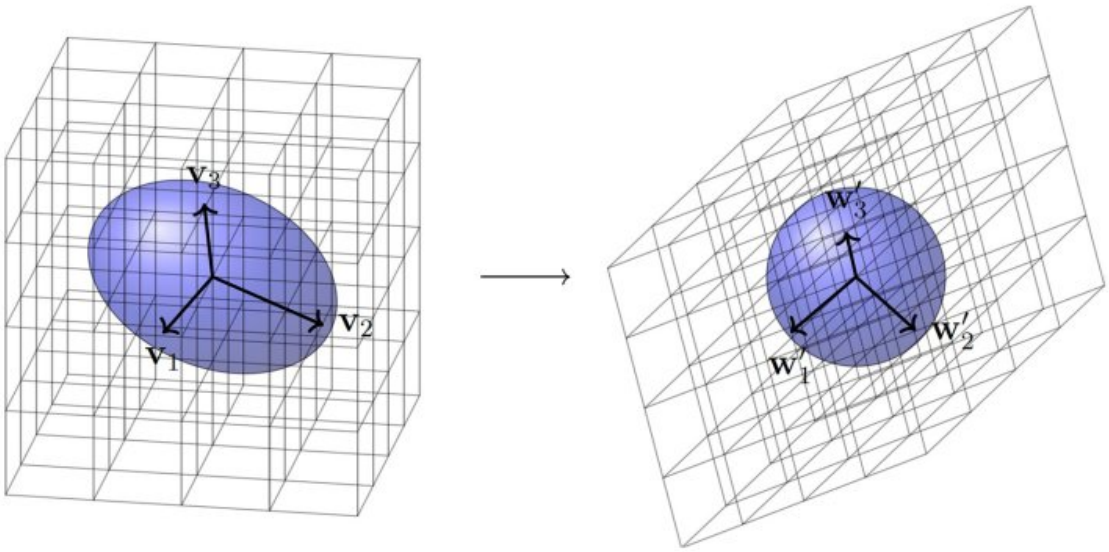
Practical Linear Algebra



Dr. Jim Newton

September 22, 2026

Contents

Chapter 0

Introduction

0.1 Course Objectives

- **Programming**
  - Develop intermediate Python programming skills.
  - Apply object-oriented design: classes, inheritance, and polymorphism.
  - Use functional programming features: comprehensions and function objects.
  - Work fluently with vectors and matrices in both mathematical notation and Python code.
- **Linear Algebra**
  - Understand why linear algebra is central to computer science: solving systems of linear equations, transforming data, and the mathematics underlying graphics, machine learning, and signal processing.
  - Progress from concrete $\mathbb{R}^n$ intuition to the abstract definition of a vector space, and recognise examples beyond $\mathbb{R}^n$.

- Apply Gaussian elimination (Bareiss algorithm) to solve linear systems, compute determinants, and invert matrices using exact integer arithmetic. Gauss-Jordan elimination is covered as optional enrichment.
- Understand bases, dimension, orthonormality, and the Gram-Schmidt process.
- Compute eigenvalues and eigenvectors and use them to diagonalize matrices.

0.2 How to Read This Document

Throughout this document, margin annotations and colored boxes serve as navigation aids.

Margin annotations. Icons appearing in the margin signal the nature of the adjacent content.

- **Essential** — a core concept you must understand and be able to apply.
- **Recall** — a result or technique introduced elsewhere; look it up if needed.
- **Warning** — a common mistake or a subtle point requiring care.
- **Enrichment** — optional deeper material for the curious reader.

Colored boxes. Different types of mathematical content are distinguished by background color.

Chapter Objectives — the skills and understanding you should have by the end of the chapter.

Activity — a hands-on task to run or observe, followed by a discussion question.

Theorem — a mathematical statement that has been proven true.

Definition — a precise meaning assigned to a mathematical term.

Example — a worked illustration of a concept or technique.

Proposition — a useful mathematical claim, typically with a short proof.

Corollary — a result that follows directly from the preceding theorem.

Lemma — a technical intermediate result used in the proof of a larger theorem.

Proof — a logical argument establishing a theorem or proposition.

Pattern — a reusable strategy or recipe to recognise and apply; the 🍴 margin icon marks later uses.

0.3 Overview

This course is at the same time a course in intermediate programming and also an introductory course in Linear Algebra.

There are two common approaches to introducing Linear Algebra. Some texts start with the practical—vectors and linear equations—and generalize to the abstract notion of a vector space. Other courses start abstractly with vector spaces and later specialize to $\mathbb{R}^n$. We will follow the former route, delaying the introduction of vector spaces until Chapter 7.

The particularity of our treatment of linear algebra is that we accompany each chapter with practical programming exercises which are

designed both to reinforce your understanding of the theoretical material learned and also to further develop your programming skills.

Even though we will use Python as the programming language, the programming concepts you use will also apply to other programming languages not covered in this course. In particular, we will exploit many of the functional programming features of the Python language.

0.4 Python

0.4.1 PyCharm

We will use the PyCharm IDE in the lectures of this course. It is a good idea for each student to also install PyCharm and use it to work through the exercises. This IDE has many features which make life easier for beginners.

0.4.2 Lists, Tuples, and Comprehensions

The Python language supports a built-in data structure which it refers to as a *list*. In most programming languages, a *list* is a linked list structure, but in Python it is an array. We will use the words *list* and *array* interchangeably. One important feature (and danger) of Python lists is that they are mutable. This means that the content of a list might change whenever it is passed to an insecure function. Accidental changes to data are difficult to debug.

The second problem with Python lists is that they cannot share tails. Many programming languages allow the programmer to easily prepend an element to a list, without affecting, neither intentionally nor accidentally, other variables which reference the same list. In Python there is no efficient way to extend a list in a way which is invisible to other references to the list. Instead for the most part we'll

use the Python *comprehension* to build lists and tuples efficiently and functionally. This approach eliminates most of the need to create a list, and thereafter modify it.

We will generally use the *tuple* data structure to prevent accidental modification of data.

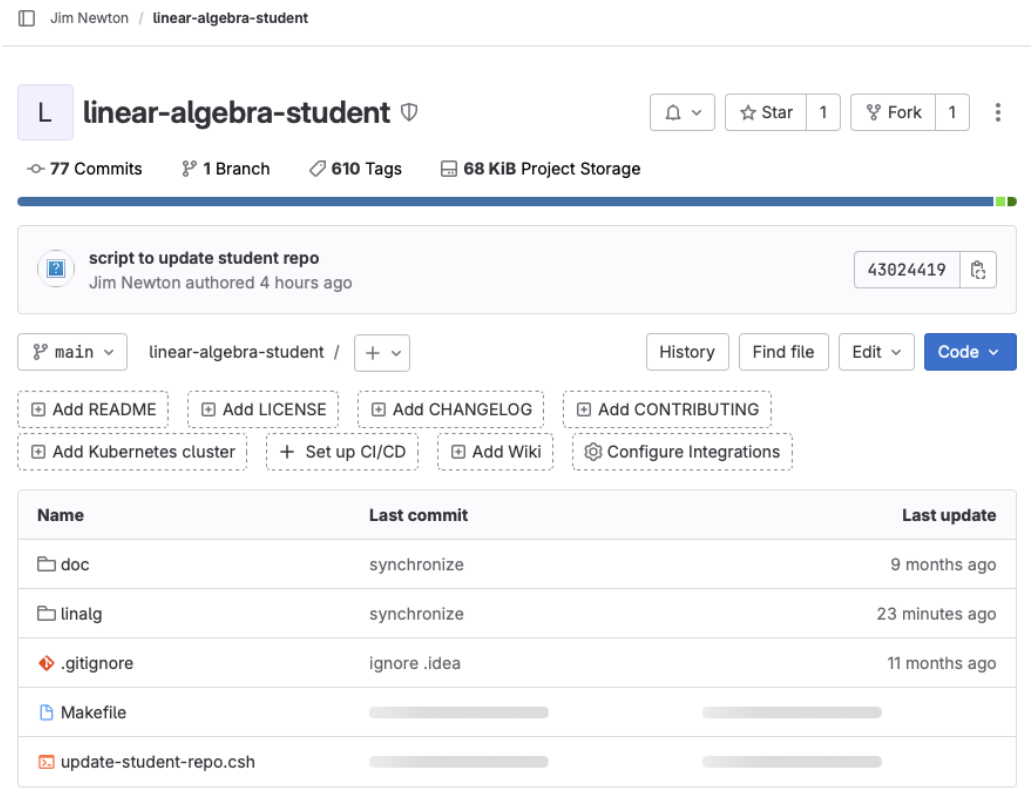
0.4.3 Object Orientation

Object-oriented programming is covered in depth in Chapter 2, using `Vector` as the running example.

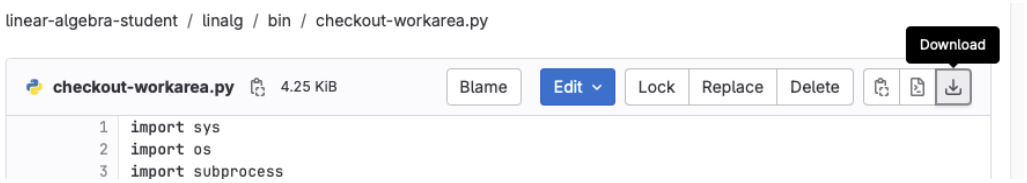
0.5 Downloading the Repository

You'll need to create a directory by running a script `checkout-workarea-linear-algebra.py`, but you must first download the script.

- Open a shell (for Windows, open PowerShell).
- Create a workarea: Change directory (`cd`) into the directory where you would like a directory to be created. The script will create a directory named, `linear-algebra-grader`, in this directory.
- With your web browser, open `https://gitlab.cri.epita.fr/jim.newton/linear-algebra-student`. See the following image:



- Click subdirectory: `linalg`
- Click subdirectory: `bin`
- Download: `checkout-workarea-linear-algebra.py` into the workarea, as shown in the following image



- Execute this command in the shell: `python checkout-workarea-linear-algebra.py`.

- You should now have a subdirectory named `linear-algebra-grader`.

0.6 Weekly Assignments

Each week you will be required to submit code which fulfills some expectations. Code should be readable. What makes code readable?

- Express the intent *vs.* Express the computation
- Use standardized coding conventions
- Make code concise

0.6.1 Factoring and Refactoring

Write code to be reused, and then reuse it! As a problem description changes over time, refactor existing code to maintain previous features which are still important but to provide new features. When code is obsolete, delete it. When code is generalized, remove specific, redundant cases.

0.6.2 Content of an Assignment

What does the weekly assignment entail?

- Coding Style: Code must be properly formatted. We will use the coding style *suggested* by PyCharm. **Reformat File ...**
- For each assignment, each student will complete some code (methods and functions) so that certain, specified, tests pass.

- The student will submit the code hierarchy which includes the code for the assignment in question PLUS all the previous assignments.
- The grade will be computed as a function of:
  - Number of tests passed for current assignment
  - Number of lines of code in your solution.
  - Code style

0.7 Syllabus

We will cover one chapter in the handout per week, more or less. Some chapters will take two weeks.

You have one assignment per chapter, usually due at 23h59 the day before the next lecture.

Session		Chapter	Assignment
1	0, 1	Vectors (due after session 2)	● hello_test.py ● Group_test.py ● Vector_test.py ● Vector_geometry_test.py ● Scaling_test.py
2	2	Object Orientation	(assignments due from session 1)
3	3	Matrices	● Matrix_test.py ● SqMatrix_test.py ● Ring_test.py
4	4	Determinant & Cramer's Rule	● row_operations_test.py ● laplacian_determinant_test.py ● cramer_test.py
5	5	Gaussian Elimination	● gaussian_test.py ● bareiss_test.py
6	6	Metric Spaces	● MetricSpace_test.py
7	7	Vector Space	● VectorSpace_test.py
8	8	Geometry	● proj_test.py ● Vector_norm_test.py ● Vector_inner_product_test.py
9	9	Basis	● orthogonal_test.py ● gram_schmidt_test.py ● coordinates_test.py
10	10	Eigenvalues	● eigenvalues_test.py

Implementation Order

Complete each row in order: implement the listed methods, then run the test before proceeding. Files appear more than once when only a subset of methods is needed at that step — leave the rest as `raise NotImplementedError()` stubs.

Session	Implement	Run test
1	hello.py	hello_test.py
	Group.py: is_monoid, is_group, is_commutative, is_abelian_group	Group_test.py
2	Vector.py: basic operations	Vector_test.py
	Vector.py: inner_product, norm, distance, angle, normalize	Vector_geometry_test.py
	Scaling.py: is_scaling	Scaling_test.py
3	Matrix.py	Matrix_test.py
	SqMatrix.py	SqMatrix_test.py
	Ring.py: is_ring, is_commutative_ring	Ring_test.py
4	Matrix.py: suppress_rc, row_operation, swap_rows, scale_row, adjoin_col, adjoin_row, extract_rows	row_operations_test.py
	SqMatrix.py: laplacian_expansion	laplacian_determinant_test.py
	SqMatrix.py: cramer	cramer_test.py
5	Matrix.py: back_substitution, make_upper_triangular_bareiss	gaussian_test.py
	Matrix.py: make_diagonal_bareiss	bareiss_test.py
6	MetricSpace.py: is_metric_space, geodesic_distance, manhattan_distance, discrete_distance, lcp_distance	MetricSpace_test.py
7	VectorSpace.py: is_vector_space	VectorSpace_test.py
8	VectorSpace.py: proj	proj_test.py
	VectorSpace.py: is_norm	Vector_norm_test.py
	VectorSpace.py: is_inner_product	Vector_inner_product_test.py
9	Vector.py: is_orthogonal	orthogonal_test.py
	VectorSpace.py: gram_schmidt	gram_schmidt_test.py
	SqMatrix.py, Vector.py: coordinates	coordinates_test.py
10	(TBD)	eigenvalues_test.py

Chapter 1

Vectors

Chapter Objectives

- Represent vectors in $\mathbb{R}^n$ as n-tuples and perform vector addition and scalar multiplication algebraically and in Python.
- Compute the dot product of two vectors and use it to find lengths, distances, and angles between vectors.
- Distinguish row vectors from column vectors and understand how context determines which representation is appropriate.
- Implement vector operations in Python using tuples and list comprehensions.
- Define monoid and group; verify that n-dimensional vectors form an Abelian group under addition.
- State the four axioms of a scaling operation; verify that componentwise multiplication of $\mathbb{R}^n$ by real scalars satisfies all four axioms.
- Derive from the axioms that $(-1)v$ is the additive inverse of v and that $v + v = 2v$.
- Implement `is_group` and `is_abelian_group` using property-based testing on randomly selected elements.

1.1 Monoids and Groups

We begin with two algebraic structures that will appear repeatedly throughout the course. We start with the slightly weaker notion of a *monoid*, then extend it to a *group* by requiring inverses.

Definition 1.1.1 (*monoid*)

A *monoid* is a set M along with an operator $\circ$ for which

1. (closure) $x_1, x_2 \in M \implies x_1 \circ x_2 \in M$
2. (associativity) $x_1, x_2, x_3 \in M \implies (x_1 \circ x_2) \circ x_3 = x_1 \circ (x_2 \circ x_3)$
3. (identity) $\exists e \in M$ such that $x \in M \implies e \circ x = x \circ e = x$

A *group* is a monoid in which every element also has an inverse.

Definition 1.1.2 (*group*)

A *group* is a monoid $(G, \circ)$ that additionally satisfies

(4) (inversion) $x \in G \implies \exists y \in G$ such that $x \circ y = y \circ x = e$

Definition 1.1.3 (*Abelian group*)

A group $(G, \circ)$ is *Abelian* if the operation is commutative: $x_1, x_2 \in G \implies x_1 \circ x_2 = x_2 \circ x_1$.

1.2 Vectors in Euclidean Space

We don't give a general definition of *vector* at this time. The definition will come in Chapter 7.

In this chapter we will talk about vectors in finite dimensional space. $\mathbb{R}^2$ represents the plane of (x, y) ordered pairs, or (x_1, x_2). $\mathbb{R}^3$

represents the 3-d space of (x, y, z) ordered triples, or (x_1, x_2, x_3). This pattern continues to every finite dimension so that $\mathbb{R}^n$ represents the n-dimensional space of $(x_1, x_2, \ldots, x_n)$ ordered n-tuples.

We will refer to an element, $(x_1, x_2, \ldots, x_n)$, of n-dimensional space as a vector, an n-tuple, or an n-vector. For example an ordered pair of real numbers such as $\left(\sqrt{2}, 3\sqrt{5}\right)$. However, we can consider vectors of arbitrary finite dimension, $(1.0, 3.0, 5.0)$, or $(1.0, 3.0, 5.0, 6.0, 7.7)$. We can talk about how many such components there are, 2, 3, ..., n.

We can talk about vectors whose components come from a given set such as:

- $\mathbb{N}$: $(1,\ 3,\ 5)$
- $\mathbb{Z}$: $(1,\ -3,\ -5)$
- $\mathbb{Q}$: $\left(\frac{3}{4},\ \frac{2}{5},\ \frac{-17}{13}\right)$
- $\mathbb{R}$: $\left(\sqrt{2},\ -3\sqrt{5},\ \pi\right)$
- $\mathbb{C}$: $(1.1 + 4i,\ -2 + 4.2i,\ 0 - i)$.

We sometimes write a vector as a *row*: $[1.2,\ 3.4]$; however we usually write a vector as a column $\begin{bmatrix} 1.2 \\ 3.4 \end{bmatrix}$.

The set of n-dimensional vectors whose components come from set, S is precisely the set S^n. *E.g.*, $\mathbb{R}^3 = \mathbb{R}\times\mathbb{R}\times\mathbb{R}$ is the set of 3-dimensional real vectors.

These are called *row-vectors* and *column-vectors* respectively. Whether a vector is written as a row or a column, or whether it is written using round parentheses or square brackets is only a matter of notation.

1.3 Arithmetic on Vectors

In this section we will define arithmetic operations on vectors so that they satisfy the monoid and group axioms by construction.

1.3.1 Equality

Two vectors are equal if they are equal component-wise.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} ,$$

but

$$\begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix} \neq \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix} .$$

Furthermore, vectors are not ordered. It makes no sense to claim that

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} < \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} .$$

We might however talk about whether the *length* of one vector is more or less than the length of another; see Section 1.5.

1.3.2 Addition

Two vectors of the same dimension may be added component-wise.

Definition 1.3.1 (*Addition in* $\mathbb{R}^n$)

In general if $u = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ and $v = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$, then the k'th component of $u + v$ is given by $(u + v)_k = a_k + b_k$.

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

$$c_k = a_k + b_k \tag{1.1}$$

The *zero vector* $\mathbf{0}_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ is an identity for addition: $v + \mathbf{0}_n = v$ for every $v \in \mathbb{R}^n$.

Theorem 1.3.2 ($\mathbb{R}^n$ *is a commutative monoid under addition*)

$(\mathbb{R}^n, +)$ is a commutative monoid.

Proof. We verify each monoid axiom and commutativity.

1. (closure) The sum of two n-vectors is an n-vector by definition.
2. (associativity) Let $u = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}$, $v = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$, $w = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$. The k'th component of $(u + v) + w$ is $(a_k + b_k) + c_k$, and the k'th

component of $u+(v+w)$ is $a_k+(b_k+c_k)$. These are equal by associativity of addition in $\mathbb{R}$, so $(u+v)+w = u+(v+w)$.

3. (identity) The zero vector $\mathbf{0}_n$ satisfies $v + \mathbf{0}_n = v$ for every $v \in \mathbb{R}^n$.

4. (commutativity) The k'th component of $u+v$ is a_k+b_k and of $v+u$ is b_k+a_k. These are equal by commutativity of addition in $\mathbb{R}$, so $u+v = v+u$.

1.3.3 Additive Inverse and Subtraction

We can extend $(\mathbb{R}^n, +)$ from a monoid to a group by providing an inverse for every vector.

Definition 1.3.3 (*Additive inverse in* $\mathbb{R}^n$)

The *additive inverse* of $v = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ is

$$-v = \begin{bmatrix} -a_1 \\ -a_2 \\ \vdots \\ -a_n \end{bmatrix}.$$

One checks that $v + (-v) = \mathbf{0}_n$, so every element has an inverse. Furthermore, addition is commutative. Therefore $(\mathbb{R}^n, +)$ is an *Abelian group*.

Subtraction is defined as addition of the inverse.

Definition 1.3.4 (*Subtraction in* $\mathbb{R}^n$)

$$u - v = u + (-v)$$

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \left(- \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right)$$

$$= \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} -b_1 \\ -b_2 \\ \vdots \\ -b_n \end{bmatrix} = \begin{bmatrix} a_1 - b_1 \\ a_2 - b_2 \\ \vdots \\ a_n - b_n \end{bmatrix}$$

1.4 Scaling

A vector may be multiplied by a scalar (a real number). We first define what it means for such an operation to be well-behaved, then show that component-wise multiplication satisfies those requirements.

Definition 1.4.1 (*Scaling operation*)

A *scaling operation* on an Abelian group $(G, +)$ by scalars from $\mathbb{R}$ is a function $\mathbb{R} \times G \to G$, written $(\alpha, v) \mapsto \alpha v$, satisfying for all $s, t \in \mathbb{R}$ and $u, v \in G$:

1. $1v = v$
2. $sv + tv = (s + t)v$
3. $su + sv = s(u + v)$
4. $(st)v = s(tv)$

We now define a specific operation on $\mathbb{R}^n$ and verify that it satisfies these axioms.

Definition 1.4.2 (*Scaling in* $\mathbb{R}^n$)

Given $\alpha \in \mathbb{R}$ and $u = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \in \mathbb{R}^n$, define αu component-wise:

$$\alpha \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \alpha a_1 \\ \alpha a_2 \\ \vdots \\ \alpha a_n \end{bmatrix}$$

i.e., $(\alpha u)_k = \alpha a_k$.

Theorem 1.4.3 (*Component-wise scaling is a scaling operation*)

Component-wise scaling on $\mathbb{R}^n$ satisfies all four axioms.

Proof. Let $s, t \in \mathbb{R}$ and $u = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}$, $v = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \in \mathbb{R}^n$. We verify each axiom by checking the k'th component.

1. $(1v)_k = 1 \cdot b_k = b_k = (v)_k$. ✓
2. $(sv + tv)_k = sb_k + tb_k = (s+t)b_k = ((s+t)v)_k$. ✓
3. $(su + sv)_k = sa_k + sb_k = s(a_k + b_k) = (s(u+v))_k$. ✓
4. $((st)v)_k = (st)b_k = s(tb_k) = (s(tv))_k$. ✓

In an entirely analogous way one may define a $\mathbb{C}$-*scaling operation* on an Abelian group by replacing $\mathbb{R}$ with $\mathbb{C}$ throughout the definition above. The four axioms are identical; only the scalar set changes. An Abelian group equipped with such an operation is called a *complex vector space*, and arises naturally in quantum mechanics and signal processing.

From these axioms we can derive several useful identities. First, $(-1)v$ is exactly the additive inverse of v:

$$v + (-1)v = 1v + (-1)v = (1 + (-1))v = 0v = \mathbf{0}_n \, .$$

(Here $0v = \mathbf{0}_n$ follows from axiom 2 with $s = 1, t = -1$ after noting $0v = 0v + \mathbf{0}_n$.) This connects scaling to the group structure: $-v = (-1)v$.

Second, adding a vector to itself gives twice the vector:

$$v + v = 1v + 1v = (1 + 1)v = 2v \, .$$

1.5 Length

The Euclidean-length of a vector, v, is denoted, $\|v\|$ and can be computed as:

Definition 1.5.1 (*Euclidean length*)

Let $v = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$. We define the *norm* of $v \in \mathbb{R}^n$ as

$$\|v\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2} = \sqrt{\sum_{k=1}^{n} a_k^2}$$

1.6 Distance

Given two vectors we can compute the *distance* between them by computing the norm of their difference.

In general a distance function must satisfy several axioms:

1. $d(u, v) = d(v, u)$ (symmetric)
2. $d(u, v) \geq 0$
3. $d(u, v) = 0 \iff u = v$
4. $d(u, v) + d(v, w) \geq d(u, w)$ (triangle inequality)

In Section 6.1 we will talk more generally about the concept of distance, and sets which have a distance defined on them. For the time being, we explicitly define the distance between two vectors of the same dimension as follows:

Definition 1.6.1 (*Euclidean distance*)

Let $u = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ and $v = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$ be elements of $\mathbb{R}^n$. The *distance* from u to v (equivalently from v to u) is defined as:

$$d(u, v) = \|u - v\| = \left\| \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right\| = \sqrt{\sum_{k=1}^{n} (a_k - b_k)^2} \qquad (1.2)$$

The dot product of two vectors having the same dimension is a quantity (a number, not a vector) whose value is zero if the vectors are perpendicular and maximum (in some sense) when the vectors are

parallel. The value of the dot product of two vectors pointing in the same *direction* is the product of the respective lengths of the vectors, and is the negative of that value if the vectors point in exactly opposite direction.

