Coding the Matrix

Linear Algebra
through Applications to Computer Science

Edition 1

PHILIP N. KLEIN

Brown University

The companion website is at `codingthematrix.com`. There you will find, in digital form, the data, examples, and support code you need to solve the problems given in the book. Auto-grading will be provided for many of the problems.

Contents

Introduction

Tourist on Fifty-Seventh Street, Manhattan: “Pardon me—could you tell me how to get to Carnegie Hall?”

Native New Yorker: “Practice, practice!”

There’s a scene in the movie *The Matrix* in which Neo is strapped in a chair and Morpheus inserts into a machine what looks like a seventies-era videotape cartridge. As the tape plays, knowledge of how to fight streams into Neo’s brain. After a very short time, he has become an expert.

I would be delighted if I could strap my students into chairs and quickly stream knowledge of linear algebra into their brain, but brains don’t learn that way. The input device is rarely the bottleneck. Students need lots of practice—but what kind of practice?

No doubt students need to practice the basic numerical calculations, such as matrix-matrix multiplication, that underlie elementary linear algebra and that seem to fill all their time in traditional cookbook-style courses on linear algebra. No doubt students need to find proofs and counterexamples to exercise their understanding of the abstract concepts of which linear algebra is constructed.

However, they also need practice in using linear algebra to think about problems in other domains, and in actually *using* linear-algebra computations to address these problems. These are the skills they most need from a linear-algebra class when they go on to study other topics such as graphics and machine learning. This book is aimed at students of computer science; such students are best served by seeing applications from their field because these are the applications that will be most meaningful for them.

Moreover, a linear-algebra instructor whose pupils are students of computer science has a special advantage: her students are computationally sophisticated. They have a learning modality that most students don’t—they can learn through reading, writing, debugging, and using computer programs.

For example, there are several ways of writing a program for matrix-vector or matrix-matrix multiplication, each providing its own kernel of insight into the meaning of the operation—and the experience of writing such programs is more effective in conveying this meaning and cementing the relationships between the operations than spending the equivalent time carrying out hand calculations.

Computational sophistication also helps students in the more abstract, mathematical aspects of linear algebra. Acquaintance with object-oriented programming helps a student grasp the notion of a *field*—a set of values together with operations on them. Acquaintance with subtyping prepares a student to understand that some vector spaces are inner product spaces. Familiarity with loops or recursion helps a student understand procedural proofs, e.g. of the existence of a basis or an orthogonal basis.

Computational thinking is the term suggested by Jeannette Wing, former head of the National Science Foundation's directorate on Computer and Information Science and Engineering, to refer to the skills and concepts that a student of computer science can bring to bear. For this book, computational thinking is the road to mastering elementary linear algebra.

Companion website

The companion website is at `codingthematrix.com`. There you will find, in digital form, the data, examples, and support code you need to solve the problems given in the book.

Intended audience

This book is accessible to a student who is an experienced programmer. Most students who take my course have had at least two semesters of introductory computer science, or have previously learned programming on their own. In addition, it is desirable that the student has has some exposure (in prior semesters or concurrently) in proof techniques such as are studied in a Discrete Math course.

The student's prior programming experience can be in pretty much any programming language; this book uses Python, and the first two labs are devoted to bringing the student up to speed in Python programming. Moreover, the programs we write in this book are not particularly sophisticated. For example, we provide stencil code that obviates the need for the student to have studied object-oriented programming.

Some sections of the text, marked with *, provide supplementary mathematical material but are not crucial for the reader's understanding.

Labs

An important part of the book is the labs. For each chapter, there is a lab assignment in which the student is expected to write small programs and use some modules we provide, generally to carry out a task or series of tasks related to an application of the concepts recently covered or about to be covered. Doing the labs "keeps it real", grounding the student's study of linear algebra in getting something done, something meaningful in its own right but also illustrative of the concepts.

In my course, there is a lab section each week, a two-hour period in which the students carry out the lab assignment. Course staff are available during this period, not to supervise but to assist when necessary. The goal is to help the students move through the lab assignment efficiently, and to get them unstuck when they encounter obstacles. The students are expected to have prepared by reviewing the previous week's course material and reading through the lab assignment.

Most students experience the labs as the most fun part of the course—it is where they discover the power of the knowledge they are acquiring to help them accomplish something that has meaning in the world of computer science.

Programming language

The book uses Python, not a programming language with built-in support for vectors and matrices. This gives students the opportunity to build vectors and matrices out of the data structures that Python does provide. Using their own implementations of vector and matrix provides transparency. Python does provide complex numbers, sets, lists (sequences), and dictionaries (which we use for representing functions). In addition, Python provides *comprehensions*, expressions that create sets, lists, or dictionaries using a simple and powerful syntax that resembles the mathematical notation for defining sets. Using this syntax, many of the procedures we write require only a single line of code.

Students are not expected to know Python at the beginning; the first two labs form an introduction to Python, and the examples throughout the text reinforce the ideas.

