python linear algebra. In mathematics, the dimensionality of a vector is the number of elements in that vector, while the length is a geometric distance; in Python, the function `len()` (where `len` is short for `length`) returns the *dimensionality* of an array, while the function `np.norm()` returns the geometric length (magnitude). In this book, I will use the term *magnitude* (or *geometric length*) instead of *length* to avoid confusion:

```
v = np.array([1,2,3,7,8,9])
v_dim = len(v) # math dimensionality
v_mag = np.linalg.norm(v) # math magnitude, length, or norm
```

There are some applications where we want a vector that has a geometric length of one, which is called a *unit vector*. Example applications include orthogonal matrices, rotation matrices, eigenvectors, and singular vectors.

A unit vector is defined as $\| v \| = 1$.

Needless to say, lots of vectors are not unit vectors. (I'm tempted to write “most vectors are not unit vectors,” but there is an infinite number of unit vectors and nonunit vectors, although the set of infinite nonunit vectors is larger than the set of

² Python still broadcasts, but the result is a 3×2 matrix instead of a 2×3 matrix.

infinite unit vectors.) Fortunately, any nonunit vector has an associated unit vector. That means that we can create a unit vector in the same direction as a nonunit vector. Creating an associated unit vector is easy; you simply scalar multiply by the reciprocal of the vector norm (Equation 2-8):

Equation 2-8. Creating a unit vector

$$\hat{\mathbf{v}} = \frac{1}{\|\mathbf{v}\|} \mathbf{v}$$

You can see the common convention for indicating unit vectors ($\hat{\mathbf{v}}$) in the same direction as their parent vector \mathbf{v} . Figure 2-4 illustrates these cases.

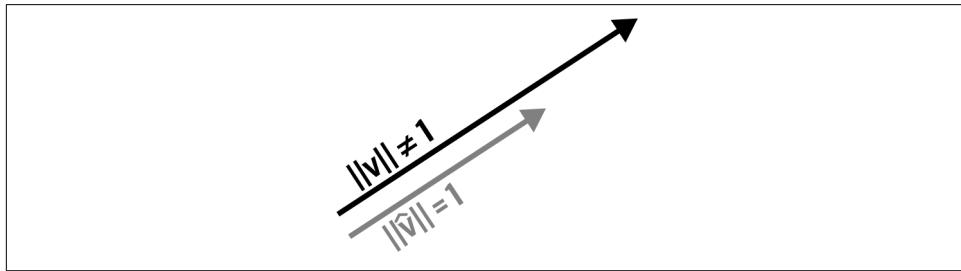


Figure 2-4. A unit vector (gray arrow) can be crafted from a nonunit vector (black arrow); both vectors have the same angle but different magnitudes

Actually, the claim that “any nonunit vector has an associated unit vector” is not entirely true. There is a vector that has nonunit length and yet has no associated unit vector. Can you guess which vector it is?³

I’m not showing Python code to create unit vectors here, because that’s one of the exercises at the end of this chapter.

The Vector Dot Product

The *dot product* (also sometimes called the *inner product*) is one of the most important operations in all of linear algebra. It is the basic computational building block from which many operations and algorithms are built, including convolution, correlation, the Fourier transform, matrix multiplication, linear feature extraction, signal filtering, and so on.

³ The zeros vector has a length of 0 but no associated unit vector, because it has no direction and because it is impossible to scale the zeros vector to have nonzero length.

There are several ways to indicate the dot product between two vectors. I will mostly use the common notation $\mathbf{a}^T \mathbf{b}$ for reasons that will become clear after learning about matrix multiplication. In other contexts you might see $\mathbf{a} \cdot \mathbf{b}$ or $\langle \mathbf{a}, \mathbf{b} \rangle$.

The dot product is a single number that provides information about the relationship between two vectors. Let's first focus on the algorithm to compute the dot product, and then I'll discuss how to interpret it.

To compute the dot product, you multiply the corresponding elements of the two vectors, and then sum over all the individual products. In other words: element-wise multiplication and sum. In [Equation 2-9](#), \mathbf{a} and \mathbf{b} are vectors, and a_i indicates the i th element of \mathbf{a} .

Equation 2-9. Dot product formula

$$\delta = \sum_{i=1}^n a_i b_i$$

You can tell from the formula that the dot product is valid only between two vectors of the same dimensionality. [Equation 2-10](#) shows a numerical example:

Equation 2-10. Example dot product calculation

$$\begin{aligned}[1 & 2 & 3 & 4] \cdot [5 & 6 & 7 & 8] &= 1 \times 5 + 2 \times 6 + 3 \times 7 + 4 \times 8 \\ &= 5 + 12 + 21 + 32 \\ &= 70\end{aligned}$$



Irritations of Indexing

Standard mathematical notation, and some math-oriented numerical processing programs like MATLAB and Julia, start indexing at 1 and stop at N , whereas some programming languages like Python and Java start indexing at 0 and stop at $N - 1$. We need not debate the merits and limitations of each convention—though I do sometimes wonder how many bugs this inconsistency has introduced into human civilization—but it is important to be mindful of this difference when translating formulas into Python code.

There are multiple ways to implement the dot product in Python; the most straightforward way is to use the `np.dot()` function:

```
v = np.array([1,2,3,4])
w = np.array([5,6,7,8])
np.dot(v,w)
```



Note About np.dot()

The function `np.dot()` does not actually implement the vector dot product; it implements matrix multiplication, which is a collection of dot products. This will make more sense after learning about the rules and mechanisms of matrix multiplication ([Chapter 5](#)). If you want to explore this now, you can modify the previous code to endow both vectors with orientations (row versus column). You will discover that the output is the dot product only when the first input is a row vector and the second input is a column vector.

Here is an interesting property of the dot product: scalar multiplying one vector scales the dot product by the same amount. We can explore this by expanding the previous code:

```
s = 10
np.dot(s*v,w)
```

The dot product of v and w is 70, and the dot product using $s*v$ (which, in math notation, would be written as $s v^T w$) is 700. Now try it with a negative scalar, e.g., $s = -1$. You'll see that the dot product magnitude is preserved but the sign is reversed. Of course, when $s = 0$ then the dot product is zero.

Now you know how to compute the dot product. What does the dot product mean and how do we interpret it?

The dot product can be interpreted as a measure of *similarity* or *mapping* between two vectors. Imagine that you collected height and weight data from 20 people, and you stored those data in two vectors. You would certainly expect those variables to be related to each other (taller people tend to weigh more), and therefore you could expect the dot product between those two vectors to be large. On the other hand, the magnitude of the dot product depends on the scale of the data, which means the dot product between data measured in grams and centimeters would be larger than the dot product between data measured in pounds and feet. This arbitrary scaling, however, can be eliminated with a normalization factor. In fact, the normalized dot product between two variables is called the *Pearson correlation coefficient*, and it is one of the most important analyses in data science. More on this in [Chapter 4](#)!

The Dot Product Is Distributive

The distributive property of mathematics is that $a(b + c) = ab + ac$. Translated into vectors and the vector dot product, it means that:

$$\mathbf{a}^T(\mathbf{b} + \mathbf{c}) = \mathbf{a}^T\mathbf{b} + \mathbf{a}^T\mathbf{c}$$

In words, you would say that the dot product of a vector sum equals the sum of the vector dot products.

The following Python code illustrates the distributivity property:

```
a = np.array([ 0,1,2 ])
b = np.array([ 3,5,8 ])
c = np.array([ 13,21,34 ])

# the dot product is distributive
res1 = np.dot( a, b+c )
res2 = np.dot( a,b ) + np.dot( a,c )
```

The two outcomes `res1` and `res2` are the same (with these vectors, the answer is 110), which illustrates the distributivity of the dot product. Notice how the mathematical formula is translated into Python code; translating formulas into code is an important skill in math-oriented coding.

Geometry of the Dot Product

There is also a geometric definition of the dot product, which is the product of the magnitudes of the two vectors, scaled by the cosine of the angle between them (Equation 2-11).

Equation 2-11. Geometric definition of the vector dot product

$$\alpha = \cos(\theta_{v,w}) \|v\| \|w\|$$

Equation 2-9 and Equation 2-11 are mathematically equivalent but expressed in different forms. The proof of their equivalence is an interesting exercise in mathematical analysis, but would take about a page of text and relies on first proving other principles including the Law of Cosines. That proof is not relevant for this book and so is omitted.

Notice that vector magnitudes are strictly positive quantities (except for the zeros vector, which has $\|0\| = 0$), while the cosine of an angle can range between -1 and $+1$. This means that the sign of the dot product is determined entirely by the geometric relationship between the two vectors. Figure 2-5 shows five cases of the dot product sign, depending on the angle between the two vectors (in 2D for visualization, but the principle holds for higher dimensions).

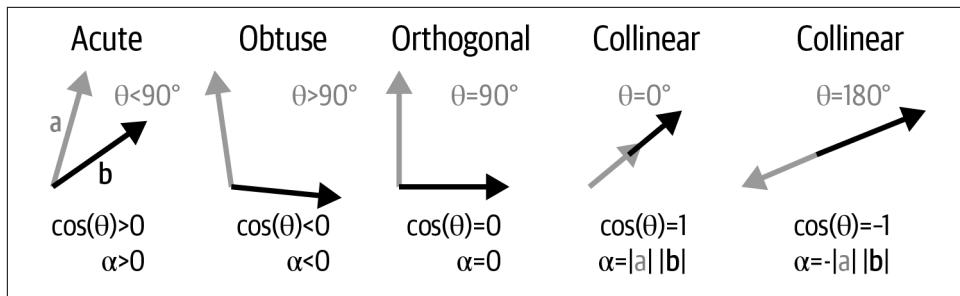


Figure 2-5. The sign of the dot product between two vectors reveals the geometric relationship between those vectors



Memorize This: Orthogonal Vectors Have a Zero Dot Product

Some math teachers insist that you shouldn't memorize formulas and terms, and instead should understand procedures and proofs. But let's be honest: memorization is an important and inescapable part of learning mathematics. Fortunately, linear algebra isn't excessively memorization-heavy, but there are a few things you'll simply need to commit to memory.

Here is one: orthogonal vectors have a dot product of zero (that claim goes both ways—when the dot product is zero, then the two vectors are orthogonal). So, the following statements are equivalent: two vectors are orthogonal; two vectors have a dot product of zero; two vectors meet at a 90° angle. Repeat that equivalence until it's permanently etched into your brain.

Other Vector Multiplications

The dot product is perhaps the most important, and most frequently used, way to multiply vectors. But there are several other ways to multiply vectors.

Hadamard Multiplication

This is just a fancy term for element-wise multiplication. To implement Hadamard multiplication, each corresponding element in the two vectors is multiplied. The product is a vector of the same dimensionality as the two multiplicands. For example:

$$\begin{bmatrix} 5 \\ 4 \\ 8 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 0 \\ .5 \\ -1 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 4 \\ -2 \end{bmatrix}$$

In Python, the asterisk indicates element-wise multiplication for two vectors or matrices:

```
a = np.array([5,4,8,2])
b = np.array([1,0,.5])
a*b
```

Try running that code in Python and...uh oh! Python will give an error. Find and fix the bug. What have you learned about Hadamard multiplication from that error? Check the footnote for the answer.⁴

Hadamard multiplication is a convenient way to organize multiple scalar multiplications. For example, imagine you have data on the number of widgets sold in different shops and the price per widget at each shop. You could represent each variable as a vector, and then Hadamard-multiply those vectors to compute the widget revenue *per shop* (this is different from the total revenue across *all shops*, which would be computed as the dot product).

Outer Product

The outer product is a way to create a matrix from a column vector and a row vector. Each row in the outer product matrix is the row vector scalar multiplied by the corresponding element in the column vector. We could also say that each column in the product matrix is the column vector scalar multiplied by the corresponding element in the row vector. In [Chapter 6](#), I'll call this a "rank-1 matrix," but don't worry about the term for now; instead, focus on the pattern illustrated in the following example:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} d & e \end{bmatrix} = \begin{bmatrix} ad & ae \\ bd & be \\ cd & ce \end{bmatrix}$$

Using Letters in Linear Algebra

In middle school algebra, you learned that using letters as abstract placeholders for numbers allows you to understand math at a deeper level than arithmetic. Same concept in linear algebra: teachers sometimes use letters inside matrices in place of numbers when that facilitates comprehension. You can think of the letters as variables.

⁴ The error is that the two vectors have different dimensionalities, which shows that Hadamard multiplication is defined only for two vectors of equal dimensionality. You can fix the problem by removing one number from a or adding one number to b.

The outer product is quite different from the dot product: it produces a matrix instead of a scalar, and the two vectors in an outer product can have different dimensionalities, whereas the two vectors in a dot product must have the same dimensionality.

The outer product is indicated as \mathbf{vw}^T (remember that we assume vectors are in column orientation; therefore, the outer product involves multiplying a column by a row). Note the subtle but important difference between notation for the **dot product** ($\mathbf{v}^T \mathbf{w}$) and the **outer product** (\mathbf{vw}^T). This might seem strange and confusing now, but I promise it will make perfect sense after learning about matrix multiplication in [Chapter 5](#).

The outer product is similar to broadcasting, but they are not the same: *broadcasting* is a general coding operation that is used to expand vectors in arithmetic operations such as addition, multiplication, and division; the *outer product* is a specific mathematical procedure for multiplying two vectors.

NumPy can compute the outer product via the function `np.outer()` or the function `np.dot()` if the two input vectors are in, respectively, column and row orientation.

Cross and Triple Products

There are a few other ways to multiply vectors such as the cross product or triple product. Those methods are used in geometry and physics, but don't come up often enough in tech-related applications to spend any time on in this book. I mention them here only so you have passing familiarity with the names.

Orthogonal Vector Decomposition

To “decompose” a vector or matrix means to break up that matrix into multiple simpler pieces. Decompositions are used to reveal information that is “hidden” in a matrix, to make the matrix easier to work with, or for data compression. It is no understatement to write that much of linear algebra (in the abstract and in practice) involves matrix decompositions. Matrix decompositions are a big deal.

Let me introduce the concept of a decomposition using two simple examples with scalars:

- We can decompose the number 42.01 into two pieces: 42 and .01. Perhaps .01 is noise to be ignored, or perhaps the goal is to compress the data (the integer 42 requires less memory than the floating-point 42.01). Regardless of the motivation, the decomposition involves representing one mathematical object as the sum of simpler objects ($42 = 42 + .01$).

- We can decompose the number 42 into the product of prime numbers 2, 3, and 7. This decomposition is called *prime factorization* and has many applications in numerical processing and cryptography. This example involves products instead of sums, but the point is the same: decompose one mathematical object into smaller, simpler pieces.

In this section, we will begin exploring a simple yet important decomposition, which is to break up a vector into two separate vectors, one of which is orthogonal to a reference vector while the other is parallel to that reference vector. Orthogonal vector decomposition directly leads to the Gram-Schmidt procedure and QR decomposition, which is used frequently when solving inverse problems in statistics.

Let's begin with a picture so you can visualize the goal of the decomposition. [Figure 2-6](#) illustrates the situation: we have two vectors \mathbf{a} and \mathbf{b} in standard position, and our goal is find the point on \mathbf{a} that is as close as possible to the head of \mathbf{b} . We could also express this as an optimization problem: project vector \mathbf{b} onto vector \mathbf{a} such that the projection distance is minimized. Of course, that point on \mathbf{a} will be a scaled version of \mathbf{a} ; in other words, $\beta\mathbf{a}$. So now our goal is to find the scalar β . (The connection to orthogonal vector decomposition will soon be clear.)

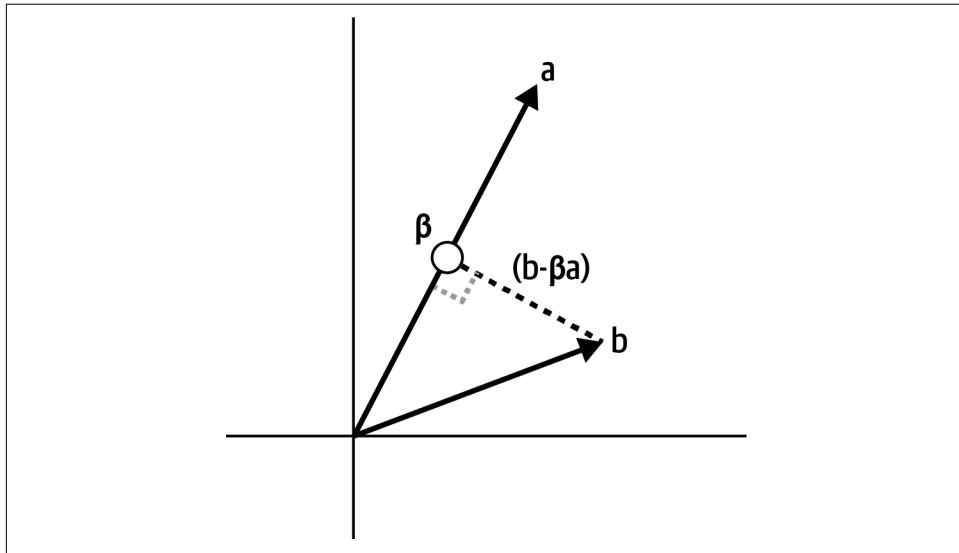


Figure 2-6. To project a point at the head of \mathbf{b} onto a vector \mathbf{a} with minimum distance, we need a formula to compute β such that the length of the projection vector $(\mathbf{b} - \beta\mathbf{a})$ is minimized

Importantly, we can use vector subtraction to define the line from \mathbf{b} to $\beta\mathbf{a}$. We could give this line its own letter, e.g., vector \mathbf{c} , but the subtraction is necessary for discovering the solution.

The key insight that leads to the solution to this problem is that the point on \mathbf{a} that is closest to the head of \mathbf{b} is found by drawing a line from \mathbf{b} that meets \mathbf{a} at a right angle. The intuition here is to imagine a triangle formed by the origin, the head of \mathbf{b} , and $\beta\mathbf{a}$; the length of the line from \mathbf{b} to $\beta\mathbf{a}$ gets longer as the angle $\angle\beta\mathbf{a}$ gets smaller than 90° or larger than 90° .

Putting this together, we have deduced that $(\mathbf{b} - \beta\mathbf{a})$ is orthogonal to $\beta\mathbf{a}$, which is the same thing as saying that those vectors are perpendicular. And that means that the dot product between them must be zero. Let's transform those words into an equation:

$$\mathbf{a}^T(\mathbf{b} - \beta\mathbf{a}) = 0$$

From here, we can apply some algebra to solve for β (note the application of the distributive property of dot products), which is shown in [Equation 2-12](#):

Equation 2-12. Solving the orthogonal projection problem

$$\begin{aligned}\mathbf{a}^T\mathbf{b} - \beta\mathbf{a}^T\mathbf{a} &= 0 \\ \beta\mathbf{a}^T\mathbf{a} &= \mathbf{a}^T\mathbf{b} \\ \beta &= \frac{\mathbf{a}^T\mathbf{b}}{\mathbf{a}^T\mathbf{a}}\end{aligned}$$

This is quite beautiful: we began with a simple geometric picture, explored the implications of the geometry, expressed those implications as a formula, and then applied a bit of algebra. And the upshot is that we discovered a formula for projecting a point onto a line with minimum distance. This is called *orthogonal projection*, and it is the basis for many applications in statistics and machine learning, including the famous least squares formula for solving linear models (you'll see orthogonal projections in [Chapters 9, 10, and 11](#)).

I can imagine that you're super curious to see what the Python code would look like to implement this formula. But you're going to have to write that code yourself in [Exercise 2-8](#) at the end of this chapter. If you can't wait until the end of the chapter, feel free to solve that exercise now, and then continue learning about orthogonal decomposition.

You might be wondering how this is related to orthogonal vector decomposition, i.e., the title of this section. The minimum distance projection is the necessary grounding, and you're now ready to learn the decomposition.

As usual, we start with the setup and the goal. We begin with two vectors, which I'll call the "target vector" and the "reference vector." Our goal is to decompose the target

vector into two other vectors such that (1) those two vectors sum to the target vector, and (2) one vector is orthogonal to the reference vector while the other is parallel to the reference vector. The situation is illustrated in [Figure 2-7](#).

Before starting with the math, let's get our terms straight: I will call the target vector \mathbf{t} and the reference vector \mathbf{r} . Then, the two vectors formed from the target vector will be called the *perpendicular component*, indicated as $\mathbf{t}_{\perp \mathbf{r}}$, and the *parallel component*, indicated as $\mathbf{t}_{\parallel \mathbf{r}}$.

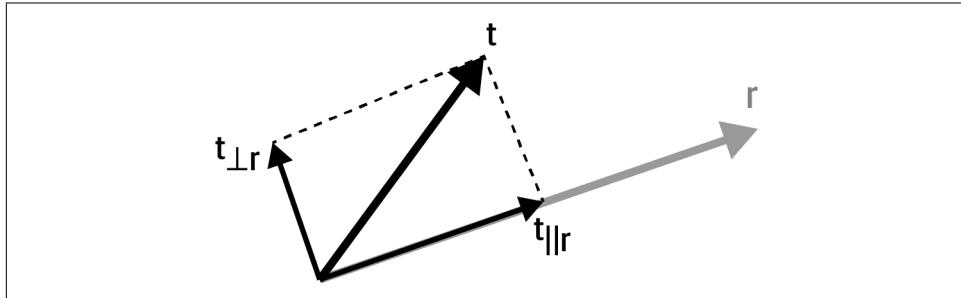


Figure 2-7. Illustration of orthogonal vector decomposition: decompose vector \mathbf{t} into the sum of two other vectors that are orthogonal and parallel to vector \mathbf{r}

We begin by defining the parallel component. What is a vector that is parallel to \mathbf{r} ? Well, any scaled version of \mathbf{r} is obviously parallel to \mathbf{r} . So, we find $\mathbf{t}_{\parallel \mathbf{r}}$ simply by applying the orthogonal projection formula that we just discovered ([Equation 2-13](#)):

Equation 2-13. Computing the parallel component of \mathbf{t} with respect to \mathbf{r}

$$\mathbf{t}_{\parallel \mathbf{r}} = \mathbf{r} \frac{\mathbf{t}^T \mathbf{r}}{\mathbf{r}^T \mathbf{r}}$$

Note the subtle difference to [Equation 2-12](#): there we only computed the scalar β ; here we want to compute the scaled vector $\beta \mathbf{r}$.

That's the parallel component. How do we find the perpendicular component? That one is easier, because we already know that the two vector components must sum to the original target vector. Thus:

$$\mathbf{t} = \mathbf{t}_{\perp \mathbf{r}} + \mathbf{t}_{\parallel \mathbf{r}}$$

$$\mathbf{t}_{\perp \mathbf{r}} = \mathbf{t} - \mathbf{t}_{\parallel \mathbf{r}}$$

In other words, we subtract off the parallel component from the original vector, and the residual is our perpendicular component.

But is that perpendicular component *really* orthogonal to the reference vector? Yes, it is! To prove it, you show that the dot product between the perpendicular component and the reference vector is zero:

$$\begin{aligned}(\mathbf{t}_{\perp} \mathbf{r})^T \mathbf{r} &= 0 \\ \left(\mathbf{t} - \mathbf{r} \frac{\mathbf{t}^T \mathbf{r}}{\mathbf{r}^T \mathbf{r}}\right)^T \mathbf{r} &= 0\end{aligned}$$

Working through the algebra of this proof is straightforward but tedious, so I've omitted it. Instead, you'll work on building intuition using Python code in the exercises.

I hope you enjoyed learning about orthogonal vector decomposition. Note again the general principle: we break apart one mathematical object into a combination of other objects. The details of the decomposition depend on our constraints (in this case, orthogonal and parallel to a reference vector), which means that different constraints (that is, different goals of the analysis) can lead to different decompositions of the same vector.