Definition 1.6.2 (*dot product*)

The dot product of two vectors having the same dimension is:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{k=1}^{n} a_k b_k \,. \tag{1.3}$$

Note that the length from Definition 1.5.1 can be rewritten in terms of the dot product from Definition 1.6.2 as follows.

$$\|v\| = \sqrt{v \cdot v} \tag{1.4}$$

The dot product has another important use. We may compute the angle between two vectors using the dot product. If θ is the angle swept out from vector u to vector v, then $\cos\theta$ is given by Equation (1.5). Notice as well that if you would like to compute the angle moving from v to u, the value of the dot product, (1.3), is the same, which corresponds to the fact that $\cos -\theta = \cos\theta$.

$$\cos\theta = \frac{u \cdot v}{\|u\| \ \|v\|} \tag{1.5}$$

1.7 Almost Equal

In Python we can check equivalence of integers (*i.e.*, `int`) and vectors of integers. However, we may not be able to check equivalence of real numbers (*i.e.*, `float`) or vectors of real numbers. For real numbers we

normally check *almost equal* rather than *equal*. Rather than testing $a = b$, we check whether the distance between a and b is less than some fixed threshold. $|a - b| < \varepsilon$. Since we can compute distance between vectors, we can make an analogous comparison to decide whether two vectors are *almost equal*.

$$\|u - v\| = \left\| \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right\| = \sqrt{\sum_{k=1}^{n} (a_k - b_k)^2} \qquad < \varepsilon$$

Sometimes we can make this check faster by recognizing that $a < \sqrt{b}$, if and only if $a^2 < b$. Thus for vectors we may check:

$$\|u - v\|^2 = \left\| \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right\|^2 = \sum_{k=1}^{n} (a_k - b_k)^2 \qquad < \varepsilon^2$$

The Python implementation of vectors, along with the programming exercises, is covered in Chapter 2.

1.8 Programming Exercises

1.8.1 Homework

The following assignments are given at the end of this session. You can begin working on them immediately using the mathematical definitions in this chapter. Chapter 2 explains how Python classes and magic methods connect these definitions to Python syntax; you will need that material to fully complete and understand every exercise.

- Implement `Group.py` — see Section 1.1. Verify each group axiom by testing randomly selected elements. No knowledge of Python OO is required: the functions take plain callables and return a `bool`.

  - `is_monoid`, `is_group`, `is_commutative`, `is_abelian_group`

  Test with `Group_test.py`.

- Implement `Vector.py` — the methods listed below only require arithmetic on the vector components (`self.content`). You do not need to understand magic methods to implement them; Chapter 2 will explain why they are named `__add__` etc. and how Python calls them.

  - `zero`, `__add__`, `__sub__`, `scale`, `__mul__` — basic arithmetic
  - `inner_product`, `norm`, `normalize`, `distance`, `angle` — geometry (see Sections **??**–**??**)

  Test with `Vector_test.py` and `Vector_geometry_test.py`.

- Implement `Vector_scale.py`: `is_scaling` — verify the four scaling axioms by sampling random vectors and scalars. Test with `Vector_scale_test.py`.

Chapter 2

Object Orientation

Chapter Objectives

- Understand Python classes and instances: define a class, allocate an instance, access instance variables.
- Implement magic methods (`__init__`, `__repr__`, `__eq__`, `__add__`, `__mul__`, `__getitem__`) and explain how Python invokes them.
- Implement static methods using `@staticmethod` and explain the difference from ordinary methods.
- Use `isinstance` to implement multiple dispatch inside a single method.
- Understand inheritance: a subclass inherits all methods of its parent class and may override them; we will see `SqMatrix` inheriting from `Matrix` in Chapter 3.
- Read `Polynomial.py` and identify the OO patterns it uses beyond `Vector`: immutability, `@property`, `__call__`, `__hash__`, `__bool__`, and mixed-type coercion.
- Implement `Vector.py` and test it with the provided test suite.
- Implement `Group.py` and `Vector_scale.py` using property-based testing to verify algebraic axioms.

This chapter covers the Python programming techniques needed to implement the mathematical objects studied in this course. We use the `Vector` class from Chapter 1 as the running example throughout.

2.1 Object-Oriented Programming

2.1.1 Classes and Instances

A *class* is a template for creating objects. An *instance* is a concrete object created from that template. In Python, a class is defined with the `class` keyword.

```
class Vector:
    def __init__(self, content):
        self.dim = len(content)
        self.content = tuple(content)
```

The function `__init__` is called automatically when a new instance is created. The call `Vector([1.0, 2.0, 3.0])` allocates an empty object and then calls `__init__` with that object as `self` and `[1.0, 2.0, 3.0]` as `content`. After `__init__` returns, the instance has two attributes: `self.dim = 3` and `self.content = (1.0, 2.0, 3.0)`.

```
v = Vector([1.0, 2.0, 3.0])
print(v.dim)      # 3
print(v.content)  # (1.0, 2.0, 3.0)
```

2.1.2 Magic Methods

Magic methods have names that begin and end with double underscores. Python calls them automatically in response to built-in operations.

Syntax	Magic method called	Purpose
`print(v)`	`__repr__`	string representation
`v == w`	`__eq__`	equality test
`v + w`	`__add__`	addition
`v * s`	`__mul__`	right-multiply by scalar
`s * v`	`__rmul__`	left-multiply by scalar
`v[i]`	`__getitem__`	indexing

For example, when Python evaluates `u + v`, it calls `u.__add__(v)`. You implement the method; Python decides when to call it.

2.1.3 Ordinary Methods

An ordinary method is called explicitly with `obj.method(args)`.

```
    def scale(v, s):
        return Vector.tabulate(v.dim, lambda i: s * v[i])
```

The call `v.scale(3)` passes `v` as the first argument (`self`, which here is named `v`) and `3` as `s`. Many IDEs will warn if the first argument is not named `self`, but Python does not require it.

2.1.4 Static Methods

A *static method* is a function defined inside a class but not bound to any instance. It is decorated with `@staticmethod` and called as `ClassName.method(args)`, without a `self` argument.

```
    @staticmethod
    def tabulate(dim, f):
        return Vector(tuple(f(k) for k in range(dim)))
```

Static methods are useful for *factory functions* — functions that create new instances — because at the point of the call there is no existing instance to call the method on.

```
v = Vector.tabulate(4, lambda i: i * 2)
# v.content == (0, 2, 4, 6)
```

2.1.5 isinstance and Multiple Dispatch

The built-in function `isinstance(obj, C)` returns `True` if `obj` is an instance of class `C` (or a subclass of `C`). It is useful inside `__mul__` when the same operator should behave differently depending on the type of the right-hand argument.

```
    def __mul__(v, other):
        if isinstance(other, Vector):
            return v.inner_product(other)   # dot product
        else:
            return v.scale(other)           # scalar multiplication
```

2.1.6 Inheritance

A *subclass* inherits all methods of its *parent class*. If the subclass defines a method with the same name, it *overrides* the parent's version. If not, Python looks up the method in the parent.

```
class Animal:
    def speak(self):
        return "..."

class Dog(Animal):
    def speak(self):        # overrides Animal.speak
        return "Woof"

class Cat(Animal):
    pass                    # inherits Animal.speak unchanged
```

In Chapter 3 we will see `SqMatrix` inherit from `Matrix`. A `SqMatrix` is a `Matrix` with equal numbers of rows and columns; it inherits addition and multiplication from `Matrix` but adds square-specific methods such as `identity` and `power`. Crucially, `isinstance(m, Matrix)` returns `True` for any `SqMatrix` instance `m`, so methods written for `Matrix` automatically accept `SqMatrix` arguments.

2.2 Vectors in Python

In the Python programming language we will represent a vector as an instance of the class `Vector`. The definition of `Vector` is incomplete and as homework exercises, you will finish the methods in the class definition. Each method corresponds to a mathematical operation defined in Chapter 1.

```
class Vector:
    def __init__(self, content):
      ...

    def __repr__(self):
      ...

    def __eq__(self, other):
      ...

    def __getitem__(self, items):
      ...

    @staticmethod
    def tabulate(dim, f):
      ...

    @staticmethod
    def random(dim, randomizer=randomize_int(-100, 100)):
      ...

    @staticmethod
    def zero(dim):
      ...

    def __add__(self, other):
      ...

    def scale(self, scalar):
      ...

    def __mul__(self, other):
      ...

    def __sub__(self, other):
```

```
36        ...
37
38    def norm(self):
39        ...
40
41    def normalize(self):
42        ...
43
44    def distance(self, other):
45        ...
46
47    def inner_product(self, other):
48        ...
49
50    def angle(self, other):
51        ...
```

Within a class definition, like the one above, `def` no longer defines a normal function, but rather defines what we refer to as a *method*. In this class definition as displayed here, all of the bodies of the method definitions are omitted.

Once a class definition has been loaded, subsequent calls to the method name using function syntax, allocates and initializes an instance of the class. For example `v = Vector([1.1, 2.3, -3.4])` will instantiate an object of type `Vector` and assign that object to the variable `v`. Thereafter, a method such as `scale` may be called using the syntax `v.scale(3)`. The mathematical meaning of vector scaling is discussed in Section 1.4. For now just notice, that an ordinary method call uses the syntax `v.method(args...)`.

There are several things to notice about the method definitions.

2.2.1 Magic Methods

So-called *magic* methods are indicated by the name beginning and ending with a double underscore.

```
1     def __init__(self, content):
2         ...
```

```
3
4     def __repr__(self):
5       ...
6
7     def __eq__(self, other):
8       ...
9
10    def __getitem__(self, items):
11      ...
12
13    def __add__(self, other):
14      ...
15
16    def __mul__(self, other):
17      ...
18
19    def __sub__(self, other):
20      ...
```

These methods, each with a reserved name, is part of the Python implementation, and the methods extend the semantics of the language in some way.

The `__init__` method gives the class definition a place to specify how to transform the initialization arguments, from the call-site, into the state saved on the instance itself. For example, below is a definition of the `__init__` magic method. When a call is made such as `Vector([1.1, 2.3, -3.4])`, the system allocates an *empty*, *uninitialized* instance of an object of type `Vector`, and calls the `__init__` method with the uninitialized instance as the `self` parameter, and with `[1.1, 2.3, -3.4]` as the `content` parameter. As indicated in the code the length of the content (3 in this case) is saved onto the instance as `self.dim = len(content)`, and the list `[1.1, 2.3, -3.4]` is converted to a `tuple` so that it becomes read-only, and stored as `self.content = tuple(content)`.

```
1     def __init__(self, content):
2         self.dim = len(content)
3         self.content = tuple(content)
```

With this initialization function in place, after code is evaluated

such as `v = Vector([1.1, 2.3, -3.4])`, subsequent calls to `v.dim` will evaluate to 3 and `v.content` will evaluate to the tuple `(1.1, 2.3, -3.4)`.

The method, `__repr__`, is called (automatically by Python) whenever it needs to print an object of type `Vector`. The responsibility of this method is to return the string to be printed.

```
    def __repr__(self):
        return f"Vector[{self.dim}]({self.content})"
```

This definition of `__repr__` causes a `Vector` to be printed revealing its dimension and content.

```
v = Vector([1.1, 2.3, -3.4])
print(v)  # evaluates to Vector[3]((1.1, 2.3, -3.4))
```

The `__eq__` method is called whenever Python needs to decide whether some object is equal to a given `Vector`. This implements component-wise equality as defined in Chapter 1.

```
    def __eq__(self, other):
        if not isinstance(other, Vector):
            return False
        return other.content == self.content
```

Whenever `v == w` is encountered, and the left-hand of the `==` is a `Vector` then this method will be called with the `Vector` as first argument. The second argument will be whatever is on the right-hand side of the `==`. The implementation above returns `False` if right-hand object fails to be of type `Vector`. If both objects are of type `Vector` then we test whether the content of the two objects are equal, by calling `==` on the two tuples.

The magic method `__getitem__` is used when the square brackets, indexing operator is used either to retrieve the value at an index, or for splicing.

```
    def __getitem__(self, items):
        return self.content[items]
```

The definition of this method allows us to index directly into vectors as if they were tuples.

```
v = Vector([1.1, 2.3, -3.4])
print(v[0])   # prints 1.1
print(v[-1])  # prints -3.4
print(v[1:])  # prints (2.3, -3.4)
```

2.2.2 Ordinary Methods

The `Vector` definition contains a method `scale` which implements scalar multiplication (Section 1.4):

```
    def scale(v, s):
        return Vector.tabulate(v.dim, lambda i: s * v[i])
```

Once an instance has been allocated, an ordinary method may be called.

```
v = Vector([1.1, 2.3, -3.4])
v.scale(3)   # evaluates to Vector((3.3, 6.9, -10.2))
```

The call to `v.scale(3)` calls the `scale` method in the `Vector` class with `v` as the first argument, `self`, and 3 as the second argument, `scalar`. You are free to name the arguments of a method any valid variable name you wish. However, many IDEs such as PyCharm will complain if the first argument is not `self`.

The magic methods `__add__`, `__mul__`, `__rmul__`, and `__sub__` implement the arithmetic defined in Chapter 1.

```
    def __add__(u, v):
        assert isinstance(v, Vector)
        assert u.dim == v.dim
        return Vector.tabulate(u.dim, lambda i: u[i] + v[i])

    def __mul__(v, s):
      return v.scale(s)

    def __rmul__(v, s):
      return v.scale(s)

    def __sub__(v, u):
      return v + u * -1
```

In mathematical notation the scalar is written on the left, as in $3v$, and Python supports both orderings: `3 * v` and `v * 3` both work. The former uses the `__rmul__` method, which Python calls on the right-hand operand when the left-hand operand does not know how to multiply by a `Vector`.

```
v = Vector([1,2,3])
u = Vector([2,3,4])
print(v + u)  # Prints Vector[3]((3,5,7))
print(3*v)    # Prints Vector[3]((3,6,9))
print(v*3)    # Prints Vector[3]((3,6,9))
print(u - v)  # Prints Vector[3]((1, 1, 1))
```

The methods `norm`, `distance`, and `inner_product` implement the corresponding sections of Chapter 1.

```
    def norm(v):
        import math
        return math.sqrt(v.inner_product(v))

    def distance(u, v):
        return (u - v).norm()

    def inner_product(u, v):
        assert u.dim == v.dim
        return sum(u[i] * v[i] for i in range(u.dim))
```

We must be careful with `angle` because the `math.acos` function will fail if its argument is not between -1 and 1. Normally $\frac{u \cdot v}{\|u\|\|v\|}$ will be in this range, but round-off error can push it slightly outside.

```
    def angle(u, v):
        assert isinstance(v, Vector)
        assert u.dim == v.dim
        import math

        tmp = u.inner_product(v) / (u.norm() * v.norm())
        if -1 <= tmp <= 1:
            return math.acos(tmp)
        elif tmp > 1:
            return math.acos(1)
        else:
            return math.acos(-1)
```

2.2.3 Static Methods

Ordinary methods are called using the `obj.method(args)` syntax where the `obj` to the left of the dot is an object of the type you are interested in, an object of type `Vector` in this case. However, sometimes we need to call a *function* within a class definition, but we don't have (or don't yet have) an object to invoke the method on. This is where static methods come in. These are normal functions, which just happen to be defined inside a `class` definition. The `tabulate` method is an example.

```
    @staticmethod
    def tabulate(dim, f):
        return Vector(tuple(f(k) for k in range(dim)))
```

This method allocates a new `Vector` object, not given the exact content, but rather given a size (`dim`), and a function which behaves as a formula to compute each entry. If the dimension is 4, then the `Vector` gets populated with `[f(0), f(1), f(2), f(3)]`. I.e. `f` is called `dim`-many times with a sort of counter argument which returns the values to populate the `Vector`. In such a case, you do not have an object of type `Vector`, because that is exactly what you are trying to create.

2.3 The Polynomial Class

A complete `Polynomial.py` is provided in the homework directory. You do not need to write it, but you should read through it: it demonstrates several OO patterns that go beyond what `Vector` shows. The following subsections highlight the most instructive ones.

2.3.1 Immutability

`Vector` stores its content as a `tuple`, making it accidentally hard to modify, but Python does not prevent you from adding new attributes to an object. `Polynomial` goes further and *enforces* immutability by overriding `__setattr__` and `__delattr__`:

```
    def __setattr__(self, name, value):
        raise AttributeError("Polynomial is immutable")

    def __delattr__(self, name):
        raise AttributeError("Polynomial is immutable")
```

With these two methods in place, any attempt to write `p.foo = 3` or `del p.foo` raises an `AttributeError` at runtime.

There is a bootstrapping problem: `__init__` itself must set the initial attribute, yet `__setattr__` is already active by the time `__init__` runs. The solution is to bypass the overridden method and call the base-class version directly:

```
    def __init__(self, coefficients):
        ...
        object.__setattr__(self, 'coefficients', tuple(coeffs))
```

The expression `object.__setattr__(self, ...)` invokes the original built-in attribute setter, skipping `Polynomial`'s override. This is the standard idiom for immutable classes in Python.

2.3.2 Properties

The `@property` decorator turns a no-argument method into an attribute that is computed on demand:

```
    @property
    def degree(self) -> int:
        return len(self.coefficients) - 1
```

After this, `p.degree` reads like a plain attribute but actually calls the method. Callers cannot accidentally set `p.degree = 0` (that would

raise `AttributeError`). Use `@property` for derived quantities that should feel like data but be computed from stored state.

2.3.3 Callable Instances

Implementing `__call__` makes an instance *callable*: the object itself behaves like a function.

```
    def __call__(self, x):
        return sum(c * x**i for i, c in enumerate(self.coefficients)
    )
```

This evaluates the polynomial at the point `x`:

```
p = Polynomial([1, 0, 1])    # 1 + x^2
print(p(3))                   # 10  (= 1 + 0*3 + 1*9)
print(p(0))                   # 1
```

2.3.4 `__hash__` and `__bool__`

Defining `__eq__` has a side-effect: Python automatically sets `__hash__` to `None`, making instances *unhashable* (they cannot be used as dictionary keys or set members). To restore hashability, define `__hash__` explicitly:

```
    def __hash__(self) -> int:
        return hash(self.coefficients)
```

Since `coefficients` is a tuple, `hash` works on it directly. Two equal polynomials produce the same hash, as required.

The `__bool__` method controls how an object is treated in a boolean context (`if p:`, `not p`, `while p:`):

```
    def __bool__(self) -> bool:
        return any(c != 0 for c in self.coefficients)
```

The zero polynomial is falsy; all others are truthy. This lets the division algorithm write `if not other:` to guard against dividing by zero, instead of the clumsier `if other == Polynomial([0]):`.

2.3.5 Mixed-Type Arithmetic and `__radd__`

`Vector` only adds vectors to vectors. `Polynomial` is more flexible: `p + 3` should produce the polynomial $p(\lambda) + 3$. The private helper method `_coerce` normalises the right-hand argument:

```
    def _coerce(self, other):
        if isinstance(other, Polynomial):
            return other
        if isinstance(other, numbers.Number):
            return Polynomial([other])
        return NotImplemented
```

Returning `NotImplemented` (not raising an exception) tells Python "I can't handle this type; try the other operand." Every arithmetic method calls `_coerce` first:

```
    def __add__(self, other) -> 'Polynomial':
        other = self._coerce(other)
        if other is NotImplemented:
            return NotImplemented
        ...
```

The *reflected* methods `__radd__`, `__rsub__`, and `__rmul__` handle the case where the scalar is on the *left*: Python calls `p.__radd__(3)` when evaluating `3 + p` and the integer `3` does not know how to add a polynomial.

```
    def __radd__(self, other) -> 'Polynomial':
        return self.__add__(other)   # addition is commutative
```

2.3.6 Rational Arithmetic: `fractions.Fraction`

For rational numbers, Python's standard library provides `fractions.Fraction`. A `Fraction` stores a numerator and denominator as integers and performs all arithmetic exactly, with no floating-point rounding. For example:

```
from fractions import Fraction
a = Fraction(1, 3)   # exactly 1/3
```

```
b = Fraction(1, 6)    # exactly 1/6
print(a + b)          # Fraction(1, 2)  -- exact, not 0.4999...
print(a * b)          # Fraction(1, 18)
print(a / b)          # Fraction(2, 1)
```

`Fraction` objects interoperate with Python integers and with each other under the usual arithmetic operators, so a matrix whose entries are `Fraction` values can be used directly with the `Matrix` and `SqMatrix` classes. This gives us $\mathbb{Q}^{n\times n}$: exact rational matrix arithmetic, useful whenever floating-point precision is a concern.

2.4 Programming

1. Type Annotations
    - MyPy
    - Function return value
    - argument values
    - `Optional`
    - type parameters
2. Object Orientation
    - class definition: `class`
    - creating an instance:
        - class name
        - factory function
    - initialization protocol: `__init__`
    - printed representation: `__repr__`
    - method definition

    - `def`, `self`
    - `@staticmethod`
  - calling a method
    - normal method: `self.foo(...)`
    - static method: `ClassName.foo()`
  - Currying method call:
    - `lambda :  self.foo()`
    - `lambda a:  self.foo(a)`
    - `lambda a, b:  self.foo(a, b)`

3. Operator Overloading
   - `a==b`: `__eq__`
   - `a+b`: `__add__`
   - `a*b`: `__mul__`
   - `a[b]`: `__getitem__`
4. Comprehensions
   - generator
   - list comprehension
   - tuple comprehension
5. Working with Arrays
   - list and array
   - zero-based index
   - appending arrays

6. Randomization
    - dimension
    - type of components
    - range of components
7. Scoping
    - `import` and `from/import`
    - `nonlocal`
8. Exceptions
    - `raise NotImplementedError`
    - `assert`

2.5 Programming Exercises

2.5.1 Property-Based Testing

In a previous course you may have verified algebraic axioms by *exhaustive checking*: given a finite set such as $\mathbb{Z}/5$, you can enumerate every element (or every pair, or every triple) and confirm that the axiom holds for each one. For a set of size n, checking an axiom involving two elements requires at most n^2 tests — manageable when n is small.

The sets in this course are *infinite*: $\mathbb{R}^n$, the integers, the real numbers, the set of all $n \times n$ matrices. Exhaustive checking is impossible. Instead we use *property-based testing*: choose the axiom to verify, generate a large number of random elements from the set, and check that the axiom holds for every sampled combination. If any sample fails, we have a concrete counterexample that proves the axiom does not

hold. If 1000 independently drawn samples all pass, we have strong (though not mathematically conclusive) evidence that the axiom holds in general.

The technique is not limited to algebraic axioms. Whenever a specification can be stated as a predicate over inputs — *a sorting function always returns a sorted list*, *a parser round-trips every string it accepts*, *a compression function is lossless* — property-based testing gives cheap, broad coverage. The industrial library for this approach in Python is `hypothesis`; in this course we implement the same idea by hand, which also makes the structure of each proof obligation explicit.

2.5.2 Classwork

- Object orientation in Python
- Introduce implementation of `Vector` in class.
- Testing
- Property based tests
- Test the triangle inequality
- Start `Group.py`: discuss monoid and group axioms; verify $(\mathbb{R}^n, +)$ is an Abelian group
- Start `Vector_scale.py`: discuss the four scaling axioms and how to test them

2.5.3 Homework

These assignments were given at the end of Chapter 1. Now that you have studied this chapter you should be able to complete them fully.

The OO concepts here — magic methods, `self`, `@staticmethod` — explain *why* the methods are structured the way they are; use the `Vector.py` examples from Section **??** as models.

- Complete `Vector.py` (an n-dimensional Euclidean vector):
  - `zero`, `__add__`, `scale`, `__mul__`, `__sub__` — basic arithmetic
  - `inner_product`, `norm`, `normalize`, `distance`, `angle` — geometry

  Test with `Vector_test.py` and `Vector_geometry_test.py`.
- Complete `Group.py`:
  - `is_monoid`, `is_group`, `is_commutative`, `is_abelian_group`

  Test with `Group_test.py`.
- Complete `Vector_scale.py`: `is_scaling`. Test with `Vector_scale_test.py`.

Chapter 3

Matrices

Chapter Objectives

- Represent $m \times n$ matrices as tuples of tuples in Python using the `Matrix` and `SqMatrix` classes.
- Show that $\mathbb{R}^{n \times m}$ is an Abelian group under component-wise addition, identifying the zero matrix as identity and entry-negation as inverse.
- Show that $\mathbb{R}^{n \times n}$ is a monoid under matrix multiplication: identify the unique identity I_n, prove associativity, and give examples showing multiplication is not commutative.
- Define a ring; explain why $\mathbb{R}^{n \times n}$ is a non-commutative ring, and why matrices with entries from any ring form a ring.
- Show that $\mathbb{R}^{n \times n}$ is *not* a group under multiplication: zero divisors are not invertible; define invertible and singular matrices.
- Implement `is_ring` and `is_commutative_ring` using property-based testing, and apply them to square matrices.

3.1 Definition

Mathematically a matrix is a finite set of numbers of size $m \times n$ arranged into m rows and n columns. Typically represented as:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}$$

We denote the number in the i'th row j'th column as a_{ij} or $a_{i,j}$. In mathematical computations we refer to elements indexed starting at 1 and ending (including) m and n.

A matrix can be represented in Python as a m-length tuple of n-length tuples of numbers. We will additionally declare classes to allow us to programmatically distinguish a matrix from just any arbitrary tuple. The classes will be named `SqMatrix` and `Matrix`.

Even if mathematically we index vectors and matrices from $1, 1$ to m, n, in the Python we use indices ranging from 0 to $m-1$ and $n-1$. This distinction is common. Some programming languages use 1-based indices, some use 0-based indices, and some languages allow the user to decide. Python uses 0-based indices.

3.2 Additive Group

Definition 3.2.1 (*matrix addition*)

If two matrices A and B each have dimensions $n \times m$, *i.e.* n rows and m columns, then A and B can be added and the component in the row i column j of the sum is exactly $c_{ij} = a_{ij} + b_{ij}$.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1}+b_{n1} & a_{n2}+b_{n2} & \cdots & a_{nn}+b_{nn} \end{bmatrix}$$

$$c_{i,j} = a_{i,j} + b_{i,j} \tag{3.1}$$

Theorem 3.2.2 ($\mathbb{R}^{n\times m}$ *is an Abelian group under addition*)

The set $\mathbb{R}^{n\times m}$ of all $n \times m$ real matrices forms an Abelian group under component-wise addition. The identity is the zero matrix $\mathbf{0}$; the inverse of A is $-A$ (negate every entry).

Proof. We verify each axiom at row i, column j.