Vector and Matrix representations

The traditional concrete representation for a vector is as a sequence of field elements. This book uses that representation but also uses another, especially in Python programs: a vector as a function mapping a finite set D to a field. Similarly, the traditional representation for a matrix is as a two-dimensional array or grid of field elements. We use this representation but also use another: a matrix as a function from the Cartesian product $R \times C$ of two finite sets to a field.

These more general representations allow the vectors and matrices to be more directly connected to the application. For example, it is traditional in information retrieval to represent a document as a vector in which, for each word, the vector specifies the number of occurences of the word in the document. In this book, we define such a vector as a function from the domain D of English words to the set of real numbers. Another example: when representing, say, a 1024×768 black-and-white image as a vector, we define the vector as a function from the domain $D = \{1, \ldots, 1024\} \times \{1, .., 768\}$ to the real numbers. The function specifies, for each pixel (i, j), the image intensity of that pixel.

From the programmer's perspective, it is certainly more convenient to directly index vectors by strings (in the case of words) or tuples (in the case of pixels). However, a more important advantage is this: having to choose a domain D for vectors gets us thinking about the application from the vector perspective.

Another advantage is analogous to that of type-checking in programs or unit-checking in physical calculations. For an $R \times C$ matrix A, the matrix-vector product $A\boldsymbol{x}$ is only legal if $\boldsymbol{x}$ is a C-vector; the matrix-matrix product AB is only legal if C is the set of row-labels of B. These constraints further reinforce the meanings of the operations.

Finally, allowing arbitrary finite sets (not just sequences of consecutive integers) to label the elements helps make it clear that the order of elements in a vector or matrix is not always (or even often) significant.

Fundamental Questions

The book is driven not just by applications but also by fundamental questions and computational problems that arise in studying these applications. Here are some of the fundamental questions:

- How can we tell whether a solution to a linear system is unique?
- How can we find the number of solutions to a linear system over $GF(2)$?
- How can we tell if a set $\mathcal{V}$ of vectors is equal to the span of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$?
- For a system of linear equations, what other linear equations are implied?
- How can we tell if a matrix is invertible?
- Can every vector space be represented as the solution set of a homogeneous linear system?

Fundamental Computational Problems

There are a few computational problems that are central to linear algebra. In the book, these arise in a variety of forms as we examine various applications, and we explore the connections between them. Here are examples:

- Find the solution to a matrix equation $M\boldsymbol{x} = \boldsymbol{b}$.
- Find the vector $\boldsymbol{x}$ minimizing the distance between $M\boldsymbol{x}$ and $\boldsymbol{b}$.
- Given vector $\boldsymbol{b}$, find the closest vector to $\boldsymbol{b}$ whose representation in a given basis is k-sparse.
- Find the solution to a matrix inequality $M\boldsymbol{x} \leq \boldsymbol{b}$.
- Given a matrix M, find the closest matrix to M whose rank is at most k.

Multiple representations

The most important theme of this book is the idea of multiple different representations for the same object. This theme should be familiar to computer scientists. In linear algebra, it arises again and again:

- Representing a vector space by generators or by homogeneous linear equations.
- Different bases for the same vector space.
- Different data structures used to represent a vector or a matrix.
- Different decompositions of a matrix.

Multiple fields

In order to illustrate the generality of the ideas of linear algebra and in order to address a broader range of applications, the book deals with three different fields: the real numbers, the complex numbers, and the finite field $GF(2)$. Most examples are over the real numbers because they are most familiar to the reader. The complex numbers serve as a warm-up for vectors since they can be used to represent points in the plane and transformations on these points. The complex numbers also come up in the discussion of the finite Fourier transform and in eigenvalues. The finite field $GF(2)$ comes up in many applications involving information, such as encryption, authentication, checksums, network coding, secret-sharing, and error-correcting codes.

The multiple codes help to illustrate the idea of an inner-product space. There is a very simple inner product for vectors over the reals; there is a slightly more complicated inner product for vectors over the complex numbers; and there is no inner-product for vectors over a finite field.

Acknowledgements

Thanks to my Brown University colleague John F. Hughes, computer scientist and recovering mathematician. I have learned a great deal from our conversations, and this book owes much to him.

Thanks to my Brown University colleague Dan Abramovich, mathematician, who has shared his insights on the tradeoffs between abstraction and simplicity of presentation.

Thanks to the students who have served as teaching assistants for the Brown University course on which this book is based and who have helped prepare some of the problems and labs, especially Sarah Meikeljohn, Shay Mozes, Olga Ohrimenko, Matthew Malin, Alexandra Berke, Anson Rosenthal, and Eli Fox-Epstein.

Thanks to Rosemary Simpson for her work on the index for this book.

Thanks to the creator of xkcd, Randall Munroe, for giving permission to include some of his work in this book.

Thanks, finally, to my family for their support and understanding.

Chapter 0

The Function (and other mathematical and computational preliminaries)

Later generations will regard
Mengenlehre [set theory] as a disease
from which one has recovered.
attributed to Poincáre

The basic mathematical concepts that inform our study of vectors and matrices are sets, sequences (lists), functions, and probability theory.

This chapter also includes an introduction to Python, the programming language we use to (i) model the mathematical objects of interest, (ii) write computational procedures, and (iii) carry out data analyses.