Summary

The beauty of linear algebra is that even the most sophisticated and computationally intense operations on matrices are made up of simple operations, most of which can be understood with geometric intuition. Do not underestimate the importance of studying simple operations on vectors, because what you learned in this chapter will form the basis for the rest of the book—and the rest of your career as an *applied linear algebra* (which is what you really are if you do anything with data science, machine learning, AI, deep learning, image processing, computational vision, statistics, blah blah blah).

Here are the most important take-home messages of this chapter:

- A vector is an ordered list of numbers that is placed in a column or in a row. The number of elements in a vector is called its dimensionality, and a vector can be represented as a line in a geometric space with the number of axes equal to the dimensionality.
- Several arithmetic operations (addition, subtraction, and Hadamard multiplication) on vectors work element-wise.
- The dot product is a single number that encodes the relationship between two vectors of the same dimensionality, and is computed as element-wise multiplication and sum.

- The dot product is zero for vectors that are orthogonal, which geometrically means that the vectors meet at a right angle.
- Orthogonal vector decomposition involves breaking up a vector into the sum of two other vectors that are orthogonal and parallel to a reference vector. The formula for this decomposition can be rederived from the geometry, but you should remember the phrase “mapping over magnitude” as the concept that that formula expresses.

Code Exercises

I hope you don’t see these exercises as tedious work that you need to do. Instead, these exercises are opportunities to polish your math and coding skills, and to make sure that you really understand the material in this chapter.

I also want you to see these exercises as a springboard to continue exploring linear algebra using Python. Change the code to use different numbers, different dimensionalities, different orientations, etc. Write your own code to test other concepts mentioned in the chapter. Most importantly: have fun and embrace the learning experience.

As a reminder: the solutions to all the exercises can be viewed or downloaded from <https://github.com/mikexcohen/LA4DataScience>.

Exercise 2-1.

The online code repository is “missing” code to create [Figure 2-2](#). (It’s not really *missing*—I moved it into the solution to this exercise.) So, your goal here is to write your own code to produce [Figure 2-2](#).

Exercise 2-2.

Write an algorithm that computes the norm of a vector by translating [Equation 2-7](#) into code. Confirm, using random vectors with different dimensionalities and orientations, that you get the same result as `np.linalg.norm()`. This exercise is designed to give you more experience with indexing NumPy arrays and translating formulas into code; in practice, it’s often easier to use `np.linalg.norm()`.

Exercise 2-3.

Create a Python function that will take a vector as input and output a unit vector in the same direction. What happens when you input the zeros vector?

Exercise 2-4.

You know how to create *unit* vectors; what if you want to create a vector of any arbitrary magnitude? Write a Python function that will take a vector and a desired magnitude as inputs and will return a vector in the same direction but with a magnitude corresponding to the second input.

Exercise 2-5.

Write a `for` loop to transpose a row vector into a column vector without using a built-in function or method such as `np.transpose()` or `v.T`. This exercise will help you create and index orientation-endowed vectors.

Exercise 2-6.

Here is an interesting fact: you can compute the squared norm of a vector as the dot product of that vector with itself. Look back to [Equation 2-8](#) to convince yourself of this equivalence. Then confirm it using Python.

Exercise 2-7.

Write code to demonstrate that the dot product is *commutative*. Commutative means that $a \times b = b \times a$, which, for the vector dot product, means that $\mathbf{a}^T \mathbf{b} = \mathbf{b}^T \mathbf{a}$. After demonstrating this in code, use equation [Equation 2-9](#) to understand why the dot product is commutative.

Exercise 2-8.

Write code to produce [Figure 2-6](#). (Note that your solution doesn't need to look *exactly* like the figure, as long as the key elements are present.)

Exercise 2-9.

Implement orthogonal vector decomposition. Start with two random-number vectors \mathbf{t} and \mathbf{r} , and reproduce [Figure 2-8](#) (note that your plot will look somewhat different due to random numbers). Next, confirm that the two components sum to \mathbf{t} and that $\mathbf{t} \perp \mathbf{r}$ and $\mathbf{t} \parallel \mathbf{r}$ are orthogonal.

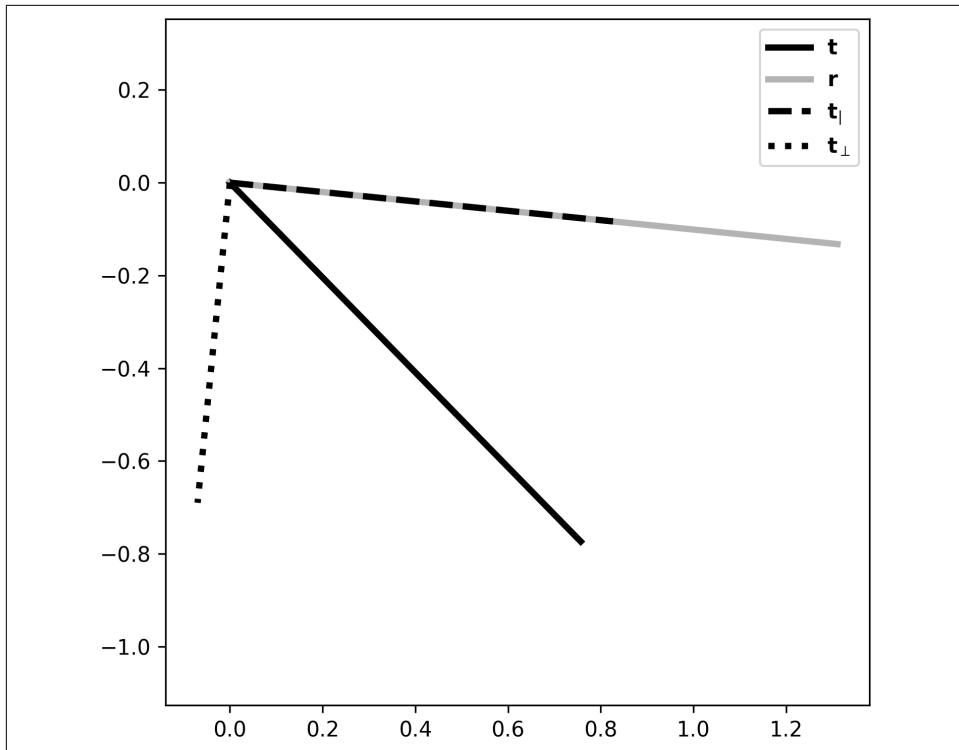


Figure 2-8. Exercise 9

Exercise 2-10.

An important skill in coding is finding bugs. Let's say there is a bug in your code such that the denominator in the projection scalar of [Equation 2-13](#) is $t^T t$ instead of $r^T r$ (an easy mistake to make, speaking from personal experience while writing this chapter!). Implement this bug to check whether it really deviates from the accurate code. What can you do to check whether the result is correct or incorrect? (In coding, confirming code with known results is called *sanity-checking*.)

Vectors, Part 2

The previous chapter laid the groundwork for understanding vectors and basic operations acting on vectors. Now you will expand the horizons of your linear algebra knowledge by learning about a set of interrelated concepts including linear independence, subspaces, and bases. Each of these topics is crucially important for understanding operations on matrices.

Some of the topics here may seem abstract and disconnected from applications, but there is a very short path between them, e.g., vector subspaces and fitting statistical models to data. The applications in data science come later, so please keep focusing on the fundamentals so that the advanced topics are easier to understand.

Vector Sets

We can start the chapter with something easy: a collection of vectors is called a *set*. You can imagine putting a bunch of vectors into a bag to carry around. Vector sets are indicated using capital italics letters, like S or V . Mathematically, we can describe sets as the following:

$$V = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$$

Imagine, for example, a dataset of the number of Covid-19 positive cases, hospitalizations, and deaths from one hundred countries; you could store the data from each country in a three-element vector, and create a vector set containing one hundred vectors.

Vector sets can contain a finite or an infinite number of vectors. Vector sets with an infinite number of vectors may sound like a uselessly silly abstraction, but vector

subspaces are infinite vector sets and have major implications for fitting statistical models to data.

Vector sets can also be empty, and are indicated as $V = \{\}$. You'll encounter empty vector sets when you learn about matrix spaces.

Linear Weighted Combination

A *linear weighted combination* is a way of mixing information from multiple variables, with some variables contributing more than others. This fundamental operation is also sometimes called *linear mixture* or *weighted combination* (the *linear* part is assumed). Sometimes, the term *coefficient* is used instead of *weight*.

Linear weighted combination simply means scalar-vector multiplication and addition: take some set of vectors, multiply each vector by a scalar, and add them to produce a single vector (Equation 3-1).

Equation 3-1. Linear weighted combination

$$\mathbf{w} = \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 + \dots + \lambda_n \mathbf{v}_n$$

It is assumed that all vectors \mathbf{v}_i have the same dimensionality; otherwise, the addition is invalid. The λ s can be any real number, including zero.

Technically, you could rewrite Equation 3-1 for subtracting vectors, but because subtraction can be handled by setting a λ_i to be negative, it's easier to discuss linear weighted combinations in terms of summation.

Equation 3-2 shows an example to help make it more concrete:

Equation 3-2. Linear weighted combination

$$\begin{aligned} \lambda_1 &= 1, \lambda_2 = 2, \lambda_3 = -3, & \mathbf{v}_1 &= \begin{bmatrix} 4 \\ 5 \\ 1 \end{bmatrix}, \mathbf{v}_2 &= \begin{bmatrix} -4 \\ 0 \\ -4 \end{bmatrix}, \mathbf{v}_3 &= \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} \\ \mathbf{w} &= \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 + \lambda_3 \mathbf{v}_3 & = & \begin{bmatrix} -7 \\ -4 \\ -13 \end{bmatrix} \end{aligned}$$

Linear weighted combinations are easy to implement, as the following code demonstrates. In Python, the data type is important; test what happens when the vectors are lists instead of NumPy arrays.¹

¹ As shown in Chapters 2 and 16, list-integer multiplication repeats the list instead of scalar multiplying it.

```

l1 = 1
l2 = 2
l3 = -3
v1 = np.array([4,5,1])
v2 = np.array([-4,0,-4])
v3 = np.array([1,3,2])
l1*v1 + l2*v2 + l3*v3

```

Storing each vector and each coefficient as separate variables is tedious and does not scale up to larger problems. Therefore, in practice, linear weighted combinations are implemented via the compact and scalable matrix-vector multiplication method, which you'll learn about in [Chapter 5](#); for now, the focus is on the concept and coding implementation.

Linear weighted combinations have several applications. Three of those include:

- The predicted data from a statistical model are created by taking the linear weighted combination of regressors (predictor variables) and coefficients (scalars) that are computed via the least squares algorithm, which you'll learn about in [Chapters 11 and 12](#).
- In dimension-reduction procedures such as principal components analysis, each component (sometimes called factor or mode) is derived as a linear weighted combination of the data channels, with the weights (the coefficients) selected to maximize the variance of the component (along with some other constraints that you'll learn about in [Chapter 15](#)).
- Artificial neural networks (the architecture and algorithm that powers deep learning) involve two operations: linear weighted combination of the input data, followed by a nonlinear transformation. The weights are learned by minimizing a cost function, which is typically the difference between the model prediction and the real-world target variable.

The concept of a linear weighted combination is the mechanism of creating vector subspaces and matrix spaces, and is central to linear independence. Indeed, linear weighted combination and the dot product are two of the most important elementary building blocks from which many advanced linear algebra computations are built.

Linear Independence

A set of vectors is *linearly dependent* if at least one vector in the set can be expressed as a linear weighted combination of other vectors in that set. And thus, a set of vectors is *linearly independent* if no vector can be expressed as a linear weighted combination of other vectors in the set.

Following are two vector sets. Before reading the text, try to determine whether each set is dependent or independent. (The term *linear independence* is sometimes shortened to *independence* when the *linear* part is implied.)

$$V = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 7 \end{bmatrix} \right\} \quad S = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 6 \end{bmatrix} \right\}$$

Vector set V is linearly independent: it is impossible to express one vector in the set as a linear multiple of the other vector in the set. That is to say, if we call the vectors in the set \mathbf{v}_1 and \mathbf{v}_2 , then there is no possible scalar λ for which $\mathbf{v}_1 = \lambda\mathbf{v}_2$.

How about set S ? This one is dependent, because we can use linear weighted combinations of some vectors in the set to obtain other vectors in the set. There is an infinite number of such combinations, two of which are $\mathbf{s}_1 = .5^*\mathbf{s}_2$ and $\mathbf{s}_2 = 2^*\mathbf{s}_1$.

Let's try another example. Again, the question is whether set T is linearly independent or linearly dependent:

$$T = \left\{ \begin{bmatrix} 8 \\ -4 \\ 14 \\ 6 \end{bmatrix}, \begin{bmatrix} 4 \\ 6 \\ 0 \\ 3 \end{bmatrix}, \begin{bmatrix} 14 \\ 2 \\ 4 \\ 7 \end{bmatrix}, \begin{bmatrix} 13 \\ 2 \\ 9 \\ 8 \end{bmatrix} \right\}$$

Wow, this one is a lot harder to figure out than the previous two examples. It turns out that this is a linearly dependent set (for example, the sum of the first three vectors equals twice the fourth vector). But I wouldn't expect you to be able to figure that out just from visual inspection.

So how do you determine linear independence in practice? The way to determine linear independence is to create a matrix from the vector set, compute the rank of the matrix, and compare the rank to the smaller of the number of rows or columns. That sentence may not make sense to you now, because you haven't yet learned about matrix rank. Therefore, focus your attention now on the concept that a set of vectors is linearly dependent if at least one vector in the set can be expressed as a linear weighted combination of the other vectors in the set, and a set of vectors is linearly independent if no vector can be expressed as a combination of other vectors.

Independent Sets

Independence is a property of a *set* of vectors. That is, a set of vectors can be linearly independent or linearly dependent; independence is not a property of an individual vector within a set.

The Math of Linear Independence

Now that you understand the concept, I want to make sure you also understand the formal mathematical definition of linear dependence, which is expressed in [Equation 3-3](#).

*Equation 3-3. Linear dependence*²

$$\mathbf{0} = \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 + \dots + \lambda_n \mathbf{v}_n, \quad \lambda \in \mathbb{R}$$

This equation says that linear dependence means that we can define some linear weighted combination of the vectors in the set to produce the zeros vector. If you can find some λ s that make the equation true, then the set of vectors is linearly dependent. Conversely, if there is no possible way to linearly combine the vectors to produce the zeros vector, then the set is linearly independent.

That might initially be unintuitive. Why do we care about the zeros vector when the question is whether we can express at least one vector in the set as a weighted combination of other vectors in the set? Perhaps you'd prefer rewriting the definition of linear dependence as the following:

$$\lambda_1 \mathbf{v}_1 = \lambda_2 \mathbf{v}_2 + \dots + \lambda_n \mathbf{v}_n, \quad \lambda \in \mathbb{R}$$

Why not start with that equation instead of putting the zeros vector on the left-hand side? Setting the equation to zero helps reinforce the principle that the *entire set* is dependent or independent; no individual vector has the privileged position of being the “dependent vector” (see “[Independent Sets](#)” on [page 36](#)). In other words, when it comes to independence, vector sets are purely egalitarian.

But wait a minute. Careful inspection of [Equation 3-3](#) reveals a trivial solution: set all λ 's to zero, and the equation reads $\mathbf{0} = \mathbf{0}$, regardless of the vectors in the set. But, as I wrote in [Chapter 2](#), trivial solutions involving zeros are often ignored in linear algebra. So we add the constraint that at least one $\lambda \neq 0$.

This constraint can be incorporated into the equation by dividing through by one of the scalars; keep in mind that \mathbf{v}_1 and λ_1 can refer to any vector/scalar pair in the set:

$$\mathbf{0} = \mathbf{v}_1 + \dots + \frac{\lambda_n}{\lambda_1} \mathbf{v}_n, \quad \lambda \in \mathbb{R}, \lambda_1 \neq 0$$

² This equation is an application of linear weighted combination!

Independence and the Zeros Vector

Simply put, any vector set that includes the zeros vector is automatically a linearly dependent set. Here's why: any scalar multiple of the zeros vector is still the zeros vector, so the definition of linear dependence is always satisfied. You can see this in the following equation:

$$\lambda_0 \mathbf{0} = 0\mathbf{v}_1 + 0\mathbf{v}_2 + 0\mathbf{v}_n$$

As long as $\lambda_0 \neq 0$, we have a nontrivial solution, and the set fits with the definition of linear dependence.



What About Nonlinear Independence?

“But Mike,” I imagine you protesting, “isn’t life, the universe, and everything *nonlinear*?” I suppose it would be an interesting exercise to count the total number of linear versus nonlinear interactions in the universe and see which sum is larger. But linear algebra is all about, well, *linear* operations. If you can express one vector as a nonlinear (but not linear) combination of other vectors, then those vectors still form a linearly independent set. The reason for the linearity constraint is that we want to express transformations as matrix multiplication, which is a linear operation. That’s not to throw shade on nonlinear operations—in my imaginary conversation, you have eloquently articulated that a purely linear universe would be rather dull and predictable. But we don’t need to explain the entire universe using linear algebra; we need linear algebra only for the linear parts. (It’s also worth mentioning that many nonlinear systems can be well approximated using linear functions.)

Subspace and Span

When I introduced linear weighted combinations, I gave examples with specific numerical values for the weights (e.g., $\lambda_1 = 1, \lambda_3 = -3$). A *subspace* is the same idea but using the infinity of possible ways to linearly combine the vectors in the set.

That is, for some (finite) set of vectors, the infinite number of ways to linearly combine them—using the same vectors but different numerical values for the weights—creates a *vector subspace*. And the mechanism of combining all possible linear weighted combinations is called the *span* of the vector set. Let’s work through a few examples. We’ll start with a simple example of a vector set containing one vector:

$$V = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix} \right\}$$

The span of this vector set is the infinity of vectors that can be created as linear combinations of the vectors in the set. For a set with one vector, that simply means all possible scaled versions of that vector. [Figure 3-1](#) shows the vector and the subspace it spans. Consider that any vector in the gray dashed line can be formed as some scaled version of the vector.

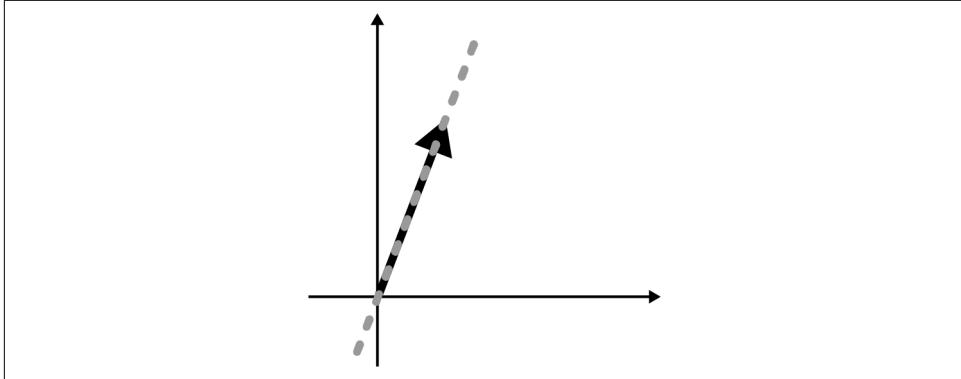


Figure 3-1. A vector (black) and the subspace it spans (gray)

Our next example is a set of two vectors in \mathbb{R}^3 :

$$V = \left\{ \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix} \right\}$$

The vectors are in \mathbb{R}^3 , so they are graphically represented in a 3D axis. But the subspace that they span is a 2D plane in that 3D space ([Figure 3-2](#)). That plane passes through the origin, because scaling both vectors by zero gives the zeros vector.

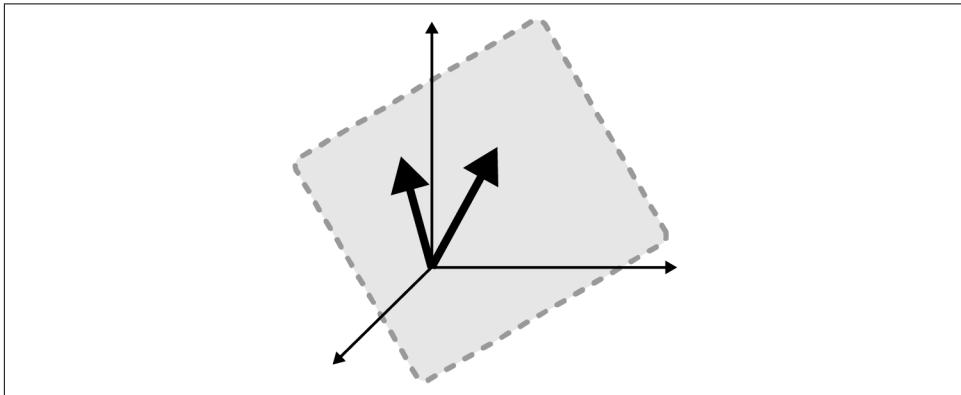


Figure 3-2. Two vectors (black) and the subspace they span (gray)

The first example had one vector and its span was a 1D subspace, and the second example had two vectors and their span was a 2D subspace. There seems to be a pattern emerging—but looks can be deceiving. Consider the next example:

$$V = \left\{ \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \right\}$$

Two vectors in \mathbb{R}^3 , but the subspace that they span is still only a 1D subspace—a line (Figure 3-3). Why is that? It's because one vector in the set is already in the span of the other vector. Thus, in terms of span, one of the two vectors is redundant.

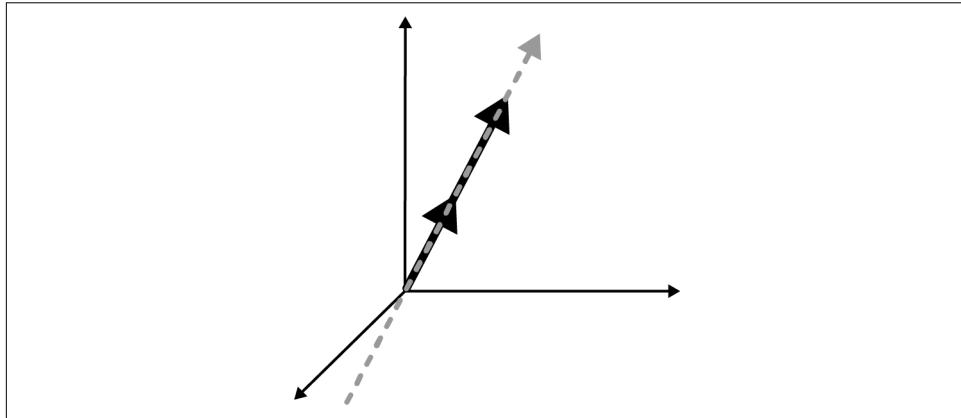


Figure 3-3. The 1D subspace (gray) spanned by two vectors (black)

So then, what is the relationship between the dimensionality of the spanned subspace and the number of vectors in the set? You might have guessed that it has something to do with linear independence.

The dimensionality of the subspace spanned by a set of vectors is the smallest number of vectors that forms a linearly independent set. If a vector set is linearly independent, then the dimensionality of the subspace spanned by the vectors in that set equals the number of vectors in that set. If the set is dependent, then the dimensionality of the subspace spanned by those vectors is necessarily less than the number of vectors in that set. Exactly how much smaller is another matter—to know the relationship between the number of vectors in a set and the dimensionality of their spanning subspace, you need to understand matrix rank, which you'll learn about in [Chapter 6](#).

The formal definition of a **vector subspace** is a subset that is closed under addition and scalar multiplication, and includes the origin of the space. That means that any linear weighted combination of vectors in the subspace must also be in the same

subspace, including setting all weights to zero to produce the zeros vector at the origin of the space.

Please don't lose sleep meditating on what it means to be "closed under addition and scalar multiplication"; just remember that a vector subspace is created from all possible linear combinations of a set of vectors.

What's the Difference Between Subspace and Span?