1. **Closure**: $(A+B)_{ij} = a_{ij} + b_{ij} \in \mathbb{R}$, so $A + B \in \mathbb{R}^{n\times m}$. ✓
2. **Associativity**: $((A+B)+C)_{ij} = (a_{ij} + b_{ij}) + c_{ij} = a_{ij} + (b_{ij} + c_{ij}) = (A + (B+C))_{ij}$. ✓
3. **Identity**: Let $\mathbf{0}$ have all entries zero. $(\mathbf{0} + A)_{ij} = 0 + a_{ij} = a_{ij}$. ✓
4. **Inverse**: Define $(-A)_{ij} = -a_{ij}$. $(A + (-A))_{ij} = a_{ij} + (-a_{ij}) = 0$. ✓

5. **Commutativity**: $(A+B)_{ij} = a_{ij}+b_{ij} = b_{ij}+a_{ij} = (B+A)_{ij}$. ✓

Note that Theorem 3.2.2 holds for *rectangular* matrices of any shape $n \times m$; no assumption of squareness is needed.

Definition 3.2.3 (*matrix scaling*)

If a matrix A has dimensions $n \times m$, *i.e.* n rows and m columns, and $s \in \mathbb{R}$, then A can be scaled by s, denoted sM. The component in the row i column j of sM is exactly $c_{ij} = s \times a_{ij}$.

$$s \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} sa_{11} & sa_{12} & \cdots & sa_{1n} \\ sa_{21} & sa_{22} & \cdots & sa_{2n} \\ \vdots & \vdots & & \vdots \\ sa_{n1} & sa_{n2} & \cdots & sa_{nn} \end{bmatrix}$$

$$c_{i,j} = sa_{i,j} \tag{3.2}$$

Given this definition of matrix scaling, we can define matrix subtraction:

Definition 3.2.4 (*Matrix subtraction*)

Assuming A and B are matrices of the same size,

$$A - B = A + (-1)B\,.$$

3.3 Matrix Multiplication

Two matrices, A and B, can be multiplied if their dimensions are compatible. The requirement is that an $n \times k$ (on the left) can be multiplied by a $k \times m$ matrix to obtain an $n \times m$ matrix.

We may multiply two matrices with compatible sizes.

Definition 3.3.1 (*matrix multiplication*)

If A has dimension $n \times k$ and B has dimension $k \times m$, then $A \times B$ (also written AB) has dimension $n \times m$. In the product, AB, the component c_{ij} (row i, column j) is given by

$$c_{ij} = \sum_{\ell=1}^{k} a_{i\ell} b_{\ell j} \tag{3.3}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix},$$

Notice that to compute c_{ij} using Equation (3.3), requires k-many multiplications. Furthermore, we must compute c_{ij} many times to populate an $n \times m$ matrix, *i.e.*, $n \times m$ times. So the matrix multiplication as shown in Definition 3.3.1 requires $n \times k \times m$ scalar multiplications. This cubic complexity can be improved in many special cases; see Section 3.8.7.

There are many ways to think about this. One way is that we *multiply* the rows of A by the columns of B. Another way to think about it is that each c_{ij} is the dot product of two vectors, the i'th row vector from A with the j'th column vector of B. See Definition 1.6.2 (page 31) for a reminder of dot products.

Example 3.3.2 (*Matrix Multiplication*)

Let's compute the product of the following two matrices.

$$
\begin{aligned}
A &= \begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix} \\
B &= \begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix} \\
AB &= \begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix} \times \begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix} \\
&= \begin{bmatrix} 0-15+16 & 0+18-18 & 0+6-6 \\ 4-20+16 & -5+24-18 & -2+8-6 \\ -12+20-8 & 15-14+9 & 6-8+3 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

3.3.1 Multiplication is Not Closed for Rectangular Matrices

For multiplication to define a monoid on a set, the operation must be *closed*: the product of any two elements of the set must again be

an element of the set. Matrix multiplication fails this requirement for rectangular matrices. If A and B are both $n \times m$ with $n \neq m$, the product AB requires the inner dimensions to match, *i.e.*, the number of columns of A must equal the number of rows of B. When $n \neq m$ this fails and AB is simply undefined.

Example 3.3.3 (*Commutation not defined*)

$$A = \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$$AB = \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

$$BA \neq \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} = \text{undefined}$$

BA is undefined because the number of columns of B is different from the number of rows of A.

In many cases both AB and BA are defined but they have different sizes, and are necessarily unequal.

Example 3.3.4 (*Commutation with different dimensions*)

$$A = \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

$$AB = \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \\ 1 & 0 & 1 \end{bmatrix}$$

$$BA = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 2 & 1 \end{bmatrix}$$

$AB \neq BA$, because they have different dimensions.

For $n \times n$ square matrices, however, the product of two $n \times n$ matrices is always an $n \times n$ matrix. Multiplication is closed on $\mathbb{R}^{n \times n}$.

3.4 Square Matrices Form a Monoid

Multiplication is closed on $\mathbb{R}^{n \times n}$. To show that $(\mathbb{R}^{n \times n}, \times)$ is a monoid we must also demonstrate an identity element and associativity.

Theorem 3.4.1 (*Unique identity*)

If $n \in \mathbb{N}$ $(n > 0)$, then there exists exactly one $n \times n$ matrix, I_r and exactly one $n \times n$ matrix, I_ℓ, such that for every $n \times n$ matrix,

M, we have $MI_r = M$ and $I_\ell M = M$. Moreover,

$$I_r = I_\ell = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Since $I_r = I_\ell$ we shall denote it simply as I.

Proof. **Part 1: $I_r = I_\ell$.** Setting $M = I_\ell$ in $MI_r = M$ gives $I_\ell I_r = I_\ell$. Setting $M = I_r$ in $I_\ell M = M$ gives $I_\ell I_r = I_r$. Therefore $I_r = I_\ell$; call their common value I.

Part 2: Form of I. Let E_{ij} denote the $n \times n$ matrix with 1 in row i, column j and 0 elsewhere. Applying $E_{ij}\, I = E_{ij}$ and looking at entry (i, ℓ):

$$(E_{ij}\, I)_{i\ell} = \sum_{k=1}^{n} (E_{ij})_{ik}\, I_{k\ell} = I_{j\ell}\,.$$

This must equal $(E_{ij})_{i\ell}$, which is 1 when $\ell = j$ and 0 otherwise. Hence $I_{j\ell} = \delta_{j\ell}$ (the Kronecker delta) for every j and ℓ: ones on the diagonal, zeros everywhere else.

From Theorem 3.4.1, we can say that for any square matrix A, we have $I \times A = A \times I = A$.

Theorem 3.4.2 (*Matrix Multiplication is Associative*)

Let A be a $n \times p$ matrix, B be a $p \times q$ matrix and C be a $q \times m$ matrix. The $(A \times B) \times C = A \times (B \times C)$.

To prove this theorem, we'll show that the ith row jth column

element of $(A \times B) \times C$ and $A \times (B \times C)$ are the same.

Proof. Let $P = (AB)C$. Denote the ith row jth column of A as $a_{i,j}$, and of B as $b_{i,j}$. The ith row jth column: $(A \times B)_{i,j} = \sum_{k=1}^{p} a_{i,k}\ b_{k,j}$. The ith row jth column: $(B \times C)_{i,j} = \sum_{k=1}^{q} b_{i,k}\ c_{k,j}$.

To multiply $(A \times B) \times C$ we multiply the ith row of $A \times B$ by the jth column of C.

$$P_{i,j} = (\overbrace{(A \times B)}^{n \times q} \times \overbrace{C}^{q \times m})_{i,j}$$

$$= \overbrace{\left[\sum_{k=1}^{p} a_{i,k}\ b_{k,1} \quad \sum_{k=1}^{p} a_{i,k}\ b_{k,2} \quad \cdots \quad \sum_{k=1}^{p} a_{i,k}\ b_{k,q}\right]}^{i\text{th row of } A \times B} \times \overbrace{\begin{bmatrix} c_{1,j} \\ c_{2,j} \\ \vdots \\ c_{q,j} \end{bmatrix}}^{j\text{th column of } C}$$

$$= \sum_{\ell=1}^{q} \sum_{k=1}^{p} a_{i,k}\ b_{k,\ell}\ c_{\ell,j}$$

Now the ith row of A times the jth column of $B \times C$:

$$(\overbrace{A}^{n\times p} \times \overbrace{(B \times C)}^{p\times m})_{i,j} = \overbrace{\begin{bmatrix} a_{i,1} & a_{i,2} & \cdots & a_{i,p} \end{bmatrix}}^{i\text{th row of } A} \times \overbrace{\begin{bmatrix} \sum_{\ell=1}^{q} b_{1,\ell}\ c_{\ell,j} \\ \sum_{\ell=1}^{q} b_{2,\ell}\ c_{\ell,j} \\ \vdots \\ \sum_{\ell=1}^{q} b_{p,\ell}\ c_{\ell,j} \end{bmatrix}}^{j\text{th column of } B\times C}$$

$$= \sum_{k=1}^{p}\sum_{\ell=1}^{q} a_{i,k}\ b_{k,\ell}\ c_{\ell,j}$$

Therefore, $((A \times B) \times C)_{i,j} = (A \times (B \times C))_{i,j}$.

Since multiplication is closed on $\mathbb{R}^{n\times n}$, and we have a unique identity (Theorem 3.4.1) and associativity (Theorem 3.4.2), we conclude:

Theorem 3.4.3 ($(\mathbb{R}^{n\times n}, \times)$ *is a monoid*)

The set $\mathbb{R}^{n\times n}$ of $n \times n$ real matrices forms a monoid under matrix multiplication. The identity element is I_n.

Uniqueness of the identity also follows from a general abstract argument: if e and e' are both identities in any monoid, then $e = e \cdot e' = e'$. Theorem 3.4.1 is more informative, however, because it derives the *form* of the identity — the diagonal matrix with ones — rather than merely asserting that only one such matrix can exist.

3.4.1 Multiplication is Not Commutative

Matrix multiplication is, in general, not commutative.

In many cases AB and BA are defined and have the same dimension

but simply are different.

Example 3.4.4 ($AB \neq BA$)

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$B = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$
$$AB = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$
$$BA = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}$$

Even if matrix multiplication is not commutative in general, there are special cases where it is–in some cases we find that $AB = BA$. There exist pairs of matrices which are commutative under multiplication. *E.g.*, if a matrix is invertible, then $A \times A^{-1} = I = A^{-1} \times A$, Equation (3.21). Also $A^n \times A^m = A^{n+m} = A^{m+n} = A^m \times A^n$, Equation (3.22).

Example 3.4.5 ($AB = BA$)

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$B = A \times A = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$
$$AB = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}$$
$$BA = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}$$

In this case, since $B = A \times A$, we have

$$AB = A(AA) = (AA)A = BA\,.$$

This principle is generalized in Equation (3.22)

See Example 4.2.1 (page 109) for another example.

3.4.2 Associativity and Matrix Chaining

Theorem 3.4.2 shows that matrix multiplication is associative when it is defined. This means you are free to rearrange the parentheses as you like, and mathematically speaking you will get the same result. However, that does not mean that the same amount of work and storage is needed when computing $A(BC)$ vs $(AB)C$. In Examples 3.4.6 and 3.4.7 we multiply three matrices two different ways and examine the amount of memory needed and the number of scalar multiplications needed in each case.

$$A = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & -1 \end{bmatrix} \tag{3.4}$$

$$B = \begin{bmatrix} 1 & 1 & 1 & 0 & -1 \\ 1 & -1 & 0 & 1 & 1 \\ -1 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{3.5}$$

$$C = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \tag{3.6}$$

Example 3.4.6 (*Associating on the left*)

We first compute $(AB)C$.

$$
\begin{aligned}
ABC &= (AB)C \\
&= \left(\begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 0 & -1 \\ 1 & -1 & 0 & 1 & 1 \\ -1 & 0 & 1 & 1 & 1 \end{bmatrix} \right) \times C \qquad (3.7) \\
&= \begin{bmatrix} -1 & 2 & 2 & 0 & -1 \\ 2 & 1 & 0 & -1 & -2 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \qquad (3.8) \\
&= \begin{bmatrix} 0 & 2 & 2 & 2 \\ 2 & 5 & 0 & 3 \end{bmatrix}
\end{aligned}
$$

In step 3.7 we compute $A \times B$ which is a 2×5 matrix. To multiply A (dimension 2×3) by B (dimension 3×5) requires $2 \times 5 \times 3 = 30$ multiplications (using Definition 3.3.1). Additionally, we must allocate an interim matrix of size $2 \times 5 = 10$.

In step 3.8, we multiply $(AB) \times C$, *i.e.*, a 2×5 by a 5×4 to obtain ABC which has dimensions 2×4. This multiplication requires $2 \times 5 \times 4 = 40$ multiplications.

To perform the two multiplications requires $30 + 40 = 70$ total scalar multiplications.

Example 3.4.7 (*Associating on the right*)

Now, let's compute $ABC = A(BC)$.

$$\begin{aligned} ABC &= A(BC) \\ &= A\times\left(\begin{bmatrix} 1 & 1 & 1 & 0 & -1 \\ 1 & -1 & 0 & 1 & 1 \\ -1 & 0 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}\right) \qquad (3.9) \\ &= \begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 & 2 & 3 \\ 0 & -1 & 2 & 1 \\ -1 & -2 & 2 & 0 \end{bmatrix} \qquad (3.10) \\ &= \begin{bmatrix} 0 & 2 & 2 & 2 \\ 2 & 5 & 0 & 3 \end{bmatrix} \end{aligned}$$

In step 3.9 we compute $B \times C$ which is a 3×4 matrix. To multiply B (dimension 3×5) by C (dimension 5×4) requires $3\times5\times4 = 60$ multiplications (using Definition 3.3.1). Additionally, we must allocate an interim matrix of size $3 \times 4 = 12$.

In step 3.10, we multiply $A \times (BC)$, *i.e.*, a 2×3 by a 3×4 to obtain ABC which has dimensions 2×4. This multiplication requires $2 \times 3 \times 4 = 24$ multiplications.

To perform the two multiplications requires $60 + 24 = 84$ total scalar multiplications.

To summarize Examples 3.4.6 and 3.4.7, computing $(AB) \times C$ requires 70 scalar multiplications and size 10 of interim storage, while computing $A \times (BC)$ requires 84 scalar multiplications and size 12 of interim storage.

We generalize by assuming we wish to compute

$$A_{n\times p} \times B_{p\times q} \times C_{q\times m}\,.$$

Computing $\underbrace{(A_{n\times p}B_{p\times q})}_{n\times q}\times C_{q\times m}$ requires $npq + nqm = nq(p+m)$ scalar multiplications. Computing $A_{n\times p} \times \underbrace{(B_{p\times q}C_{q\times m})}_{p\times m}$ requires $npm + pqm = pm(n+q)$ scalar multiplications.

The conclusion we draw from Examples 3.4.6 and 3.4.7 is that to decide whether to compute $(AB)C$ vs $A(BC)$ depends on the respective sizes of the matrices A, B, and C. If $nq(p+m) < pm(n+q)$ then compute $(AB)C$, otherwise compute $A(BC)$. This choice is straightforward when multiplying three matrices. However, when multiplying four matrices there are 5 ways to draw the parentheses.

$$\begin{aligned} ABCD &= A(B(CD)) \\ &= A((BC)D) \\ &= (AB)(CD) \\ &= ((AB)C)D \\ &= (A(BC))D \end{aligned}$$

For 5 matrices there are 14 ways, for 6 matrices there are 42 ways, and for 7 matrices there are 132 ways. In general for n matrices there are $\frac{1}{n}\binom{2(n-1)}{n-1}$ ways.[1] Consequently, finding the most efficient way to multiply n matrices is difficult. In this course, we will not solve the problem of finding the most efficient grouping to minimize the work, or minimize the memory of the so-called *matrix chaining* problem. There is a discussion on this on Wikipedia: `https://en.wikipedia.org/wiki/Matrix_chain_multiplication`.

[1] See a discussion on StackExchange: `https://math.stackexchange.com/questions/1630457/how-many-ways-to-multiply-n-matrices`.

3.5 Ring Structure of Matrices

We now have two operations on $\mathbb{R}^{n\times n}$:

- $(\mathbb{R}^{n\times n}, +)$ is an Abelian group (Theorem 3.2.2).
- $(\mathbb{R}^{n\times n}, \times)$ is a monoid (Theorem 3.4.3).

A set equipped with two operations satisfying the above properties *and* a distributivity condition linking them is called a *ring*.

Recall from Chapter 1 that a *monoid* is a set with an associative binary operation and an identity element, and that a *group* is a monoid in which every element has an inverse. We now combine two such operations into a single structure.

Definition 3.5.1 (*ring*)

A *ring* is a set R with two binary operations $+$ and $\cdot$ for which

1. $(R, +)$ is an Abelian group (with identity 0).
2. $(R, \cdot)$ is a monoid (with identity 1).
3. (distributivity) $a \cdot (b + c) = a \cdot b + a \cdot c \quad$ and $\quad (a + b) \cdot c = a \cdot c + b \cdot c \quad$ for all $a, b, c \in R$.

If additionally $a \cdot b = b \cdot a$ for all $a, b \in R$, the ring is called *commutative*.

Examples of commutative rings include the integers $\mathbb{Z}$, the rationals $\mathbb{Q}$, the reals $\mathbb{R}$, and the complex numbers $\mathbb{C}$. The set $\mathbb{R}[\lambda]$ of polynomials with real coefficients is also a commutative ring, worth examining in more detail because it will reappear throughout the course.

A polynomial $p \in \mathbb{R}[\lambda]$ is a finite sum $p(\lambda) = a_0 + a_1\lambda + a_2\lambda^2 + \cdots + a_d\lambda^d$ with $a_i \in \mathbb{R}$. Two polynomials are added coefficient-wise

and multiplied by the usual convolution rule (expand and collect like powers). The ring axioms are satisfied:

- **Additive identity**: the zero polynomial 0 (all coefficients zero).
- **Additive inverse**: $-p$ (negate every coefficient).
- **Multiplicative identity**: the constant polynomial 1.
- **Commutativity of multiplication**: $p \cdot q = q \cdot p$ (follows from commutativity of real multiplication).

Note that most polynomials do *not* have a multiplicative inverse inside $\mathbb{R}[\lambda]$: there is no polynomial q such that $\lambda \cdot q = 1$. In a ring there is no requirement that multiplicative inverses exist, so this is perfectly consistent with the ring definition.

A complete `Polynomial.py` is provided in the homework directory; you do not need to write it, but you should read through it. It implements the full ring interface (`__add__`, `__mul__`, `__divmod__`, `__neg__`, `__pow__`, `__eq__`) as well as `real_roots`. Chapter 2 (Section 2.3) examines its OO design in detail. For rational arithmetic, Python's `fractions.Fraction` provides exact $\mathbb{Q}^{n\times n}$ entries; see the same section.

3.5.1 $\mathbb{R}^{n\times n}$ is a Non-Commutative Ring

The set $\mathbb{R}^{n\times n}$ of $n \times n$ real matrices forms a ring under $+$ and $\times$:

- $(\mathbb{R}^{n\times n}, +)$ is an Abelian group (Theorem 3.2.2).
- $(\mathbb{R}^{n\times n}, \times)$ is a monoid (Theorem 3.4.3).
- Distributivity: $A(B+C) = AB+AC$ and $(A+B)C = AC+BC$, which follow directly from Definition 3.3.1.

However, as we saw in Section 3.4.1, matrix multiplication is in general *not* commutative: $AB \neq BA$. Therefore $\mathbb{R}^{n\times n}$ is a *non-commutative ring* for $n \geq 2$.

3.5.2 Matrices over a Ring

In Section 3.1, we defined a matrix as a rectangular array of *numbers*, which was intentionally ambiguous, for what is a number? However, nothing in the definition of matrix addition or multiplication requires that the entries be real numbers. Addition uses $+$ and multiplication uses $+$ and $\times$ of the entries: both operations are available in any ring. We can therefore generalise:

Definition 3.5.2 (*matrix over a ring*)

Let R be a ring. An $n \times m$ matrix *over* R is a rectangular array with entries in R. The set of all such matrices is denoted $R^{n\times m}$; for square matrices, $R^{n\times n}$.

Theorem 3.5.3 ($R^{n\times n}$ *is a ring when* R *is a ring*)

If R is a ring, then $R^{n\times n}$ is also a ring (non-commutative for $n \geq 2$ even if R is commutative).

This generalisation is not merely theoretical. Useful matrix rings include:

- $\mathbb{Z}^{n\times n}$ — matrices of integers; Gaussian elimination over $\mathbb{Z}$ avoids fractions (see the Bareiss algorithm in Chapter 5).
- $\mathbb{Q}^{n\times n}$ — matrices of rationals; elimination is exact with no floating-point error.
- $\mathbb{R}[\lambda]^{n\times n}$ — matrices whose entries are polynomials in λ. This arises in Chapter 10 when forming $A - \lambda I$ to find eigenvalues.

Proposition 3.5.4 (*Matrix Arithmetic Identities*)

The following identities hold whenever the operations are defined, for matrices A, B, and C, and scalars s and t.

$$(A + B) + C = A + (B + C) \tag{3.11}$$
$$A + B = B + A \tag{3.12}$$
$$s(A + B) = sA + sB \tag{3.13}$$
$$(s + t)A = sA + tA \tag{3.14}$$
$$A + 0 = 0 + A = A \tag{3.15}$$
$$(A \times B) \times C = A \times (B \times C) \tag{3.16}$$
$$A \times (B + C) = A \times B + A \times C \tag{3.17}$$
$$(B + C) \times A = B \times A + C \times A \tag{3.18}$$
$$(sA) \times B = A \times (sB) = s(A \times B) \tag{3.19}$$

Proposition 3.5.5 (*Square Matrix Arithmetic Identities*)

The following identities hold for a *square* matrix A, scalars s and t, and natural numbers n and m.

$$A \times I = I \times A = A \tag{3.20}$$
$$A \times A^{-1} = A^{-1} \times A = I \qquad (A \text{ invertible}) \tag{3.21}$$
$$A^n \times A^m = A^m \times A^n \tag{3.22}$$

3.5.3 Vectors Do Not Form a Ring

The set of n-dimensional vectors forms an Abelian group under addition, but there is no natural closed multiplication on vectors that gives another vector of the same dimension. (The dot product returns a scalar, not a vector.) Therefore vectors do *not* form a ring.

3.6 Is the Multiplicative Monoid a Group?

$(\mathbb{R}^{n\times n}, \times)$ is a monoid. We now ask: is it a group? A group would require every element to have a multiplicative inverse. It turns out that $\mathbb{R}^{n\times n}$ is *not* a group — some matrices have no inverse.

When working with integers, rationals, real numbers and complex numbers an equation like $xy = 0$ means that either $x = 0$ or $y = 0$ (or perhaps both). However, this fails to be the case when working with matrices. Example 3.6.1 shows two matrices whose product is zero, despite the fact that neither of the factors is zero.

Example 3.6.1

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} (1)(0)+(0)(0) & (1)(0)+(0)(1) \\ (0)(0)+(0)(0) & (0)(0)+(0)(1) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Theorem 3.6.2 (*Divisors of Zero are not Invertible*)

If two non-zero, multiplicatively compatible matrices N and M are such that $NM = 0$, then N is not invertible.

Proof by Contradiction. Let $NM = 0$ and suppose $N \neq 0$, $M \neq 0$, and N^{-1} exists.

$$
\begin{aligned}
0 &= N^{-1} - N^{-1} \\
&= N^{-1}I - N^{-1} \\
&= N^{-1}(NN^{-1}) - N^{-1} \\
&= N^{-1}(NN^{-1} + NM) - N^{-1} && \text{because } NM = 0 \\
&= N^{-1}(N(N^{-1} + M)) - N^{-1} \\
&= (N^{-1}N)(N^{-1} + M) - N^{-1} && \text{by Theorem 3.4.2} \\
&= (I)(N^{-1} + M) - N^{-1} \\
&= N^{-1} + M - N^{-1} \\
&= M
\end{aligned}
$$

$0 = M$ is a contradiction of our assumption that $M \neq 0$.

Since zero divisors exist in $\mathbb{R}^{n\times n}$ and have no multiplicative inverse (Theorem 3.6.2), $(\mathbb{R}^{n\times n}, \times)$ is *not* a group. $\mathbb{R}^{n\times n}$ is a ring whose multiplicative monoid is not a group.

Nevertheless, some square matrices *do* have inverses.

Definition 3.6.3 (*invertible, singular*)

A square matrix for which an inverse exists is called *invertible* or *non-singular*. A square matrix which is not invertible is called *non-invertible* or *singular*.

In Example 3.3.2 we saw that $AB = I$. Since the product of the two matrices is the identity, we say the matrices are inverses of each other, and we use the notation $A^{-1} = B$ as well as $B^{-1} = A$.

Theorem 3.6.4 (*Inverse Matrix*)

If A is a square matrix, and if there exists a square matrix B such that $AB = I$, then there exists exactly one such B, and $AB = BA$.

Proof. Let A and B be matrices such that $AB = I$. We have:

$$\begin{aligned} A &= IA \\ &= (AB)A && \text{by substitution} \\ &= A(BA) && \text{by Theorem 3.4.2} \end{aligned}$$

Thus BA is an identity matrix. And from Theorem 3.4.1, the identity is unique. So $BA = I = AB$.

We saw in Section 3.4.1 that in general matrix multiplication is not commutative. However, in the special case of inverse matrices: a matrix commutes with its inverse.

Recall again, Example 3.3.2 where we saw that $AB = I$. Theorem 3.6.4 predicts that $BA = AB$. We verify this in Example 3.6.5.