0.1 Set terminology and notation

The reader is likely to be familiar with the idea of a *set*, a collection of mathematical objects in which each object is considered to occur at most once. The objects belonging to a set are its *elements*. We use curly braces to indicate a set specified by explicitly enumerating its elements. For example, $\{\heartsuit, \spadesuit, \clubsuit, \diamondsuit\}$ is the set of suits in a traditional deck of cards. The order in which elements are listed is not significant; a set imposes no order among its elements.

The symbol $\in$ is used to indicate that an object belongs to a set (equivalently, that the set *contains* the object). For example, $\heartsuit \in \{\heartsuit, \spadesuit, \clubsuit, \diamondsuit\}$.

One set S_1 is *contained in* another set S_2 (written $S_1 \subseteq S_2$) if every element of S_1 belongs to S_2. Two sets are equal if they contain exactly the same elements. A convenient way to prove that two sets are equal consists of two steps: (1) prove the first set is contained in the second, and (2) prove the second is contained in the first.

A set can be infinite. In Chapter 1, we discuss the set $\mathbb{R}$, which consists of all real numbers, and the set $\mathbb{C}$, which consists of all complex numbers.

If a set S is not infinite, we use $|S|$ to denote its *cardinality*, the number of elements it contains. For example, the set of suits has cardinality 4.

0.2 Cartesian product

One from column A, one from column B.

The *Cartesian product* of two sets A and B is the set of all pairs (a, b) where $a \in A$ and $b \in B$.

Example 0.2.1: For the sets $A = \{1, 2, 3\}$ and $B = \{\heartsuit, \spadesuit, \clubsuit, \diamondsuit\}$, the Cartesian product is

$$\{(1, \heartsuit), (2, \heartsuit), (3, \heartsuit), (1, \spadesuit), (2, \spadesuit), (3, \spadesuit), (1, \clubsuit), (2, \clubsuit), (3, \clubsuit), (1, \diamondsuit), (2, \diamondsuit), (3, \diamondsuit)\}$$

Quiz 0.2.2: What is the cardinality of $A \times B$ in Example 0.2.1 (Page 2)?

Answer

$|A \times B| = 12$.

Proposition 0.2.3: For finite sets A and B, $|A \times B| = |A| \cdot |B|$.

Quiz 0.2.4: What is the cardinality of $\{1, 2, 3, \ldots, 10, J, Q, K\} \times \{\heartsuit, \spadesuit, \clubsuit, \diamondsuit\}$?

Answer

We use Proposition 0.2.3. The cardinality of the first set is 13, and the cardinality of the second set is 4, so the cardinality of the Cartesian product is $13 \cdot 4$, which is 52.

The Cartesian product is named for René Descartes, whom we shall discuss in Chapter 6.

0.3 The function

Mathematicians never die—they just lose function.

Loosely speaking, a function is a rule that, for each element in some set D of possible inputs, assigns a possible output. The output is said to be the *image* of the input under the function and the input is a *pre-image* of the output. The set D of possible inputs is called the *domain* of the function.

Formally, a *function* is a (possibly infinite) set of pairs (a, b) no two of which share the same first entry.

Example 0.3.1: The doubling function with domain $\{1, 2, 3, \ldots\}$ is

$$\{(1, 2), (2, 4), (3, 6), (4, 8), \ldots\}$$

The domain can itself consist of pairs of numbers.

Example 0.3.2: The multiplication function with domain $\{1, 2, 3, \ldots\} \times \{1, 2, 3, \ldots\}$ looks something like this:

$$\{((1, 1), 1), ((1, 2), 2), \ldots, ((2, 1), 2), ((2, 2), 4), ((2, 3), 6), \ldots\}$$

For a function named f, the image of q under f is denoted by $f(q)$. If $r = f(q)$, we say that q *maps to* r *under* f. The notation for "q maps to r" is $q \mapsto r$. (This notation omits specifying the function; it is useful when there is no ambiguity about which function is intended.)

It is convenient when specifying a function to specify a *co-domain* for the function. The co-domain is a set from which the function's output values are chosen. Note that one has some leeway in choosing the co-domain since not all of its members need be outputs.

The notation

$$f : D \longrightarrow F$$

means that f is a function whose domain is the set D and whose *co-domain* (the set of possible outputs) is the set F. (More briefly: "a function from D to F", or "a function that maps D to F.")

Example 0.3.3: Caesar was said to have used a cryptosystem in which each letter was replaced with the one three steps forward in the alphabet (wrapping around for X,Y, and Z).[a] Thus the plaintext MATRIX would be encrypted as the cyphertext PDWULA. The function that maps each plaintext letter to its cyphertext replacement could be written as

$$A \mapsto D, B \mapsto E, C \mapsto F, D \mapsto G, W \mapsto Z, X \mapsto A, Y \mapsto B, Z \mapsto C$$

This function's domain and co-domain are both the alphabet $\{A, B, \ldots, Z\}$.

[a] Some imaginary historians have conjectured that Caesar's assasination can be attributed to his use of such a weak cryptosystem.

Example 0.3.4: The cosine function, *cos*, maps from the set of real numbers (indicated by $\mathbb{R}$)