Many students are confused about the difference between *span* and *subspace*. That's understandable, because they are highly related concepts and often refer to the same thing. I will explain the difference between them, but don't stress about the subtleties—span and subspace so often refer to identical mathematical objects that using the terms interchangeably is usually correct.

I find that thinking of *span* as a verb and *subspace* as a noun helps understand their distinction: a set of vectors spans, and the result of their spanning is a subspace. Now consider that a *subspace* can be a smaller portion of a larger space, as you saw in [Figure 3-3](#). Putting these together: span is the mechanism of creating a subspace. (On the other hand, when you use span as a noun, then span and subspace refer to the same infinite vector set.)

Basis

How far apart are Amsterdam and Tenerife? Approximately 2,000. What does "2,000" mean? That number makes sense only if we attach a basis unit. A basis is like a ruler for measuring a space.

In this example, the unit is *mile*. So our basis measurement for Dutch-Spanish distance is 1 mile. We could, of course, use different measurement units, like nanometers or light-years, but I think we can agree that mile is a convenient basis for distance at that scale. What about the length that your fingernail grows in one day—should we still use miles? Technically we can, but I think we can agree that *millimeter* is a more convenient basis unit. To be clear: the amount that your fingernail has grown in the past 24 hours is the same, regardless of whether you measure it in nanometers, miles, or light-years. But different units are more or less convenient for different problems.

Back to linear algebra: a *basis* is a set of rulers that you use to describe the information in the matrix (e.g., data). Like with the previous examples, you can describe the same data using different rulers, but some rulers are more convenient than others for solving certain problems.

The most common basis set is the Cartesian axis: the familiar XY plane that you've used since elementary school. We can write out the basis sets for the 2D and 3D Cartesian graphs as follows:

$$S_2 = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\} \quad S_3 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

Notice that the Cartesian basis sets comprise vectors that are mutually orthogonal and unit length. Those are great properties to have, and that's why the Cartesian basis sets are so ubiquitous (indeed, they are called the *standard basis set*).

But those are not the only basis sets. The following set is a different basis set for \mathbb{R}^2 .

$$T = \left\{ \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \begin{bmatrix} -3 \\ 1 \end{bmatrix} \right\}$$

Basis set S_2 and T both span the same subspace (all of \mathbb{R}^2). Why would you prefer T over S ? Imagine we want to describe data points p and q in [Figure 3-4](#). We can describe those data points as their relationship to the origin—that is, their coordinates—using basis S or basis T .

In basis S , those two coordinates are $p = (3, 1)$ and $q = (-6, 2)$. In linear algebra, we say that the points are expressed as the linear combinations of the basis vectors. In this case, that combination is $3s_1 + 1s_2$ for point p , and $-6s_1 + 2s_2$ for point q .

Now let's describe those points in basis T . As coordinates, we have $p = (1, 0)$ and $q = (0, 2)$. And in terms of basis vectors, we have $1t_1 + 0t_2$ for point p and $0t_1 + 2t_2$ for point q (in other words, $p = t_1$ and $q = 2t_2$). Again, the data points p and q are the same regardless of the basis set, but T provided a compact and orthogonal description.

Bases are extremely important in data science and machine learning. In fact, many problems in applied linear algebra can be conceptualized as finding the best set of basis vectors to describe some subspace. You've probably heard of the following terms: dimension reduction, feature extraction, principal components analysis, independent components analysis, factor analysis, singular value decomposition, linear discriminant analysis, image approximation, data compression. Believe it or not, all of those analyses are essentially ways of identifying optimal basis vectors for a specific problem.

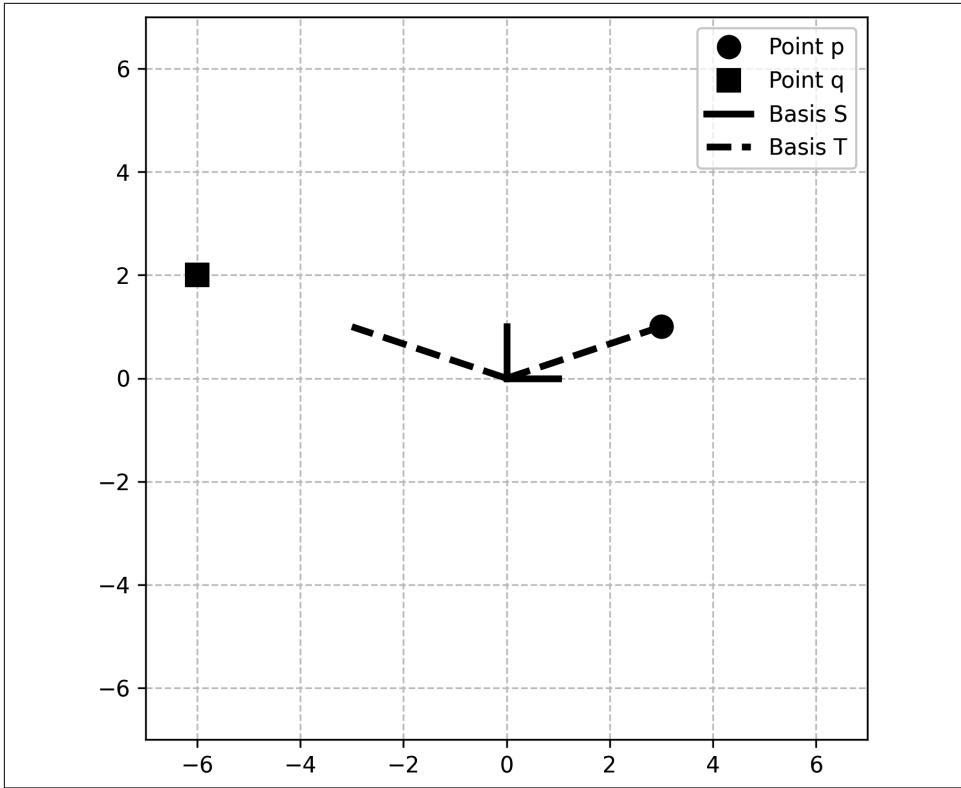


Figure 3-4. The same points (p and q) can be described by basis set S (black solid lines) or T (black dashed lines)

Consider Figure 3-5: this is a dataset of two variables (each dot represents a data point). The figure actually shows three distinct bases: the “standard basis set” corresponding to the $x = 0$ and $y = 0$ lines, and basis sets defined via a principal components analysis (PCA; left plot) and via an independent components analysis (ICA; right plot). Which of these basis sets provides the “best” way of describing the data? You might be tempted to say that the basis vectors computed from the ICA are the best. The truth is more complicated (as it tends to be): no basis set is intrinsically better or worse; different basis sets can be more or less helpful for specific problems based on the goals of the analysis, the features of the data, constraints imposed by the analyses, and so on.

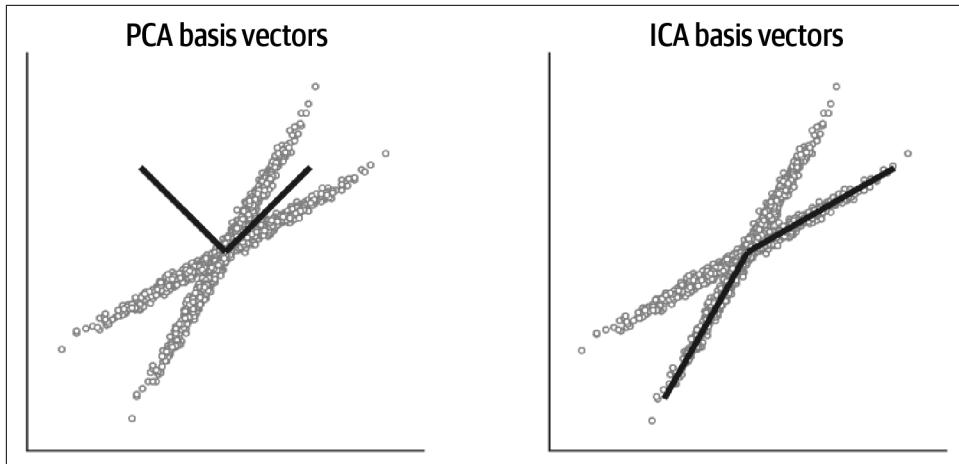


Figure 3-5. A 2D dataset using different basis vectors (black lines)

Definition of Basis

Once you understand the concept of a basis and basis set, the formal definition is straightforward. In fact, basis is simply the combination of span and independence: a set of vectors can be a basis for some subspace if it (1) spans that subspace and (2) is an independent set of vectors.

The basis needs to span a subspace for it to be used as a basis for that subspace, because you cannot describe something that you cannot measure.³ Figure 3-6 shows an example of a point outside of a 1D subspace. A basis vector for that subspace cannot measure the point r . The black vector is still a valid basis vector for the subspace it spans, but it does not form a basis for any subspace beyond what it spans.

³ A general truism in science.

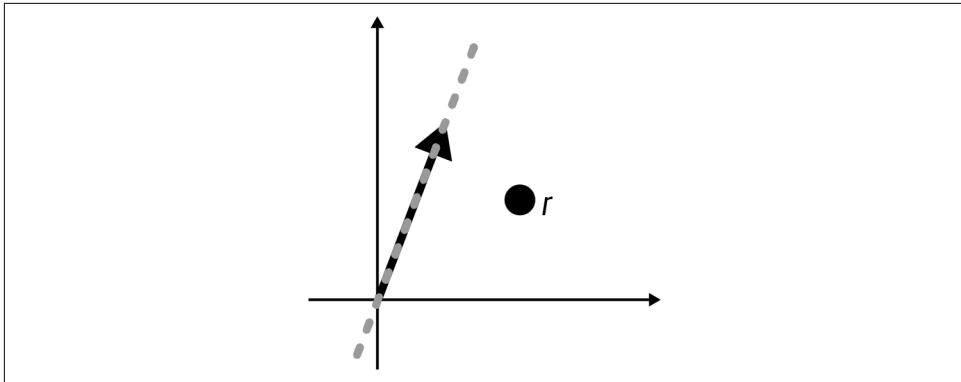


Figure 3-6. A basis set can measure only what is contained inside its span

So a basis needs to span the space that it is used for. That's clear. But why does a basis set require linear independence? The reason is that any given vector in the subspace must have a unique coordinate using that basis. Let's imagine describing point p from Figure 3-4 using the following vector set:

$$U = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$$

U is a perfectly valid vector set, but it is definitely *not* a basis set. Why not?⁴

What linear weighted combination describes point p in set U ? Well, the coefficients for the linear weighted combination of the three vectors in U could be $(3, 0, 1)$ or $(0, 1.5, 1)$ or...a bajillion other possibilities. That's confusing, and so mathematicians decided that a vector must have *unique* coordinates within a basis set. Linear independence guarantees uniqueness.

To be clear, point p (or any other point) can be described using an infinite number of basis sets. So the measurement is not unique in terms of the plethora of possible basis sets. But *within* a basis set, a point is defined by exactly one linear weighted combination. It's the same thing with my distance analogy at the beginning of this section: we can measure the distance from Amsterdam to Tenerife using many different measurement units, but that distance has only one value per measurement unit. The distance is not simultaneously 3,200 miles and 2,000 miles, but it is simultaneously 3,200 *kilometers* and 2,000 *miles*. (Note for nerds: I'm approximating here, OK?)

⁴ Because it is a linearly dependent set.

Summary

Congratulations on finishing another chapter! (Well, almost finished: there are coding exercises to solve.) The point of this chapter was to bring your foundational knowledge about vectors to the next level. Below is a list of key points, but please remember that underlying all of these points is a very small number of elementary principles, primarily linear weighted combinations of vectors:

- A vector set is a collection of vectors. There can be a finite or an infinite number of vectors in a set.
- Linear weighted combination means to scalar multiply and add vectors in a set. Linear weighted combination is one of the single most important concepts in linear algebra.
- A set of vectors is linearly dependent if a vector in the set can be expressed as a linear weighted combination of other vectors in the set. And the set is linearly independent if there is no such linear weighted combination.
- A subspace is the infinite set of all possible linear weighted combinations of a set of vectors.
- A basis is a ruler for measuring a space. A vector set can be a basis for a subspace if it (1) spans that subspace and (2) is linearly independent. A major goal in data science is to discover the best basis set to describe datasets or to solve problems.

Code Exercises

Exercise 3-1.

Rewrite the code for linear weighted combination, but put the scalars in a list and the vectors as elements in a list (thus, you will have two lists, one of scalars and one of NumPy arrays). Then use a `for` loop to implement the linear weighted combination operation. Initialize the output vector using `np.zeros()`. Confirm that you get the same result as in the previous code.

Exercise 3-2.

Although the method of looping through lists in the previous exercise is not as efficient as matrix-vector multiplication, it is more scalable than without a `for` loop. You can explore this by adding additional scalars and vectors as elements in the lists. What happens if the new added vector is in \mathbb{R}^4 instead of \mathbb{R}^3 ? And what happens if you have more scalars than vectors?

Exercise 3-3.

In this exercise, you will draw random points in subspaces. This will help reinforce the idea that subspaces comprise *any* linear weighted combination of the spanning vectors. Define a vector set containing one vector $[1, 3]$. Then create 100 numbers drawn randomly from a uniform distribution between -4 and $+4$. Those are your random scalars. Multiply the random scalars by the basis vector to create 100 random points in the subspace. Plot those points.

Next, repeat the procedure but using two vectors in \mathbb{R}^3 : $[3, 5, 1]$ and $[0, 2, 2]$. Note that you need 100×2 random scalars for 100 points and two vectors. The resulting random dots will be on a plane. [Figure 3-7](#) shows what the results will look like (it's not clear from the figure that the points lie on a plane, but you'll see this when you drag the plot around on your screen).

I recommend using the `plotly` library to draw the dots, so you can click-drag the 3D axis around. Here's a hint for getting it to work:

```
import plotly.graph_objects as go
fig = go.Figure( data=[go.Scatter3d(
    x=points[:,0], y=points[:,1], z=points[:,2],
    mode='markers' )])
fig.show()
```

Finally, repeat the \mathbb{R}^3 case but setting the second vector to be $1/2$ times the first.

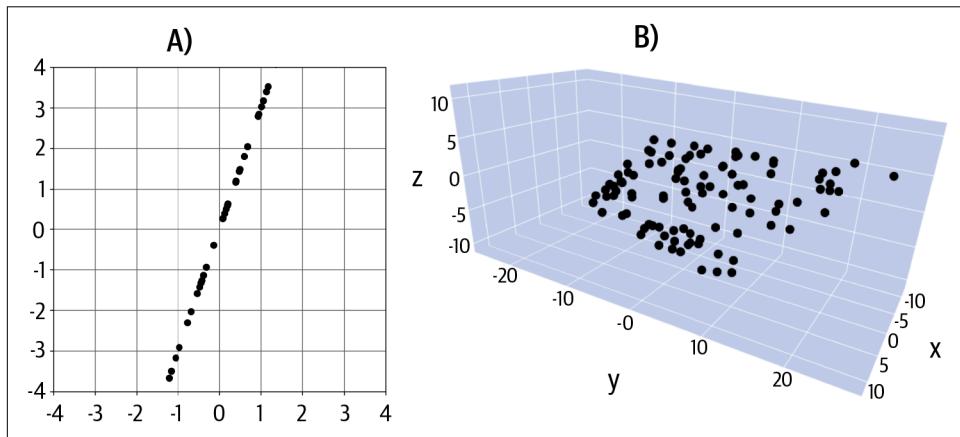


Figure 3-7. Exercise 3-3

CHAPTER 4

Vector Applications

While working through the previous two chapters, you may have felt that some of the material was esoteric and abstract. Perhaps you felt that the challenge of learning linear algebra wouldn't pay off in understanding real-world applications in data science and machine learning.

I hope that this chapter dispels you of these doubts. In this chapter, you will learn how vectors and vector operations are used in data science analyses. And you will be able to extend this knowledge by working through the exercises.

Correlation and Cosine Similarity

Correlation is one of the most fundamental and important analysis methods in statistics and machine learning. A *correlation coefficient* is a single number that quantifies the linear relationship between two variables. Correlation coefficients range from -1 to $+1$, with -1 indicating a perfect negative relationship, $+1$ a perfect positive relationships, and 0 indicating no linear relationship. [Figure 4-1](#) shows a few examples of pairs of variables and their correlation coefficients.

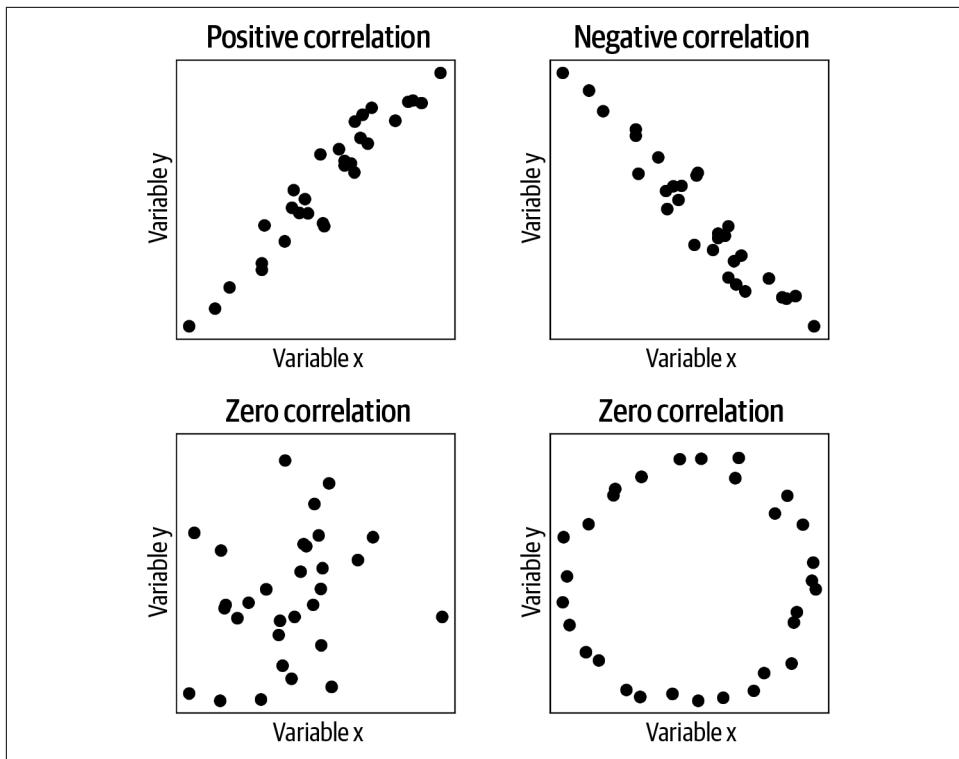


Figure 4-1. Examples of data exhibiting positive correlation, negative correlation, and zero correlation. The lower-right panel illustrates that correlation is a linear measure; nonlinear relationships between variables can exist even if their correlation is zero.

In [Chapter 2](#), I mentioned that the dot product is involved in the correlation coefficient, and that the magnitude of the dot product is related to the magnitude of the numerical values in the data (remember the discussion about using grams versus pounds for measuring weight). Therefore, the correlation coefficient requires some normalizations to be in the expected range of -1 to $+1$. Those two normalizations are:

Mean center each variable

Mean centering means to subtract the average value from each data value.

Divide the dot product by the product of the vector norms

This divisive normalization cancels the measurement units and scales the maximum possible correlation magnitude to $|1|$.

Equation 4-1 shows the full formula for the Pearson correlation coefficient.

Equation 4-1. Formula for Pearson correlation coefficient

$$\rho = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

It may not be obvious that the correlation is simply three dot products. Equation 4-2 shows this same formula rewritten using the linear algebra dot-product notation. In this equation, $\tilde{\mathbf{x}}$ is the mean-centered version of \mathbf{x} (that is, variable \mathbf{x} with normalization #1 applied).

Equation 4-2. The Pearson correlation expressed in the parlance of linear algebra

$$\rho = \frac{\tilde{\mathbf{x}}^T \tilde{\mathbf{y}}}{\|\tilde{\mathbf{x}}\| \|\tilde{\mathbf{y}}\|}$$

So there you go: the famous and widely used Pearson correlation coefficient is simply the dot product between two variables, normalized by the magnitudes of the variables. (By the way, you can also see from this formula that if the variables are unit normed such that $\|\mathbf{x}\| = \|\mathbf{y}\| = 1$, then their correlation equals their dot product. (Recall from Exercise 2-6 that $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T \mathbf{x}}$.)

Correlation is not the only way to assess similarity between two variables. Another method is called *cosine similarity*. The formula for cosine similarity is simply the geometric formula for the dot product (Equation 2-11), solved for the cosine term:

$$\cos(\theta_{x,y}) = \frac{\alpha}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

where α is the dot product between \mathbf{x} and \mathbf{y} .

It may seem like correlation and cosine similarity are exactly the same formula. However, remember that Equation 4-1 is the full formula, whereas Equation 4-2 is a simplification under the assumption that the variables have already been mean centered. Thus, cosine similarity does not involve the first normalization factor.

Correlation Versus Cosine Similarity

What does it mean for two variables to be “related”? Pearson correlation and cosine similarity can give different results for the same data because they start from different assumptions. In the eyes of Pearson, the variables [0, 1, 2, 3] and [100, 101, 102, 103] are perfectly correlated ($\rho = 1$) because changes in one variable are exactly mirrored

in the other variable; it doesn't matter that one variable has larger numerical values. However, the cosine similarity between those variables is .808—they are not in the same numerical scale and are therefore not perfectly related. Neither measure is incorrect nor better than the other; it is simply the case that different statistical methods make different assumptions about data, and those assumptions have implications for the results—and for proper interpretation. You'll have the opportunity to explore this in [Exercise 4-2](#).

From this section, you can understand why the Pearson correlation and cosine similarity reflects the *linear* relationship between two variables: they are based on the dot product, and the dot product is a linear operation.

There are four coding exercises associated with this section, which appear at the end of the chapter. You can choose whether you want to solve those exercises before reading the next section, or continue reading the rest of the chapter and then work through the exercises. (My personal recommendation is the former, but you are the master of your linear algebra destiny!)

Time Series Filtering and Feature Detection

The dot product is also used in time series filtering. Filtering is essentially a feature-detection method, whereby a template—called a *kernel* in the parlance of filtering—is matched against portions of a time series signal, and the result of filtering is another time series that indicates how much the characteristics of the signal match the characteristics of the kernel. Kernels are carefully constructed to optimize certain criteria, such as smooth fluctuations, sharp edges, particular waveform shapes, and so on.

The mechanism of filtering is to compute the dot product between the kernel and the time series signal. But filtering usually requires *local* feature detection, and the kernel is typically much shorter than the entire time series. Therefore, we compute the dot product between the kernel and a short snippet of the data of the same length as the kernel. This procedure produces one time point in the filtered signal ([Figure 4-2](#)), and then the kernel is moved one time step to the right to compute the dot product with a different (overlapping) signal segment. Formally, this procedure is called convolution and involves a few additional steps that I'm omitting to focus on the application of the dot product in signal processing.