Example 3.6.5 (*Matrix Multiplication*)

In this example, we multiply the matrices from Example 3.3.2 in

the opposite order.

$$A = \begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix}$$

$$AB = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$BA = \begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0+-5+6 & -12+20-8 & -8+10-2 \\ 0-6+6 & -15+24-8 & -10+12-2 \\ 0+9-3 & 24-36+12 & 16-18+3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Theorem 3.6.4 claims that if a square matrix is invertible, then the matrix commutes (under multiplication) with its inverse. However, the inverse of a matrix may not exist. There are many examples of singular matrices. An easy example is the zero matrix, another example is a matrix with a row or column of zeros. However, given an arbitrary matrix, it is computationally difficult to determine whether it is invertible.

3.7 Scaling

In Chapter 1 (Section 1.4) we defined a *scaling operation* on an Abelian group G as a function $\mathbb{R} \times G \to G$ satisfying four axioms. Component-wise scalar multiplication gives such an operation on $\mathbb{R}^{n \times m}$.

Theorem 3.7.1 (*Component-wise scaling satisfies all four axioms*)

$\mathbb{R}^{n \times m}$ equipped with component-wise scalar multiplication is an $\mathbb{R}$-scalable Abelian group.

Proof. Let $s, t \in \mathbb{R}$ and $A, B \in \mathbb{R}^{n \times m}$. We verify each axiom at entry (i, j).

1. $(1 \cdot A)_{ij} = 1 \cdot a_{ij} = a_{ij}$. ✓
2. $(sA + tA)_{ij} = sa_{ij} + ta_{ij} = (s + t)a_{ij} = ((s + t)A)_{ij}$. ✓
3. $(sA + sB)_{ij} = sa_{ij} + sb_{ij} = s(a_{ij} + b_{ij}) = (s(A + B))_{ij}$. ✓
4. $((st)A)_{ij} = (st)a_{ij} = s(ta_{ij}) = (s(tA))_{ij}$. ✓

Definition 3.7.2 (*real vector space*)

A *real vector space* is an $\mathbb{R}$-scalable Abelian group: an Abelian group $(V, +)$ equipped with an $\mathbb{R}$-scaling operation.

Therefore $\mathbb{R}^{n \times m}$ is a real vector space. We will study vector spaces in depth in Chapter 7.

3.8 Further Matrix Topics

3.8.1 Transpose

Definition 3.8.1 (*transpose*)

The *transpose* of an $m \times n$ matrix A is the $n \times m$ matrix A^T defined by

$$(A^T)_{i,j} = A_{j,i}\,.$$

A square matrix A is called *symmetric* if $A^T = A$.

Example 3.8.2 (*Transpose*)

Suppose

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

To form A^T we build a matrix whose first column is the first row of A, the second column of A^T is the second row of A, etc.

$$A^T = \begin{bmatrix} 0 & 1 & -1 \\ -3 & -4 & 4 \\ -2 & -2 & 1 \\ 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

3.8.2 Matrix-Vector Multiplication

There is an important special case of matrix multiplication which occurs often in practice. We can embed an n-dimensional vector v either as an $n \times 1$ column matrix or as a $1 \times n$ row matrix. First an example:

Example 3.8.3

$$
\begin{aligned}
Av &= \begin{bmatrix} 1 & 2 & -1 & 1 \\ 2 & -1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 0 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} (1)(2)+(2)(-1)+(-1)(0)+(1)(1) \\ (2)(2)+(-1)(-1)+(1)(0)+(0)(1) \\ (0)(2)+(0)(-1)+(1)(0)+(1)(1) \\ (1)(2)+(-1)(-1)+(-1)(0)+(-1)(1) \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 1 \\ 2 \end{bmatrix}
\end{aligned}
$$

When v is embedded as an $n \times 1$ column matrix, the product Av via (3.3) reduces to

$$c_i = \sum_{\ell=1}^{k} a_{i\ell} v_\ell \,. \tag{3.23}$$

Although it is less common, a vector v can also be embedded as a $1 \times n$ row matrix and multiplied on the left of a matrix, provided the matrix has n rows. We write this as $v^T A$, where v^T indicates that v is treated as a $1 \times n$ row rather than an $n \times 1$ column.

First an example:

Example 3.8.4

$$
\begin{aligned}
v^T A &= \begin{bmatrix} 2 \\ -2 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} 1 & 2 & -1 & 1 \\ 2 & -1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & -1 & -1 \end{bmatrix} \\
&= \begin{bmatrix} 2 & -2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 & 1 \\ 2 & -1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & -1 & -1 \end{bmatrix} \\
&= \begin{bmatrix} (2)(1)+(-2)(2)+(0)(0)+(1)(1) \\ (2)(2)+(-2)(-1)+(0)(0)+(1)(-1) \\ (2)(-1)+(-2)(1)+(0)(1)+(1)(-1) \\ (2)(1)+(-2)(0)+(0)(1)+(1)(-1) \end{bmatrix}^T \\
&= \begin{bmatrix} 2-4+0+1 \\ 4+2+0-1 \\ -2-2+0-1 \\ 2+0+0-1 \end{bmatrix}^T \\
&= \begin{bmatrix} -1 & 5 & -5 & 1 \end{bmatrix}
\end{aligned}
$$

When v is embedded as a $1 \times n$ row matrix, the product $v^T B$ via (3.3) reduces to

$$c_j = \sum_{\ell=1}^{k} v_\ell b_{\ell j} \,. \tag{3.24}$$

3.8.3 Matrix as Linear Transform

We may think of matrix-vector multiplication as a linear transformation.

Theorem 3.8.5 (*Matrix as linear transform*)

If A is an $n \times n$ matrix and u and v are $n-$vectors, and $\alpha \in \mathbb{R}$, then

$$A(u+v) = Au + Av \tag{3.25}$$
$$A(\alpha u) = \alpha Au \tag{3.26}$$

From Theorem 3.8.5 we see that any set, S of vectors can be transformed to a new set of vectors by applying the *transform*, A, to each of them. We may think of such a transformation is a transformation of the set of all vectors of dimension n. Such a transformation might reflect, rotate, stretch, or compress the vectors. The exact behavior of this transformation depends on the matrix itself.

We will explore more about this in Section 4.1.

3.8.4 Distance between Matrices

In Section 1.6, we defined a distance function for vectors. In this section we define a distance function for matrices.

As a reminder, a distance function satisfies several axioms:

1. $d(u,v) = d(v,u)$ (symmetric)
2. $d(u,v) \geq 0$
3. $d(u,v) = 0 \iff u = v$
4. $d(u,v) + d(v,w) \geq d(u,w)$ (triangle inequality)

In order to test matrix related functions, we will need a way to test *almost equal*. We will do this by defining a distance function as follows for a matrix of size $m \times n$. Let

Definition 3.8.6 (*matrix distance*)

If, $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$, and $B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$, 

then the distance between A and B, is given by

$$d(A, B) = \max_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} |a_{ij} - b_{ij}| \tag{3.27}$$

3.8.5 Exponentiation

In this section we talk about how to compute integer powers of a matrix.

How many operations are needed to compute a matrix A raised to a whole number power? For example $A^5 = A \times A \times A \times A \times A$. Clearly such a multiplication can be achieved by evoking the matrix multiply function $n - 1$ times, each of which has cubic complexity. But can it be done with fewer matrix multiplications? 

An insight comes by realizing that $A^5 = \left(A^2\right)^2 \times A$. We can compute A^2 with one call to `matrix-multiply`, then we can compute $\left(A^2\right)^2$ with a second call, and finally $\left(A^2\right)^2 \times A$ with a third call. Notice that $3 < n - 1$.

In general Equation (3.28) specifies how to raise a matrix, A, to a whole number power.

Definition 3.8.7 (*matrix exponentiation*)

If A is a square matrix, and p is 0 or a positive integer, then A 

raised to the p power is

$$A^p = \begin{cases} I & \text{if } p = 0 \\ A\ A^{p-1} & \text{if } p \text{ odd} \\ \left(A^2\right)^{\frac{p}{2}} & \text{if } p \text{ even} \end{cases} \tag{3.28}$$

Even though (3.28) is correct, it is not the best formula to use for implementing matrix exponentiation programmatically. The problem is that computing the identity, I, requires memory allocation, and since $A^1 = A$, we should be able to compute A^p when $p = 1$ without any additional allocation; *i.e.*, if $p = 1$ we should simply evaluate to A itself. Thus Equation (3.29) is a better model to use for programmatic purpose.

$$A^p = \begin{cases} I & \text{if } p = 0 \\ A & \text{if } p = 1 \\ A\ A^{p-1} & \text{if } p \text{ odd} \\ \left(A^2\right)^{\frac{p}{2}} & \text{if } p \text{ even} \end{cases} \tag{3.29}$$

We can also extend the matrix power function to negative powers. If we understand A^{-1} to mean the inverse of the matrix (Section A.4), then we can extend (3.29) to (3.30).

$$A^p = \begin{cases} I & \text{if } p = 0 \\ A & \text{if } p = 1 \\ \left(A^{-1}\right)^{-p} & \text{if } p < 0 \\ A\ A^{p-1} & \text{if } p \text{ odd} \\ \left(A^2\right)^{\frac{p}{2}} & \text{if } p \text{ even} \end{cases} \tag{3.30}$$

Let's look at an example application of matrix exponentiation. Suppose we have a social network of 6 people (numbered 1 through 6), some

of whom know each other and some of whom don't. We may represent the knowledge graph by placing a 1 in row k column j if person k knows person j, and a 0 otherwise. If we assume that person j knows person k whenever person k knows person j then this matrix is symmetric. However, the matrix might be asymmetric. For example, maybe Jean Dupont knows Emmanuel Macron, but Emmanuel Macron may not remember exactly who Jean Dupont is. Thus perhaps there is a 1 in row j column k but a zero in row k column j.

Suppose the matrix looks as in (3.31). Notice this matrix has 1's along the main diagonal, because we assume that every person knows himself.

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.31}$$

We see from (3.31), that person 3 does not know person 6. We see this because there is a 0 in row 3 column 6. However, does person 3 know someone who knows 6, or does person 3 know someone who knows someone who ...? If so, what is the length of the shortest string of people which completes this acquaintance. If we look at the sequence $A^2, A^3, \ldots, A^k$, we see

$$A^1 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \qquad A^2 = \begin{bmatrix} 1 & 2 & 1 & 0 & 1 & 0 \\ 2 & 2 & 2 & 0 & 2 & 1 \\ 2 & 1 & 1 & 0 & 0 & 0 \\ 2 & 3 & 1 & 1 & 1 & 0 \\ 2 & 3 & 2 & 1 & 2 & 2 \\ 2 & 1 & 2 & 2 & 0 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 3 & 4 & 3 & 0 & 3 & 1 \\ 6 & 6 & 5 & 1 & 4 & 3 \\ 3 & 3 & 2 & 0 & 1 & 0 \\ 5 & 7 & 4 & 1 & 4 & 1 \\ 7 & 8 & 7 & 3 & 5 & 4 \\ 6 & 5 & 4 & 3 & 1 & 1 \end{bmatrix} \qquad A^4 = \begin{bmatrix} 9 & 10 & 8 & 1 & 7 & 4 \\ 16 & 17 & 14 & 4 & 10 & 7 \\ 6 & 7 & 5 & 0 & 4 & 1 \\ 14 & 17 & 12 & 2 & 11 & 5 \\ 22 & 23 & 19 & 7 & 13 & 9 \\ 14 & 15 & 10 & 4 & 6 & 2 \end{bmatrix}$$

The matrix A^1 indicates how paths of length 1 (containing two people) connect person j with person k. A^2 indicates how paths of length 2 (containing three people) connect person j with person k. A^3 indicates how paths of length 3 (containing 4 people) connect person j with person k. We see that the in A^1, A^2, and A^3, there is a 0 in row 3 column 6. However, in A^4 there is a 1 in row 3 column 6. This means there is exactly 1 string of 5 people (including person 3 and person 6) who form an acquaintance string. *I.e.*, there exist persons α, β, γ so that 3 knows α, α knows β, β knows γ, and γ knows 6.

3.8.6 Complexity of Matrix Multiplication

If we construct an algorithm based directly on Definition 3.3.1 we see that we must compute $n \times m$ entries in the product matrix, and to compute each such entry we must perform k multiplications and $k-1$ additions. In the case that we are multiplying two square matrices, then $n = m = k$, so the complexity of matrix multiplication is at least

$O(n^3)$, assuming the multiplication and addition of numbers is constant. By $O(n^3)$ we simply mean that to multiply two square matrices of dimension, n, we must perform at least n^3 operations. As the dimension of the matrix grows linearly, the work needed to multiply the matrices grows cubicly. There are more efficient algorithms which we will not study in this course. The curious student might take a look at the *Strassen* algorithm.

The complexities of (3.23) and (3.24) are both $O(n^2)$ assuming the matrix has dimension $n \times n$, because to compute each of the n components in the resulting vector, we must perform n multiplications and $n - 1$ additions.

3.8.7 Strassen Matrix Multiplication

Given a two square matrices whose dimension is a power of 2 (4×4, 8×8, ..., $2^k \times 2^k$) we may multiply the matrices by partitioning them each into a 2×2 matrix where each entry is another matrix, this time with dimension $2^{k-1} \times 2^{k-1}$. Then those 2×2 matrices can be multiplied by the normal matrix multiplication rules.

This is precisely an instance of the construction in Section 3.5: since $\mathbb{R}^{(2^{k-1})\times(2^{k-1})}$ is itself a ring, we may form matrices whose entries come from that ring. A 4×4 real matrix, viewed as a 2×2 matrix of 2×2 blocks, is an element of $(\mathbb{R}^{2\times 2})^{2\times 2}$. The Strassen algorithm works because the 2×2 matrix multiplication formula (Theorem 3.8.9) requires only the ring operations $+$ and $\times$ of its entries — no commutativity of entry multiplication is assumed — so it applies equally whether the entries are real numbers or matrices.

Example 3.8.8 (4×4 ***Matrix Multiplication***)

$$
\begin{aligned}
AB &= \begin{bmatrix} 1 & 2 & 3 & -1 \\ 2 & 1 & -1 & 0 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & -1 & 0 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 2 & 0 \\ 1 & 2 & 3 & -1 \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} & \begin{bmatrix} 3 & -1 \\ -1 & 0 \end{bmatrix} \\ \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix} & \begin{bmatrix} -1 & 0 \\ 1 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} & \begin{bmatrix} 2 & 1 \\ 3 & -1 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} && (3.32) \\
&= \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} && (3.33)
\end{aligned}
$$

Where

$$
\begin{aligned}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} && (3.34) \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} && (3.35) \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} && (3.36) \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22} && (3.37)
\end{aligned}
$$

Terms such as $A_{11}B_{11}$, $A_{12}B_{21}$, ..., $A_{12}B_{22}$ are matrix multiplications. Each has half as many rows and half as many columns as A and B.

To compute AB using the traditional multiplication algorithm, from Section 3.3, requires us to compute $4 \times 4 = 16$ matrix entries, each requiring 4 integer multiplications; *i.e.* $4 \times 4 \times 4 = 4^3 = 64$ multiplications. Compare that with the number of multiplications using (3.33), in which we 4 times make a computation such as $A_{11}B_{11} + A_{12}B_{21}$,

which in turn is 2 matrix multiplications of 2×2 matrices. Each 2×2 matrix multiplication requires $2^3 = 8$ multiplications. So in total this is $4 \times 2 \times 8 = 64$; *i.e.* exactly as many multiplications as the standard Section 3.3 algorithm.

The refactoring as in (3.32) of $AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ may not seem, at first glance, as very useful, at least not when using the standard matrix multiplication algorithm. The fact that (3.33) requires 8 matrix multiplications means that this algorithm is just as slow as the standard algorithm. However, surprisingly, there is a way to multiply two 2×2 matrices using only 7 multiplications. This trick applies both to 2×2 matrices of numbers, as well as 2×2 matrices of matrices. We first compute the following seven values, which require a single multiplication each.

Theorem 3.8.9 (*Strassen Multiply*)

If we define $M_1, \ldots, M_7$ as follows.

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})\,; \end{aligned}$$

then we can now compute AB with no additional multiplications.

$$AB = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix} \tag{3.38}$$

$$= \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{3.39}$$

Proof. It is straightforward to verify 3.38 with a bit of algebra by verifying the following. We simply substitute for each M_j, distribute multiplication over addition (and subtraction), and remark that many of the terms cancel.

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) \\
&\quad + A_{22}(B_{21} - B_{11}) - (A_{11} + A_{12})B_{22} \\
&\quad + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + \\
&\quad + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} \\
&\quad + A_{12}B_{21} - A_{22}B_{21} + A_{12}B_{22} - A_{22}B_{22} \\
&= A_{11}B_{11} + A_{12}B_{21}
\end{aligned}$$

$$\begin{aligned}
C_{12} &= M_3 + M_5 \\
&= A_{11}(B_{12} - B_{22}) + (A_{11} + A_{12})B_{22} \\
&= A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{22} + A_{12}B_{22} \\
&= A_{11}B_{12} + A_{12}B_{22}
\end{aligned}$$

$$\begin{aligned}
C_{21} &= M_2 + M_4 \\
&= (A_{21} + A_{22})B_{11} + A_{22}(B_{21} - B_{11}) \\
&= A_{21}B_{11} + A_{22}B_{11} + A_{22}B_{21} - A_{22}B_{11} \\
&= A_{21}B_{11} + A_{22}B_{21}
\end{aligned}$$

$$
\begin{aligned}
C_{22} &= M_1 - M_2 + M_3 + M_6 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) \\
&\quad - (A_{21} + A_{22})B_{11} + A_{11}(B_{12} - B_{22}) \\
&\quad + (A_{21} - A_{11})(B_{11} + B_{12}) \\
&= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\
&\quad - A_{21}B_{11} - A_{22}B_{11} + A_{11}B_{12} - A_{11}B_{22} \\
&\quad + A_{21}B_{11} + A_{21}B_{12} - A_{11}B_{11} - A_{11}B_{12} \\
&= A_{21}B_{12} + A_{22}B_{22}
\end{aligned}
$$

Using Theorem 3.8.9 we can multiply the two matrices in Example 3.8.8, using 7 multiplications of 2×2 matrix, each of which requires 7 multiplications of numbers. *I.e.*, 4×4 matrix multiplication does not need 64 multiplications, but rather $7 \times 7 = 49$ multiplications.

The Strassen multiplication algorithm is not an optimization for small matrices because of the excessive number of additions. Notice in Theorem 3.8.9, that 18 additions are necessary along with the 7 multiplications. In the recursive case (4×4 matrix case, Example 3.8.8) there are 18 matrix additions, each of which require 4 numerical additions, and 7 multiplications of 2×2 matrices, each of which requires 18 numerical additions. Thus $7 \times 18 + 18 \times 4 = 198$ additions are required by the Strassen algorithm to perform a single 4×4 matrix multiplication, compared with $4 \times 4 \times 3 = 48$ additions for the standard algorithm.

In fact it is not until $2^{14} \times 2^{14}$ when the number of additions in the Strassen algorithm becomes fewer than that in the standard algorithm.

If the number of additions required for a $2^{n-1} \times 2^{n-1}$ matrix is a_{n-1}, then the number of additions required for $2^n \times 2^n$ is $a_n = 7a_{n-1} + 18\left(\frac{2^n}{2}\right)^2$; whereas the number of additions and multiplications needed for the naive multiplication are $2^n \times 2^n \times (2^n - 1)$ and

$2^n \times 2^n \times 2^n$ respectively.

In the following table for each size of a square matrix ($2^n \times 2^n$) we list the number of multiplications necessary for the Strassen and naive algorithms as well as the number of additions necessary for each. The rightmost column shows the ratio of the sum, number of multiplications plus number of additions of Strassen divided by naive. When this ratio is larger than unity, then the naive algorithms has fewer operations; when the ratio is less than unity, then the Strassen algorithm requires fewer operations.

dim	**Strassen mult.**	**Naive mult.**	**Strassen additions**	**Naive additions**	**ratio**
$2^n \times 2^n$	a_n				
2×2	7	8	18	4	4.5
4×4	49	64	198	48	2.21
8×8	3.4×10^2	512	1674	4.5×10^2	2.10
16×16	2.4×10^3	4.1×10^3	1.3×10^4	3.8×10^3	1.92
32×32	1.7×10^4	3.3×10^4	9.5×10^4	3.2×10^4	1.73
64×64	1.2×10^5	2.6×10^5	6.8×10^5	2.6×10^5	1.54
$2^7 \times 2^7$	8.2×10^5	2.1×10^6	4.8×10^6	2.1×10^6	1.36
$2^8 \times 2^8$	5.8×10^6	1.7×10^7	3.4×10^7	1.7×10^7	1.19
$2^9 \times 2^9$	4.0×10^7	1.3×10^8	2.4×10^8	1.3×10^8	1.05
$2^{10} \times 2^{10}$	2.8×10^8	1.1×10^9	1.7×10^9	1.1×10^9	0.92
$2^{11} \times 2^{11}$	2.0×10^9	8.6×10^9	1.2×10^{10}	8.6×10^9	0.80
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2^{19} \times 2^{19}$	1.1×10^{16}	1.4×10^{17}	6.8×10^{16}	1.4×10^{17}	0.28
$2^{20} \times 2^{20}$	8.0×10^{16}	1.2×10^{18}	4.8×10^{17}	1.2×10^{18}	0.24

We notice that somewhere between 512×512 and 1024×1024 is the crossover point. This result should not be interpreted too literally, because we are making several dubious assumptions.

- We are assuming that floating point multiplication and floating point addition have the same cost.

- We are assuming we can partition a $2n \times 2n$ matrix into four $n \times n$ (as in (3.39)) with no cost, and after the recursive call the resulting 4 matrices, each $n \times n$ can be stitched back together into a $2n \times 2n$ matrix free of cost.

- If the matrix dimension is not even, we cannot divide it into four square matrices (as in (3.39)). Instead before each recursive call, we may need to pad an extra row and column of zeros, as a pre-processing step, and un-pad them after the recursive call returns. We are also ignoring this cost in the approximation.

3.9 Matrices in Python

The algebraic structures established in this chapter — Abelian group, ring, vector space — each correspond directly to operations on the `Matrix` and `SqMatrix` classes. Addition and scaling implement the group and vector-space structure; multiplication implements the monoid and ring structure; the `distance` method supports the metric-space view introduced in Chapter 6. In this section we present the implementation of each operation in turn. The full class skeletons are given here; the method bodies are left as homework exercises.

3.9.1 Rectangular Matrices

Recall from Section 2.2 that we represented a vector in Python by a class named `Vector`. We will use a slightly more advanced technique to represent a matrix. In fact we will use the class `Matrix` to represent rectangular matrices; an instance, `M`, of class `Matrix` will have two instance variables `M.rows` and `M.cols` indicating the number of rows and columns the matrix has. As we will see in the upcoming sections, certain operations on matrices require certain kinds of *compatibility* of

the number of rows and columns of one matrix with respect to another matrix.

On the other hand, certain kind of matrix operations only make sense on square matrices, *i.e.*, a matrix whose number of rows is the same as its number of columns. In this special case, the matrix will be represented (and implemented) by a subclass of `Matrix` called `SqMatrix`. The `__init__` magic methods of `Matrix` and `SqMatrix` will control which class gets instantiated.

First we look at the `Matrix` class. Here we omit many of the methods and all the method bodies. Some of the implementations (bodies) of the methods are given in the class definition; however, some must be implemented by the student as homework exercises.

```
T = TypeVar('T')

class Matrix(Generic[T]):

    def __init__(self, content):        # provided

    def __repr__(self):

    def __getitem__(self, items):

    def __eq__(self, other) -> bool:

    def __neg__(self) -> 'Matrix':

    def __add__(self, other: 'Matrix') -> 'Matrix':

    def __sub__(self, other: 'Matrix') -> 'Matrix':

    def __mul__(self, other) -> 'Matrix':

    def __rmul__(self, other) -> 'Matrix':

    @staticmethod
    def tabulate(rows: int, cols: int, f) -> 'Matrix':

    @staticmethod
    def random(rows: int, cols: int, randomizer) -> 'Matrix':

```

```
29      @staticmethod
30      def zero(rows: int, cols: int) -> 'Matrix':
31
32      def scale(self, s: T) -> 'Matrix':
33
34      def transpose(self) -> 'Matrix':
35
36      def row_vec(self, r: int) -> Vector:
37
38      def col_vec(self, c: int) -> Vector:
39
40      def norm(self) -> T:
41
42      def distance(self, other) -> T:
```

The initialization magic method is given as follows. This method is called by Python when `Matrix(...)` the constructor, is called as a function. The method sets `self.rows` and `self.cols` by examining the given content. It also asserts that all the rows have the same length, and asserts that there is at least one row and one column. If necessary, it converts content to a tuple of tuples, to ensure that the data is immutable. Most notably, if number of rows is the same as number of columns, the instance gets destructively converted to an instance of `SqMatrix`. However, the code explicitly avoids calling the `__init__`, as that would cause an infinite loop.

```
1       def __init__(self, content):
2           assert len(content) > 0
3           assert all(len(row) > 0 for row in content)
4           self.rows = len(content)
5           self.cols = len(content[0])
6           self.content = tuple(tuple(row) for row in content)
7           assert all(self.cols == len(row) for row in content)
8           if self.rows == self.cols:
9               from matrix.SqMatrix import SqMatrix
10              self.__class__ = SqMatrix
11              self.dim = self.rows
```

The `__repr__`, `__getitem__`, and `__eq__` magic methods are left as an exercise for the student. However, there are examples which might help in the `Vector` class definition as well as `SqMatrix`.

As was the case in the `Vector` class, the `Matrix` class also has a static method named `tabulate`. In the `Vector` case `tabulate` took two arguments, a dimension and a unary function. In this case (`Matrix`), `tabulate` takes three arguments, `rows`, `cols`, and a binary function `f`. The `tabulate` method will call the function, `f`, as `f(i,j)` for each `i in range(rows)` and for each `j in range(cols)`, and the matrix will be populated with the respective return values. The result is illustrated here.

```
m = Matrix.tabulate(17,19, lambda i, j: i*i + j)
# now m is a matrix with 17 rows and 19 columns

print(m[12][8])  # will print 152 because 152 = 12*12+8
```

The `tabulate` method is extremely useful in defining other matrix operations, because it allows us to directly translate the mathematical definition into the definition in the Python code. A trivial example is the definition of the `random` method, which allocates a `Matrix` of the requested size and populates it by calling the given `randomizer` function $cols \times rows$ many times.

```
    @staticmethod
    def random(rows: int, cols: int, randomizer) -> 'Matrix[T]':
      return Matrix.tabulate(rows, cols, lambda r, c: randomizer())
```

3.9.2 Square Matrices

A square matrix is a special case of a rectangular matrix. We exploit this fact by defining a subclass of `Matrix` called `SqMatrix`. Whenever a method is called on an instance of `SqMatrix` but the method is missing in the `SqMatrix` definition, then the method is invoked in its superclass `Matrix` if found. We say, therefore, that methods in `SqMatrix` override those methods in `Matrix` if they exist in both places. However, in some cases a method is defined in `SqMatrix` for which there is no like-named method in `Matrix`. Python uses the same syntax for both. *I.e.*, there is

no syntactical indicator that a method is a new method or an overriding method.

The following is an excerpt of the `SqMatrix` definition. Some methods are omitted, and all bodies are omitted.

```
class SqMatrix(Matrix):
    def __init__(self, content):

    def __repr__(self):

    @staticmethod
    def random(dim, randomizer):

    @staticmethod
    def identity(dim):

    @staticmethod
    def zero(dim):

    @staticmethod
    def tabulate(dim, f):

    @staticmethod
    def diagonal(entries):

    def power(b, p):
```

Let's look at the initialization function for `SqMatrix`. The `__init__` magic method expects a sequence (probably a list) of sequences which define a square matrix. Otherwise an exception is thrown. If the given array of numbers is square, then `self.dim` is set to this common dimension, and `self.content` is set to a tuple of tuples containing the 2-dimensional array of numbers. Finally `super().__init__(content)` is called which causes the code in the `__init__` method in the class `Matrix` to be executed, which in turn sets the `rows` and `cols` instance variables.

```
    def __init__(self, content):
        self.dim = len(content)
        self.content = tuple(tuple(row) for row in content)
        assert all(len(row) == self.dim for row in content)
```

```
        super().__init__(content)
```

Next is the `__repr__` magic method, which attempts to print each row of the matrix on a different line.

```
    def __repr__(self):
        lines = ",\n        ".join(f"{row}" for row in self.content)
        return f"SqMatrix(({lines}))"
```

Finally we look at the `random` method because the pattern here is important for several other methods. Because a square matrix has a so-called `dimension`, many methods defined on `SqMatrix` have one fewer argument than the corresponding method on `Matrix`. So these methods (in `SqMatrix`) simply call the corresponding method in `Matrix` but with a duplicated argument.

Here we explicitly call `Matrix.random`, but we must pass `dim` twice, because `Matrix.random` is expecting `rows, cols` as the first two arguments. The `randomizer` can be passed as-is, because it is a 0-ary function.

```
    @staticmethod
    def random(dim, randomizer):
        return Matrix.random(dim, dim, randomizer)
```

Another method which follows this pattern is `SqMatrix.tabulate` which is a shallow wrapper around `Matrix.tabulate`. `SqMatrix.tabulate` takes two arguments, while `Matrix.tabulate` takes 3.

```
    @staticmethod
    def tabulate(dim, f):
        return Matrix.tabulate(dim, dim, f)
```

The identity matrix is constructed using `tabulate` with a function that returns 1 on the diagonal and 0 elsewhere. An equivalent formulation uses the `diagonal` method.

```
class SqMatrix(Matrix):
    ...

    @staticmethod
```

```
    def diagonal(entries):
        return SqMatrix.tabulate(len(entries),
                                 lambda r, c:
                                   entries[r] if r == c else 0)

    @staticmethod
    def identity(dim: int):
        return SqMatrix.diagonal([1]*dim)
```

3.9.3 Matrix Addition

When Python evaluates `a + b`, what happens behind the scenes is that it calls the `__add__` method on the object on the left-hand side of the `+` sign. If that object has type precisely `Matrix` then the `__add__` in the `Matrix` class is called. However, if the matrix is a square matrix, then the object will have class `SqMatrix`. In such a case the system will find no method of that name in the `SqMatrix` class, so the system will search for the method in the super class, `Matrix`. The method will be found and called. This means the same method implements addition for `Matrix` and also for `SqMatrix`.

We can implement the addition operation with a call to `tabulate` taking advantage of Equation (3.1).

```
    def __add__(a, b):
      assert isinstance(b, Matrix)
      assert a.rows == b.rows
      assert a.cols == b.cols
      return Matrix.tabulate(a.rows,
                             a.cols,
                             lambda i, j: a[i][j] + b[i][j])
```

3.9.4 Matrix Scaling and Subtraction

In Python we can define scaling and subtraction using the above definitions. The `__mul__` method handles `M * s` (scalar on the right); a sep-

arate `__rmul__` method handles `s * M` (scalar on the left). The `__mul__` method and its cases are discussed in more depth in Section 3.9.5.

```
class Matrix(...):

    def scale(m, s):
      return Matrix.tabulate(m.rows, m.cols,
                             lambda i, j: s * m[i][j])

    def __sub__(a, b):
      return a + b * -1

    def __mul__(a, b):
      if isinstance(b, Matrix):
        ...

      if isinstance(b, Vector):
        ...

      if isinstance(b, numbers.Number):
        return a.scale(b)

      raise RuntimeError(f"cannot multiply {a} by {b}")

    def __rmul__(a, b):
      if isinstance(b, numbers.Number):
        return a.scale(b)
      return NotImplemented
```

3.9.5 Matrix Multiplication

There are four cases we need to consider when implementing matrix multiplication in Python. When `a` is of type `Matrix` and `b` is any type at all, Python evaluates `a * b` by calling the `Matrix` implementation of `__mul__`. That function examines the type of the right-hand operand.

We can handle four different types here:

- If `b` is a `Matrix`
- If `b` is a `Vector`

- If `b` is a scalar (any `numbers.Number`, including `int` and `float`)
- If a scalar appears on the *left*, *e.g.*, `3 * a`, Python first tries the scalar's own `__mul__`, which does not know about matrices and returns `NotImplemented`. Python then falls back to `a.__rmul__(3)`.

Because of Equation (3.3), we can use the `tabulate` method to make the `__mul__` method straightforward to write. We just have to translate Equation (3.3) into a Python `lambda` function.

```
import numbers

class Matrix(...):

    def __mul__(a, b):
      if isinstance(b, Matrix):
        assert a.cols == b.rows
        return Matrix.tabulate(a.rows, b.cols,
                               lambda i, j:
                                 sum(a[i][k] * b[k][j]
                                     for k in range(a.cols)))

      if isinstance(b, Vector):
        ...

      if isinstance(b, numbers.Number):
        return a.scale(b)

      raise RuntimeError(f"cannot multiply {a} by {b}")

    def __rmul__(a, b):
      if isinstance(b, numbers.Number):
        return a.scale(b)
      return NotImplemented
```

The `__rmul__` method is Python's mechanism for handling expressions where the matrix appears on the right of the `*` operator. When Python evaluates `3 * m`, it first asks the integer `3` whether it knows how to multiply itself by a matrix; it does not, so Python calls `m.__rmul__(3)` instead. Returning `NotImplemented` (rather than raising an exception) for non-scalar arguments lets Python produce a clean

`TypeError` when the multiplication is genuinely undefined.

3.9.6 Matrix-Vector Multiplication

We can extend the code for `__mul__` defined in Section 3.9.5 to also accommodate `m*v` where `m` is a `Matrix` and `v` is a `Vector`.

```
class Matrix(...):

    def __mul__(a, b):
      if isinstance(b, Matrix):
        ...

      if isinstance(b, Vector):
        assert a.cols == b.dim
        return Vector.tabulate(a.rows,
                               lambda i: sum(a[i][k] * b[k]
                                 for k in range(a.cols)))

      if isinstance(b, numbers.Number):
        ...

      raise RuntimeError(f"cannot multiply {a} by {b}")
```

3.9.7 Matrix Distance

In Python the `distance` and `norm` methods can be implemented as follows.

```
class Matrix(...):

    def norm(a):
        return max(abs(a[i][j])
                   for i in range(a.rows)
                   for j in range(a.cols))

    def distance(a, b):
        assert a.rows == b.rows
        assert a.cols == b.cols
        return (a - b).norm()
```

3.10 Programming Exercises

1. Working with Arrays
    - working with references, array of array-references
2. Object Orientation
    - emulating multiple dispatch: `isinstance`
3. tabulate
4. functions which work on generators, `max`, `sum`, `reduce`
5. recursion

3.10.1 Classwork

1. Begin implementation of `SqMatrix.py`
2. Implement `fast-power` (generic)
3. Start `Ring.py`: discuss ring axioms; verify $\mathbb{R}^{n\times n}$ is a non-commutative ring

3.10.2 Homework

- Implement `Matrix.py`
    - `__repr__`
    - `tabulate`
    - `random`
    - `zero`

  - `__getitem__`
  - `__eq__`
  - `__add__`
  - `__sub__`
  - `scale`
  - `__mul__`
  - `transpose`
  - `row_vec`
  - `col_vec`
  - `norm`
  - `distance`

- Implement `SqMatrix.py`
  - `random`
  - `identity`
  - `zero`
  - `tabulate`
  - `power`

- Test with:
  - `Matrix_test.py`
  - `SqMatrix_test.py`

- Implement `Ring.py` (see Section 3.5.1). You are given a generator function, a membership predicate, two operations $+$ and $\cdot$, and identity elements 0 and 1. Verify each ring axiom by testing a large number of randomly selected elements.

  - `is_ring` — tests closure, associativity, identity, additive inverse, and distributivity for both operations
  - `is_commutative_ring` — additionally tests that multiplication is commutative

  Use $\mathbb{R}^{n\times n}$ (square matrices under $+$ and $\cdot$) as the primary test case; verify it is a ring but *not* a commutative ring.

- Test with:
  - `Ring_test.py`

Chapter 4

Determinant

Chapter Objectives

- Compute the determinant of a square matrix by Laplace (cofactor) expansion.
- State and apply the invertibility criterion: a square matrix has an inverse if and only if its determinant is non-zero.
- Interpret the determinant geometrically as the signed scaling factor of area or volume under the associated linear transformation.
- Identify and apply the three elementary row operations (swap, scale, row combination) and state how each one affects the determinant.
- Implement determinant computation and row operations in Python.
- Recognise that Laplacian expansion requires only ring operations ($+$, $\times$, negation) and therefore works for matrices whose entries come from any commutative ring, such as $\mathbb{Q}$ or $\mathbb{R}[\lambda]$.

In this chapter we will present a concept called *determinant*. The *determinant* is a scalar quantity which, in some sense, characterizes a

square matrix. It is not difficult to understand how to compute a determinant (Definition 4.4.7), but we do not yet have enough background to have a good intuition of what a *determinant* really is.

Arguably, the most important feature of the determinant is that if the determinant of a matrix is zero, then the matrix has no inverse.

Theorem 4.0.1 (*Determinant of singular matrix*)

If M is a square matrix, then there exists a matrix L such that $ML = I$ if and only if $\det(M) \neq 0$.

Sometimes $\det(A)$ is denoted as $|A|$. However, this is not an absolute value nor a norm, as the value may be positive or negative and does not obey the triangle inequality.

4.1 Matrix as Transformation

Given a 2×2 matrix and a set of (x, y) points in the plane, we may generate a new set of points in the plane by multiplying each point (as a column vector) by the matrix on the left. This can be seen as a single matrix multiply if the column vectors are adjoined, left-to-right.

$$\begin{bmatrix} 1.1 & 1.5 \\ -2.1 & 2.7 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \\ y_1 & y_2 & y_3 & \cdots & y_n \end{bmatrix} = \begin{bmatrix} x'_1 & x'_2 & x'_3 & \cdots & x'_n \\ y'_1 & y'_2 & y'_3 & \cdots & y'_n \end{bmatrix}$$

Not only can a matrix be seen as a transformation of a set of points, it can also be thought of as a transformation of the set of *all points* in the plane, *i.e.* a transformation of the plane itself. For example, the matrix, $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, reflects the plane about the line $y = x$. The matrix, $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, maps the plane to itself. The matrix, $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, rotates the plane counter-clockwise through an angle, θ.

The actual values chosen in the transformation matrix, R_θ, determine different characteristics of the transformation. If the determinant is non-zero, then each point in the plane is mapped one-to-one to another point in the plane. *I.e.*, if the determinant is non-zero then the transformation is an isomorphism.

However, if the determinant is zero, then all points are mapped to a single line, or to a single point.

4.2 Composition of Transforms

If matrix $R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ rotates the plane by θ, and $F = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ reflects the plane about the line $y = x$, how can we reflect the plane and then rotate it? For any point $z = \begin{bmatrix} x \\ y \end{bmatrix}$, we have the point reflected is Fz, and that point rotated is $R_\theta(Fz)$. But we know that matrix multiplication is associative so $R_\theta(Fz) = (R_\theta F)z$. The fact that such multiplication is associative allows us to represent as a single operator the matrix which first reflects and then rotates as:

$$\begin{aligned} R_\theta F &= \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -\sin\theta & \cos\theta \\ \cos\theta & \sin\theta \end{bmatrix} \end{aligned}$$

If we want to rotate first, and then reflect, we can simply multiply the matrices in the opposite order. The right-most transform is the one that applies first.

$$FR_\theta = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \sin\theta & \cos\theta \\ \cos\theta & -\sin\theta \end{bmatrix}$$

We observe that $FR_\theta \neq R_\theta F$, which is not surprising as matrix multiplication is not commutative except in special cases. One such special case is if we rotate first by 31° and then by 17° we should get the same result as if we first rotate by 17° and then by 31°.

Example 4.2.1 (*Rotation matrices are commutative*)

$$\begin{aligned} R_{48^\circ} &= R_{17^\circ} R_{31^\circ} \\ &= \begin{bmatrix} \cos 17^\circ & -\sin 17^\circ \\ \sin 17^\circ & \cos 17^\circ \end{bmatrix} \begin{bmatrix} \cos 31^\circ & -\sin 31^\circ \\ \sin 31^\circ & \cos 31^\circ \end{bmatrix} \\ &= \begin{bmatrix} \cos 17^\circ \cos 31^\circ - \sin 17^\circ \sin 31^\circ & -\cos 17^\circ \sin 31^\circ - \sin 17^\circ \cos 31^\circ \\ \sin 17^\circ \cos 31^\circ + \cos 17^\circ \sin 31^\circ & -\sin 17^\circ \sin 31^\circ - \cos 17^\circ \cos 31^\circ \end{bmatrix} \\ &= \begin{bmatrix} \cos(17^\circ + 31^\circ) & -\sin(17^\circ + 31^\circ) \\ \sin(17^\circ + 31^\circ) & \cos(17^\circ + 31^\circ) \end{bmatrix} \\ &= \begin{bmatrix} \cos 48^\circ & -\sin 48^\circ \\ \sin 48^\circ & \cos 48^\circ \end{bmatrix} \end{aligned}$$

If we compute $R_{31^\circ} R_{17^\circ}$, again we have

$$\begin{aligned} R_{31^\circ} R_{17^\circ} &= \begin{bmatrix} \cos(31^\circ + 17^\circ) & -\sin(31^\circ + 17^\circ) \\ \sin(31^\circ + 17^\circ) & \cos(31^\circ + 17^\circ) \end{bmatrix} \\ &= \begin{bmatrix} \cos 48^\circ & -\sin 48^\circ \\ \sin 48^\circ & \cos 48^\circ \end{bmatrix}. \end{aligned}$$

I.e., $R_{31^\circ} R_{17^\circ} = R_{17^\circ} R_{31^\circ}$, which confirms our intuition.

If we repeat the previous computation with θ and ϕ rather than 31° and 17°, we find that:

$$R_\theta R_\phi = R_\phi R_\theta = \begin{bmatrix} \cos(\theta+\phi) & -\sin(\theta+\phi) \\ \sin(\theta+\phi) & \cos(\theta+\phi) \end{bmatrix} \tag{4.1}$$

We would also suspect that R_θ and $R_{-\theta}$ are inverses, because if we rotate the plane by θ then rotate it in the opposite direction by the same θ then we are back where we started. This is verified by substituting $\phi = -\theta$ into (4.1).

$$\begin{aligned} R_\theta R_{-\theta} &= \begin{bmatrix} \cos(\theta-\theta) & -\sin(\theta-\theta) \\ \sin(\theta-\theta) & \cos(\theta-\theta) \end{bmatrix} \\ &= \begin{bmatrix} \cos 0 & -\sin 0 \\ \sin 0 & \cos 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I \end{aligned}$$

Therefore $R_\theta = R_{-\theta}^{-1}$, or $R_\theta^{-1} = R_{-\theta}$ as suspected.

While on the subject of inverses, what happens if we reflect twice about the line $y = x$? In fact we restore the plane to its original state. This would suggest that $F^2 = FF = I$, otherwise stated that $F^{-1} = F$, that F is its own inverse. We can easily verify this fact.

$$\begin{aligned} F^2 = FF &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} (0)(0)+(1)(1) & (0)(1)+(1)(0) \\ (1)(0)+(0)(1) & (1)(1)+(0)(0) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= I \end{aligned}$$

We make one final observation regarding the rotation matrix. What happens if we rotate twice by 90°? If we apply (4.1) directly we see that it is the same as rotating 180°. However, if we look at the actual multiplication of the matrix R_{90° by itself, we observe something interesting.

$$\begin{aligned}
(R_{90^\circ})^2 = R_{90^\circ}R_{90^\circ} &= \begin{bmatrix} \cos 90^\circ & -\sin 90^\circ \\ \sin 90^\circ & \cos 90^\circ \end{bmatrix} \begin{bmatrix} \cos 90^\circ & -\sin 90^\circ \\ \sin 90^\circ & \cos 90^\circ \end{bmatrix} \\
&= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \\
&= -\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
&= -I
\end{aligned}$$

So we see that R_{90° is a square root of $-I$. This is reminiscent of the fact that the imaginary number $i = \sqrt{-1}$. We know from elementary algebra also that $(-i)^2 = -1$. This corresponds to the fact that also $R^2_{-90^\circ} = -I$; *i.e.* we can rotate 180° either by rotating twice by 90° or twice by −90°.

$$
\begin{aligned}
(R_{-90^\circ})^2 &= \begin{bmatrix} \cos(-90^\circ) & -\sin(-90^\circ) \\ \sin(-90^\circ) & \cos(-90^\circ) \end{bmatrix} \begin{bmatrix} \cos(-90^\circ) & -\sin(-90^\circ) \\ \sin(-90^\circ) & \cos(-90^\circ) \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \\
&= -\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
&= -I
\end{aligned}
$$

4.3 Geometry of the Determinant

In Section 3.8.2, we mentioned that matrix-vector multiplication behaves as a linear transform on the set of all n-dimensional vectors. We did not prove (and will not now) that such a transform is *continuous*. *I.e.*, it maps continuous paths to continuous paths, and connected regions to connected regions. This mapping might stretch or compress areas, and it might reverse the orientation of curves. If the transform stretches or compresses, then it does so uniformly. *I.e.*, two different shapes will be stretched or compressed by the same amount.

Definition 4.3.1 (*Unit cube*)

Let S be the set of vectors, $\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ such that $0 \leq a_k \leq 1$ for $1 \leq k \leq n$. S is called the *unit cube* in n dimensions.

Definition 4.3.2 (*determinant*)

The *determinant* of a square matrix A is the number α such that the n-dimensional volume of the n-dimensional unit square, scales by α under transformation A.

If the determinant is negative, then the orientation of curves is reversed. Recall that we mentioned in Section 4.1 that the matrix, $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, reflects the plane about the line $y = x$, and notice that

$$\det\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\right) = (0)(0) - (1)(1) = -1\,.$$

If the determinant is unity, then the transformation preserves area and orientation. Recall the rotation matrix from Section 4.1, $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$. If we compute its determinant we get

$$\det\left(\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}\right) = \cos^2\theta + \sin^2\theta = 1\,,$$

so we can be sure that even if the matrix performs a rotation, that it does not reflect, and does not stretch shapes.

If the determinant is zero, then at least one dimension is lost, *i.e.*, volume of the unit cube is mapped to 0. In two dimensional space, the unit square is mapped to a line segment or to a point. In three-dimensional space, the 3-d unit cube is mapped to a 2-d or 1-d surface or to a point.

4.4 Computation of Determinant

We will present a method to compute the *determinant* of a square matrix, called the *Laplacian expansion* method.

The determinant of a matrix which tells us whether the matrix has an inverse. Some square matrices have an inverse and some do not. If $\det(A) \neq 0$, then there exists a matrix, which we denote A^{-1}, for which $A \times A^{-1} = A^{-1}A = I$. The inverse of a matrix is compute-intensive to calculate. In Section A.4 we will talk about a way to compute the inverse.

Definition 4.4.1 (*suppression matrix*)

Given an $n \times m$ matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}, \tag{4.2}$$

we define the *suppression* matrix, $\frac{A}{\neg i,j}$, as the $(n-1) \times (m-1)$ matrix formed by suppressing the i'th row and j'th column of A.

Example 4.4.2 (*Compute suppression matrix*)

As an example of a suppression matrix, let

$$A = \begin{bmatrix} 1 & 2 & 5 & 4 \\ 0 & 1 & 3 & 7 \\ 2 & -3 & 5 & -4 \\ 2 & 1 & -2 & -6 \end{bmatrix}.$$

If we suppress row 2, $\begin{bmatrix} 0 & 1 & 3 & 7 \end{bmatrix}$, and column 3, $\begin{bmatrix} 5 \\ 3 \\ 5 \\ -2 \end{bmatrix}$, then

we obtain the 3×3 matrix:

$$\frac{A}{\neg 2, 3} = \begin{bmatrix} 1 & 2 & \cancel{3} & 4 \\ \cancel{0} & \cancel{1} & \cancel{2} & \cancel{3} \\ 2 & -3 & \cancel{4} & -4 \\ 2 & 1 & \cancel{-2} & -6 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 \\ 2 & -3 & -4 \\ 2 & 1 & -6 \end{bmatrix}.$$

What does the Python code to compute the suppression matrix look like? One question to ask is whether this operation makes sense for `class Matrix` or for `class SqMatrix`. The answer is that suppressing a row and columns is an operation we can do to all matrices (provided there are at least two rows and at least two columns. Thus the method should go in the `class Matrix`.

Given a matrix, `m`, and a row and column indices `sr` and `sc`, we wish to return a new matrix with one fewer rows and one fewer columns. In particular, we would like to suppress row `sr` (suppress-row) and suppress column `sc` (suppress-column). To do this, we can use the `Matrix.tabulate` function. Its first two call-site arguments should be the number of rows and columns of `m` but decremented by 1. However, we must use a cleverly written local function as third call-site argument.

The local function `f` should be a binary function which will accept a row and column index as input, and it should return an element which goes in that position in the matrix being constructed. Furthermore, `f` must extract that element out of the matrix `m`. The function `f` must be written so that if its given `r,c` are both less than the respective `sr,sc`, then the value `m[r][c]` should be returned. However, if $r \geq sr$ then `m[r+1][...]` should be returned; if $c \geq sc$, then `m[...][c+1]` should be returned. Notice that either one or both of the inequalities

may be satisfied or non satisfied, so the indices of `m[..][...]` must be calculated appropriately in all four cases.

```
    def suppress_rc(m, sr, sc):
        assert 0 <= sr < m.rows
        assert 0 <= sc < m.cols
        assert m.rows > 1
        assert m.cols > 1

        def f(r, c):
            ...

        return Matrix.tabulate(m.rows - 1,
                               m.cols - 1,
                               f)
```

We can now use the suppression matrix to define the determinant.

Definition 4.4.3 (*Laplacian expansion*)

We may compute the determinant of a square, $n \times n$ matrix as:

$$\det(A) = \begin{cases} a_{11} & \text{if } n = 1 \\ \sum_{k=1}^{n} (-1)^{k+1} a_{1k} \det\left(\frac{A}{\neg 1,k}\right) & \text{if } n > 1 \end{cases} \tag{4.3}$$

Equation (4.3) is called the *Laplacian expansion* of matrix A.

In Equation (4.3), we see the term $(-1)^{k+1}$, since when $k = 1, 3, 5, \ldots$ we would like the term $a_{1k} \det\left(\frac{A}{\neg 1,k}\right)$ to be added, and when $k = 2, 4, 6, \ldots$, we would like the term to be subtracted.

ATTENTION, when writing this code in Python, this iteration variable will start at 0, not 1. So the exponent of (-1) must be chosen appropriately.

In the following code, part of the computation is omitted. Each iteration of the `for k in` loop will multiply three terms

1. $(-1)^{k+1}$; careful! $k + 1$ assumes 1-based indices

2. $a_{1,k}$; careful! the 1 subscript assumes 1-based indices

3. $\det\left(\frac{A}{\neg 1,k}\right)$; careful! the 1 subscript again assumes 1-based indices

The student must complete the formula which somehow involves a call to `suppress_rc` and a recursive call to `laplacian_expansion`.

```
class SqMatrix(Matrix):

  def laplacian_expansion(A):
    if A.dim == 1:
      return ...
    else:
      return sum((-1)**k \
                 * ... \
                 * ...
                 for k in range(A.dim)
                 )
```

There are two additional optimizations which can be made to the above Python code.