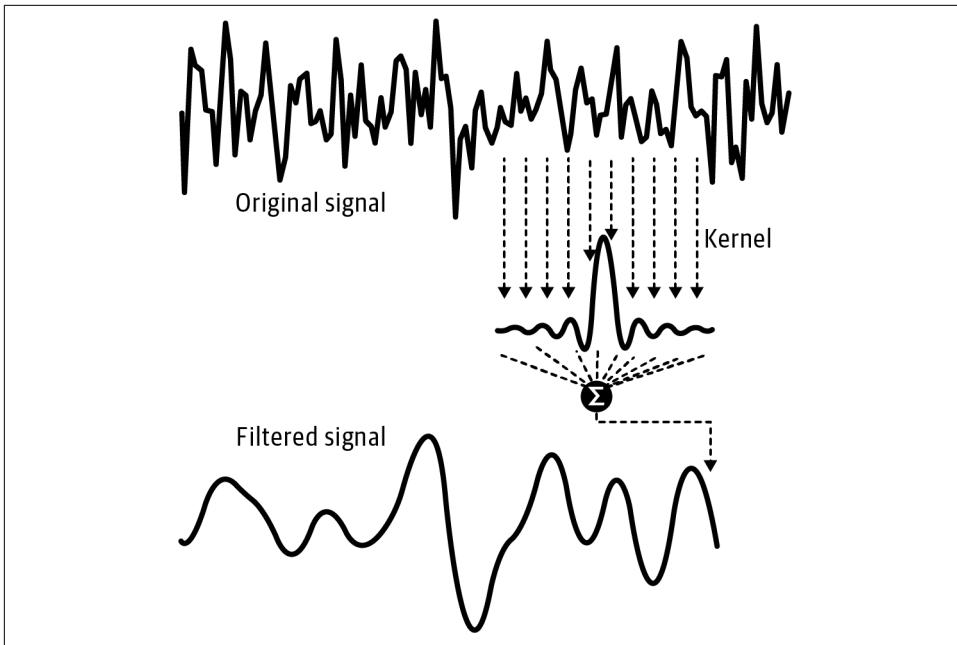


Figure 4-2. Illustration of time series filtering

Temporal filtering is a major topic in science and engineering. Indeed, without temporal filtering there would be no music, radio, telecommunications, satellites, etc. And yet, the mathematical heart that keeps your music pumping is the vector dot product.

In the exercises at the end of the chapter, you will discover how dot products are used to detect features (edges) and to smooth time series data.

***k*-Means Clustering**

k-means clustering is an unsupervised method of classifying multivariate data into a relatively small number of groups, or categories, based on minimizing distance to the group center.

k-means clustering is an important analysis method in machine learning, and there are sophisticated variants of *k*-means clustering. Here we will implement a simple version of *k*-means, with the goal of seeing how concepts about vectors (in particular: vectors, vector norms, and broadcasting) are used in the *k*-means algorithm.

Here is a brief description of the algorithm that we will write:

1. Initialize k centroids as random points in the data space. Each centroid is a *class*, or category, and the next steps will assign each data observation to each class. (A *centroid* is a center generalized to any number of dimensions.)
2. Compute the Euclidean distance between each data observation and each centroid.¹
3. Assign each data observation to the group with the closest centroid.
4. Update each centroid as the average of all data observations assigned to that centroid.
5. Repeat steps 2–4 until a convergence criteria is satisfied, or for N iterations.

If you are comfortable with Python coding and would like to implement this algorithm, then I encourage you to do that before continuing. Next, we will work through the math and code for each of these steps, with a particular focus on using vectors and broadcasting in NumPy. We will also test the algorithm using randomly generated 2D data to confirm that our code is correct.

Let's start with step 1: initialize k random cluster centroids. k is a parameter of k -means clustering; in real data, it is difficult to determine the optimal k , but here we will fix $k = 3$. There are several ways to initialize random cluster centroids; to keep things simple, I will randomly select k data samples to be centroids. The data are contained in variable `data` (this variable is 150×2 , corresponding to 150 observations and 2 features) and visualized in the upper-left panel of [Figure 4-3](#) (the online code shows how to generate these data):

```
k = 3
ridx = np.random.choice(range(len(data)), k, replace=False)
centroids = data[ridx, :] # data matrix is samples by features
```

Now for step 2: compute the distance between each data observation and each cluster centroid. Here is where we use linear algebra concepts you learned in the previous chapters. For one data observation and centroid, Euclidean distance is computed as

$$\delta_{i,j} = \sqrt{(d_i^x - c_j^x)^2 + (d_i^y - c_j^y)^2}$$

where $\delta_{i,j}$ indicates the distance from data observation i to centroid j , d_i^x is feature x of the i th data observation, and c_j^x is the x -axis coordinate of centroid j .

¹ Reminder: Euclidean distance is the square root of the sum of squared distances from the data observation to the centroid.

You might think that this step needs to be implemented using a double `for` loop: one loop over k centroids and a second loop over N data observations (you might even think of a third `for` loop over data features). However, we can use vectors and broadcasting to make this operation compact and efficient. This is an example of how linear algebra often looks different in equations compared to in code:

```
dists = np.zeros((data.shape[0],k))
for ci in range(k):
    dists[:,ci] = np.sum((data-centroids[ci,:])**2, axis=1)
```

Let's think about the sizes of these variables: `data` is 150×2 (observations by features) and `centroids[ci,:]` is 1×2 (cluster ci by features). Formally, it is not possible to subtract these two vectors. However, Python will implement broadcasting by repeating the cluster centroids 150 times, thus subtracting the centroid from each data observation. The exponent operation `**` is applied element-wise, and the `axis=1` input tells Python to sum across the columns (separately per row). So, the output of `np.sum()` will be a 150×1 array that encodes the Euclidean distance of each point to centroid ci .

Take a moment to compare the code to the distance formula. Are they really the same? In fact, they are not: the square root in Euclidean distance is missing from the code. So is the code wrong? Think about this for a moment; I'll discuss the answer later.

Step 3 is to assign each data observation to the group with minimum distance. This step is quite compact in Python, and can be implemented using one function:

```
groupidx = np.argmin(dists, axis=1)
```

Note the difference between `np.min`, which returns the minimum *value*, versus `np.argmin`, which returns the *index* at which the minimum occurs.

We can now return to the inconsistency between the distance formula and its code implementation. For our k -means algorithm, we use distance to assign each data point to its closest centroid. Distance and squared distance are monotonically related, so both metrics give the same answer. Adding the square root operation increases code complexity and computation time with no impact on the results, so it can simply be omitted.

Step 4 is to recompute the centroids as the mean of all data points within the class. Here we can loop over the k clusters, and use Python indexing to find all data points assigned to each cluster:

```
for ki in range(k):
    centroids[ki,:] = [ np.mean(data[groupidx==ki,0]),
                        np.mean(data[groupidx==ki,1]) ]
```

Finally, Step 5 is to put the previous steps into a loop that iterates until a good solution is obtained. In production-level k -means algorithms, the iterations continue until a stopping criteria is reached, e.g., that the cluster centroids are no longer moving around. For simplicity, here we will iterate three times (an arbitrary number selected to make the plot visually balanced).

The four panels in [Figure 4-3](#) show the initial random cluster centroids (iteration 0), and their updated locations after each of three iterations.

If you study clustering algorithms, you will learn sophisticated methods for centroid initialization and stopping criteria, as well as quantitative methods to select an appropriate k parameter. Nonetheless, all k -means methods are essentially extensions of the above algorithm, and linear algebra is at the heart of their implementations.

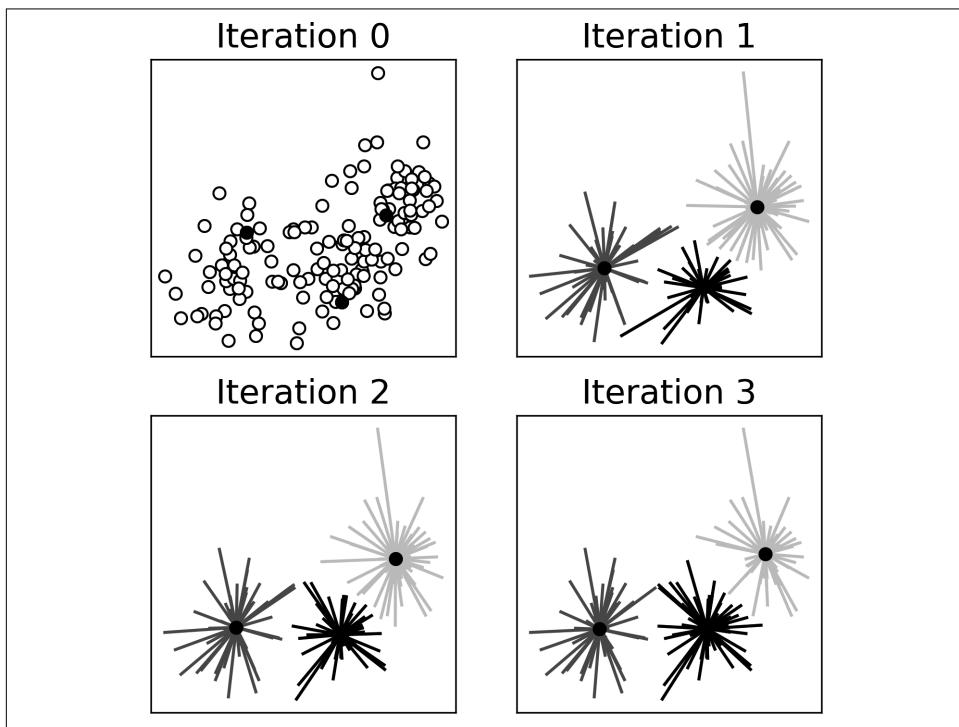


Figure 4-3. k-means

Code Exercises

Correlation Exercises

Exercise 4-1.

Write a Python function that takes two vectors as input and provides two numbers as output: the Pearson correlation coefficient and the cosine similarity value. Write code that follows the formulas presented in this chapter; don't simply call `np.corrcoef` and `spatial.distance.cosine`. Check that the two output values are identical when the variables are already mean centered and different when the variables are not mean centered.

Exercise 4-2.

Let's continue exploring the difference between correlation and cosine similarity. Create a variable containing the integers 0 through 3, and a second variable equaling the first variable plus some offset. You will then create a simulation in which you systematically vary that offset between -50 and $+50$ (that is, the first iteration of the simulation will have the second variable equal to $[-50, -49, -48, -47]$). In a `for` loop, compute the correlation and cosine similarity between the two variables and store these results. Then make a line plot showing how the correlation and cosine similarity are affected by the mean offset. You should be able to reproduce Figure 4-4.

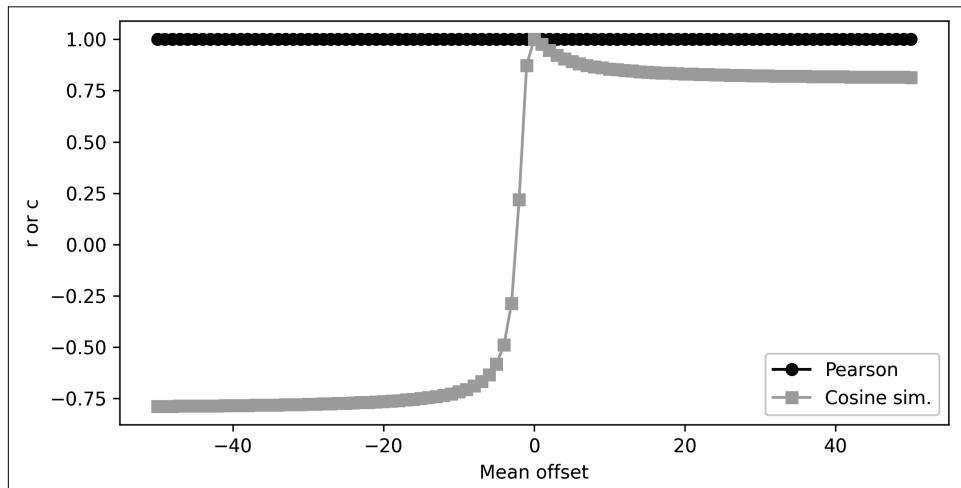


Figure 4-4. Results of Exercise 4-2

Exercise 4-3.

There are several Python functions to compute the Pearson correlation coefficient. One of them is called `pearsonr` and is located in the `stats` module of the SciPy library. Open the source code for this file (hint: `??functionname`) and make sure you understand how the Python implementation maps onto the formulas introduced in this chapter.

Exercise 4-4.

Why do you ever need to code your own functions when they already exist in Python? Part of the reason is that writing your own functions has huge educational value, because you see that (in this case) the correlation is a simple computation and not some incredibly sophisticated black-box algorithm that only a computer-science PhD could understand. But another reason is that built-in functions are sometimes slower because of myriad input checks, dealing with additional input options, converting data types, etc. This increases usability but at the expense of computation time.

Your goal in this exercise is to determine whether your own bare-bones correlation function is faster than NumPy's `corrcoef` function. Modify the function from [Exercise 4-2](#) to compute only the correlation coefficient. Then, in a `for` loop over 1,000 iterations, generate two variables of 500 random numbers and compute the correlation between them. Time the `for` loop. Then repeat but using `np.corrcoef`. In my tests, the custom function was about 33% faster than `np.corrcoef`. In these toy examples, the differences are measured in milliseconds, but if you are running billions of correlations with large datasets, those milliseconds really add up! (Note that writing your own functions without input checks has the risk of input errors that would be caught by `np.corrcoef`.) (Also note that the speed advantage breaks down for larger vectors. Try it!)

Filtering and Feature Detection Exercises

Exercise 4-5.

Let's build an edge detector. The kernel for an edge detector is very simple: $[-1 \ 1]$. The dot product of that kernel with a snippet of a time series signal with constant value (e.g., $[10 \ 10]$) is 0. But that dot product is large when the signal has a steep change (e.g., $[1 \ 10]$ would produce a dot product of 9). The signal we'll work with is a plateau function. Graphs A and B in [Figure 4-5](#) show the kernel and the signal. The first step in this exercise is to write code that creates these two time series.

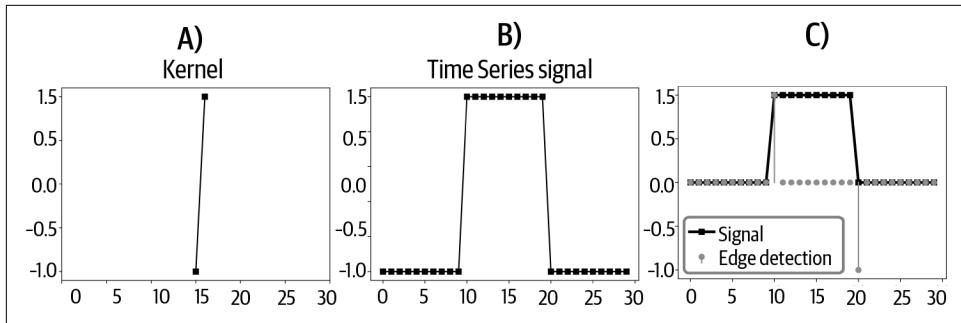


Figure 4-5. Results of Exercise 4-5

Next, write a `for` loop over the time points in the signal. At each time point, compute the dot product between the kernel and a segment of the time series data that has the same length as the kernel. You should produce a plot that looks like graph C in [Figure 4-5](#). (Focus more on the result than on the aesthetics.) Notice that our edge detector returned 0 when the signal was flat, +1 when the signal jumped up, and -1 when the signal jumped down.

Feel free to continue exploring this code. For example, does anything change if you pad the kernel with zeros ($[0 \ 1 \ 1 \ 0]$)? What about if you flip the kernel to be $[1 \ -1]$? How about if the kernel is asymmetric ($[-1 \ 2]$)?

Exercise 4-6.

Now we will repeat the same procedure but with a different signal and kernel. The goal will be to smooth a rugged time series. The time series will be 100 random numbers generated from a Gaussian distribution (also called a normal distribution). The kernel will be a bell-shaped function that approximates a Gaussian function, defined as the numbers $[0, .1, .3, .8, 1, .8, .3, .1, 0]$ but scaled so that the sum over the kernel is 1. Your kernel should match graph A in [Figure 4-6](#), although your signal won't look exactly like graph B due to random numbers.

Copy and adapt the code from the previous exercise to compute the sliding time series of dot products—the signal filtered by the Gaussian kernel. Warning: be mindful of the indexing in the `for` loop. Graph C in [Figure 4-6](#) shows an example result. You can see that the filtered signal is a smoothed version of the original signal. This is also called low-pass filtering.

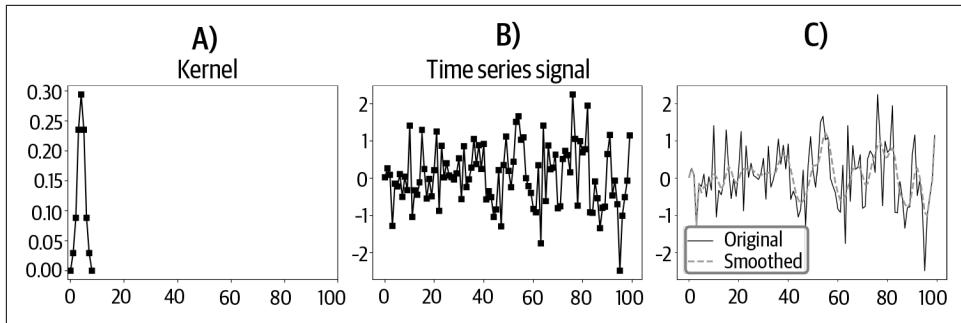


Figure 4-6. Results of Exercise 4-6

Exercise 4-7.

Replace the 1 in the center of the kernel with -1 and mean center the kernel. Then rerun the filtering and plotting code. What is the result? It actually accentuates the sharp features! In fact, this kernel is now a high-pass filter, meaning it dampens the smooth (low-frequency) features and highlights the rapidly changing (high-frequency) features.

k-Means Exercises

Exercise 4-8.

One way to determine an optimal k is to repeat the clustering multiple times (each time using randomly initialized cluster centroids) and assess whether the final clustering is the same or different. Without generating new data, rerun the k -means code several times using $k = 3$ to see whether the resulting clusters are similar (this is a qualitative assessment based on visual inspection). Do the final cluster assignments generally seem similar even though the centroids are randomly selected?

Exercise 4-9.

Repeat the multiple clusterings using $k = 2$ and $k = 4$. What do you think of these results?

Matrices, Part 1

A matrix is a vector taken to the next level. Matrices are highly versatile mathematical objects. They can store sets of equations, geometric transformations, the positions of particles over time, financial records, and myriad other things. In data science, matrices are sometimes called data tables, in which rows correspond to observations (e.g., customers) and columns correspond to features (e.g., purchases).

This and the following two chapters will take your knowledge about linear algebra to the next level. Get a cup of coffee and put on your thinking cap. Your brain will be bigger by the end of the chapter.

Creating and Visualizing Matrices in NumPy

Depending on the context, matrices can be conceptualized as a set of column vectors stacked next to each other (e.g., a data table of observations-by-features), as a set of row vectors layered on top of each other (e.g., multisensor data in which each row is a time series from a different channel), or as an ordered collection of individual matrix elements (e.g., an image in which each matrix element encodes pixel intensity value).

Visualizing, Indexing, and Slicing Matrices

Small matrices can simply be printed out in full, like the following examples:

$$\begin{bmatrix} 1 & 2 \\ \pi & 4 \\ 6 & 7 \end{bmatrix}, \quad \begin{bmatrix} -6 & 1/3 \\ e^{4.3} & -1.4 \\ 6/5 & 0 \end{bmatrix}$$

But that's not scalable, and matrices that you work with in practice can be large, perhaps containing billions of elements. Therefore, larger matrices can be visualized

as images. The numerical value of each element of the matrix maps onto a color in the image. In most cases, the maps are pseudo-colored because the mapping of numerical value onto color is arbitrary. [Figure 5-1](#) shows some examples of matrices visualized as images using the Python library `matplotlib`.

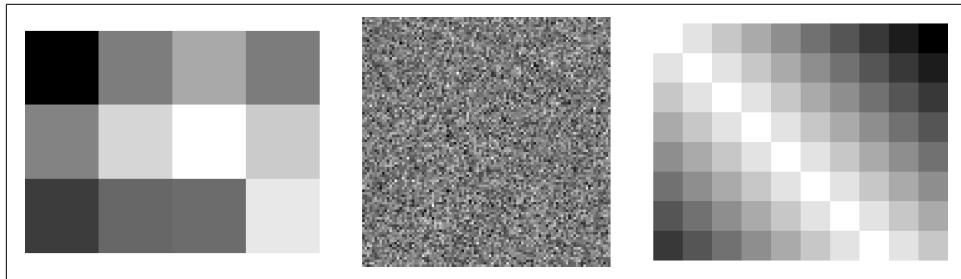


Figure 5-1. Three matrices, visualized as images

Matrices are indicated using bold-faced capital letters, like matrix **A** or **M**. The size of a matrix is indicated using (row, column) convention. For example, the following matrix is 3×5 because it has three rows and five columns:

$$\begin{bmatrix} 1 & 3 & 5 & 7 & 9 \\ 0 & 2 & 4 & 6 & 8 \\ 1 & 4 & 7 & 8 & 9 \end{bmatrix}$$

You can refer to specific elements of a matrix by indexing the row and column position: the element in the 3rd row and 4th column of matrix **A** is indicated as $a_{3,4}$ (and in the previous example matrix, $a_{3,4} = 8$). *Important reminder:* math uses 1-based indexing whereas Python uses 0-based indexing. Thus, element $a_{3,4}$ is indexed in Python as `A[2,3]`.

Extracting a subset of rows or columns of a matrix is done through slicing. If you are new to Python, you can consult [Chapter 16](#) for an introduction to slicing lists and NumPy arrays. To extract a section out of a matrix, you specify the start and end rows and columns, and that the slicing goes in steps of 1. The online code walks you through the procedure, and the following code shows an example of extracting a submatrix from rows 2–4 and columns 1–5 of a larger matrix:

```
A = np.arange(60).reshape(6,10)
sub = A[1:4:1,0:5:1]
```

And here are the full and submatrices:

```
Original matrix:
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]]
```

```
[30 31 32 33 34 35 36 37 38 39]  
[40 41 42 43 44 45 46 47 48 49]  
[50 51 52 53 54 55 56 57 58 59]]
```

Submatrix:

```
[[10 11 12 13 14]  
[20 21 22 23 24]  
[30 31 32 33 34]]
```

Special Matrices

There is an infinite number of matrices, because there is an infinite number of ways of organizing numbers into a matrix. But matrices can be described using a relatively small number of characteristics, which creates “families” or categories of matrices. These categories are important to know, because they appear in certain operations or have certain useful properties.

Some categories of matrices are used so frequently that they have dedicated NumPy functions to create them. Following is a list of some common special matrices and Python code to create them;¹ you can see what they look like in [Figure 5-2](#):

Random numbers matrix

This is a matrix that contains numbers drawn at random from some distribution, typically Gaussian (a.k.a. normal). Random-numbers matrices are great for exploring linear algebra in code, because they are quickly and easily created with any size and rank (matrix rank is a concept you’ll learn about in [Chapter 16](#)).

There are several ways to create random matrices in NumPy, depending on which distribution you want to draw numbers from. In this book, we’ll mostly use Gaussian-distributed numbers:

```
Mrows = 4 # shape 0  
Ncols = 6 # shape 1  
A = np.random.randn(Mrows,Ncols)
```

Square versus nonsquare

A square matrix has the same number of rows as columns; in other words, the matrix is in $\mathbb{R}^{N \times N}$. A nonsquare matrix, also sometimes called a rectangular matrix, has a different number of rows and columns. You can create square and rectangular matrices from random numbers by adjusting the shape parameters in the previous code.

Rectangular matrices are called *tall* if they have more rows than columns and *wide* if they have more columns than rows.

¹ There are more special matrices that you will learn about later in the book, but this list is enough to get started.

Diagonal

The *diagonal* of a matrix is the elements starting at the top-left and going down to the bottom-right. A *diagonal matrix* has zeros on all the off-diagonal elements; the diagonal elements may also contain zeros, but they are the only elements that may contain nonzero values.

The NumPy function `np.diag()` has two behaviors depending on the inputs: input a matrix and `np.diag` will return the diagonal elements as a vector; input a vector and `np.diag` will return a matrix with those vector elements on the diagonal. (Note: extracting the diagonal elements of a matrix is *not* called “diagonalizing a matrix”; that is a separate operation introduced in [Chapter 13](#).)

Triangular

A triangular matrix contains all zeros either above or below the main diagonal. The matrix is called *upper triangular* if the nonzero elements are above the diagonal and *lower triangular* if the nonzero elements are below the diagonal.

NumPy has dedicated functions to extract the upper (`np.triu()`) or lower (`np.tril()`) triangle of a matrix.

Identity

The identity matrix is one of the most important special matrices. It is the equivalent of the number 1, in that any matrix or vector times the identity matrix is that same matrix or vector. The identity matrix is a square diagonal matrix with all diagonal elements having a value of 1. It is indicated using the letter **I**. You might see a subscript to indicate its size (e.g., I_5 is the 5×5 identity matrix); if not, then you can infer the size from context (e.g., to make the equation consistent).

You can create an identity matrix in Python using `np.eye()`.

Zeros

The zeros matrix is comparable to the zeros vector: it is the matrix of all zeros. Like the zeros vector, it is indicated using a bold-faced zero: **0**. It can be a bit confusing to have the same symbol indicate both a vector and a matrix, but this kind of overloading is common in math and science notation.

The zeros matrix is created using the `np.zeros()` function.

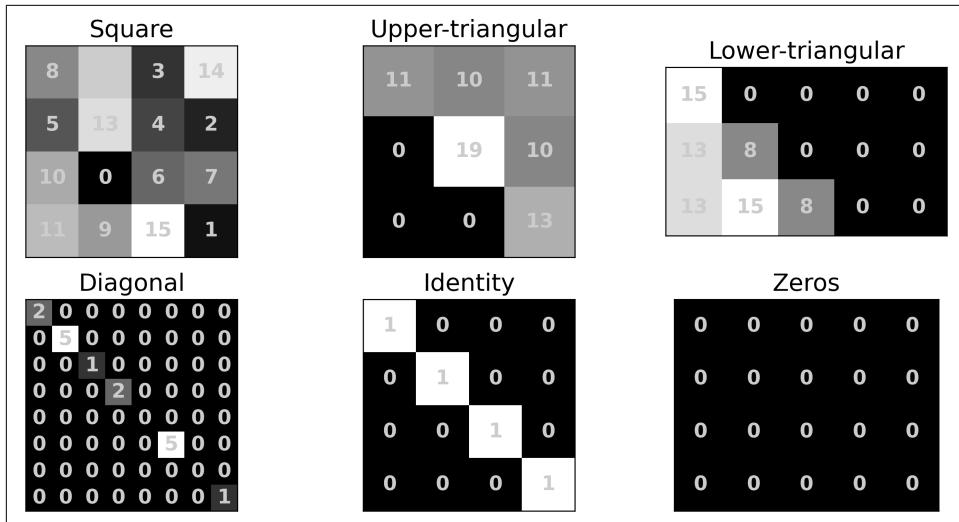


Figure 5-2. Some special matrices. Numbers and grayscale values indicate the matrix value at each element.

Matrix Math: Addition, Scalar Multiplication, Hadamard Multiplication

Mathematical operations on matrices fall into two categories: intuitive and unintuitive. In general, the intuitive operations can be expressed as element-wise procedures whereas the unintuitive operations take longer to explain and a bit of practice to understand. Let's start with the intuitive operations.

Addition and Subtraction

You add two matrices by adding their corresponding elements. Here's an example:

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 2 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 3 & 1 \\ -1 & -4 & 2 \end{bmatrix} = \begin{bmatrix} (2+0) & (3+3) & (4+1) \\ (1-1) & (2-4) & (4+2) \end{bmatrix} = \begin{bmatrix} 2 & 6 & 5 \\ 0 & -2 & 6 \end{bmatrix}$$

As you might guess from the example, matrix addition is defined only between two matrices of the same size.

“Shifting” a Matrix

As with vectors, it is not formally possible to add a scalar to a matrix, as in $\lambda + \mathbf{A}$. Python allows such an operation (e.g., `3+np.eye(2)`), which involves broadcast-adding the scalar to each element of the matrix. That is a convenient computation, but it is not formally a linear algebra operation.

But there is a linear-algebra way to add a scalar to a square matrix, and that is called *shifting* a matrix. It works by adding a constant value to the diagonal, which is implemented by adding a scalar multiplied identity matrix:

$$\mathbf{A} + \lambda \mathbf{I}$$

Here's a numerical example:

$$\begin{bmatrix} 4 & 5 & 1 \\ 0 & 1 & 11 \\ 4 & 9 & 7 \end{bmatrix} + 6 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 5 & 1 \\ 0 & 7 & 11 \\ 4 & 9 & 13 \end{bmatrix}$$

Shifting in Python is straightforward:

```
A = np.array([ [4,5,1],[0,1,11],[4,9,7] ])
s = 6
A + s # NOT shifting!
A + s*np.eye(len(A)) # shifting
```

Notice that only the diagonal elements change; the rest of the matrix is unadulterated by shifting. In practice, one shifts a relatively small amount to preserve as much information as possible in the matrix while benefiting from the effects of shifting, including increasing the numerical stability of the matrix (you'll learn later in the book why that happens).

Exactly how much to shift is a matter of ongoing research in multiple areas of machine learning, statistics, deep learning, control engineering, etc. For example, is shifting by $\lambda = 6$ a little or a lot? How about $\lambda = .001$? Obviously, these numbers are “big” or “small” relative to the numerical values in the matrix. Therefore, in practice, λ is usually set to be some fraction of a matrix-defined quantity such as the norm or the average of the eigenvalues. You'll get to explore this in later chapters.

“Shifting” a matrix has two primary (extremely important!) applications: it is the mechanism of finding the eigenvalues of a matrix, and it is the mechanism of regularizing matrices when fitting models to data.

Scalar and Hadamard Multiplications

These two types of multiplication work the same for matrices as they do for vectors, which is to say, element-wise.

Scalar-matrix multiplication means to multiply each element in the matrix by the same scalar. Here is an example using a matrix comprising letters instead of numbers:

$$\gamma \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} \gamma a & \gamma b \\ \gamma c & \gamma d \end{bmatrix}$$

Likewise, Hadamard multiplication involves multiplying two matrices element-wise (hence the alternative terminology *element-wise multiplication*). Here is an example:

$$\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \odot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 2a & 3b \\ 4c & 5d \end{bmatrix}$$

In NumPy, Hadamard multiplication can be implemented using the `np.multiply()` function. But it's often easier to implement using an asterisk between the two matrices: `A*B`. This can cause some confusion, because standard matrix multiplication (next section) is indicated using an `@` symbol. That's a subtle but important difference! (This will be particularly confusing for readers coming to Python from MATLAB, where `*` indicates matrix multiplication.)

```
A = np.random.randn(3,4)
B = np.random.randn(3,4)

A*B # Hadamard multiplication
np.multiply(A,B) # also Hadamard
A@B # NOT Hadamard!
```

Hadamard multiplication does have some applications in linear algebra, for example, when computing the matrix inverse. However, it is most often used in applications as a convenient way to store many individual multiplications. That's similar to how vector Hadamard multiplication is often used, as discussed in [Chapter 2](#).

Standard Matrix Multiplication

Now we get to the unintuitive way to multiply matrices. To be clear, standard matrix multiplication is not particularly difficult; it's just different from what you might expect. Rather than operating element-wise, standard matrix multiplication operates row/column-wise. In fact, standard matrix multiplication reduces to a systematic collection of dot products between rows of one matrix and columns of the other matrix. (This form of multiplication is formally simply called *matrix multiplication*;

I've added the term *standard* to help disambiguate from Hadamard and scalar multiplications.)

But before I get into the details of how to multiply two matrices, I will first explain how to determine whether two matrices can be multiplied. As you'll learn, two matrices can be multiplied only if their sizes are *concordant*.

Rules for Matrix Multiplication Validity

You know that matrix sizes are written out as $M \times N$ —rows by columns. Two matrices multiplying each other can have different sizes, so let's refer to the size of the second matrix as $N \times K$. When we write out the two multiplicand matrices with their sizes underneath, we can refer to the “inner” dimensions N and the “outer” dimensions M and K .

Here's the important point: *matrix multiplication is valid only when the “inner” dimensions match, and the size of the product matrix is defined by the “outer” dimensions*. See Figure 5-3.

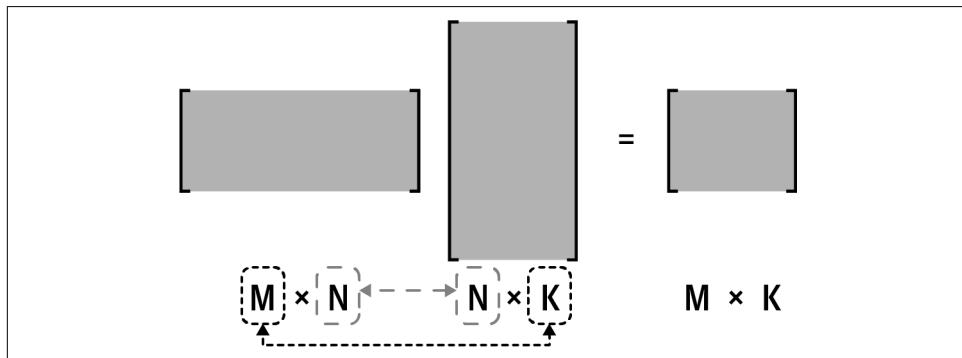


Figure 5-3. Matrix multiplication validity, visualized. Memorize this picture.

More formally, matrix multiplication is valid when the number of columns in the left matrix equals the number of rows in the right matrix, and the size of the product matrix is defined by the number of rows in the left matrix and the number of columns in the right matrix. I find the “inner/outer” rubric easier to remember.

You can already see that matrix multiplication does not obey the commutative law: \mathbf{AB} may be valid while \mathbf{BA} is invalid. Even if both multiplications are valid (for example, if both matrices are square), they may produce different results. That is, if $\mathbf{C} = \mathbf{AB}$ and $\mathbf{D} = \mathbf{BA}$, then in general $\mathbf{C} \neq \mathbf{D}$ (they are equal in some special cases, but we cannot generally assume equality).

Note the notation: Hadamard multiplication is indicated using a dotted-circle ($\mathbf{A} \odot \mathbf{B}$) whereas matrix multiplication is indicated as two matrices side-by-side without any symbol between them (\mathbf{AB}).

Now it's time to learn about the mechanics and interpretation of matrix multiplication.

Matrix Multiplication

The reason why matrix multiplication is valid only if the number of columns in the left matrix matches the number of rows in the right matrix is that the (i,j) th element in the product matrix is the dot product between the i th row of the left matrix and the j th column in the right matrix.

Equation 5-1 shows an example of matrix multiplication, using the same two matrices that we used for Hadamard multiplication. Make sure you understand how each element in the product matrix is computed as dot products of corresponding rows and columns of the left-hand side matrices.

Equation 5-1. Example of matrix multiplication. Parentheses added to facilitate visual grouping.

$$\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} (2a + 3c) & (2b + 3d) \\ (4a + 5c) & (4b + 5d) \end{bmatrix}$$

If you are struggling to remember how matrix multiplication works, **Figure 5-4** shows a mnemonic trick for drawing out the multiplication with your fingers.

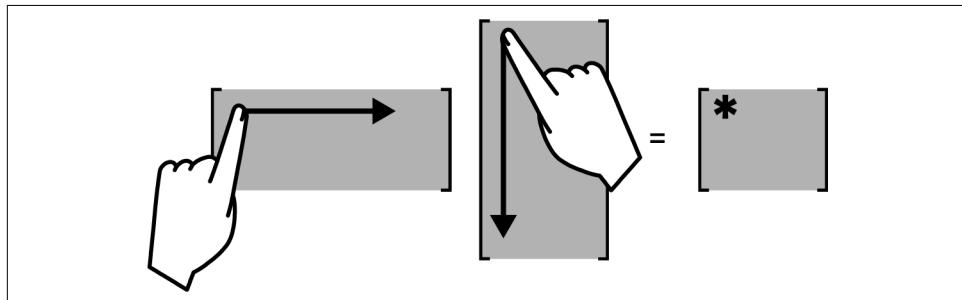


Figure 5-4. Finger movements for matrix multiplication

How do you interpret matrix multiplication? Remember that the dot product is a number that encodes the relationship between two vectors. So, the result of matrix multiplication is a matrix that stores all the pairwise linear relationships between rows of the left matrix and columns of the right matrix. That is a beautiful thing, and is the basis for computing covariance and correlation matrices, the general linear

model (used in statistics analyses including ANOVAs and regressions), singular-value decomposition, and countless other applications.

Matrix-Vector Multiplication

In a purely mechanical sense, matrix-vector multiplication is nothing special and does not deserve its own subsection: multiplying a matrix and a vector is simply matrix multiplication where one “matrix” is a vector.

But matrix-vector multiplication does have many applications in data science, machine learning, and computer graphics, so it’s worth spending some time on. Let’s start with the basics:

- A matrix can be right-multiplied by a column vector but not a row vector, and it can be left-multiplied by a row vector but not a column vector. In other words, \mathbf{Av} and $\mathbf{v}^T \mathbf{A}$ are valid, but \mathbf{Av}^T and \mathbf{vA} are invalid.

That is clear from inspecting matrix sizes: an $M \times N$ matrix can be premultiplied by a $1 \times M$ matrix (a.k.a. a row vector) or postmultiplied by an $N \times 1$ matrix (a.k.a. a column vector).

- The result of matrix-vector multiplication is always a vector, and the orientation of that vector depends on the orientation of the multiplicand vector: premultiplying a matrix by a row vector produces another row vector, while postmultiplying a matrix by a column vector produces another column vector. Again, this is obvious when you think about matrix sizes, but it’s worth pointing out.

Matrix-vector multiplication has several applications. In statistics, the model-predicted data values are obtained by multiplying the design matrix by the regression coefficients, which is written out as $\mathbf{X}\boldsymbol{\beta}$. In principal components analysis, a vector of “feature-importance” weights is identified that maximizes variance in dataset \mathbf{Y} , and is written out as $(\mathbf{Y}^T \mathbf{Y})\mathbf{v}$ (that feature-importance vector \mathbf{v} is called an eigenvector). In multivariate signal processing, a reduced-dimensional component is obtained by applying a spatial filter to multichannel time series data \mathbf{S} , and is written out as $\mathbf{w}^T \mathbf{S}$. In geometry and computer graphics, a set of image coordinates can be transformed using a mathematical transformation matrix, and is written out as $\mathbf{T}\mathbf{p}$, where \mathbf{T} is the transformation matrix and \mathbf{p} is the set of geometric coordinates.

There are so many more examples of how matrix-vector multiplication is used in applied linear algebra, and you will see several of these examples later in the book. Matrix-vector multiplication is also the basis for matrix spaces, which is an important topic that you’ll learn about later in the next chapter.

For now, I want to focus on two specific interpretations of matrix-vector multiplication: as a means to implement linear weighted combinations of vectors, and as the mechanism of implementing geometric transformations.

Linear weighted combinations

In the previous chapter, we calculated linear weighted combinations by having separate scalars and vectors, and then multiplying them individually. But you are now smarter than when you started the previous chapter, and so you are now ready to learn a better, more compact, and more scalable method for computing linear weighted combinations: put the individual vectors into a matrix, and put the weights into corresponding elements of a vector. Then multiply. Here's a numerical example:

$$4 \begin{bmatrix} 3 \\ 0 \\ 6 \end{bmatrix} + 3 \begin{bmatrix} 1 \\ 2 \\ 5 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 1 \\ 0 & 2 \\ 6 & 5 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Please take a moment to work through the multiplication, and make sure you understand how the linear weighted combination of the two vectors can be implemented as a matrix-vector multiplication. The key insight is that each element in the vector scalar multiplies the corresponding column in the matrix, and then the weighted column vectors are summed to obtain the product.

This example involved linear weighted combinations of column vectors; what would you change to compute linear weighted combinations of row vectors?²

Geometric transforms

When we think of a vector as a geometric line, then matrix-vector multiplication becomes a way of rotating and scaling that vector (remember that scalar-vector multiplication can scale but not rotate).

Let's start with a 2D case for easy visualization. Here are our matrix and vectors:

```
M = np.array([ [2,3],[2,1] ])
x = np.array([ [1,1.5] ]).T
Mx = M@x
```

Notice that I created x as a row vector and then transposed it into a column vector; that reduced the number of square brackets to type.

Graph A in [Figure 5-5](#) visualizes these two vectors. You can see that the matrix M both rotated and stretched the original vector. Let's try a different vector with the same matrix. Actually, just for fun, let's use the same vector elements but with swapped positions (thus, vector $v = [1.5, 1]$).

Now a strange thing happens in graph B ([Figure 5-5](#)): the matrix-vector product is no longer rotated into a different direction. The matrix still scaled the vector, but its

² Put the coefficients into a row vector and premultiply.

direction was preserved. In other words, the *matrix*-vector multiplication acted as if it were *scalar*-vector multiplication. That is not a random event: in fact, vector v is an eigenvector of matrix M , and the amount by which M stretched v is its eigenvalue. That's such an incredibly important phenomenon that it deserves its own chapter ([Chapter 13](#)), but I just couldn't resist introducing you to the concept now.

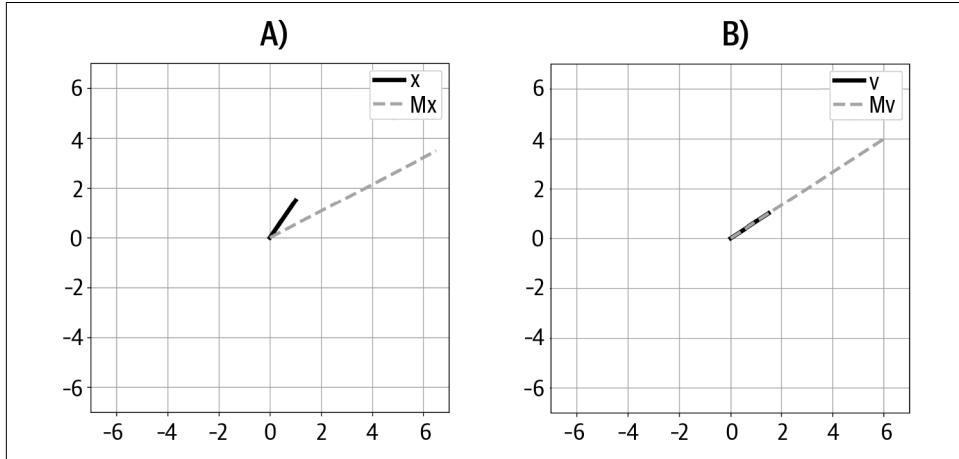


Figure 5-5. Examples of matrix-vector multiplication

Segues to advanced topics aside, the main point of these demonstrations is that one of the functions of matrix-vector multiplication is for a matrix to house a transformation that, when applied to a vector, can rotate and stretch that vector.

Matrix Operations: Transpose

You learned about the transpose operation on vectors in [Chapter 2](#). The principle is the same with matrices: swap the rows and columns. And just like with vectors, the transpose is indicated with a superscripted T (thus, C^T is the transpose of C). And double-transposing a matrix returns the original matrix ($C^{TT} = C$).

The formal mathematical definition of the transpose operation is printed in [Equation 5-2](#) (essentially repeated from the previous chapter), but I think it's just as easy to remember that *transposing swaps rows and columns*.

Equation 5-2. Definition of the transpose operation

$$a_{i,j}^T = a_{j,i}$$

Here's an example:

$$\begin{bmatrix} 3 & 0 & 4 \\ 9 & 8 & 3 \end{bmatrix}^T = \begin{bmatrix} 3 & 9 \\ 0 & 8 \\ 4 & 3 \end{bmatrix}$$

There are a few ways to transpose matrices in Python, using a function and a method acting on NumPy arrays:

```
A = np.array([ [3,4,5],[1,2,3] ])
A_T1 = A.T # as method
A_T2 = np.transpose(A) # as function
```

The matrix in this example uses a 2D NumPy array; what do you think will happen if you apply the transpose method to a vector encoded as a 1D array? Try it and find out!³

Dot and Outer Product Notation

Now that you know about the transpose operation and about the rules for matrix multiplication validity, we can return to the notation of the vector dot product. For two-column vectors of $M \times 1$, transposing the first vector and not the second gives two “matrices” of sizes $1 \times M$ and $M \times 1$. The “inner” dimensions match and the “outer” dimensions tell us that the product will be 1×1 , a.k.a. a scalar. That is the reason why the dot product is indicated as $\mathbf{a}^T \mathbf{b}$.

Same reasoning for the outer product: multiplying a column vector by a row vector has sizes $M \times 1$ and $1 \times N$. The “inner” dimensions match, and the size of the result will be $M \times N$.

Matrix Operations: LIVE EVIL (Order of Operations)

LIVE EVIL is a palindrome (a palindrome is a word or phrase that is spelled the same forwards and backwards) and a cute mnemonic for remembering how transposing affects the order of multiplied matrices. Basically, the rule is that the transpose of multiplied matrices is the same as the individual matrices transposed and multiplied, but reversed in order. In [Equation 5-3](#), L, I, V, and E are all matrices, and you can assume that their sizes match to make multiplication valid.

Equation 5-3. Example of the LIVE EVIL rule

$$(\mathbf{L}\mathbf{I}\mathbf{V}\mathbf{E})^T = \mathbf{E}^T \mathbf{V}^T \mathbf{I}^T \mathbf{L}^T$$

³ Nothing. NumPy will return the same 1D array without altering it, and without giving a warning or error.

Needless to say, this rule applies for multiplying any number of matrices, not just four, and not just with these “randomly selected” letters.

This does seem like a strange rule, but it’s the only way to make transposing multiplied matrices work. You’ll have the opportunity to test this yourself in [Exercise 5-7](#) at the end of this chapter. If you like, you may skip to that exercise before moving on.

Symmetric Matrices

Symmetric matrices have lots of special properties that make them great to work with. They also tend to be numerically stable and thus convenient for computer algorithms. You’ll learn about the special properties of symmetric matrices as you work through this book; here I will focus on what symmetric matrices are and how to create them from nonsymmetric matrices.

What does it mean for a matrix to be symmetric? It means that the corresponding rows and columns are equal. And that means that when you swap the rows and columns, nothing happens to the matrix. And that in turn means that a *symmetric matrix equals its transpose*. In math terms, a matrix \mathbf{A} is symmetric if $\mathbf{A}^T = \mathbf{A}$.

Check out the symmetric matrix in [Equation 5-4](#).

Equation 5-4. A symmetric matrix; note that each row equals its corresponding column

$$\begin{bmatrix} a & e & f & g \\ e & b & h & i \\ f & h & c & j \\ g & i & j & d \end{bmatrix}$$

Can a nonsquare matrix be symmetric? Nope! The reason is that if a matrix is of size $M \times N$, then its transpose is of size $N \times M$. Those two matrices could not be equal unless $M = N$, which means the matrix is square.

Creating Symmetric Matrices from Nonsymmetric Matrices

This may be surprising at first, but multiplying *any* matrix—even a nonsquare and nonsymmetric matrix—by its transpose will produce a square symmetric matrix. In other words, $\mathbf{A}^T \mathbf{A}$ is square symmetric, as is $\mathbf{A} \mathbf{A}^T$. (If you lack the time, patience, or keyboard skills to format the superscripted T , you can write $\mathbf{A} \mathbf{A}^T$ and $\mathbf{A}^T \mathbf{A}$ or $\mathbf{A}' \mathbf{A}$ and $\mathbf{A} \mathbf{A}'$.)

Let’s prove this claim rigorously before seeing an example. On the one hand, we don’t actually need to prove separately that $\mathbf{A}^T \mathbf{A}$ is square *and* symmetric, because the latter

implies the former. But proving squareness is straightforward and a good exercise in linear algebra proofs (which tend to be shorter and easier than, e.g., calculus proofs).

The proof is obtained simply by considering the matrix sizes: if \mathbf{A} is $M \times N$, then $\mathbf{A}^T \mathbf{A}$ is $(N \times M)(M \times N)$, which means the product matrix is of size $N \times N$. You can work through the same logic for $\mathbf{A} \mathbf{A}^T$.