- We may avoid recurring (avoid computing the determinant of an $(n-1) \times (n-1)$ matrix in case the coefficient $A[0][k]$ is zero.
- We can hard code the formula for the determinant in the case the matrix has size 2×2. See Theorem 4.4.6.

```
class SqMatrix(Matrix):

  def laplacian_expansion(A):
    if A.dim == 1:
      return ...
    elif A.dim == 2:
      return A[...][...] * A[...][...] - A[...][...] * A[...][...]
    else:
      return sum((-1)**k \
                 * ... \
                 * ...
                 for k in range(A.dim)
                 if ...
                 )
```

4.4.1 Matrices over a Commutative Ring

Look carefully at Equation (4.3) and at the Python code above. The only operations used on the matrix entries are:

- addition ($+$), through the `sum(...)`,
- multiplication ($\times$), through $a_{1k} \cdot \det\left(\frac{A}{\neg 1,k}\right)$, and
- negation ($-$), through $(-1)^{k+1}$.

These are precisely the operations guaranteed by a *commutative ring* (Chapter 3, Section 3.5.1). The Laplacian expansion therefore works correctly for any square matrix whose entries come from a commutative ring — not just integers or floating-point numbers.

Example 4.4.4

The following are all valid entry types for `laplacian_expansion`:

- integers $\mathbb{Z}$ or rationals $\mathbb{Q}$ — exact arithmetic, no round-off error.
- polynomials $\mathbb{R}[\lambda]$ — the determinant of a matrix of polynomials is itself a polynomial. In particular, $\det(A - \lambda I)$ is the *characteristic polynomial* of A, whose roots are the eigenvalues. We will use this in Chapter 10.

The Python implementation of `laplacian_expansion` must **not** assume that entries are `int` or `float`. In particular:

- Do not use `abs()` on entries — it is not defined for polynomials.
- Do not compare entries with `== 0` using a float tolerance — use exact equality.

- The `sum(...)` built-in starts from the integer `0`; for polynomial entries, initialise the accumulator with the additive identity of the entry type, or use `functools.reduce` with the ring's addition.

Definition 4.4.3 also serves as an *inefficient* algorithm for computing the determinant. To compute the determinant of an $n \times n$ matrix, we recursively compute n-many determinants of $(n-1) \times (n-1)$ matrices. The recursion terminates either when we reach a 1×1 matrix, or 2×2 matrix depending on how the function is optimized.

As an example, we will compute the determinant of a 4×4 matrix.

Example 4.4.5 (*Determinant using Laplacian expansion*)

Let

$$A = \begin{bmatrix} 1 & 2 & 5 & 4 \\ 0 & 1 & 3 & 7 \\ 2 & -3 & 5 & -4 \\ 2 & 1 & -2 & -6 \end{bmatrix}$$

$$\begin{aligned} \det(A) = 1 & \left| \frac{A}{\neg 1, 1} \right| \\ & - 2 \left| \frac{A}{\neg 1, 2} \right| \\ & + 5 \left| \frac{A}{\neg 1, 3} \right| \\ & - 4 \left| \frac{A}{\neg 1, 4} \right|, \end{aligned}$$

where

$$\frac{A}{\neg 1,1} = \begin{bmatrix} 1 & 3 & 7 \\ -3 & 5 & -4 \\ 1 & -2 & -6 \end{bmatrix} \qquad \text{removing row 1 and column 1}$$

$$\frac{A}{\neg 1,2} = \begin{bmatrix} 0 & 3 & 7 \\ 2 & 5 & -4 \\ 2 & -2 & -6 \end{bmatrix} \qquad \text{removing row 1 and column 2}$$

$$\frac{A}{\neg 1,3} = \begin{bmatrix} 0 & 1 & 7 \\ 2 & -3 & -4 \\ 2 & 1 & -6 \end{bmatrix} \qquad \text{removing row 1 and column 3}$$

$$\frac{A}{\neg 1,4} = \begin{bmatrix} 0 & 1 & 3 \\ 2 & -3 & 5 \\ 2 & 1 & -2 \end{bmatrix} \qquad \text{removing row 1 and column 4}$$

To compute determinants, $\det\left(\frac{A}{\neg 1,1}\right)$ through $\det\left(\frac{A}{\neg 1,n}\right)$ we apply Def. 4.4.3 recursively, terminating at $n = 1$.

Let's continue the computation a few more steps by computing $\det\left(\frac{A}{\neg 1,1}\right)$.

$$\det\left(\frac{A}{\neg 1,1}\right) = \det\left(\begin{bmatrix} 1 & 3 & 7 \\ -3 & 5 & -4 \\ 1 & -2 & -6 \end{bmatrix}\right)$$

$$= (1)\det\left(\begin{bmatrix} 5 & -4 \\ -2 & -6 \end{bmatrix}\right) \quad \text{removing row 1, col 1}$$

$$- (3)\det\left(\begin{bmatrix} -3 & -4 \\ 1 & -6 \end{bmatrix}\right) \quad \text{removing row 1, col 2}$$

$$+ (7)\det\left(\begin{bmatrix} -3 & 5 \\ 1 & -2 \end{bmatrix}\right) \quad \text{removing row 1, col 3}$$

Similar for $\det\left(\frac{A}{\neg 1,2}\right)$, $\det\left(\frac{A}{\neg 1,3}\right)$, and $\det\left(\frac{A}{\neg 1,4}\right)$.

The computation in Example 4.4.5 is intensive; however, some optimizations are possible.

There is a shortcut to speed up the algorithm somewhat. The determinant of a 2×2 matrix can be hard coded.

Theorem 4.4.6 (*Determinant of* 2×2)

The determinant of a 2×2 matrix can be calculated by Equation (4.4).

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc \tag{4.4}$$

Proof. To prove this identity, we simply apply the Laplacian expansion to the matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

When we suppress a row and column from a 2×2 matrix we are left with a 1×1 matrix. And the determinant of a 1×1 matrix

(according to Definition 4.4.3, case $n = 1$) is simply the element inside the matrix.

$$\begin{aligned}\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) &= (a)\det\big(\begin{bmatrix} d \end{bmatrix}\big) && \text{removing row 1, col 1} \\ &\quad - (b)\det\big(\begin{bmatrix} c \end{bmatrix}\big) && \text{removing row 1, col 2} \\ &= ad - bc\end{aligned}$$

Another possible optimization is that the Laplacian expansion works just as well traversing any row or column. So if there is some row which contains many zeros, then expand along that row, and avoid the recursive calls. However, we must be careful to use the correct sign.

Definition 4.4.7 (*Laplacian expansion optimization*)

Let A be an $n \times n$ matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}. \tag{4.5}$$

We may compute the determinant of a square, $n \times n$ matrix along row r similar to Definition 4.4.3.

$$\det(A) = \begin{cases} a_{11} & \text{if } n = 1 \\ a_{11}a_{22} - a_{12}a_{21} & \text{if } n = 2 \\ \sum_{k=1}^{n} (-1)^{r+k} a_{rk} \det\left(\frac{A}{\neg r,k}\right) & \text{if } n > 2 \end{cases} \tag{4.6}$$

Equivalently we may expand along column c:

$$\det(A) = \begin{cases} a_{11} & \text{if } n = 1 \\ a_{11}a_{22} - a_{12}a_{21} & \text{if } n = 2 \\ \sum_{k=1}^{n} (-1)^{c+k} a_{kc} \det\left(\frac{A}{\neg k, c}\right) & \text{if } n > 2 \end{cases} \tag{4.7}$$

4.5 Complexity of the Laplacian Determinant Algorithm

This recursive definition is easy to implement programmatically, but is very inefficient for large matrices, requiring approximately $(e-1)n!$ multiplications.

We would like to compute the number of multiplication operations necessary to compute the determinant of an $n \times n$ matrix using the Laplacian expansion. If we have a row or column containing many zeros, then the algorithm can be optimized. However, we will compute the worst case complexity assuming no such optimization is possible.

If $n = 1$, then $\det(A)$ requires 0 multiplications.

If $n = 2$, then $\det(A) = a_{11}a_{22} - a_{12}a_{21}$, or 2 multiplications.

If $n = 3$, then

$$\begin{aligned} \det(A) = a_{11} \left| \frac{A}{\neg 1, 1} \right| \\ - a_{12} \left| \frac{A}{\neg 1, 2} \right| \\ + a_{13} \left| \frac{A}{\neg 1, 3} \right| . \end{aligned}$$

That is 3 multiplications in addition to 3 computations of the determinant of a 2×2 matrix, which we just argued was 2 multiplications

each; *i.e.* $3 + 3 \times 2 = 9$.

This pattern continues. If $n = k$, then $\det(A)$ needs k multiplications plus k computations of the determinant of an $(k-1) \times (k-1)$ matrix.

$$
\begin{aligned}
T(2) &= 2 \\
T(3) &= 3 + 3\ T(2) \\
&= 3 + 3 \times 2 \\
T(4) &= 4 + 4\ T(3) \\
&= 4 + 4 \times (3 + 3 \times 2) \\
&= 4 + (4 \times 3) + (4 \times 3 \times 2) \\
T(5) &= 5 + 5\ T(4) \\
&= 5 + 5 \times (4 + 4 \times 3 + 4 \times 3 \times 2) \\
&= 5 + (5 \times 4) + (5 \times 4 \times 3) + (5 \times 4 \times 3 \times 2) \\
&\vdots \\
T(n) &= n + n\ T(n-1) \\
&= \underbrace{n}_{\frac{n!}{(n-1)!}} + \underbrace{n \times (n-1)}_{\frac{n!}{(n-2)!}} + \underbrace{n \times (n-1)(n-2)}_{\frac{n!}{(n-3)!}} + \cdots + \underbrace{n!}_{\frac{n!}{1!}}
\end{aligned}
$$

We can generalize $T(n)$ as in Theorem 4.5.1.

Theorem 4.5.1 (*Complexity of Laplacian algorithm*)

The number of multiplications needed to compute the determinant of an $n \times n$ matrix using the Laplacian expansion algorithm is given

by:

$$T(n) = \frac{n!}{(n-1)!} + \frac{n!}{(n-2)!} + \cdots + \frac{n!}{1!}$$
$$= n! \sum_{k=1}^{n-1} \frac{1}{k!} \tag{4.8}$$

We present a proof by induction on n.

Proof.

Base Case : First we show (4.8) holds for $n = 2$.

$$\begin{aligned} T(n) &= T(2) \\ &= 2! \sum_{k=1}^{1} \frac{1}{k!} \\ &= 2! \left(\frac{1}{1!} \right) \\ &= 2 \end{aligned}$$

This result agrees with (4.4).

Inductive Case : Suppose that (4.8) holds for a particular $n > 2$, we show that it holds for $n+1$. *I.e.*, we must show

$$T(n+1) = (n+1)! \sum_{k=1}^{(n+1)-1} \frac{1}{k!} .$$

We consider $T(n+1)$. The Laplacian expansion algorithm states that to compute the determinant of an $(n+1) \times (n+1)$

matrix we need $(n+1)$ recursive calls, each of complexity $T(n)$, and after the recursive calls, each sub-determinant must be multiplied by an element of the $(n+1) \times (n+1)$ matrix. So:

$$\begin{aligned} T(n+1) &= (n+1) + (n+1)T(n) \\ &= (n+1) + (n+1) \underbrace{\left(n! \sum_{k=1}^{n-1} \frac{1}{k!} \right)}_{\text{by inductive hypothesis}} \\ &= (n+1) + (n+1)n! \sum_{k=1}^{n-1} \frac{1}{k!} \end{aligned}$$

But since $(n+1)n! = (n+1)!$, we have

$$\begin{aligned}
T(n+1) &= (n+1) + (n+1)n! \sum_{k=1}^{n-1} \frac{1}{k!} \\
&= (n+1) + (n+1)! \sum_{k=1}^{n-1} \frac{1}{k!} \\
&= (n+1)\left(\frac{n!}{n!}\right) + (n+1)! \sum_{k=1}^{n-1} \frac{1}{k!} \\
&= \frac{(n+1)!}{n!} + (n+1)! \sum_{k=1}^{n-1} \frac{1}{k!} \\
&= (n+1)!\left(\frac{1}{n!} + \sum_{k=1}^{n-1} \frac{1}{k!}\right) \\
&= (n+1)! \sum_{k=1}^{n} \frac{1}{k!} \\
&= (n+1)! \sum_{k=1}^{(n+1)-1} \frac{1}{k!}
\end{aligned}$$

If we wish to understand the value of $n! \sum_{k=1}^{n-1} \frac{1}{k!}$ from Theorem 4.5.1, we might ask what is its asymptotic behavior, as $n \to \infty$. In Theorem 4.5.3 we argue that this $T(n) \to (e-1)n!$, but to show that we make use of Lemma 4.5.2.

Lemma 4.5.2

$$\lim_{n\to\infty}\sum_{k=n}^{\infty}\frac{1}{k!}=0$$

Proof. Consider the sequence of partial sums $a_n=\sum_{k=n}^{\infty}\frac{1}{k!}$. We will show $(a_n)_{n=0}^{\infty}$ converges to 0. Note that if $k>2$, $k!>2^{k-1}$, because $k!=\prod_{j=2}^{k}j\geq\prod_{j=2}^{k}2=2^{k-1}$. Let $\varepsilon>0$, let $n>2$ and $n>\log_2\frac{1}{\varepsilon}$, so that $\frac{1}{2^n}<\varepsilon$.

$$a_n=\sum_{k=n}^{\infty}\frac{1}{k!}<\sum_{k=n}^{\infty}\frac{1}{2^{k-1}}$$

Let $k'=k-n-1$, so $k=k'+n+1$.

$$\begin{aligned}\sum_{k=n+1}^{\infty}\frac{1}{2^{k-1}}&=\sum_{k'+n+1=n+1}^{\infty}\frac{1}{2^{(k'+n+1)-1}}=\sum_{k'=0}^{\infty}\frac{1}{2^{(k'+n)}}\\&=\frac{1}{2^n}\sum_{k'=0}^{\infty}\frac{1}{2^{k'}}=\left(\frac{1}{2^n}\right)(1)=\frac{1}{2^n}<\varepsilon\end{aligned}$$

Theorem 4.5.3 (*Laplacian Asymptotic behavior*)

The complexity of the Laplacian expansion algorithm goes to $(e-1)n!$ for large n.

$$\lim_{n\to\infty}T(n)=(e-1)n! \tag{4.9}$$

To prove Theorem 4.5.3, will show that $\lim_{n\to\infty}\frac{T(n)}{(e-1)n!}=1$. We will

exploit the fact that the sum $\sum_{k=1}^{n-1} \frac{1}{k!}$ is part of the infinite series expansion for $e = \sum_{k=0}^{\infty} \frac{1}{k!}$. We simply need to break the infinite sum into three parts: $k = 0$, $1 \leq k \leq n-1$, and $k \geq n$, so that the middle part $1 \leq k \leq n-1$ cancels away.

Proof. First we notice that the $n!$ in the numerator cancels with that in the denominator.

$$\lim_{n\to\infty} \frac{T(n)}{(e-1)\ n!} = \lim_{n\to\infty} \frac{n! \sum_{k=1}^{n-1} \frac{1}{k!}}{n!\ (e-1)} = \lim_{n\to\infty} \frac{\sum_{k=1}^{n-1} \frac{1}{k!}}{e-1}$$

So we need to show $\lim_{n\to\infty} \frac{\sum_{k=1}^{n-1} \frac{1}{k!}}{e-1} = 1$.

$$\lim_{n\to\infty} \frac{\sum_{k=1}^{n-1} \frac{1}{k!}}{e-1} = \lim_{n\to\infty} \frac{e + \sum_{k=1}^{n-1} \frac{1}{k!} - e}{e-1} = \lim_{n\to\infty} \frac{e + \sum_{k=1}^{n-1} \frac{1}{k!} - \sum_{k=0}^{\infty} \frac{1}{k!}}{e-1}$$

$$= \lim_{n\to\infty} \frac{e + \overbrace{\sum_{k=1}^{n-1} \frac{1}{k!} - \sum_{k=1}^{n-1} \frac{1}{k!}}^{=0} - \overbrace{\sum_{k=0}^{0} \frac{1}{k!}}^{=1} - \sum_{k=n}^{\infty} \frac{1}{k!}}{e-1}$$

$$= \lim_{n\to\infty} \frac{e - 1 - \sum_{k=n}^{\infty} \frac{1}{k!}}{e-1} = \lim_{n\to\infty} \frac{e-1}{e-1} - \frac{\overbrace{\lim_{n\to\infty} \sum_{k=n}^{\infty} \frac{1}{k!}}^{=\ 0 \text{ by Lemma 4.5.2}}}{e-1}$$

$$= 1 - \frac{0}{e-1} = 1 - 0 = 1$$

To get an idea of how quickly $T(n)$ converges to $(e-1)n!$, we need an impression of how fast the values of $\sum_{k=n}^{\infty} \frac{1}{k!}$ converge to zero. Some such values are shown here.

n	$\sum_{k=n+1}^{\infty} \frac{1}{k!}$
1	0.71828175
2	0.21828175
3	0.051615
4	0.009948492
5	0.0016150475
6	0.00022625923
7	0.000027894974
8	0.000002861023
9	0.00000023841858

4.6 Cramer's Rule

In Chapter 5 we will look at more computationally efficient algorithms to solve the matrix equation $Ax = b$, but in this section first we look at an algorithm which is very easy to program. This is an algorithm which is easy to implement but is very inefficient to compute. The technique is called *Cramer's Rule*.

Theorem 4.6.1 (*Cramer's Rule*)

Given an invertible matrix A of dimension, $n \times n$, and a vector b of dimension n, the system

$$Ax = b$$

can be solved as follows. Let A_k be the matrix which is identical to A except that column k has been replaced by the column vector b.

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

where

$$x_k = \frac{\det(A_k)}{\det(A)} \tag{4.10}$$

We look at a simple example.

Example 4.6.2 (*Cramer's Rule*)

We wish to again solve the system from Equation (5.7).

$$\overbrace{\begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}^{x} = \overbrace{\begin{bmatrix} 5 \\ 0 \\ 11 \end{bmatrix}}^{b}$$

For A_1 we substitute b for column 1 of A.

$$A_1 = \begin{bmatrix} \bullet & 2 & 3 \\ \bullet & 2 & -1 \\ \bullet & 1 & 5 \end{bmatrix} = \begin{bmatrix} \boxed{5} & 2 & 3 \\ \boxed{0} & 2 & -1 \\ \boxed{11} & 1 & 5 \end{bmatrix}$$

For A_2 we substitute b for column 2 of A.

$$A_2 = \begin{bmatrix} 1 & \bullet & 3 \\ -1 & \bullet & -1 \\ 2 & \bullet & 5 \end{bmatrix} = \begin{bmatrix} 1 & \boxed{5} & 3 \\ -1 & \boxed{0} & -1 \\ 2 & \boxed{11} & 5 \end{bmatrix}$$

For A_3 we substitute b for column 3 of A.

$$A_3 = \begin{bmatrix} 1 & 2 & \bullet \\ -1 & 2 & \bullet \\ 2 & 1 & \bullet \end{bmatrix} = \begin{bmatrix} 1 & 2 & \boxed{5} \\ -1 & 2 & \boxed{0} \\ 2 & 1 & \boxed{11} \end{bmatrix}$$

Now we can compute $\det(A)$, $\det(A_1)$, $\det(A_2)$, and $\det(A_3)$, for example using the Laplacian expansion, Section 4.4, to find

$$\begin{aligned} \det(A) &= 2 \\ \det(A_1) &= -33 \\ \det(A_2) &= -7 \\ \det(A_3) &= 19 \end{aligned}$$

So

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \frac{1}{\det(A)} \begin{bmatrix} \det(A_1) \\ \det(A_2) \\ \det(A_3) \end{bmatrix} = \begin{bmatrix} -33/2 \\ -7/2 \\ 19/2 \end{bmatrix}$$

The code for Cramer's rule in Python is straightforward assuming the method `replace_col` exists. This is a method on `Matrix` which takes a column index and a correctly sized vector (object of type `Vector`), and returns a new matrix of the same size as the original one, but with the indicated column replaced with the entries from the vector. This computation can easily be done using the `Matrix.tabulate` function to create a new matrix the same size as the given one, but a cleverly coded local function, `f`, must be provided. Of course a `lambda` function can be used in place of the local function name if the programmer prefers it.

The local function, `f`, must retrieve `m[r][c]` whenever $c \neq k$, and it must retrieve `v[r]` if $c = k$

```
class Matrix:

    def replace_col(m, k, v):
        assert isinstance(v, Vector)
        assert isinstance(k, int)
        assert m.rows == v.dim
        assert k < m.cols

        def f(r,c):
          ...

        return Matrix.tabulate(m.rows,
                               m.cols,
                               f)
```

Now, assuming `replace_col` is available, the Cramer's rule method, `cramers_rule` method in `class SqMatrix` is straightforward. The method accepts a `Vector`, `b`, of the same dimension as the square matrix, and either returns `None` if there is no uniquely determined solution, else it returns a `Vector` containing the v vector which solves $Av = b$.

In the following code, partially implementing the method `cramers_rule`, the vector is generated by a call to `Vector.tabulate`, given the desired dimension (same as the dimension of A). The second argument is the function `f`, which is a local function which is not complete in the example.

The definition of `f` must be completed so that for a given k, it (nondestructively) replaces column k in A with the vector b (using the method `replace_col`; then computes the determinant of that matrix, using the method `laplacian_expansion`; and finally divides the result by the determinate of A which has been precalculated in the variable `detA`. *I.e.*, $f(k)$ must compute $\frac{\det(A_k)}{\det(A)}$ from Equation (4.10).

```
class SqMatrix(Matrix):

    def cramers_rule(A, b):
        assert isinstance(b, Vector)
        assert A.dim == b.dim

        detA = A.laplacian_expansion()
        if detA == 0:
            return None

        def f(k):
          ...

        return Vector.tabulate(A.dim, f)
```

4.7 Complexity of Cramer's Rule

We saw in Theorem 4.5.3 that the computation of the determinant of an $n \times n$ matrix using the Laplacian expansion requires approximately $(e-1)n! \approx 1.7183(n!) \approx 2(n!)$ multiplications. If we use Cramer's rule to solve a system of n simultaneous linear equations, we must compute $n+1$ determinants of $n \times n$ matrices; one $\det(A_k)$ for each variable and the common denominator $\det(A)$. Thus we need $(2)(n+1)(n!) = (2)(n+1)!$ total multiplications.

The following table assumes 3 trillion, 3×10^{12}, multiplications per second (assuming the Apple M4 chip can perform almost 3 trillion per second).

n	$(2)(n+1)!$	time
6	10,080	
7	80,640	
8	725,760	
9	7,257,600	
10	79,833,600	
11	958,003,200	
12	12,454,041,600	
13	174,356,582,400	
14	2,615,348,736,000	0.8 seconds
15	41.845×10^{12}	14 seconds
16	711.374×10^{12}	4 minutes
17	12.804×10^{15}	71 minutes
18	243.290×10^{15}	22 hours
19	4.86510^{18}	19 days
20	102.181×10^{18}	1.1 year
21	2.248×10^{21}	24 years

4.8 Inverse of a 2×2 Matrix

The determinant gives a direct formula for the inverse of a 2×2 matrix, requiring nothing more than the four entries and one division.

Theorem 4.8.1 (*Inverse of* 2×2 *matrix*)

If $\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc \neq 0$, then

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

Proof. Multiply in both directions and apply Proposition 3.5.4 (3.19) to factor the scalar:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{ad-bc} \begin{bmatrix} ad-bc & 0 \\ 0 & ad-bc \end{bmatrix} = I\,.$$

The product in the other order gives the same result by Theorem 3.6.4.

4.9 Elementary Row Operations

There are three types of operations one may perform on the rows of a matrix while solving a linear system. These are called *elementary row operations*.

Definition 4.9.1 (*elementary row operations*)

There are three types of elementary row operations.

1. $R_i \leftrightarrow R_j$: exchange row i with row j $(i \neq j)$.
2. $\beta R_i \to R_i$: replace a row with a scaled version of the row $(\beta \neq 0)$.
3. $\alpha R_i + \beta R_j \to R_j$: replace a row by a linear combination of that row and another row $(i \neq j,\ \beta \neq 0)$.

Notice that if $\alpha = 0$ then operation 2 is simply a special case of operation 3. Thus, the purist might argue that there are only 2 distinct row operations. Nevertheless, we will consider operations 2 and 3 as distinct operations.

Each row operation has a predictable effect on the determinant.

Proposition 4.9.2 (*Row operations and the determinant*)

Let A be an $n \times n$ matrix.

1. **Swap:** $R_i \leftrightarrow R_j$ multiplies $\det(A)$ by -1.
2. **Scale:** $\beta R_i \to R_i$ multiplies $\det(A)$ by β.
3. **Row combination:** $\alpha R_i + \beta R_j \to R_j$ multiplies $\det(A)$ by β. In particular, the special case $R_i + \beta R_j \to R_j$ ($\alpha = 1$, $\beta = 1$) leaves $\det(A)$ *unchanged*.

The third item is the key insight behind Gaussian elimination: adding a multiple of one row to another does not change the determinant. This means we can zero out all entries below the diagonal — turning A into an upper-triangular matrix — without altering $\det(A)$. The determinant then reads off as the product of the diagonal entries. Chapter 5 builds on this observation to develop the Bareiss algorithm, which performs these row operations using only integer arithmetic.