Now to prove symmetry. Recall that the definition of a symmetric matrix is one that equals its transpose. So let's transpose $\mathbf{A}^T \mathbf{A}$, do some algebra, and see what happens. Make sure you can follow each step here; the proof relies on the LIVE EVIL rule:

$$(\mathbf{A}^T \mathbf{A})^T = \mathbf{A}^T \mathbf{A}^{TT} = \mathbf{A}^T \mathbf{A}$$

Taking the first and final terms, we get $(\mathbf{A}^T \mathbf{A})^T = (\mathbf{A}^T \mathbf{A})$. The matrix equals its transpose, hence it is symmetric.

Now repeat the proof on your own using $\mathbf{A} \mathbf{A}^T$. Spoiler alert! You'll come to the same conclusion. But writing out the proof will help you internalize the concept.

So $\mathbf{A} \mathbf{A}^T$ and $\mathbf{A}^T \mathbf{A}$ are both square symmetric. But they are not the same matrix! In fact, if \mathbf{A} is nonsquare, then the two matrix products are not even the same size.

$\mathbf{A}^T \mathbf{A}$ is called the *multiplicative method* for creating symmetric matrices. There is also an *additive method*, which is valid when the matrix is square but nonsymmetric. This method has some interesting properties but doesn't have a lot of application value, so I won't focus on it. [Exercise 5-9](#) walks you through the algorithm; if you're up for a challenge, you can try to discover that algorithm on your own before looking at the exercise.

Summary

This chapter is the first of a three-chapter series on matrices. Here you learned the groundwork from which all matrix operations are based. In summary:

- Matrices are spreadsheets of numbers. In different applications, it is useful to conceptualize them as a set of column vectors, a set of row vectors, or an arrangement of individual values. Regardless, visualizing matrices as images is often insightful, or at least pleasant to look at.
- There are several categories of special matrices. Being familiar with the properties of the types of matrices will help you understand matrix equations and advanced applications.

- Some arithmetic operations work element-wise, like addition, scalar multiplication, and Hadamard multiplication.
- “Shifting” a matrix means adding a constant to the diagonal elements (without changing the off-diagonal elements). Shifting has several applications in machine learning, primarily for finding eigenvalues and regularizing statistical models.
- Matrix multiplication involves dot products between rows of the left matrix and columns of the right matrix. The product matrix is an organized collection of mappings between row-column pairs. Memorize the rule for matrix multiplication validity: $(M \times N)(N \times K) = (M \times K)$.
- LIVE EVIL:⁴ The transpose of multiplied matrices equals the individual matrices transposed and multiplied with their order reversed.
- Symmetric matrices are mirrored across the diagonal, which means that each row equals its corresponding columns, and are defined as $\mathbf{A} = \mathbf{A}^T$. Symmetric matrices have many interesting and useful properties that make them great to work with in applications.
- You can create a symmetric matrix from any matrix by multiplying that matrix by its transpose. The resulting matrix $\mathbf{A}^T \mathbf{A}$ is central to statistical models and the singular value decomposition.

Code Exercises

Exercise 5-1.

This exercise will help you gain familiarity with indexing matrix elements. Create a 3×4 matrix using `np.arange(12).reshape(3,4)`. Then write Python code to extract the element in the second row, fourth column. Use softcoding so that you can select different row/column indices. Print out a message like the following:

`The matrix element at index (2,4) is 7.`

Exercise 5-2.

This and the following exercise focus on slicing matrices to obtain submatrices. Start by creating matrix **C** in [Figure 5-6](#), and use Python slicing to extract the submatrix comprising the first five rows and five columns. Let’s call this matrix **C**₁. Try to reproduce [Figure 5-6](#), but if you are struggling with the Python visualization coding, then just focus on extracting the submatrix correctly.

⁴ LIVE EVIL is a cute mnemonic, not a recommendation for how to behave in society!

Original matrix										Submatrix				
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4
10	11	12	13	14	15	16	17	18	19	10	11	12	13	14
20	21	22	23	24	25	26	27	28	29	20	21	22	23	24
30	31	32	33	34	35	36	37	38	39	30	31	32	33	34
40	41	42	43	44	45	46	47	48	49	40	41	42	43	44
50	51	52	53	54	55	56	57	58	59	50	51	52	53	54
60	61	62	63	64	65	66	67	68	69	60	61	62	63	64
70	71	72	73	74	75	76	77	78	79	70	71	72	73	74
80	81	82	83	84	85	86	87	88	89	80	81	82	83	84
90	91	92	93	94	95	96	97	98	99	90	91	92	93	94

Figure 5-6. Visualization of Exercise 5-2

Exercise 5-3.

Expand this code to extract the other four 5×5 blocks. Then create a new matrix with those blocks reorganized according to Figure 5-7.

Original matrix										Block-shifted										
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	
10	11	12	13	14	15	16	17	18	19	55	56	57	58	59	50	51	52	53	54	
20	21	22	23	24	25	26	27	28	29	65	66	67	68	69	60	61	62	63	64	
30	31	32	33	34	35	36	37	38	39	75	76	77			70	71	72	73	74	
40	41	42	43	44	45	46	47	48	49	85	86	87	88	89		80	81	82	83	84
50	51	52	53	54	55	56	57	58	59	95	96	97	98	99	90	91	92	93	94	
60	61	62	63	64	65	66	67	68	69	5	6	7	8	9	0	1	2	3	4	
70	71	72	73	74	75	76	77			15	16	17	18	19	10	11	12	13	14	
80	81	82	83	84	85	86	87	88	89	25	26	27	28	29	20	21	22	23	24	
90	91	92	93	94	95	96	97	98	99	35	36	37	38	39	30	31	32	33	34	

Figure 5-7. Visualization of Exercise 5-3

Exercise 5-4.

Implement matrix addition element-wise using two `for` loops over rows and columns. What happens when you try to add two matrices with mismatching sizes? This

exercise will help you think about breaking down a matrix into rows, columns, and individual elements.

Exercise 5-5.

Matrix addition and scalar multiplication obey the mathematical laws of commutivity and distributivity. That means that the following equations give the same results (assume that the matrices \mathbf{A} and \mathbf{B} are the same size and that σ is some scalar):

$$\sigma(\mathbf{A} + \mathbf{B}) = \sigma\mathbf{A} + \sigma\mathbf{B} = \mathbf{A}\sigma + \mathbf{B}\sigma$$

Rather than proving this mathematically, you are going to demonstrate it through coding. In Python, create two random-numbers matrices of size 3×4 and a random scalar. Then implement the three expressions in the previous equation. You'll need to figure out a way to confirm that the three results are equal. Keep in mind that tiny computer precision errors in the range of 10^{-15} should be ignored.

Exercise 5-6.

Code matrix multiplication using `for` loops. Confirm your results against using the numpy `@` operator. This exercise will help you solidify your understanding of matrix multiplication, but in practice, it's always better to use `@` instead of writing out a double `for` loop.

Exercise 5-7.

Confirm the LIVE EVIL rule using the following five steps: (1) Create four matrices of random numbers, setting the sizes to be $\mathbf{L} \in \mathbb{R}^{2 \times 6}$, $\mathbf{I} \in \mathbb{R}^{6 \times 3}$, $\mathbf{V} \in \mathbb{R}^{3 \times 5}$, and $\mathbf{E} \in \mathbb{R}^{5 \times 2}$. (2) Multiply the four matrices and transpose the product. (3) Transpose each matrix individually and multiply them *without reversing their order*. (4) Transpose each matrix individually and multiply them reversing their order *according to the LIVE EVIL rule*. Check whether the result of step 2 matches the results of step 3 and step 4. (5) Repeat the previous steps but using all square matrices.

Exercise 5-8.

In this exercise, you will write a Python function that checks whether a matrix is symmetric. It should take a matrix as input, and should output a boolean `True` if the matrix is symmetric or `False` if the matrix is nonsymmetric. Keep in mind that small computer rounding/precision errors can make “equal” matrices appear unequal.

Therefore, you will need to test for equality with some reasonable tolerance. Test the function on symmetric and nonsymmetric matrices.

Exercise 5-9.

I mentioned that there is an additive method for creating a symmetric matrix from a nonsymmetric square matrix. The method is quite simple: average the matrix with its transpose. Implement this algorithm in Python and confirm that the result really is symmetric. (Hint: you can use the function you wrote in the previous exercise!)

Exercise 5-10.

Repeat the second part of [Exercise 3-3](#) (the two vectors in \mathbb{R}^3), but use matrix-vector multiplication instead of vector-scalar multiplication. That is, compute $\mathbf{A}\mathbf{s}$ instead of $\sigma_1\mathbf{v}_1 + \sigma_2\mathbf{v}_2$.

Exercise 5-11.

Diagonal matrices have many interesting properties that make them useful to work with. In this exercise, you will learn about two of those properties:

- Premultiplying by a diagonal matrix scales the rows of the right matrix by the corresponding diagonal elements.
- Postmultiplying by a diagonal matrix scales the columns of the left matrix by the corresponding diagonal elements.

This fact is used in several applications, including computing correlation matrices ([Chapter 7](#)) and diagonalizing a matrix ([Chapters 13](#) and [14](#)).

Let's explore an implication of this property. Start by creating three 4×4 matrices: a matrix of all ones (hint: `np.ones()`); a diagonal matrix where the diagonal elements are 1, 4, 9, and 16; and a diagonal matrix equal to the square root of the previous diagonal matrix.

Next, print out the pre- and postmultiplied ones matrix by the first diagonal matrix. You'll get the following results:

```
# Pre-multiply by diagonal:  
[[ 1.  1.  1.  1.]  
 [ 4.  4.  4.  4.]  
 [ 9.  9.  9.  9.]  
 [16. 16. 16. 16.]]  
  
# Post-multiply by diagonal:  
[[ 1.  4.  9.  16.]  
 [ 1.  4.  9.  16.]]
```

```
[ 1.  4.  9. 16.]  
[ 1.  4.  9. 16.]]
```

Finally, premultiply *and* postmultiply the ones matrix by the square root of the diagonal matrix. You'll get the following:

```
# Pre- and post-multiply by sqrt-diagonal:  
[[ 1.  2.  3.  4.]  
 [ 2.  4.  6.  8.]  
 [ 3.  6.  9. 12.]  
 [ 4.  8. 12. 16.]]
```

Notice that the rows *and* the columns are scaled such that the (i,j) th element in the matrix is multiplied by the product of the i th and j th diagonal elements. (In fact, we've created a multiplication table!)

Exercise 5-12.

Another fun fact: matrix multiplication is the same thing as Hadamard multiplication for two diagonal matrices. Figure out why this is using paper and pencil with two 3×3 diagonal matrices, and then illustrate it in Python code.

CHAPTER 6

Matrices, Part 2

Matrix multiplication is one of the most wonderful gifts that mathematicians have bestowed upon us. But to move from elementary to advanced linear algebra—and then to understand and develop data science algorithms—you need to do more than just multiply matrices.

We begin this chapter with discussions of matrix norms and matrix spaces. Matrix norms are essentially an extension of vector norms, and matrix spaces are essentially an extension of vector subspaces (which in turn are nothing more than linear weighted combinations). So you already have the necessary background knowledge for this chapter.

Concepts like linear independence, rank, and determinant will allow you to transition from understanding elementary concepts like transpose and multiplication to understanding advanced topics like inverse, eigenvalues, and singular values. And those advanced topics unlock the power of linear algebra for applications in data science. Therefore, this chapter is a waypoint in your transformation from linear algebra newbie to linear algebra knowbie.¹

Matrices seem like such simple things—just a spreadsheet of numbers. But you've already seen in the previous chapters that there is more to matrices than meets the eye. So, take a deep and calming breath and dive right in.

¹ The internet claims this is a real word; let's see if I can get it past the O'Reilly editors.

Matrix Norms

You learned about vector norms in [Chapter 2](#): the norm of a vector is its Euclidean geometric length, which is computed as the square root of the sum of the squared vector elements.

Matrix norms are a little more complicated. For one thing, there is no “*the* matrix norm”; there are multiple distinct norms that can be computed from a matrix. Matrix norms are somewhat similar to vector norms in that each norm provides one number that characterizes a matrix, and that the norm is indicated using double-vertical lines, as in the norm of matrix \mathbf{A} is indicated as $\| \mathbf{A} \|$.

But different matrix norms have different meanings. The myriad of matrix norms can be broadly divided into two families: element-wise (also sometimes called entry-wise) and induced. Element-wise norms are computed based on the individual elements of the matrix, and thus these norms can be interpreted to reflect the magnitudes of the elements in the matrix.

Induced norms can be interpreted in the following way: one of the functions of a matrix is to encode a transformation of a vector; the induced norm of a matrix is a measure of how much that transformation scales (stretches or shrinks) that vector. This interpretation will make more sense in [Chapter 7](#) when you learn about using matrices for geometric transformations, and in [Chapter 14](#) when you learn about the singular value decomposition.

In this chapter, I will introduce you to element-wise norms. I'll start with the Euclidean norm, which is actually a direct extension of the vector norm to matrices. The Euclidean norm is also called the *Frobenius norm*, and is computed as the square root of the sum of all matrix elements squared ([Equation 6-1](#)).

Equation 6-1. The Frobenius norm

$$\| \mathbf{A} \|_F = \sqrt{\sum_{i=1}^M \sum_{j=1}^N a_{ij}^2}$$

The indices i and j correspond to the M rows and N columns. Also note the subscript F indicating the Frobenius norm.

The Frobenius norm is also called the ℓ_2 norm (the ℓ is a fancy-looking letter L). And the ℓ_2 norm gets its name from the general formula for element-wise p -norms (notice that you get the Frobenius norm when $p = 2$):

$$\| \mathbf{A} \|_p = \left(\sum_{i=1}^M \sum_{j=1}^N |a_{ij}|^p \right)^{1/p}$$

Matrix norms have several applications in machine learning and statistical analysis. One of the important applications is in regularization, which aims to improve model fitting and increase generalization of models to unseen data (you'll see examples of this later in the book). The basic idea of regularization is to add a matrix norm as a cost function to a minimization algorithm. That norm will help prevent model parameters from becoming too large (ℓ_2 regularization, also called *ridge regression*) or encouraging sparse solutions (ℓ_1 regularization, also called *lasso regression*). In fact, modern deep learning architectures rely on matrix norms to achieve such impressive performance at solving computer vision problems.

Another application of the Frobenius norm is computing a measure of “matrix distance.” The distance between a matrix and itself is 0, and the distance between two distinct matrices increases as the numerical values in those matrices become increasingly dissimilar. Frobenius matrix distance is computed simply by replacing matrix \mathbf{A} with matrix $\mathbf{C} = \mathbf{A} - \mathbf{B}$ in [Equation 6-1](#).

This distance can be used as an optimization criterion in machine learning algorithms, for example, to reduce the data storage size of an image while minimizing the Frobenius distance between the reduced and original matrices. [Exercise 6-2](#) will guide you through a simple minimization example.

Matrix Trace and Frobenius Norm

The *trace* of a matrix is the sum of its diagonal elements, indicated as $tr(\mathbf{A})$, and exists only for square matrices. Both of the following matrices have the same trace (14):

$$\begin{bmatrix} 4 & 5 & 6 \\ 0 & 1 & 4 \\ 9 & 9 & 9 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 8 & 0 \\ 1 & 2 & 6 \end{bmatrix}$$

Trace has some interesting properties. For example, the trace of a matrix equals the sum of its eigenvalues and therefore is a measure of the “volume” of its eigenspace. Many properties of the trace are less relevant for data science applications, but here is one interesting exception:

$$\| \mathbf{A} \|_F = \sqrt{\sum_{i=1}^M \sum_{j=1}^N a_{ij}^2} = \sqrt{tr(\mathbf{A}^T \mathbf{A})}$$

In other words, the Frobenius norm can be calculated as the square root of the trace of the matrix times its transpose. The reason why this works is that each diagonal element of the matrix $\mathbf{A}^T \mathbf{A}$ is defined by the dot product of each row with itself.

Exercise 6-3 will help you explore the trace method of computing the Frobenius norm.

Matrix Spaces (Column, Row, Nulls)

The concept of *matrix spaces* is central to many topics in abstract and applied linear algebra. Fortunately, matrix spaces are conceptually straightforward, and are essentially just linear weighted combinations of different features of a matrix.

Column Space

Remember that a linear weighted combination of vectors involves scalar multiplying and summing a set of vectors. Two modifications to this concept will extend linear weighted combination to the column space of a matrix. First, we conceptualize a matrix as a set of column vectors. Second, we consider the infinity of real-valued scalars instead of working with a specific set of scalars. An infinite number of scalars gives an infinite number of ways to combine a set of vectors. That resulting infinite set of vectors is called the *column space of a matrix*.

Let's make this concrete with some numerical examples. We'll start simple—a matrix that has only one column (which is actually the same thing as a column vector). Its column space—all possible linear weighted combinations of that column—can be expressed like this:

$$C\begin{pmatrix} 1 \\ 3 \end{pmatrix} = \lambda \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \quad \lambda \in \mathbb{R}$$

The $C(\mathbf{A})$ indicates the column space of matrix \mathbf{A} , and the \in symbol means “is a member of” or “is contained in.” In this context, it means that λ can be any possible real-valued number.

What does this mathematical expression mean? It means that the column space is the set of all possible scaled versions of the column vector $[1 \ 3]$. Let's consider a few specific cases. Is the vector $[1 \ 3]$ in the column space? Yes, because you can express that vector as the matrix times $\lambda = 1$. How about $[-2 \ -6]$? Also yes, because you can express that vector as the matrix times $\lambda = -2$. How about $[1 \ 4]$? The answer here is no: vector $[1 \ 4]$ is *not* in the column space of the matrix, because there is simply no scalar that can multiply the matrix to produce that vector.

What does the column space look like? For a matrix with one column, the column space is a line that passes through the origin, in the direction of the column vector, and stretches out to infinity in both directions. (Technically, the line doesn't stretch out to literal infinity, because infinity is not a real number. But that line is arbitrarily long—much longer than our limited human minds can possibly fathom—so for all

intents and purposes, we can speak of that line as being infinitely long.) **Figure 6-1** shows a picture of the column space for this matrix.

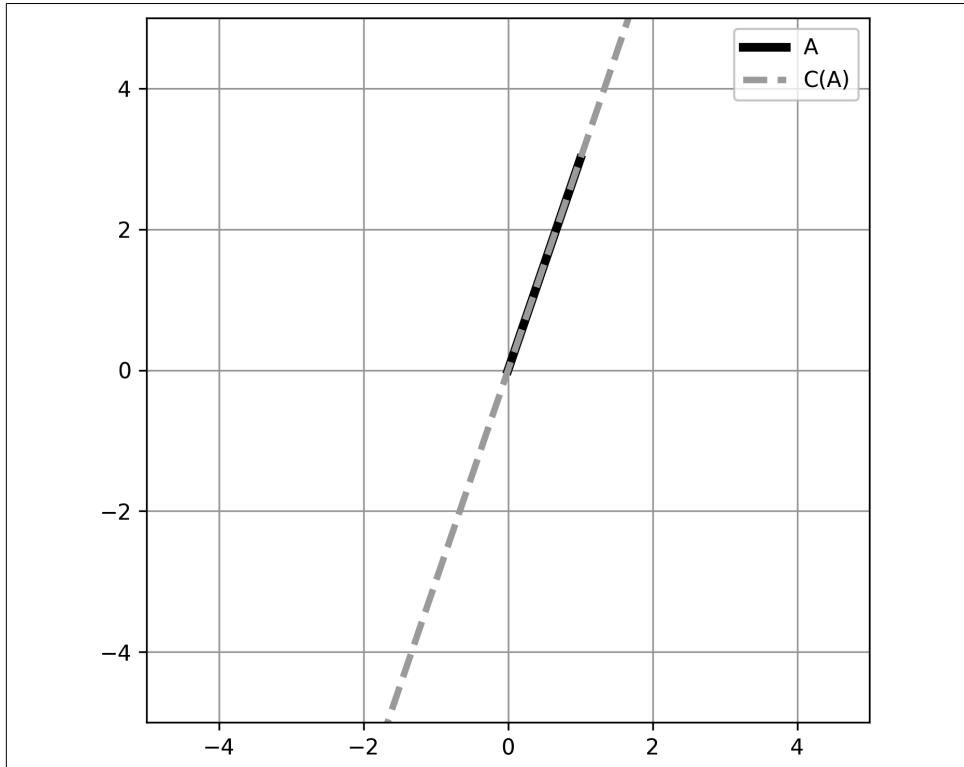


Figure 6-1. Visualization of the column space of a matrix with one column. This column space is a 1D subspace.

Now let's consider a matrix with more columns. We'll keep the column dimensionality to two so we can visualize it on a 2D graph. Here's our matrix and its column space:

$$C\left(\begin{bmatrix} 1 & 1 \\ 3 & 2 \end{bmatrix}\right) = \lambda_1 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + \lambda_2 \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \lambda \in \mathbb{R}$$

We have two columns, so we allow for two distinct λ s (they're both real-valued numbers but can be different from each other). Now the question is, what is the set of all vectors that can be reached by some linear combination of these two column vectors?

The answer is: all vectors in \mathbb{R}^2 . For example, the vector $[-4 \ 3]$ can be obtained by scaling the two columns by, respectively, 11 and -15. How did I come up with those

scalar values? I used the least squares projection method, which you'll learn about in [Chapter 11](#). For now, you can focus on the concept that these two columns can be appropriately weighted to reach any point in \mathbb{R}^2 .

Graph A in [Figure 6-2](#) shows the two matrix columns. I didn't draw the column space of the matrix because it is the entirety of the axis.

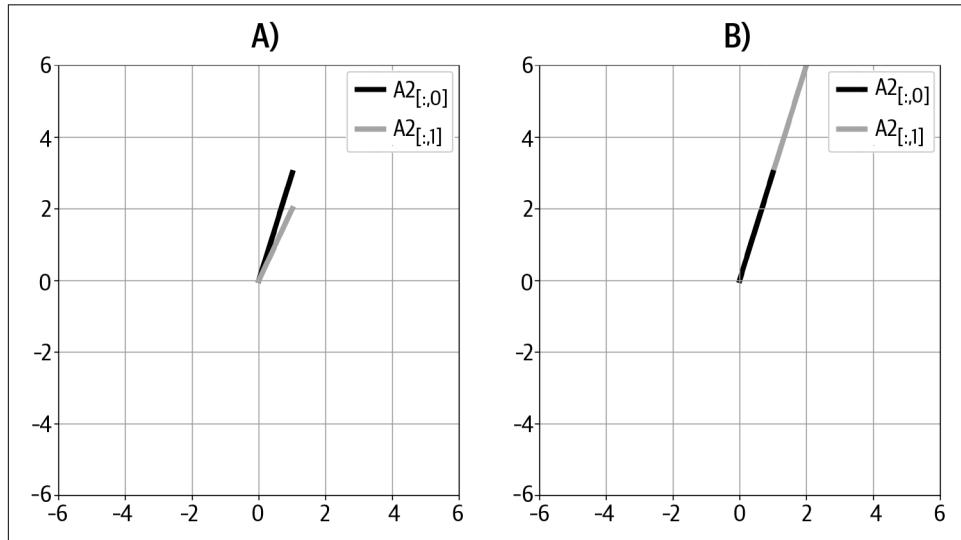


Figure 6-2. More examples of column spaces

One more example in \mathbb{R}^2 . Here's our new matrix for consideration:

$$C \begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix} = \lambda_1 \begin{pmatrix} 1 \\ 3 \end{pmatrix} + \lambda_2 \begin{pmatrix} 2 \\ 6 \end{pmatrix}, \quad \lambda \in \mathbb{R}$$

What is the dimensionality of its column space? Is it possible to reach any point in \mathbb{R}^2 by some linear weighted combination of the two columns?

The answer to the second question is no. And if you don't believe me, try to find a linear weighted combination of the two columns that produces vector $[3 \ 5]$. It is simply impossible. In fact, the two columns are collinear (graph B in [Figure 6-2](#)), because one is already a scaled version of the other. That means that the column space of this 2×2 matrix is still just a line—a 1D subspace.