4.9.1 Row Operations in Python

The three row operations correspond directly to the methods you will implement in `Matrix.py`:

- `swap_rows(i, j)` — implements $R_i \leftrightarrow R_j$
- `scale_row(beta, i)` — implements $\beta R_i \to R_i$
- `row_operation(alpha, i, beta, j)` — implements $\alpha R_i + \beta R_j \to R_j$

4.10 Programming Exercises

4.10.1 Classwork

1. Develop code to transform set of points in the plane
2. Setup for Laplacian determinant algorithm

4.10.2 Homework

- Implement `Matrix.py`
  - `suppress_rc`
  - `diagonal`
  - `laplacian_expansion`
  - `scale_row`
  - `row_operation`
  - `swap_rows`
  - `adjoin_col`
  - `adjoin_cols`
  - `adjoin_rows`
  - `adjoin_row`
  - `extract_rows`
  - `extract_cols`
  - `cols_to_matrix`
  - `rows_to_matrix`
- Implement `SqMatrix.py`

  - diagonal
  - laplacian_expansion
  - cramers_rule

- Test with:
  - laplacian_determinant_test.py
  - cramer_test.py
  - row_operations_test.py

Chapter 5

Gaussian Elimination

Chapter Objectives

- Identify whether a system of linear equations has no solution, exactly one solution, or infinitely many solutions.
- Apply the three elementary row operations (introduced in Chapter 4) to reduce an augmented matrix to row-echelon form.
- Apply back-substitution to read off the solution from row-echelon form.
- Apply the Bareiss recurrence to reduce an integer matrix to upper-triangular form without introducing fractions.
- Recognize that the determinant emerges as a byproduct of the Bareiss forward pass.
- Express back-substitution as a sequence of row operations and use this to compute matrix inverses.
- Implement Gaussian elimination and the Bareiss algorithm in Python and use them to solve square linear systems and invert matrices.

5.1 Linear Equations

A *system* of linear equations may have zero, one, or infinitely many solutions. *E.g.*., the system consisting of Equations (5.1) and (5.2) has no solution.

$$3x_1 - 3x_2 = 0 \tag{5.1}$$
$$x_1 - x_2 = -1 \tag{5.2}$$

This fact can easily be seen because in order to satisfy (5.1), x_1 and x_2 must be equal. However, no choice of $x_1 = x_2$ will satisfy (5.2).

$$3x_1 - 3x_2 = 0 \tag{5.3}$$
$$2x_1 - 2x_2 = 0 \tag{5.4}$$

The system consisting of (5.3) and (5.4) has infinitely many solutions. In particular any choice of $x_1 = x_2$ satisfies both equations.

Finally the system consisting of (5.5) and (5.6) has a unique solution, namely $x_1 = 1$, $x_2 = -1$.

$$3x_1 + 2x_2 = 1 \tag{5.5}$$
$$2x_1 + 3x_2 = -1 \tag{5.6}$$

We can represent a system of linear equations as a matrix multiplication.

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 11 \end{bmatrix} \tag{5.7}$$

If we use normal matrix multiplication rules and multiply each row of the matrix in (5.7), we obtain the system of equations: (5.8), (5.9), (5.10).

$$x_1 + 2x_2 + 3x_3 = 5 \tag{5.8}$$
$$-x_1 + 2x_2 - x_3 = 0 \tag{5.9}$$
$$2x_1 + x_2 + 5x_3 = 11 \tag{5.10}$$

Furthermore, we can verify that $\begin{bmatrix} -33/2 \\ -7/2 \\ 19/2 \end{bmatrix}$ is a solution to the system of equations, (5.8), (5.9), (5.10), by substituting

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -33/2 \\ -7/2 \\ 19/2 \end{bmatrix} \tag{5.11}$$

and performing the matrix multiplication to verify:

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix} \begin{bmatrix} -33/2 \\ -7/2 \\ 19/2 \end{bmatrix} = \begin{bmatrix} \frac{-33}{2} - \frac{2(7)}{2} + \frac{3(19)}{2} \\ \frac{-(-33)}{2} - \frac{2(7)}{2} - \frac{19}{2} \\ \frac{2(-33)}{2} - \frac{7}{2} + \frac{5(19)}{2} \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 11 \end{bmatrix} \tag{5.12}$$

5.2 Solving Systems of Linear Equations

Our goal is to solve a given system of n-many linear equations. For a unique solution to exist there must be at least n equations, which are consistent (not contradictory), and not redundant (independent). For example if we know

$$2x_1 + 3x_2 = 1$$
$$4x_1 + 6x_2 = 2\,,$$

then we don't have two independent equations because one equation is simply double the other. The second equation gives no new information apart from what is already known in the first.

As another example, suppose we are given

$$\begin{aligned} 2x_1 + 3x_2 &= 1 \\ 2x_1 + 3x_2 &= 2\,, \end{aligned}$$

then the two equations are independent but contradictory. In Section 5.4 we'll make this requirement explicit and precise. For the moment consider a system such as the following:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n\,. \end{aligned}$$

This system can be rewritten as $Ax = b$, where $A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & & & \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}$, $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$, and $b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$.

The problem we would like to be able to solve is: given A and b, find x such that $Ax = b$. Moreover, we would like an algorithm which lends itself to computer programming.

There are 4 techniques we will study (or have studied) for solving $Ax = b$.

- Cramer's Rule, Section 4.6.

- Matrix Identity, Section 5.3.
- Gaussian Elimination, Section 5.5.
- Gauss-Jordan Elimination, Chapter A.
- Bareiss Algorithm, Section 5.11.

5.3 Matrix Inverse Method

We wish to solve the equation $Ax = b$, assuming $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$. If the inverse of A is known, then we can left-multiply both sides by A^{-1}.

$$\begin{aligned} Ax &= b \\ A^{-1}(Ax) &= A^{-1}b \\ \big(A^{-1}A\big)x &= A^{-1}b \\ Ix &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

This technique sounds easy enough, but the problem is that we do not know (yet) how to compute the inverse. Let's look at an example where we know the inverse already. Recall Example 3.6.5 where $A = \begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix}$, $B = \begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix}$, and consequently $AB = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

The fact that $AB = I$ means that $B = A^{-1}$. Using this fact we can solve $Ax = b$ for any appropriately sized vector b. Lets solve for x

when $b = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}$.

$$
\begin{aligned}
Ax &= b \\
\begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} &= \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \\
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} &= \begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \\
&= \underbrace{\begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix}}_{A^{-1}=B} \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 8+5-2 \\ 10+6-2 \\ -16-9+3 \end{bmatrix} = \begin{bmatrix} 11 \\ 14 \\ -22 \end{bmatrix}
\end{aligned}
$$

5.4 Linear Independence

Definition 5.4.1 (*linear combination*)

Given a finite set of vectors, $V = \{v_1, v_2, \ldots, v_m\}$, each of same dimension $n > 1$, and scalars $\alpha_1, \alpha_2, \ldots, \alpha_m \in \mathbb{R}$. The sum,

$$\sum_{k=1}^{m} \alpha_k v_k = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_m v_m$$

is said to be a *linear combination* of V.

There is a slight confusion possible with Definition 5.4.1. The confusion is that perhaps $v_i = v_j$ for some i, j. In such a case $|V| < m$, so the notation for the sum $\sum_{k=1}^{m} \alpha_k v_k$ is confusing in that the same vector v_i occurs twice in the sum. This is not really a problem, because v_i is really scaled by $(\alpha_i + \alpha_j)$ in the sum.

Next, we define the *span* as the set of all possible linear combinations of a (finite or infinite) set of n dimensional vectors.

Definition 5.4.2 (*span*)

If B is a set of vectors the *span* of B, written $\text{span}(B)$, is defined as follows:

- If $|B| = m < \infty$, *i.e.*, $B = \{v_1, v_2, \ldots, v_m\} \subset \mathbb{R}^n$, then $\text{span}(B) \subset \mathbb{R}^n$ is the set of all vectors which can be expressed as linear combinations of B. *I.e.*,

$$\text{span}(B) = \left\{ \sum_{i=1}^{m} \alpha_i v_i \;\middle|\; \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \end{bmatrix} \in \mathbb{R}^m \right\}$$

- If $|B| = \infty$, *i.e.*, $B = \{v_1, v_2, \ldots\} \subset \mathbb{R}^n$, then $\text{span}(B) \subset \mathbb{R}^n$

is the set of vectors which can be expressed as a linear combination of some subset of B. *I.e.*, $\text{span}(B) = \bigcap_{\substack{S \subset V \\ |S| < \infty}} \text{span}(S)$.

The pursuit of solving a system of linear equations amounts to starting with an element of the span of some set of vectors, and attempting to reverse-engineer the coefficients $x = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \end{bmatrix}$. In some cases, the coefficients can be determined uniquely; in some cases it can be shown that no such coefficients exists, and in other cases the set of x values are themselves the span of some other set of vectors, potentially of smaller dimension. There is a unique solution in the case that the vectors are linearly independent, Definition 5.4.3.

Definition 5.4.3 (*linearly dependent/independent*)

A set, V, ($|V| > 1$) of vectors is said to be *linearly dependent* if some (at least one) $b \in V$, can be written as a linear combination of some (necessarily finite) subset of $V \setminus \{b\}$; *i.e.*, if there exist $\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} \in \mathbb{R}^n$, and $v_1, v_2, \ldots, v_n \in V \setminus \{b\}$ such that

$$b = \sum_{k=1}^{n} \alpha_k v_k \,. \tag{5.13}$$

If V is not linearly dependent, it is called *linearly independent*.

In the special case that each $v_i \in \mathbb{R}^n$, we can construct an $n \times n$ ma-

trix by filling columns with vectors $v_1, v_2, \ldots, v_n$, $A = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$. Furthermore if we let $x = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$, then Equation (5.13), $b = \sum_{k=1}^{n} \alpha_k v_k$, becomes simply $Ax = b$.

Determining whether a given set of n vectors each of dimension n is linearly independent, amounts to finding the coefficients, which amounts to solving a system of linear equations, or determining whether such a solution exists. *I.e.*, does $Ax = b$ have a unique solution?

Given a system of n linear equations, it suffices to compute the determinant (Sections 4.4 and A.7) of the matrix of coefficients to determine whether the row vectors are linearly independent. If the determinant is non-zero, then the system of equations has a unique solution, and the vectors are linearly independent.

Theorem 5.4.4 (*Linear Independent Vectors*)

A finite set of vectors $V = \{v_1, v_2, \ldots, v_n\} \subset \mathbb{R}^n$ of size $|V| = n$ is linearly dependent if and only if there exist a non-zero vector, $v = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} \in \mathbb{R}^n$ such that $\sum_{k=1}^{n} \alpha_k v_k = 0$.

Proof.

- ($\Longrightarrow$)

Suppose $v = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} \in \mathbb{R}^n$ with $v \neq 0$, and suppose $\sum_{k=1}^{n} \alpha_k v_k = 0$.

$$\begin{aligned} 0 = \sum_{k=1}^{n} \alpha_k v_k &= \alpha_1 v_1 + \sum_{k=2}^{n} \alpha_k v_k \\ -\alpha_1 v_1 &= \sum_{k=2}^{n} \alpha_k v_k \\ v_1 &= \frac{1}{-\alpha_1} \sum_{k=2}^{n} \alpha_k v_k = \sum_{k=2}^{n} \left(\frac{-\alpha_k}{\alpha_1} \right) v_k \end{aligned}$$

So V is linearly dependent (by Definition 5.4.3).

- ($\Longleftarrow$)

  Now suppose V is linearly dependent, and that v_j can be written as the linear combination $v_j = \sum_{\substack{k=1 \\ k \neq j}}^{n} \alpha_k v_k$. Let

$\alpha_j = -1$:

$$
\begin{aligned}
v_j &= \sum_{\substack{k=1 \\ k \neq j}}^{n} \alpha_k v_k \\
0 &= \sum_{\substack{k=1 \\ k \neq j}}^{n} \alpha_k v_k - v_j = \sum_{\substack{k=1 \\ k \neq j}}^{n} \alpha_k v_k + \alpha_j v_j \\
&= \sum_{k=1}^{n} \alpha_k v_k
\end{aligned}
$$

Since $\alpha_j \neq 0$, then $\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} \neq 0$.

Corollary 5.4.5 is what many text books use as the definition of linearly independent. The proposition is equivalent, and more mathematically useful, but we find it less intuitive.

Corollary 5.4.5 (*Alternate definition of* linearly independent)

A set $V \subset \mathbb{R}^n$ $(n > 1)$ of vectors $(|V| = n)$ is linearly independent if and only if

$$
\sum_{k=1}^{n} \alpha_k v_k = 0 \implies \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = 0 \,.
$$

Proof. This is a restatement of Theorem 5.4.4

5.5 Gaussian Elimination with Back-Substitution

Definition 5.5.1 (*row-echelon form*)

A matrix is in *row-echelon form* if

1. All rows consisting only zeros are at the bottom
2. The leading entry, (*i.e.*, the left-most non-zero element) of row i ($i \neq 1$) is strictly to the right of the leading entry in row $i-1$.

Definition 5.5.2 (*reduced row-echelon form*)

A matrix in *row-echelon* form is said in particular to be in *reduced row-echelon* form, if the leading element of every non-zero row is 1.

For example: A (5.14) is neither in row-echelon form nor in reduced row-echelon form; B (5.15) is in row-echelon form but not in reduced row-echelon form, C (5.16) is in both row-echelon form and also in reduced row-echelon form; finally D (5.17) is an upper triangular matrix in row-echelon but not in reduced row-echelon form.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ -1 & 2 & -1 & 0 & 1 \\ 2 & 1 & 5 & 3 & 7 \\ 1 & 1 & 6 & -3 & 7 \end{bmatrix} \tag{5.14}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 2 & -1 & 0 & 1 \\ 0 & 0 & 0 & 3 & 7 \\ 0 & 0 & 0 & 0 & 7 \end{bmatrix} \tag{5.15}$$

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.16}$$

$$D = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 3 & 1 & 7 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.17}$$

If we can use elementary row operations to reduce a matrix to a triangular *row-echelon* form or *reduced-row-echelon* form, then the system of linear equations has a unique solution.

In this course we will only find the solution to a system of equations which has a unique solution. However, when a unique solution does not exist, we can determine that the system has no solution, or characterize the set of all solutions.

Suppose we have a system of equations represented by the matrix/vector equation

$$Ax = b\,.$$

- If any zero row has a non-zero right-hand side ($b \neq 0$): the system has no solution.

- If all zero rows have $b = 0$: the system has infinitely many solutions.

The goal of Gaussian elimination with back-substitution is to transform a matrix into row-echelon form. This reduction is carried out by the three elementary row operations defined in Section 4.9 (Chapter 4).

If a matrix can be transformed to an upper triangular row-echelon form, such as D (5.17), then the corresponding system of linear equations can be solved using back-substitution.

5.6 Example of Gaussian Elimination

Lets start with the system of equations (5.7). Rather than performing row operations on the square matrix and the solution vector in two steps, we first adjoin the vector to the right-hand side of the matrix.

Example 5.6.1 (*Gaussian Elimination*)

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 11 \end{bmatrix}$$

$$\Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ -1 & 2 & -1 & 0 \\ 2 & 1 & 5 & 11 \end{array}\right]$$

Now we will perform a sequence of elementary row operations.

$$R_1 + R_2 \to R_2 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 2 & 1 & 5 & 11 \end{array}\right] \tag{5.18}$$

$$2R_1 - R_3 \to R_3 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 0 & 3 & 1 & -1 \end{array}\right] \tag{5.19}$$

$$3R_2 - 4R_3 \to R_3 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 0 & 0 & 2 & 19 \end{array}\right] \tag{5.20}$$

In Example 5.6.1 we created zeros below the diagonal in columns 1 and 2. Note that if performing these computations by hand, it is probably easier to perform steps (5.18) and (5.19) in a single pass rather than needlessly copying the matrix redundantly. In general zeros can be placed below the diagonal of any given column in a single pass.

5.7 Gaussian Elimination in Python

Our goal is to explain the Python functions necessary to perform Gaussian Elimination starting from the square $n \times n$ matrix, A, and n-vector, b. Eventually we want to solve $Ax = b$ for x. However, Gaussian Elimination does not result in vector x rather it results in a new matrix in row echelon form, from which we can use back-substitution (Section 5.9) to solve for x.

The first step before calling the method `make_row_echelon` is to create the augmented matrix. This is the $n \times (n+1)$ matrix such as

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ -1 & 2 & -1 & 0 \\ 2 & 1 & 5 & 11 \end{array}\right]$$ from Example 5.6.1.

This can easily be done if `A` is the matrix, and `b` is the vector, then the Python code `A.adjoin_col(b)` will create the adjoin matrix by adjoining vector `b` onto the right-hand side of matrix `A`. Note that we can only perform Gaussian Elimination on a square matrix, *i.e.*, A is square; of course the augmented matrix, $\left[A \mid b\right]$, is not. This means that `gaussian_elimination_back_substitution` is a method in `SqMatrix`, but `make_row_echelon` is a method of `Matrix`.

In general $\left[A \mid b\right]$ looks like the following.

$$\left[\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{array}\right]$$

The goal is to perform row operations to obtain non-zero values on the main diagonal, and zeros below the diagonal. Here we denote the entries with $a'_{i,j}$ and b'_i to emphasize that these are not the same values as in the original augmented matrix.

$$\left[\begin{array}{cccc|c} \boxed{a'_{1,1}} & a'_{1,2} & \cdots & a'_{1,n} & b'_1 \\ 0 & \boxed{a'_{2,2}} & \cdots & a'_{2,n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \boxed{a'_{n,n}} & b'_n \end{array}\right]$$

The method `make_row_echelon` walks down the diagonal from $k = 0$ to $k = n - 1$, where $n \times n$ is the dimension of the matrix A, or $n \times (n + 1)$ is the dimension of the augmented matrix. $\left[A \mid b\right]$. The augmented matrix is represented by the variable `m` in the Python code.

```
class Matrix:

  def make_row_echelon(m):
    for k in range(0, m.rows - 1):
      pivot = m.find_pivot_row(k)
      if pivot is None:
        return None
      else:
        ...

      for r in range(k + 1, m.rows):
        m = ...
    return m
```

On each iteration `k`, the loop does two things, (1) guarantees an optimum value is on the diagonal at coordinates k, k, and (2) performs row operations to obtain zeros below the diagonal from $k+1, k$ to $n-1, k$.

To assure that an optimum value is on the diagonal, we compute `pivot` which is the index of the value with the greatest absolute value but restricted to column `k` but below the diagonal. See Section 5.8 for more details on pivoting. In the case that there are only zeros below the diagonal in column `k`, then we have determined that the original A has no inverse, thus we return `None`, indicating that $Ax = b$ has no unique solution. Otherwise (the `else` part) must swap rows `k` and `pivot`.

After a non-zero value has been swapped onto the diagonal, there is a loop `for r in range(...)`. This loop iterates `r` over the rows below the diagonal, starting with $k+1$ and ending with $n-1$ which is the bottom-most row. Recall that mathematically, the bottom-most row has index n but in the Python code the bottom rows are numbered from 0 to $n-1$. The job of each iteration, *i.e.*, for each value of `r`, is to perform a row operation resulting in a 0 in row `r` column `k`, knowing that a non-zero value is in position k, k. The two rows in question look like the following.

$$\begin{array}{r} \\ \text{Row k:} \\ \\ \text{Row r:} \\ \\ \end{array} \left[\begin{array}{ccccccc} \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a'_{k,k} & \cdots & b'_k \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a'_{r,k} & \cdots & b'_r \\ \vdots & & \vdots & \vdots & & \vdots \end{array}\right]$$

to put a zero at position $a'_{r,k}$, we need to perform the elementary row operation $a'_{r,k}R_k - a'_{k,k}R_r \to R_r$. This row operation corresponds to the method call `m = m.row_operation(m[r][k], k, -m[k][k], r)`. Notice the negative sign on `-m[k][k]`. It would also work to put the sign on the other coefficient: $-a'_{r,k}R_k + a'_{k,k}R_r \to R_r$.

It it important to recognize that this row operation does not destroy the leading zeros in row r because $(a'_{r,k})(0) - (a'_{k,k})(0) = 0$.

5.8 Pivoting

The choice of which row to use as the pivot at each step is called the *pivot strategy*, and the right strategy depends on what kind of numbers the matrix contains.

When reducing a matrix *by hand* using integer arithmetic, the natural strategy is to pivot only when the diagonal entry is zero. When a swap is needed, any row with a non-zero entry in the column suffices. When choosing the scalars for a row operation, use the *greatest common factor* of the two leading entries so that intermediate values stay small and no fractions are introduced unnecessarily. This is the appropriate strategy for the hand calculations in this chapter.

The danger of a small pivot. A 64-bit `float` carries about 15–16 significant decimal digits. Dividing by a very small pivot magnifies ev-

ery rounding error in the row by the same factor, potentially consuming all available precision.

Example 5.8.1 (*A small pivot causes a wrong answer*)

Consider the 2×2 system with $\epsilon = 10^{-15}$:

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}.$$

The true solution is $x_1 = 2$, $x_2 = 1$. Without pivoting, the algorithm divides row 1 by ϵ, then subtracts $\frac{1}{\epsilon}$ times row 1 from row 2:

$$\left[\begin{array}{cc|c} \epsilon & 1 & 1 \\ 0 & 2 - 1/\epsilon & 4 - 1/\epsilon \end{array}\right].$$

In floating-point, $4 - 10^{15} \approx -10^{15}$ and $2 - 10^{15} \approx -10^{15}$, so $x_2 \approx 1$ — correct so far. But then $x_1 = (1 - x_2)/\epsilon = 0/10^{-15} = 0$. The correct answer is $x_1 = 2$. The subtraction $1 - x_2$ lost all significant digits; this is called *catastrophic cancellation*.

Partial pivoting. The fix is to choose the pivot with the *largest absolute value* in the column before each step. With the largest possible value as the pivot, the multipliers used to zero out other rows are at most 1 in absolute value, preventing small differences from being amplified.

Example 5.8.2 (*Partial pivoting fixes the problem*)

With $\epsilon = 10^{-15}$, partial pivoting compares $|\epsilon|$ with $|1|$ in column 1 and swaps:

$$\left[\begin{array}{cc|c} \epsilon & 1 & 1 \\ 1 & 2 & 4 \end{array}\right] \xrightarrow{R_1 \leftrightarrow R_2} \left[\begin{array}{cc|c} 1 & 2 & 4 \\ \epsilon & 1 & 1 \end{array}\right]$$

The pivot is now 1. The multiplier for row 2 is only ϵ, so the row operation barely changes it. Back-substitution gives $x_2 = 1$ and $x_1 = 4 - 2 \cdot 1 = 2$, both correct.

For this reason `find_pivot_row(k)` selects the row at or below position k with the largest absolute value in column k.

Rational numbers: minimize denominator growth. If the matrix entries are exact rational numbers, floating-point cancellation is not a concern—but a different problem arises. Each row operation can produce fractions with very large numerators and denominators even when the final answer is a small integer. For rational arithmetic, a better pivot criterion is the entry that minimizes denominator growth during elimination.

Polynomials: minimize degree growth. When eliminating over a matrix whose entries are polynomials—for example, when computing the characteristic polynomial $\det(M - \lambda I)$ by row-reducing $M - \lambda I$—the concern is not rounding error but *expression swell*: intermediate entries can become polynomials of far higher degree than necessary. A good pivot strategy chooses entries of low polynomial degree.

The design implication. Because the right pivot strategy depends on what the matrix entries *are*, a general-purpose elimination routine should not hard-code a specific criterion. The method `find_pivot_row` is kept separate from the elimination loop precisely so that different strategies can be substituted without rewriting the elimination logic. The current implementation uses partial pivoting (largest absolute value), which is the correct default for floating-point matrices.

Condition number. Even with the best pivot strategy, some systems are inherently hard to solve accurately. The *condition number* $\kappa(A) = \|A\| \, \|A^{-1}\|$ measures how much a small change in b can shift the solution x: if $\kappa(A) \approx 10^k$, solving $Ax = b$ may lose up to k significant digits regardless of algorithm.

Example 5.8.3 (*A system no algorithm can solve accurately*)

Let $\delta = 10^{-8}$. Two systems with the same matrix but right-hand sides differing by only δ:

$$\text{System A:} \quad \begin{bmatrix} 1 & 1 \\ 1 & 1+\delta \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2+\delta \end{bmatrix} \quad \Rightarrow \quad x_1 = 1, \quad x_2 = 1$$

$$\text{System B:} \quad \begin{bmatrix} 1 & 1 \\ 1 & 1+\delta \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2+2\delta \end{bmatrix} \quad \Rightarrow \quad x_1 = 0, \quad x_2 = 2$$

A difference of $\delta = 10^{-8}$ in the input produces a difference of 1 in the solution. The two rows of the matrix are nearly parallel, so their intersection shifts enormously when either line moves slightly. The condition number here is $\kappa(A) \approx 10^8$: up to 8 significant digits of the solution can be lost regardless of pivot strategy. No algorithm can recover information that was never in the data.

Cramer's Rule (Section 4.6) expresses each solution component as a ratio of determinants. Each determinant costs $O(n \cdot n!)$ operations with Laplacian expansion, and near-zero determinants amplify floating-point errors directly in the result. Cramer's Rule is never used in practice for systems larger than 3×3.

5.9 Back Substitution

In Example 5.6.1 we converted the augmented matrix $\left[A \mid b\right]$ to row echelon form: $\left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 0 & 0 & 2 & 19 \end{array}\right]$. This form makes it straightforward to solve for $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ one variable at a time using what we call *back-substitution.*

Example 5.9.1 (*Back-Substitution*)

First, solve for x_3:

$$\begin{aligned} 2x_3 &= 19 \\ x_3 &= \frac{19}{2} \end{aligned}$$

Now, solve for x_2, given x_3:

$$\begin{aligned} 4x_2 &= 5 - 2x_3 \\ 4x_2 &= 5 - 2 \cdot \frac{19}{2} \\ &= 5 - 19 = -14 \\ x_2 &= \frac{-7}{2} \end{aligned}$$

Now, solve for x_1, given x_3 and x_2:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 5 \\ x_1 &= 5 - 2x_2 - 3x_3 \\ x_1 &= 5 - 2 \cdot \frac{-7}{2} - 3 \cdot \frac{19}{2} \\ &= 5 + 7 - \frac{57}{2} = \frac{-33}{2} \end{aligned}$$

So we have

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{-33}{2} \\ \frac{-7}{2} \\ \frac{19}{2} \end{bmatrix} \tag{5.21}$$

In general if you have already solved for $x_{k+1} \ldots x_n$, then you can solve for x_k using the coefficients $a_k, a_{k+1}, \ldots, a_n$ from row k of the matrix in row-echelon form using Equation (5.22).

$$x_k = \frac{a_{k,n+1} - \sum_{j=k+1}^{n} a_{k,j} x_j}{a_{k,k}} \tag{5.22}$$

5.10 About Array and Vector Indices

Keep in mind that the indices in (5.22) are 1-based, but in the Python language, the indices are 0-based. The corresponding formula can be expressed as:


```
    x[k] = (m[k][n] - sum(m[k][j] * x[j]
                          for j in range(k+1, n))
            ) / m[k][k]
```

You should compute `x[k]` in reverse order starting at `x[n-1]` and ending at `x[0]`.