The take-home message here is that having N columns in a matrix does not guarantee that the column space will be N -D. The dimensionality of the column space equals the number of columns only if the columns form a linearly independent set. (Remember

from [Chapter 3](#) that linear independence means a set of vectors in which no vector can be expressed as a linear weighted combination of other vectors in that set.)

One final example of column spaces, to see what happens when we move into 3D. Here's our matrix and its column space:

$$C \begin{pmatrix} 3 & 0 \\ 5 & 2 \\ 1 & 2 \end{pmatrix} = \lambda_1 \begin{pmatrix} 1 \\ 5 \\ 1 \end{pmatrix} + \lambda_2 \begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix}, \quad \lambda \in \mathbb{R}$$

Now there are two columns in \mathbb{R}^3 . Those two columns are linearly independent, meaning you cannot express one as a scaled version of the other. So the column space of this matrix is 2D, but it's a 2D plane that is embedded in \mathbb{R}^3 ([Figure 6-3](#)).

The column space of this matrix is an infinite 2D plane, but that plane is merely an infinitesimal slice of 3D. You can think of it like an infinitely thin piece of paper that spans the universe.

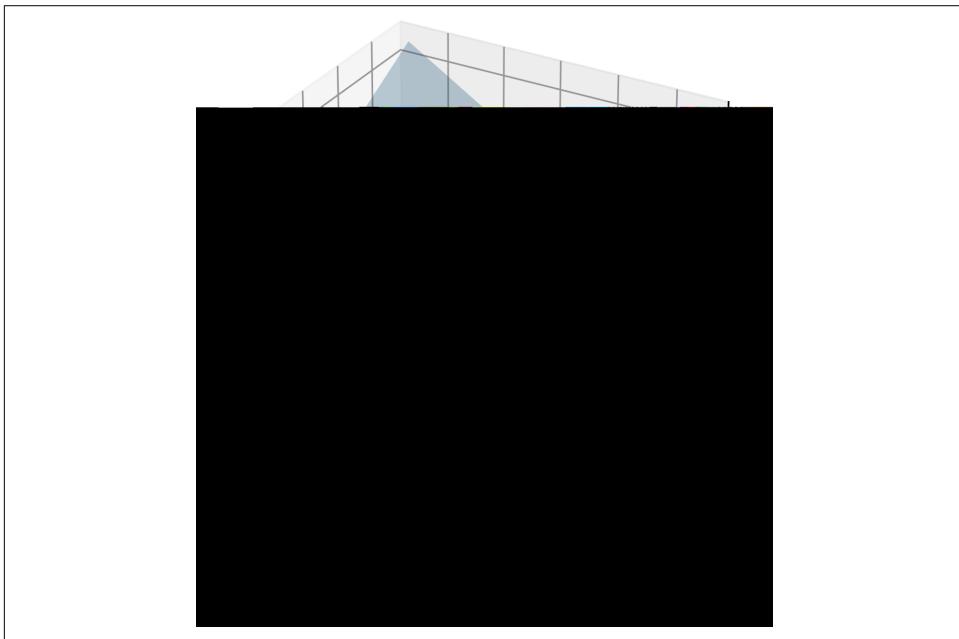


Figure 6-3. The 2D column space of a matrix embedded in 3D. The two thick lines depict the two columns of the matrix.

There are many vectors on that plane (i.e., many vectors that can be obtained as a linear combination of the two column vectors), but many *more* vectors that are not

Translating Formulas to Code

Translating mathematical equations into Python code is sometimes straightforward and sometimes difficult. But it is an important skill, and you will improve with practice. Let's start with a simple example in [Equation 16-1](#).

Equation 16-1. An equation

$$y = x^2$$

You might think that the following code would work:

```
y = x**2
```

But you'd get an error message (`NameError: name x is not defined`). The problem is that we are trying to use a variable `x` before defining it. So how do we define `x`? In fact, when you look at the mathematical equation, you defined `x` without really thinking about it: `x` goes from negative infinity to positive infinity. But you don't draw the function out that far—you would probably choose a limited range to draw that function, perhaps -4 to $+4$. That range is what we to specify in Python:

```
x = np.arange(-4,5)
y = x**2
```

[Figure 16-4](#) shows the plot of the function, created using `plt.plot(x,y, 's-')`.

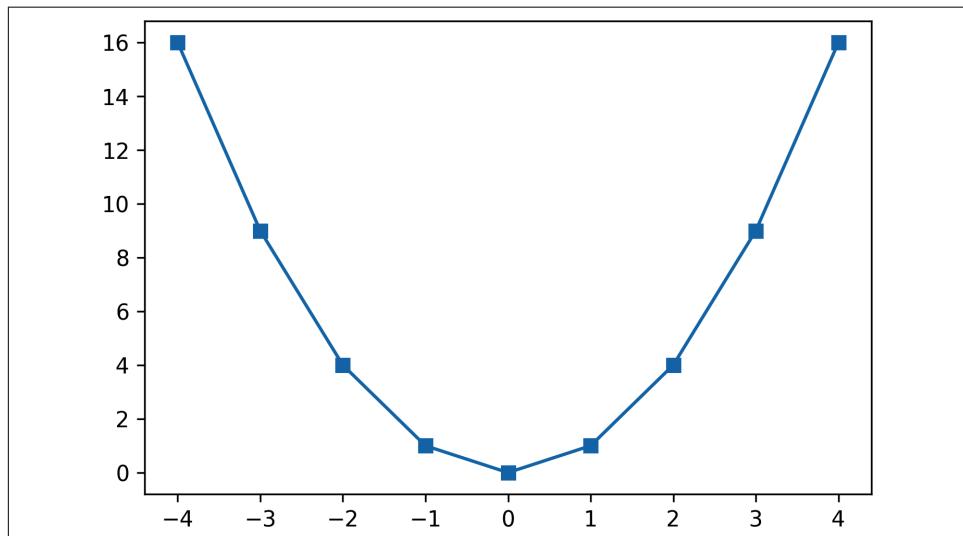


Figure 16-4. Visualizing data, part 3

It looks OK, but I think it's too choppy; I would like the line to look smoother. We can accomplish this by increasing the resolution, which means having more points between -4 and $+4$. I'll use the function `np.linspace()`, which takes three inputs: the start value, the stop value, and the number of points in between:

```
x = np.linspace(-4,4,42)
y = x**2
plt.plot(x,y, 's-')
```

Now we have 42 points linearly (evenly) spaced between -4 and $+4$. That makes the plot smoother (Figure 16-5). Note that `np.linspace` outputs a vector that ends at $+4$. This function has inclusive bounds. It is a little confusing to know which functions have inclusive and which have exclusive bounds. Don't worry, you'll get the hang of it.

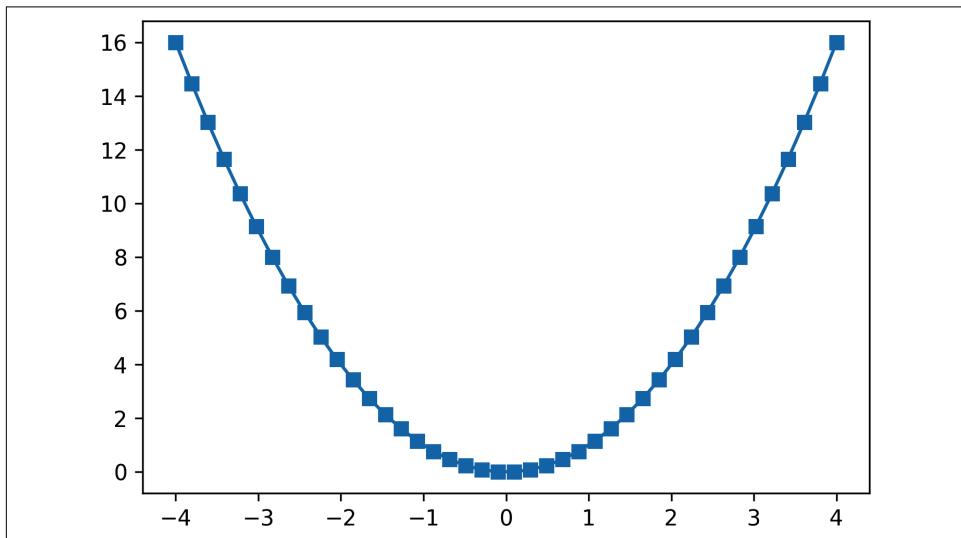


Figure 16-5. Visualizing data, part 4

Let's try another function-to-code translation. I will also use this opportunity to introduce you to a concept called *soft coding*, which means creating variables for parameters that you might want to change later.

Please translate the following mathematical function into code and generate a plot before looking at my code that follows:

$$f(x) = \frac{\alpha}{1 + e^{-\beta x}}$$

$$\alpha = 1.4$$

$$\beta = 2$$

This function is called a *sigmoid* and is used often in applied math, for example as a nonlinear activation function in deep learning models. α and β are parameters of the equation. Here I've set them to specific values. But once you have the code working, you can explore the effects of changing those parameters on the resulting graph. In fact, using code to understand math is, IMHO,³ the absolute best way to learn mathematics.

There are two ways you can code this function. One is to put the numerical values for α and β directly into the function. This is an example of *hard coding* because the parameter values are directly implemented in the function.

An alternative is to set Python variables to the two parameters, and then use those parameters when creating the mathematical function. This is *soft coding*, and it makes your code easier to read, modify, and debug:

```
x = np.linspace(-4,4,42)
alpha = 1.4
beta = 2

num = alpha # numerator
den = 1 + np.exp(-beta*x) # denominator
fx = num / den
plt.plot(x,fx, 's-');
```

Notice that I've split up the function creation into three lines of code that specify the numerator and denominator, and then their ratio. This makes your code cleaner and easier to read. Always strive to make your code easy to read, because it (1) reduces the risk of errors and (2) facilitates debugging.

Figure 16-6 shows the resulting sigmoid. Take a few moments to play with the code: change the `x` variable limits and resolution, change the `alpha` and `beta` parameter values, maybe even change the function itself. *Mathematics is beautiful, Python is your canvas, and code is your paintbrush!*

³ I'm told that this is millennial lingo for "in my humble opinion."

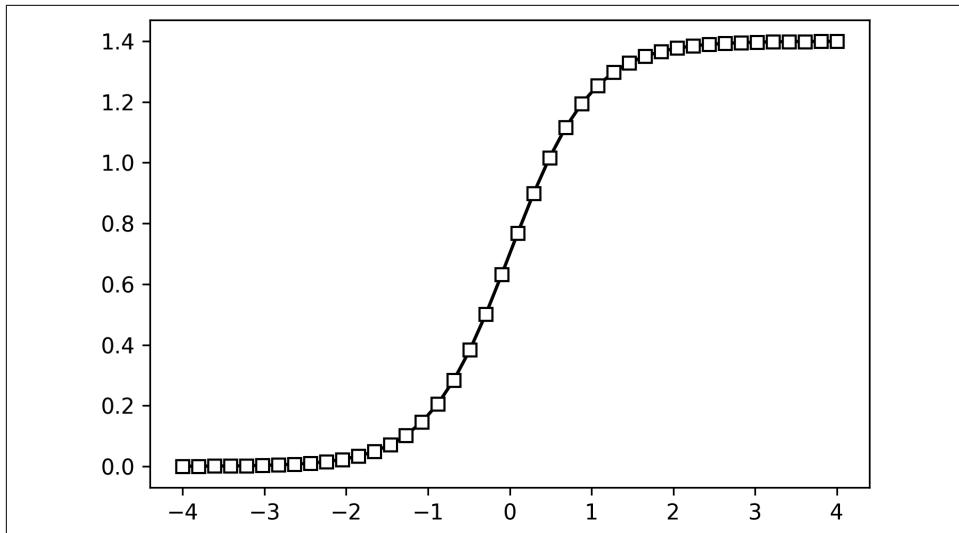


Figure 16-6. Visualizing data, part 5

Print Formatting and F-Strings

You already know how to print out variables using the `print()` function. But that's just for printing out one variable with no other text. F-strings give you more control over the output format. Observe:

```
var1 = 10.54
print(f'The variable value is {var1}, which makes me happy.')
>> The variable value is 10.54, which makes me happy.
```

Note the two key features of the f-string: the initial `f` before the first quote mark and the curly brackets `{}` encasing variable names that get replaced with variable values.

The next code block further highlights the flexibility of f-strings:

```
theList = ['Mike', 7]
print(f'{theList[0]} eats {theList[1]*100} grams of chocolate each day.')
>> Mike eats 700 grams of chocolate each day.
```

Two key points to learn from this example: (1) don't worry, I don't actually eat that much chocolate (well, not every day), and (2) you can use indexing and code inside the curly brackets, and Python will print out the result of the computation.

One final feature of f-string formatting:

```
pi = 22/7
print(f'{pi}, {pi:.3f}')
```

```
>> 3.142857142857143, 3.143
```

The key addition in that code is the `:.3f`, which controls the formatting of the output. This code tells Python to print out three numbers after the decimal point. See what happens when you change the 3 to another integer and what happens when you include an integer before the colon.

There are many other formatting options—and other ways to have flexible text outputs—but the basic implementation of f-strings is all you need to know for this book.

Control Flow

The power and flexibility of programming come from endowing your code with the ability to adapt its behavior depending on the state of certain variables or user inputs. Dynamism in code comes from *control flow* statements.

Comparators

Comparators are special characters that allow you to compare different values. The outcome of a comparator is a data type called a *Boolean*, which takes one of two values: `True` or `False`. Here are a few examples:

```
print( 4<5 ) # 1
print( 4>5 ) # 2
print( 4==5 ) # 3
```

The outputs of these lines are `True` for #1 and `False` for #2 and #3.

That third statement contains a double-equals sign. It's very different from a single equals sign, which you already know is used to assign values to a variable.

Two more comparators are `<=` (less than or equal to) and `>=` (greater than or equal to).

If Statements

If statements are intuitive because you use them all the time: *If I'm tired, then I will rest my eyes.*

The basic if statement has three parts: the `if` keyword, the *conditional statement*, and the *code content*. A conditional statement is a piece of code that evaluates to true or false, followed by a colon (`:`). If the conditional is true, then all the code beneath and indented is run; if the conditional is false, none of the indented code is run, and Python will continue running code that is not indented.

Here is an example:

```
var = 4
if var==4:
    print(f'{var} equals 4!')
    print("I'm outside the +for+ loop.")

>> 4 equals 4!
I'm outside the +for+ loop.
```

And here is another example:

```
var = 4
if var==5:
    print(f'{var} equals 5!')

print("I'm outside the +for+ loop.")

>> I'm outside the +for+ loop.
```

The first message is skipped because 4 does not equal 5; therefore, the conditional statement is false, and therefore Python ignores all of the indented code.

elif and else

Those two examples show the basic `if` statement form. `If` statements can include additional conditionals to increase the sophistication of the flow of information. Before reading my explanation of the following code and before typing this into Python on your computer, try to understand the code and make a prediction about what messages will print out:

```
var = 4

if var==5:
    print('var is 5') # code 1
elif var>5:
    print('var > 5') # code 2
else:
    print('var < 5') # code 3

print('Outside the if-elif-else')
```

When Python encounters a code statement like this, it proceeds from top to bottom. So, Python will start with the first conditional after the `if`. If that conditional is true, Python will run code 1 and then *skip all following conditionals*. That is, as soon as Python encounters a true conditional, the indented code is run and the `if` statement ends. It doesn't matter if subsequent conditionals are also true; Python won't check them or run their indented code.

If the first conditional is false, Python will proceed to the next conditional, which is `elif` (short for “else if”). Again, Python will run the subsequent indented code if the conditional is true, or it will skip the indented code if the conditional is false. This code example shows one `elif` statement, but you can have multiple such statements.

The final `else` statement has no conditional. This is like the “plan B” of the `if` statement: it is run if all the previous conditionals are false. If at least one of the conditionals is true, then the `else` code is not evaluated.

The output of this code example is:

```
var <5
Outside the if-elif-else
```

Multiple conditions

You can combine conditionals using `and` and `or`. It’s the coding analog to “If it rains *and* I need to walk, then I’ll bring an umbrella.” Here are a few examples:

```
if 4==4 and 4<10:
    print('Code example 1.')

if 4==5 and 4<10:
    print('Code example 2.')

if 4==5 or 4<10:
    print('Code example 3.')

>> Code example 1.
Code example 3.
```

The text `Code example 2` did not print because 4 does not equal 5. However, when using `or`, then *at least one* of the conditionals is true, so the subsequent code was run.

For Loops

Your Python skills are now sufficient to print out the numbers 1–10. You could use the following code:

```
print(1)
print(2)
print(3)
```

And so on. But that is not a scalable strategy—what if I asked you to print out numbers up to a million?

Repeating code in Python is done through *loops*. The most important kind of loop is called a *for loop*. To create a `for` loop, you specify an iterable (an *iterable* is a variable used to iterate over each element in that variable; a list can be used as an iterable) and

then any number of lines of code that should be run inside the `for` loop. I'll start with a very simple example, and then we'll build on that:

```
for i in range(0,10):
    print(i+1)
```

Running that code will output the numbers 0 through 10. The function `range()` creates an iterable object with its own data type called `range`, which is often used in `for` loops. The `range` variable contains integers from 0 through 9. (Exclusive upper bound! Also, if you start counting at 0, then you don't need the first input, so `range(10)` is the same as `range(0,10)`). But my instructions were to print the numbers 1 through 10, so we need to add 1 inside the `print` function. This example also highlights that you can use the iteration variable as a regular numerical variable.

`for` loops can iterate over other data types. Consider the following example:

```
theList = [ 2,'hello',np.linspace(0,1,14) ]
for item in theList:
    print(item)
```

Now we're iterating over a list, and the looping variable `item` is set to each item in the list at each iteration.

Nested Control Statements

Nesting flow-control statements inside other flow-control statements gives your code an additional layer of flexibility. Try to figure out what the code does and make a prediction for its output. Then type it into Python and test your hypothesis:

```
powers = [0]*10

for i in range(len(powers)):
    if i%2==0 and i>0:
        print(f'{i} is an even number')

    if i>4:
        powers[i] = i**2

print(powers)
```

I haven't yet taught you about the `%` operator. That's called the *modulus operator*, and it returns the remainder after division. So $7\%3 = 1$, because 3 goes into 7 twice with a remainder of 1. Likewise, $6\%2 = 0$ because 2 goes into 6 three times with a remainder of 0. In fact, $k\%2 = 0$ for *all* even numbers and $k\%2 = 1$ for *all* odd numbers. Therefore, a statement like `i%2==0` is a way to test whether the numeric variable `i` is even or odd.

Measuring Computation Time

When writing and evaluating code, you will often want to know how long the computer takes to run certain pieces of code. There are several ways to measure elapsed time in Python; one simple method is shown here, using the `time` library:

```
import time

clockStart = time.time()
# some code here...
compTime = time.time() - clockStart
```

The idea is to query the operating system's local time twice (this is the output of the function `time.time()`): once before running some code or functions, and once after running the code. The difference in clock times is the computation time. The result is the elapsed time in seconds. It's often handy to multiply the result by 1,000 to print the results in milliseconds (ms).

Getting Help and Learning More

I'm sure you've heard the phrase "Math is not a spectator sport." Same goes for coding: the only way to learn to code is by coding. You will make lots of mistakes, get frustrated because you can't figure out how to get Python to do what you want, see lots of errors and warning messages that you can't decipher, and just get really irritated at the universe and everything in it. (Yeah, you know the feeling I'm referring to.)

What to Do When Things Go Awry

Please allow me the indulgence to tell a joke: four engineers get into a car, but the car won't start. The mechanical engineer says, "It's probably a problem with the timing belt." The chemical engineer says, "No, I think the problem is the gas/air mixture." The electrical engineer says, "It sounds to me like the spark plugs are faulty." Finally, the software engineer says, "Let's just get out of the car and get back in again."

The moral of the story is that when you encounter some unexplainable issues in your code, you can try restarting the *kernel*, which is the engine running Python. That won't fix coding errors, but it may resolve errors due to variables being overwritten or renamed, memory overloads, or system failures. In Jupyter notebooks, you can restart the kernel through the menu options. Be aware that restarting the kernel clears out all of the variables and environment settings. You may need to rerun the code from the beginning.

If the error persists, then search the internet for the error message, the name of the function you're using, or a brief description of the problem you're trying to solve. Python has a huge international community, and there is a multitude of online forums where people discuss and resolve Python coding issues and confusions.

Summary

Mastering a programming language like Python takes years of dedicated study and practice. Even achieving a good beginner's level takes weeks to months. I hope this chapter provided you with enough skills to complete this book. But as I wrote in [Chapter 1](#), if you find that you understand the math but struggle with the code, then you might want to put this book down, get some more Python training, and then come back.

On the other hand, you should also see this book as a way to improve your Python coding skills. So if you don't understand some code in the book, then learning linear algebra is the perfect excuse to learn more Python!

Index

A

adding vectors, 11
 geometry of vector addition, 12
 scalar-vector addition, 14
 scalar-vector addition and multiplication, 34
 in vector dot product, 19
additive method (creating symmetric matrices), 75
adjugate matrix, 135
Anaconda installer, 280
angle (or direction) of vectors, 10
arrays
 matrix as 2D NumPy array, 73
 NumPy, 289
 vectors as NumPy arrays, 9
averaging vectors, 15

B

back substitution, 166, 187
basis, 41-46
 importance in data science and machine learning, 42
basis sets, 42
 requirement of linear independence, 45
basis vectors, 223
 null space, 222
 orthogonal, 242
bike rentals, predicting (example) (see predicting bike rentals based on weather)
Boolean type, 297
broadcasting, 55
 outer product versus, 24
vector broadcasting in Python, 16

C

Cartesian basis sets, 42
centroids, 54
CGI movies and video games, graphics in, 119
characteristic polynomial of a matrix, 219
Cholesky decomposition, 232
cloud-located Python, 280
code examples from this book, 5
code exercises, 5
coding
 intuition from versus mathematical proofs, 4
 linear algebra applications in code, 3
coefficients, 34, 161
 finding set that minimizes squared errors, 182
 regression coefficients for regressors, 198
cofactors matrix, 135
Colab environment, 280
 using Jupyter Notebooks in, 5
 working with code files in, 281
column orientation (vectors), 7
comments, 285
comparators (Python), 297
complete or full QR decomposition, 151
complex-valued eigenvalues, 226
components, 256
computation time, measuring in Python, 301
condition number of a matrix, 140, 187, 248
constants, 160, 177
control flow in Python, 297-300
 comparators, 297
 for loops, 299
 if statements, 297

elif and else, 298
multiple conditions in, 299
nested control statements, 300
convolution (image), 121-123
correlation coefficients, 49
normalizations, 50
correlation versus cosine similarity, 51
cosine similarity, 51
covariance, 113
covariance and correlation matrices, 113-116
correlation matrix in bike rental prediction example, 195
covariance matrices
eigendecomposition of, using in PCA, 257
generalized eigendecomposition on in LDA, 261
cross product, 24

D

data animations, creating, 116-120
data compression, 217, 262
data types
in Python, 283
representing vectors in Python, 9
of variables storing vectors, 14
decompositions, 147
def keyword (Python), 287
definiteness, 230, 234
positive (semi)definite, 231
dependent variables, 176, 184
in bike rental prediction example, 194
design matrices
bike rental prediction example, 196
of polynomial regression, 201
reduced-rank, 199
shifting to transform reduced-rank to full-rank matrix, 199
determinant of an eigenvalue-shifted matrix set to zero, 219
diagonal matrices, 64, 220
computing inverse of, 133
singular values matrix, 242
diagonal of a matrix, storing eigenvalue in, 222
dimension reduction, 217
dimensionality (vectors), 7
in addition of vectors, 11
dimensions in Python, 10
in dot product, 19
direction (or angle) of vectors, 10

eigenvectors, 222, 224
distance, 55
dot product, 18
in correlation coefficients, 50
distributive nature of, 20
dividing by product of vector norms, 50
geometry of, 21
in matrix-vector multiplication, 73
notation for versus outer product, 24
for orthogonal matrices, 148
Pearson correlation and cosine similarity based on, 52
in time series filtering, 52
zero dot product of orthogonal vectors, 22

E

echelon form of a matrix, 163
preferred forms of, 164
economy or reduced QR decomposition, 151
eigendecomposition, 213-239
applications of eigendecomposition and SVD, 255-278
linear discriminant analysis, 260-262
principal components analysis, 255-259
computing for a matrix times its transpose, 246
diagonalizing a square matrix, 222-223
different interpretations of, 213
finding eigenvalues, 217-220
finding the eigenvectors, 220-222
generalized, 232
interpretations of eigenvalues and eigenvectors, 214-215
dimension reduction (data compression), 217
noise reduction, 216
statistics, principal components analysis, 215
myriad subtleties of, 233
quadratic form, definiteness, and eigenvalues, 228-232
of singular matrices, 227
singular value decomposition and, 241
special properties of symmetric matrices relating to, 224-227
eigenvalues, 214-215, 234
of dataset's covariance matrix, graph of, 216
eigenvalue equation, 214
for diagonalization of a matrix, 223

vector-scalar version, 244
finding, 217-220
from eigendecomposition of singular matrix, 227
noise reduction with, 216
real-valued, 226
signs of, determining sign of quadratic form, 230
eigenvectors, 214-215, 234
finding, 220-222
sign and scale indeterminacy of eigenvectors, 221
from eigendecomposition of singular matrix, 227
in null space of the matrix shifted by its eigenvalue, 218
orthogonal, 224
real-valued in symmetric matrices, 226
stored in matrix columns, not rows, 220
element-wise multiplication, 67
(see also Hadamard multiplication)
elif and else statements (Python), 298
equations, systems of, 159-163, 165, 171
converting equations into matrices, 160
solved by matrix inverse, 167
solving with Gauss-Jordan elimination, 167
working with matrix equations, 161
equations, translating to code in Python, 293-296
errors
in Python, 283
squared errors between predicted data and observed data, 181
Euclidean distance, 55

F

f-strings, 296
feature detection, time series filtering and, 52
filtering, 52
image, 120-123
float type, 283
for loops (Python), 299
formulas, translating to code in Python, 293-296
Frobenius norm, 200
full inverse, 130
full or complete QR decomposition, 151
full-rank matrices
no zero-valued eigenvalues, 228

square, computing inverse of, 134
functions (Python), 285-290
libraries of, 288
importing NumPy, 289
indexing and slicing in NumPy, 289
methods as functions, 286
writing your own, 287

G
Gauss-Jordan elimination, 166, 187
matrix inverse via, 167
Gaussian elimination, 165
Gaussian kernel (2D), 122
general linear models, 175-178, 188
setting up, 176-178
in simple example, 183-186
solving, 178-183
exactness of the solution, 179
geometric perspective on least squares, 180
why least squares works, 181
solving with multicollinearity, 199
terminology, 176
generalized eigendecomposition, 232
on covariance matrices in LDA, 261
geometric interpretation of matrix inverse, 141
geometric length (vectors), 17
geometric transforms, 71
using matrix-vector multiplication, 116-120
geometry of vectors, 10
addition and subtraction of vectors, 12
dot product, 21
eigenvectors, 214
vector-scalar multiplication, 15
GLMs (see general linear models)
Google Colab environment, 5, 280
working with code files in, 281
Gram-Schmidt procedure, 149
grid matrix, 135
grid search to find model parameters, 204-205
GS or G-S (see Gram-Schmidt procedure)

H

Hadamard multiplication, 23
of matrices, 67
hard coding, 295
Hermitian, 244
Hilbert matrix, its inverse, and their product, 140

Householder transformation, 149

I

identity matrices, 64
 constants vector in first column of, 168
 matrix inverse containing transform of
 matrix to, 129
 produced by reduced row echelon form, 166
 replacement in generalized eigendecompo-
 sition, 232
IDEs (interactive development environments),
 5, 280
 graphical displays in Python, 290
 indentation in functions, 287
if statements (Python), 297
 elif and else, 298
 multiple conditions in, 299
ill-conditioned matrices, 248
 handling, 249
image feature detection, 120-123
images
 compression, SVD and, 262
 larger matrices visualized as, 61
independence (see linear independence)
independent components analysis (ICA), 43
independent variables, 176, 184
 in bike rental prediction example, 194
indexing
 of lists and related data types in Python, 284
 mathematical versus programming conven-
 tions for, 19
 of matrices, 62
 in NumPy, 289
int type, 283
interactive development environments (see
 IDEs)
intercepts or offsets, 160, 177, 185
 adding intercept term to observed and pre-
 dicted data, 185
inverses (matrix), 129
 (see also matrix inverse)
 QR decomposition and, 154
 types of and conditions for invertibility, 130
iterables, 299

J

Jupyter Notebooks, 5

K

k-means clustering, 53-56
kernels
 in image filtering, 121
 in time series filtering, 52
keywords in Python, 287

L

LAPACK library, 188
LDA (see linear discriminant analysis)
Learning Python (Lutz), 279
least squares, 178, 188
 applications, 193-211
 grid search to find model parameters,
 204-205
 polynomial regression, 200
 predicting bike rentals based on weather,
 193-200
 geometric perspective on, 180
 solving via QR decomposition, 187
 why it works, 181
left singular vectors, 241
left-inverse, 130, 136, 178
 versus NumPy least squares solver, 179
length (vectors), 8
 terminology differences between mathemat-
 ics and Python, 17
letters in linear algebra, 23
libraries of Python functions, 288
linear algebra
 about, 1
 applications in code, 3
 prerequisites for learning, 2-4
linear dependence, 199
linear discriminant analysis (LDA), 260-262
linear independence, 35-38, 46
 determining in practice, 36
 guaranteeing uniqueness in basis sets, 45
 independence and the zeros vector, 38
 math of, 37
linear mixture, 34
 (see also linear weighted combination)
linear models, general (see general linear mod-
 els)
linear weighted combination, 34, 46
 combined with variance in PCA, 256
 in matrix-vector multiplication, 71
lists, 284
 indexing, 284

representing vectors in Python, 9
scalar multiplying a list, 14
LIVE EVIL order of operations acronym, 73
low-rank approximations, 245
via singular value decomposition, 262
lower triangular matrices, 64
lower upper decomposition (see LU decomposition)
LU decomposition, 159, 169-171, 171
ensuring uniqueness of, 169
row swaps via permutation matrices, 170

M

magnitude (vectors), 10
eigenvectors, 222, 224
unit vectors and, 17
mapping between vectors, 20
mathematical proofs versus intuition from coding, 4
mathematics
attitudes toward learning, 3
on the chalkboard versus implemented in code, 8
learning, mathematical proofs versus intuition from coding, 4
matplotlib (Python), 62, 290
matrices, 1, 75
about, 61
applications, 113-128
geometric transforms via matrix-vector multiplication, 116-120
image feature detection, 120-123
multivariate data covariance matrices, 113-116
converting equations into, 160
creating and visualizing in NumPy, 61-65
special matrices, 63
visualizing, indexing and slicing matrices, 61-63
math, 65-67
addition and subtraction of matrices, 65
scalar and Hadamard multiplication, 67
shifting a matrix, 66
operations
LIVE EVIL, order of operations acronym, 73
transpose, 72, 73
quadratic form of, 228-230
special, 75

standard multiplication, 67-72
matrix multiplication, 69
rules for matrix multiplication validity, 68
symmetric, 74
creating from nonsymmetric matrices, 74
transposing, 15
matrix equations, 129
for a covariance matrix, 114
working with, 161, 171
matrix inverse, 129-145, 187
computing, 131-138
for diagonal matrix, 133
inverse of 2 X 2 matrix, 131
one-sided inverses, 136
via Gauss-Jordan elimination, 167
geometric interpretation of, 141
inverse of an orthogonal matrix, 148
Moore-Penrose pseudoinverse, 138
numerical stability of the inverse, 139-141
QR decomposition and, 154
types of inverses and conditions for invertibility, 130
uniqueness of, 138
matrix rank (see rank)
matrix-vector multiplication, 70-72
geometric transforms, 71
geometric transforms via, 116-120
linear weighted combinations, 71
mean centering, 50
methods, 286
minimum norm solution (min-norm), 199
minors matrix, 134
Monte Carlo simulations, 232
Moore-Penrose pseudoinverse, 138, 199
singular value decomposition and, 249
MP pseudoinverse (see Moore-Penrose pseudoinverse)
multicollinearity, 199
multiplication
geometry of vector-scalar multiplication, 15
matrix multiplication and vector dot product, 20
methods other than dot product for vectors, 22
cross and triple products, 24
Hadamard multiplication, 22
outer product, 23

multiplying matrices, 76
scalar and Hadamard multiplication of matrices, 67
scalar-vector multiplication and addition, 34
standard matrix multiplication, 67-72
vector dot product, 19
vector-scalar, 13
multiplication operator in Python (*), 23
multiplying matrices, 67
multiplicative method (creating symmetric matrices), 75
multivariate data covariance matrices, 113-116

N

nested flow-control statements, 300
noise
 ill-conditioned matrix as amplification factor, 249
 using SVD for denoising, 263
noise reduction, 216
noninvertible matrices, 130
nonlinear independence, 38
nonsquare matrices, 74
 having one-sided inverse, 130
 Moore-Penrose pseudoinverse applied to, 250
 square matrices versus, 63
norm (vectors), 17
 ratio of two norms, 257
normalizations for correlation coefficient, 50
normalized quadratic form, 257
np.argmin function, 55
np.diag function, 64
np.dot function, 19, 24
np.eye function, 64
np.min function, 55
np.multiply function, 67
np.outer function, 24
np.random function, 63
np.sum function, 55
np.tril function, 64
np.triu function, 64
null space vectors (eigenvectors), 221
numerical stability of matrix inverse, 139-141
NumPy
 diagonalizing a matrix in, 223
 importing into Python, 289
 indexing and slicing in, 289
 polynomial regression functions, 203

svd function, 243

O

offsets, 160
one-sided inverse, 130
 computing, 136
ordered list of numbers, 10
 (see also vectors)
ordinary least squares (OLS), 198
orientation (vectors), 7
 in addition of vectors, 12
orientationless arrays, 9
orthogonal eigenvectors, 224
 orthogonality proof, 225
orthogonal matrices, 117, 147-149
 matrices of singular vectors, 242
 orthogonal columns and unit-norm columns, 147
permutation matrices, 170
transforming nonorthogonal matrices to
 using Gram-Schmidt, 149
 using QR decomposition, 150-154
orthogonal projection, 26
orthogonal vector decomposition, 13, 24-28
orthogonal vector projection, 181
orthogonal vectors, zero dot product, 22
outer product, 23
 versus broadcasting, 24
 in matrix-vector multiplication, 73

P

pandas dataframes, 196
parallel component (orthogonal vector decomposition), 27
Pearson correlation coefficient, 20
 versus cosine similarity, 51
 expressed in linear algebra terms, 51
 formula for, 50
permutation matrices, 149, 187
 row swaps via, 170
perpendicular component (orthogonal vector decomposition), 27
pivots, 163
 in Gauss-Jordan elimination, 166
polynomial regression, 200
positive (semi)definite, 231, 234
positive definite, 231
predicting bike rentals based on weather
 (example), 193-200

creating regression table using statsmodels, 198
multicollinearity, 199
predicted versus observed data, 197
regularization of the matrix, 199
prime factorization, 25
principal components analysis (PCA), 43, 215
using eigendecomposition and SVD, 255-259
math of PCA, 256
PCA via SVD, 259
proof that eigendecomposition solves PCA optimization goal, 258
steps in performing PCA, 259
print formatting and f-strings, 296
projecting out a data dimension, 216
proof by negation, 138
proportion of regularization, 200
pseudoinverse, 130
Moore-Penrose pseudoinverse, 138
pure rotation matrix, 116, 148
Python, 3
creating data animations in, 116-120
IDEs, 5
least squares solution via left-inverse translated into code, 178
SciPy library, function for LU decomposition, 169
singular value decomposition in, 243
sympy library, computation of RREF, 167
terminology differences from chalkboard
linear algebra, 17
tutorial, 279-302
control flow, 297-300
functions, 285-290
getting help and learning more, 301
IDEs, 280
measuring computation time, 301
print formatting and f-strings, 296
translating formulas to code, 293-296
use cases and alternatives, 279
using Python locally and online, 280
variables, 282-285
visualizations, 290-292
vectors in, 9

Q
QR decomposition, 149, 150-154
QR and inverses, 154

sizes of Q and R, 151
solving least squares via, 187
quadratic form of a matrix, 228-230
normalized quadratic form of data covariance matrix, 257

R

random numbers matrices, 63
LU decomposition applied to, 171
range type, 300
rank (matrices)
from eigendecomposition of singular matrix, 228
singular value decomposition and rank-1 layers of a matrix, 244-246
singular values and, 243
rank-deficient matrices, 131
real-valued eigenvalues, 226
rectangular matrices, 63
left-null space and null space, 243
reduced or economy QR decomposition, 151
reduced rank matrices, 131
eigendecomposition of, 228
not invertible, 133
reduced row echelon form (RREF), 164, 166
regression
creating regression table using statsmodels, 198
polynomial, 200
regressors, 195
regression coefficients for, 198
regularization, 199
right singular vectors, 241
right-inverse, 130
rotation matrices, 116
row orientation (vectors), 7
row reduction, 163-168, 171
Gauss-Jordan elimination, 166
Gaussian elimination, 165
goal of, 163
matrix inverse via Gauss-Jordan elimination, 167
row swaps, 170
row, column index (matrix elements), 16
RREF (reduced row echelon form), 166

S

scalar equations versus matrix equations, 161
scalar inverse, 131

scalar matrices, 218
scalar-matrix multiplication, 67
scalars
 negative, changing direction of vectors, 15
 scalar-vector addition, 14
 scalar-vector multiplication and addition, 34
 vector-scalar multiplication, 13
SciPy library (Python), 169
 computation of generalized eigendecomposition, 233
scree plots, 216
self-learners, how to use this book, 6
shape (vectors in Python), 8
shifting a matrix, 66, 76, 199
 by its eigenvalue, 218
sigmoid function, 295
sign and scale indeterminacy of eigenvectors, 221
similarity between vectors, 20
simultaneous diagonalization of two matrices, 232
singular matrices, 130, 218
 eigendecomposition of, 227
singular value decomposition, 241-254
 applications of eigendecomposition and SVD, 255-278
 principal components analysis via SVD, 259
 big picture of, 241-243
 important features, 242
 singular values and matrix rank, 243
computing from eigendecomposition, 246-247
 SVD of $\mathbf{A}\mathbf{A}^T$, 247
key points, 250
low-rank approximations via, 262
and the Moore-Penrose pseudoinverse, 249
in Python, 243
 and rank-1 layers of a matrix, 244-246
 using for denoising, 263
singular values, 241
 converting to variance, 247
 important properties of, 246
 and matrix rank, 243
singular vectors, 259, 268
 orthogonal nature of, 246
slicing (in NumPy), 289
slicing a matrix, 62
soft coding, 295
soft proofs, 4
span, 38
 (see also subspace and span)
sparse variables, 195
square matrices, 74
 definiteness, 230
 diagonalizing, 222-223
 eigendecomposition for, 214
 full rank, computing inverse of, 134
 full rank, unique LU decomposition for, 169
 versus nonsquare matrices, 63
squared errors between predicted data and observed data, 181
standard basis set, 42
standard matrix multiplication, 67
 (see also matrices)
standard position (vectors), 10
statistical models, 175, 176
 (see also general linear models)
 regularization, 199
statistics
 correlation in, 49
 general linear model, 70
 principal components analysis, 215
 shifting a matrix, 66
 solving inverse problems in, 25
 variance, 114
statsmodels library, using to create regression table, 198
str (string) type, 284
stretching eigenvector of the matrix, 214
submatrices, 62
subspace and span (vector), 38-41, 46
 basis and, 44
subtracting vectors, 11
 geometry of vector subtraction, 12
 in linear weighted combination, 34
SVD (see singular value decomposition)
symmetric matrices, 74
 creating from nonsymmetric, 74
 positive (semi)definiteness and, 232
 special properties relating to eigendecomposition, 224-227, 234
 orthogonal eigenvectors, 224
 real-valued eigenvalues, 226
sympy library (Python), 167

T

tall matrices, 63

computing left-inverse, 136
economy versus full QR decomposition, 151
teachers, how to use this book, 6
tensors, 123
time series filtering, 52
transforms
 geometric transforms in matrix-vector multiplication, 71
 matrix inverse undoing geometric transform, 141
transpose operations, 9, 15
 on matrices, 72
 order of operations with matrices, LIVE EVIL rule, 73
symmetric matrix equals its transpose, 74
transposing left matrix in covariance matrix, 115
triangular matrices, 64
 lower-triangular, upper-triangular (LU) decomposition, 169
triple product, 24
trivial solution, 14
type function, 284

U

uniqueness
 linear independence guaranteeing, 45
 of QR decomposition, 153
unit vectors, 17
 creating from nonunit vectors, 18
unit-norm columns (orthogonal matrices), 147
upper triangular matrices, 64, 153, 187
 transforming dense matrix into, using row reduction, 163

V

variables (Python), 282-285
 data types, 283
 indexing, 284
 naming, 284
variance, 114
 combined with linear weighted combination in PCA, 256
 converting singular values to, 247
vector sets, 33, 46
 linear independence, 36
vectors, 1

applications, 49-60
 correlation and cosine similarity, 49-52
 k-means clustering, 53-56
 time series filtering and feature detection, 52
creating and visualizing in NumPy, 7-10
dot product, 18
 distributive nature of, 20
 geometry of, 21
linear independence, 35-38
linear weighted combination, 34
magnitude and unit vectors, 17
matrix-vector multiplication, 70-72
multiplication methods other than dot product, 22
 cross and triple products, 24
 Hadamard multiplication, 22
 outer product, 23
multiplication of left singular vector by right singular vector, 245
operations on, 11-17
 adding two vectors, 11
 averaging vectors, 15
 geometry of vector addition and subtraction, 12
 geometry of vector-scalar multiplication, 15
 transpose operation, 15
vector broadcasting in Python, 16
vector-scalar multiplication, 13
subspace and span, 38-41
visualizations
 in Python, 290-292
 visualizing matrices, 61

W

weighted combination, 34
 (see also linear weighted combination)
wide matrices, 63
 right-inverse for wide full row-rank matrix, 130

Z

zeros matrix, 64
zeros vector, 14
 eigenvalue equation set to, 219
 linear independence and, 37, 38

About the Author

Mike X Cohen is an **associate professor of neuroscience** at the Donders Institute (Radboud University Medical Centre) in the Netherlands. He has over 20 years of experience teaching scientific coding, data analysis, statistics, and related topics, and has authored several **online courses** and textbooks. He has a suspiciously dry sense of humor and enjoys anything purple.

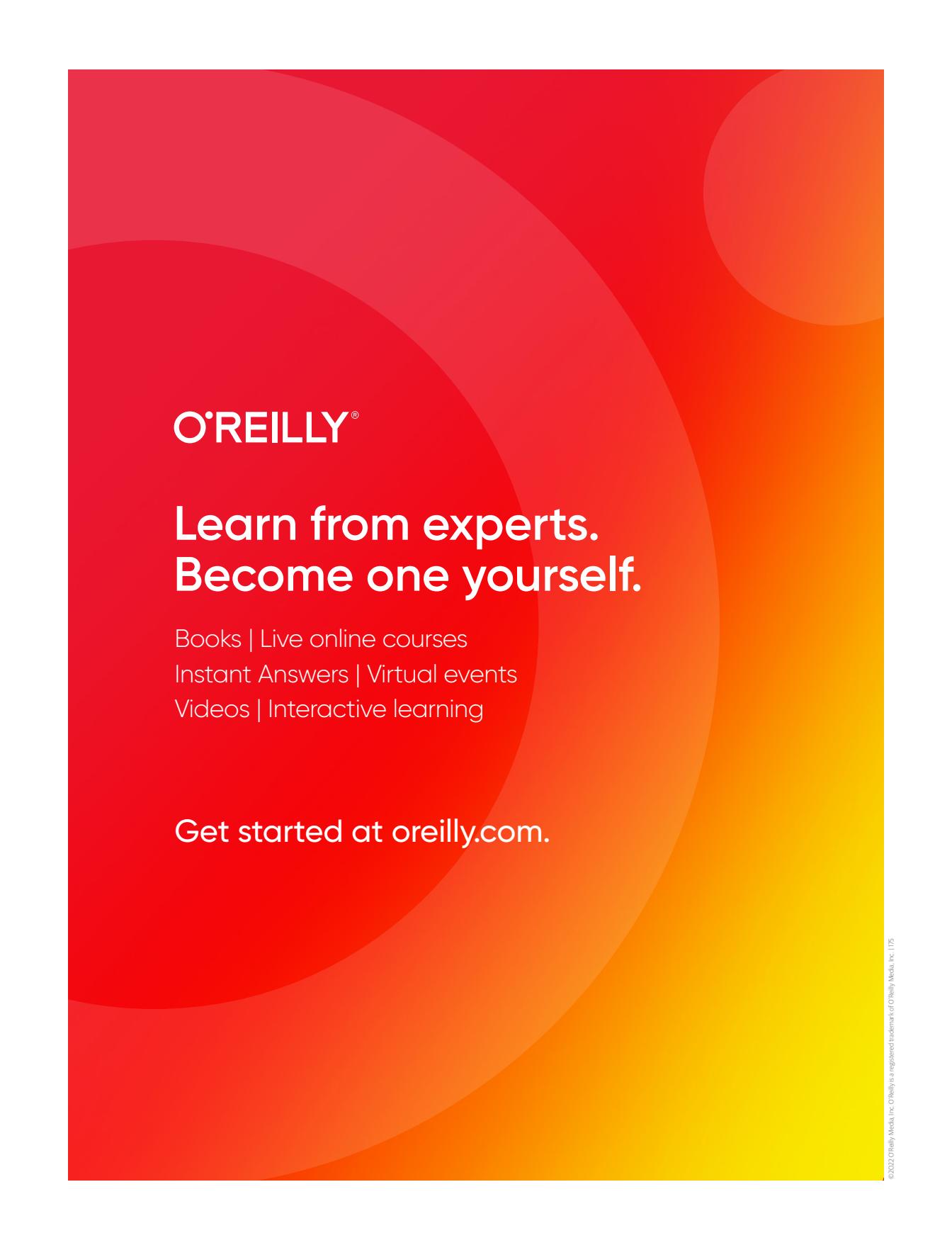
Colophon

The animal on the cover of *Practical Linear Algebra for Data Science* is a nyala antelope, also known as the lowland nyala or simply nyala (*Tragelaphus angasii*). Female and juvenile nyals are typically a light reddish-brown, while adult males have a dark brown or even grayish coat. Both males and females have white stripes along the body and white spots on the flank. Males have spiral-shaped horns that can grow up to 33 inches long, and their coats are much shaggier, with a long fringe hanging from their throats to their hindquarters and a mane of thick black hair along their spines. Females weigh about 130 pounds, while males can weigh as much as 275 pounds.

Nyals are native to the woodlands of southeastern Africa, with a range that includes Malawi, Mozambique, South Africa, Eswatini, Zambia, and Zimbabwe. They are shy creatures, preferring to graze in the early morning, late afternoon, or nighttime, and spending most of the hot part of the day resting among cover. Nyals form loose herds of up to ten animals, though older males are solitary. They are not territorial, though males will fight over dominance during mating.

Nyals are considered a species of least concern, though cattle grazing, agriculture, and habitat loss pose a threat to them. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Histoire Naturelle*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

**Learn from experts.
Become one yourself.**

Books | Live online courses
Instant Answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.