5.11 The Bareiss Algorithm

Standard Gaussian elimination requires dividing by the pivot $M_{k,k}$ when zeroing out the column below the diagonal. When matrix entries are integers, this division may introduce fractions, causing intermediate values to grow. When entries are floating-point numbers, dividing by a small pivot amplifies rounding errors.

From a programming perspective, avoiding this split also removes the need for special-case code. Without Bareiss, a general-purpose elimination routine would have to branch on the entry type—using integer division (`//`) for integer matrices and floating-point division (`/`) for float matrices—or maintain two separate implementations. Bareiss defers division to a single step where the divisor is known in advance, so the caller supplies one function (`divide=lambda a,b: a//b` for integers, the default `a/b` for floats) and the algorithm itself is identical in both cases. As we just saw in the tests, the same implementation also works unchanged for matrices whose entries are any commutative ring—Gaussian integers, polynomials, or anything else for which exact division is defined.

The *Bareiss algorithm* avoids both problems with the following recurrence. At step k, for every row $i > k$ and column $j \geq k$:

$$M_{i,j} \leftarrow \frac{M_{k,k} \cdot M_{i,j} - M_{i,k} \cdot M_{k,j}}{M_{k-1,k-1}}, \tag{5.23}$$

where we set $M_{-1,-1} = 1$. For integer matrices, the numerator is always divisible by $M_{k-1,k-1}$, so no fractions ever appear.

Example 5.11.1 (*Bareiss Elimination* (3×3))

Let

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 0 \\ 1 & 0 & 2 \end{bmatrix}.$$

Step $k = 0$ (pivot $M_{0,0} = 2$, previous pivot $= 1$):

$$M_{1,1} = \frac{2 \cdot 3 - 1 \cdot 1}{1} = 5, \qquad M_{1,2} = \frac{2 \cdot 0 - 1 \cdot 1}{1} = -1,$$
$$M_{2,1} = \frac{2 \cdot 0 - 1 \cdot 1}{1} = -1, \qquad M_{2,2} = \frac{2 \cdot 2 - 1 \cdot 1}{1} = 3.$$

All entries in column 0 below the pivot become 0:

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 5 & -1 \\ 0 & -1 & 3 \end{bmatrix}.$$

Step $k = 1$ (pivot $M_{1,1} = 5$, previous pivot $M_{0,0} = 2$):

$$M_{2,2} = \frac{5 \cdot 3 - (-1) \cdot (-1)}{2} = \frac{14}{2} = 7.$$

The matrix is now upper triangular:

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 5 & -1 \\ 0 & 0 & 7 \end{bmatrix}.$$

The bottom-right entry, 7, is $\det(A)$.

Example 5.11.2 (*Bareiss Elimination with pivoting* (4×4))

Let

$$A = \begin{bmatrix} 0 & 2 & 1 & -1 \\ 4 & 1 & 2 & 3 \\ 2 & 0 & 3 & 1 \\ 1 & 2 & 0 & 2 \end{bmatrix}.$$

The leading entry $M_{0,0} = 0$ cannot serve as a pivot. Column 0 has absolute values $0, 4, 2, 1$; the largest is 4 at row 1, so we swap rows 0 and 1. This single swap means the sign will contribute a factor of -1 to the determinant.

$$\begin{bmatrix} 4 & 1 & 2 & 3 \\ 0 & 2 & 1 & -1 \\ 2 & 0 & 3 & 1 \\ 1 & 2 & 0 & 2 \end{bmatrix}$$

Step $k = 0$ (pivot $M_{0,0} = 4$, previous pivot $= 1$):

$$M_{1,1} = \tfrac{4\cdot 2 - 0\cdot 1}{1} = 8, \quad M_{1,2} = \tfrac{4\cdot 1 - 0\cdot 2}{1} = 4, \quad M_{1,3} = \tfrac{4\cdot(-1) - 0\cdot 3}{1} = -4,$$
$$M_{2,1} = \tfrac{4\cdot 0 - 2\cdot 1}{1} = -2, \quad M_{2,2} = \tfrac{4\cdot 3 - 2\cdot 2}{1} = 8, \quad M_{2,3} = \tfrac{4\cdot 1 - 2\cdot 3}{1} = -2,$$
$$M_{3,1} = \tfrac{4\cdot 2 - 1\cdot 1}{1} = 7, \quad M_{3,2} = \tfrac{4\cdot 0 - 1\cdot 2}{1} = -2, \quad M_{3,3} = \tfrac{4\cdot 2 - 1\cdot 3}{1} = 5.$$

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

Step $k = 1$: column 1 (rows 1–3) has absolute values $8, 2, 7$; row 1 is already the largest, so no swap. Pivot $M_{1,1} = 8$, previous pivot $M_{0,0} = 4$:

$$M_{2,2} = \tfrac{8\cdot 8 - (-2)\cdot 4}{4} = \tfrac{72}{4} = 18, \quad M_{2,3} = \tfrac{8\cdot(-2) - (-2)\cdot(-4)}{4} = \tfrac{-24}{4} = -6,$$
$$M_{3,2} = \tfrac{8\cdot(-2) - 7\cdot 4}{4} = \tfrac{-44}{4} = -11, \quad M_{3,3} = \tfrac{8\cdot 5 - 7\cdot(-4)}{4} = \tfrac{68}{4} = 17.$$

$$\begin{bmatrix} 4 & 1 & 2 & 3 \\ 0 & 8 & 4 & -4 \\ 0 & 0 & 18 & -6 \\ 0 & 0 & -11 & 17 \end{bmatrix}$$

Step $k = 2$: column 2 (rows 2–3) has absolute values $18, 11$; row 2 is already the largest, so no swap. Pivot $M_{2,2} = 18$, previous pivot $M_{1,1} = 8$:

$$M_{3,3} = \frac{18 \cdot 17 - (-11) \cdot (-6)}{8} = \frac{306 - 66}{8} = \frac{240}{8} = 30.$$

The matrix is now upper triangular:

$$\begin{bmatrix} 4 & 1 & 2 & 3 \\ 0 & 8 & 4 & -4 \\ 0 & 0 & 18 & -6 \\ 0 & 0 & 0 & 30 \end{bmatrix}.$$

One row swap was performed, so $\det(A) = (-1)^1 \times 30 = -30$.

5.11.1 Determinant as a Byproduct

After the forward Bareiss pass, the bottom-right entry of the resulting upper-triangular matrix equals $\det(A)$, up to the sign from any row swaps:

$$\det(A) = (-1)^s \cdot M^{\text{final}}_{n-1,n-1},$$

where s is the number of row swaps performed. The determinant comes for free—no separate computation is needed.

5.11.2 Implementing Bareiss in Python

The method `make_upper_triangular_bareiss` in `Matrix.py` performs the forward Bareiss pass. Like `make_row_echelon`, it is a method of `Matrix` so it can be applied to augmented matrices $[A \mid b]$. It returns a pair `(tri, det)`: the upper-triangular matrix and the determinant. When the matrix is singular, `tri` is `None` and `det` is `0`.

Pass `divide=lambda a,b: a//b` to use exact integer division throughout, which keeps all entries in $\mathbb{Z}$ when the input is an integer matrix. The default `a/b` uses floating-point division.

Skipping already-zero columns. At step k, every entry $M_{i,j}$ with $j < k$ is already zero from earlier steps. The Bareiss recurrence would reproduce zero for these entries, so there is no need to compute them. Skipping them reduces the number of arithmetic operations and, for floating-point matrices, eliminates the risk of accumulating rounding error in entries that are supposed to be exactly 0.

The implementation reflects this: the `updated` closure inside `make_upper_triangular_bareiss` returns the existing value unchanged whenever `j < k`:

```
def updated(i, j, ...):
    if i <= k:
        return m[i][j]    # rows above pivot row are untouched
    if j < k:
        return m[i][j]    # left of pivot column already zero -- skip
    return divide(pk * m[i][j] - m[i][k] * m[k][j], prev)
```

For generic entry types such as matrices-of-matrices or polynomials, the "zero" stored in those positions is whatever the previous step left there—the type-specific zero of the entry type, not necessarily the Python integer `0`. Skipping the recomputation means the code never needs to construct or compare against a type-specific zero, which is another reason the implementation is type-generic.

5.11.3 Entry-Type Requirements

The Bareiss algorithm requires that entry multiplication be *commutative*: $a \cdot b = b \cdot a$ for any two entries a and b. A number system with commutative multiplication (together with the usual addition rules) is called a *commutative ring*; if you took the BCS Algebra course,

you will recognise this from the rings discussion in Chapter 3. This is a mathematical requirement of the algorithm, not just an implementation detail: the correctness proof relies on the Desnanot-Jacobi determinant identity, which only holds over commutative rings. Ordinary matrix multiplication, for example, is *not* commutative, so a matrix whose entries are themselves matrices (other than the special complex-number representation we tested) would not satisfy this requirement.

All of our concrete use cases do satisfy it: integers, floating-point numbers, complex numbers, and polynomials all have $ab = ba$.

Because the Bareiss recurrence only uses arithmetic on the matrix entries themselves—never on indices or on the size of the matrix—the implementation works for any commutative ring, provided the entry type supports the following Python magic methods:

- `__add__` and `__sub__`: addition and subtraction of two entries.
- `__mul__`: multiplication of two entries (must be commutative, as discussed above).
- `__rmul__`: left-multiplication by a scalar (*e.g.* `sign * entry`). Required so that Python can dispatch `scalar * entry` correctly when the scalar's own `__mul__` does not know about the entry type.
- A `comparator` callable that accepts an entry and returns a real number: zero (falsy) for the zero element, positive otherwise. Used by `find_pivot_row` to select the best pivot row. The default `abs` is correct for numbers; other types need a custom comparator.
- A `divide(a, b)` callable that performs *exact* division. The Bareiss recurrence guarantees the numerator is always divisible

by the previous pivot, so no remainder ever arises. The caller supplies the appropriate operation: `a // b` for integers, `a / b` for floats, polynomial exact division for polynomials.

- A `one` value—the multiplicative identity for the entry type—used to initialise the "virtual" previous pivot $M_{-1,-1} = 1$.

Python's standard library provides `fractions.Fraction`, which represents exact rational numbers p/q and satisfies all six requirements with no extra work. `abs(Fraction(0))` is falsy, so the default `comparator=abs` works unchanged; `Fraction` interoperates freely with Python integers, so the default `one=1` also works. The only argument that must be supplied is `divide`:

```
from fractions import Fraction
rat_divide = lambda a, b: a / b
```

Example 5.11.3 (*Rational entries:* 2×2 *inverse*)

Let

$$A = \begin{bmatrix} \frac{1}{2} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{5} \end{bmatrix}, \qquad \det(A) = \frac{1}{2}\cdot\frac{1}{5} - \frac{1}{3}\cdot\frac{1}{4} = \frac{1}{10} - \frac{1}{12} = \frac{1}{60}.$$

Because $\det(A) = \frac{1}{60}$, the adjugate entries are multiples of $\frac{1}{60}$, and the inverse turns out to be an *integer* matrix:

$$A^{-1} = \begin{bmatrix} 12 & -20 \\ -15 & 30 \end{bmatrix}.$$

```
A = SqMatrix([[Fraction(1, 2), Fraction(1, 3)],
              [Fraction(1, 4), Fraction(1, 5)]])

tri, det = A.make_upper_triangular_bareiss(divide=rat_divide)
# det == Fraction(1, 60)  -- exact, no floating-point error

I_q = Matrix([[Fraction(1), Fraction(0)],
```

```
                [Fraction(0), Fraction(1)]])
diag, det = A.adjoin_cols(I_q) \
              .make_diagonal_bareiss(divide=rat_divide)
n = 2
A_inv = SqMatrix.tabulate(n, lambda i, j: diag[i][n + j] / det)
# A_inv == SqMatrix([[12, -20], [-15, 30]])
```

Example 5.11.4 (*Rational entries:* 3×3 *Hilbert matrix*)

The *Hilbert matrix* H has entries $H_{ij} = \dfrac{1}{i+j+1}$ (0-indexed). Its determinant is a famous test of exact arithmetic:

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}, \qquad \det(H) = \frac{1}{2160}\,.$$

A floating-point implementation accumulates rounding error; Bareiss with `Fraction` entries computes the result exactly:

```
H = SqMatrix.tabulate(3, lambda i, j: Fraction(1, i + j + 1))
_, det = H.make_upper_triangular_bareiss(divide=rat_divide)
# det == Fraction(1, 2160)  -- exact
```

In Chapter 10 we will need exactly this generality. Computing the *characteristic polynomial* $\det(A - \lambda I)$ requires row-reducing a matrix whose entries are polynomials in λ rather than numbers. All six requirements above are met by polynomials: they support the standard arithmetic operations, exact polynomial division is well defined whenever the leading term of the divisor divides that of the dividend, and the pivot strategy should prefer entries of low degree to minimise expression growth during elimination.

5.12 Back-Substitution as Row Operations

Section 5.9 solved for the unknowns one at a time after the forward pass. Each back-substitution step is equivalent to a row operation: to eliminate $M_{i,k}$ for row $i < k$, multiply row i by the pivot $M_{k,k}$ and subtract $M_{i,k}$ times row k:

$$M_{k,k}\, R_i - M_{i,k}\, R_k \;\rightarrow\; R_i\,. \tag{5.24}$$

No new pivoting is needed—the diagonal entries are already in place from the forward pass. Applying (5.24) for $k = n-1$ down to $k = 1$, eliminating every row $i < k$ at each step, produces a diagonal matrix.

Example 5.12.1 (*Backward elimination*)

Continuing Example 5.11.1, the forward Bareiss pass gave

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 5 & -1 \\ 0 & 0 & 7 \end{bmatrix}.$$

Step $k = 2$ (pivot $M_{2,2} = 7$):

$$7R_1 - (-1)R_2 \rightarrow R_1 \Rightarrow \begin{bmatrix} 2 & 1 & 1 \\ 0 & 35 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

$$7R_0 - 1 \cdot R_2 \rightarrow R_0 \Rightarrow \begin{bmatrix} 14 & 7 & 0 \\ 0 & 35 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

Step $k = 1$ (pivot $M_{1,1} = 35$):

$$35R_0 - 7R_1 \to R_0 \Rightarrow \begin{bmatrix} 490 & 0 & 0 \\ 0 & 35 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

The matrix is now diagonal. Dividing each row by its diagonal entry gives the identity. Note that the bottom-right entry, $M_{2,2} = 7$, is unchanged by the backward pass and equals $\det(A)$.

5.12.1 Matrix Inversion

Applying the same two-pass procedure to $\left[A \mid I\right]$ computes the inverse. The row operations transform both blocks simultaneously; dividing each row i by $M_{i,i}$ at the end gives:

$$\left[A \mid I\right] \xrightarrow{\text{forward + backward}} \left[D \mid B\right] \xrightarrow{\boldsymbol{\nabla} \cdot M_{i,i}} \left[I \mid A^{-1}\right] .$$

Example 5.12.2 (*Inverse via row operations*)

Let $A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 0 \\ 1 & 0 & 2 \end{bmatrix}$. After the forward Bareiss pass on $\left[A \mid I\right]$:

$$\left[\begin{array}{ccc|ccc} 2 & 1 & 1 & 1 & 0 & 0 \\ 0 & 5 & -1 & -1 & 2 & 0 \\ 0 & 0 & 7 & -3 & 1 & 5 \end{array}\right]$$

After the backward row operations:

$$\left[\begin{array}{ccc|ccc} 490 & 0 & 0 & 420 & -140 & -210 \\ 0 & 35 & 0 & -10 & 15 & 5 \\ 0 & 0 & 7 & -3 & 1 & 5 \end{array}\right]$$

Dividing rows by 490, 35, 7 respectively:

$$A^{-1} = \frac{1}{7}\begin{bmatrix} 6 & -2 & -3 \\ -2 & 3 & 1 \\ -3 & 1 & 5 \end{bmatrix}.$$

The normalization denominators (490, 35, 7) are in general different for each row. For an integer matrix with $\det(A) = \pm 1$, however, each denominator exactly divides every entry in its row, so A^{-1} has all-integer entries. Pass `divide=lambda a,b: a//b` to `gauss_jordan_inverse` to compute such inverses without any floating-point arithmetic.

This combined algorithm—forward Gaussian elimination, backward row operations, then row normalization—is known as *Gauss-Jordan elimination* (GJE). Chapter A covers GJE in more detail as enrichment material.

5.13 Programming

1. Return multiple values as a tuple: `return (matrix, det)`
2. Callable parameters: passing a function as an argument, *e.g.* `divide=lambda a,b: a//b`

5.14 Programming Exercises

5.14.1 Homework

Two methods are *provided* in `Matrix.py` for you to study:

- `make_row_echelon` — standard floating-point Gaussian elimination (forward pass). Read this carefully: it shows exactly how

swap_rows and row_operation combine to eliminate entries below the diagonal. Your back_substitution will complete the solve once this forward pass is done.

- make_diagonal_bareiss — the backward pass of the Bareiss algorithm. Compare its inner loop with back_substitution: both perform back-substitution, but make_diagonal_bareiss expresses the same computation entirely as row operations (zeroing entries *above* the diagonal) rather than computing each solution component directly. This lets the result stay in matrix form so the inverse can be read off the right-hand block.

You must implement:

- back_substitution in Matrix.py — given an augmented matrix already in row-echelon form (as returned by make_row_echelon), compute and return the solution vector. See the back-substitution algorithm in Section **??**.
- make_upper_triangular_bareiss in Matrix.py — the forward pass of the Bareiss algorithm. The key step is the fraction-free row operation; see Section 5.11.
- Test with:
  - gaussian_test.py — exercises back_substitution (and indirectly your row-operation implementations via the provided make_row_echelon)
  - bareiss_test.py — exercises make_upper_triangular_bareiss

Chapter 6

Metric Spaces

Chapter Objectives

- Define metric spaces and verify the four distance-function axioms for concrete examples.
- Identify metric spaces arising in linear algebra: Euclidean distance, discrete metric, Manhattan metric, and geodesic distance.
- Implement `is_metric_space` using property-based testing.

A *metric space* is a set equipped with a notion of distance. Not every function can serve as a distance; the function must satisfy four axioms that capture the properties we expect distance to have. This chapter defines those axioms, studies several concrete examples, and introduces the property-based testing technique used to verify them programmatically.

6.1 Distance Functions

Not every function from $M \times M$ to $\mathbb{R}$ deserves to be called a distance function. A distance function must obey certain axioms. These properties of the distance function are abstractions from the notion of distance you already have for measuring distance in geometry. However, distance might not be physical distance, measured in centimeters, we may also talk about distance in time, or distance in relativistic spacetime, or distance in temperature, or distance in more abstract sets for which there may not be an obvious physical interpretation.

Definition 6.1.1 (*metric space, distance*)

A set M together with a function $d : M \times M \to \mathbb{R}$ is called a *metric space* provided:

1. (positivity)
   $x, y \in M \implies d(x, y) \geq 0.$
2. (definite)
   $x, y \in M \implies (d(x, y) = 0 \iff x = y).$
3. (symmetry)
   $x, y \in M \implies d(x, y) = d(y, x).$
4. (triangle inequality)
   $x, y, z \in M \implies d(x, z) \leq d(x, y) + d(y, z).$

Keep in mind that a metric space is a set. So the normal relations of subset ($\subset$), and element ($\in$), as well as notions such as $\emptyset$, still apply. However, in addition to the properties of general sets, we also have the properties of a distance function.

6.1.1 Examples of Metric Spaces

Proposition 6.1.2

The integers, $\mathbb{Z}$ together with the function $(x, y) \mapsto |x - y|$ is a simple and familiar *metric space*.

Proof. We must prove the four axioms in Definition 6.1.1 Let $x, y, z \in \mathbb{Z}$,

1. **Positivity** $d(x, y) = |x - y| \geq 0$
2. **Definite** ($\implies$) Suppose $d(x, y) = 0$ (prove $x = y$). $0 = d(x, y) = |x - y|$. So $x = y$. ($\impliedby$) Suppose $x = y$ (prove $d(x, y) = 0$). $d(x, y) = d(x, x) = |x - x| = |0| = 0$.
3. **Symmetric** $d(x, y) = |x - y| = |y - x| = d(y, x)$
4. **Triangle Inequality** $d(x, y) + d(y, z) = |x - y| + |y - z|$ [a] Recall the definition: $|a + b| = \max\{a + b, -(a + b)\}$.

$$
\begin{aligned}
x &\leq |x| \\
x + y &\leq |x| + y \\
&\leq |x| + |y| && \text{because } y \leq |y| \\
-(x + y) &= -x + -y \\
&\leq |-x| + -y && \text{because } -x \leq |-x| \\
&\leq |-x| + |-y| && \text{because } -y \leq |-y| \\
&= |x| + |y|
\end{aligned}
$$

Thus, $|x + y| = \max\{x + y, -(x + y)\} \leq |x| + |y|$.

[a] Proof taken from `https://math.stackexchange.com/a/307352/400222`. Thanks to k.stm `https://math.stackexchange.com/users/42242/k-stm`.

- An important example of a *metric space* you will see in this course is a *normed vector space*, which we will see in Chapter 8. From Section 1.6, $\mathbb{R}^n$ with Euclidean distance

$$\begin{aligned} d(u,v) &= \|u-v\| \\ &= \left\| \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right\| \\ &= \sqrt{\sum_{k=1}^{n} (a_k - b_k)^2} \end{aligned}$$

- Discrete metric:

$$d(x,y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases} \tag{6.1}$$

- Manhattan metric

$$d(u,v) = \sum_{k=1}^{n} |a_k - b_k| \tag{6.2}$$

- Geodesic distance, see Section 6.1.2.
- Levenshtein distance, see Section 6.1.4.
- Longest prefix distance, see Section 6.1.4.

6.1.2 Geodesic distance

The globe is the set of pairs $\begin{bmatrix} \phi \\ \lambda \end{bmatrix}$ representing latitude, ϕ, and longitude, λ, where $-90 \leq \phi \leq 90$ and $-180 < \lambda \leq 180$. We must be careful to

assure that each point on the globe corresponds to exactly one $\begin{bmatrix}\phi\\ \lambda\end{bmatrix}$. To avoid that multiple points such as $\begin{bmatrix}90\\ 10\end{bmatrix}$ and $\begin{bmatrix}90\\ 20\end{bmatrix}$ (and others) correspond to the north pole (similar for south pole), we exclude $\begin{bmatrix}\phi\\ \lambda\end{bmatrix}$ values for which $\phi = -90$ and $\lambda \neq 0$. *I.e.*, we exclude the lines $\phi = 90$ and $\phi = -90$. Unfortunately, this would also exclude the north and south poles entirely, so we must add back those two special points to have a complete globe. Precisely, the set which forms the metric space is:

$$S = \left\{ \begin{bmatrix}\phi\\ \lambda\end{bmatrix} \middle| -90 < \phi < 90, -180 < \lambda \leq 180 \right\} \cup \left\{ \begin{bmatrix}-90\\ 0\end{bmatrix}, \begin{bmatrix}90\\ 0\end{bmatrix} \right\} .$$

There are several ways of computing the distance on the surface of the earth, supposing that the earth is a perfect sphere.

1. Basic approach: In this case,

$$d\left(\begin{bmatrix}\phi_1\\ \lambda_1\end{bmatrix}, \begin{bmatrix}\phi_2\\ \lambda_2\end{bmatrix} \right) = r \cos^{-1}(\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos\Delta\lambda) , \tag{6.3}$$

   where r is the radius of the earth; *e.g.* $r \approx 6371$ km, and $\Delta\lambda = \lambda_1 - \lambda_2$.

2. Vincenty formula, which is less sensitive to computational round-off error.

   An alternate formula for geodesic distance is given by the *Vincenty formula*:

$$d\left(\begin{bmatrix}\phi_1\\ \lambda_1\end{bmatrix}, \begin{bmatrix}\phi_2\\ \lambda_2\end{bmatrix} \right) = r \tan^{-1} \frac{\sqrt{A^2 + B^2}}{C} \tag{6.4}$$

Where

$$A = \cos\phi_2 \sin\Delta\lambda$$
$$B = \cos\phi_1 \sin\phi_2 - \sin\phi_1 \cos\phi_2 \cos\Delta\lambda$$
$$C = \sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos\Delta\lambda$$

3. Vector version (using $\cos^{-1}$) : This version uses 3D normal vectors. Let

$$n_1 = \begin{bmatrix} \cos\phi_1 \cos\lambda_1 \\ \cos\phi_1 \sin\lambda_1 \\ \sin\phi_1 \end{bmatrix} \tag{6.5}$$

$$n_2 = \begin{bmatrix} \cos\phi_2 \cos\lambda_2 \\ \cos\phi_2 \sin\lambda_2 \\ \sin\phi_2 \end{bmatrix} \tag{6.6}$$

We have the formula for geodesic distance:

$$d\left(\begin{bmatrix}\lambda_1 \\ \phi_1\end{bmatrix}, \begin{bmatrix}\lambda_2 \\ \phi_2\end{bmatrix}\right) = r\cos^{-1}(n_1 \cdot n_2) \tag{6.7}$$

4. Vector version (using $\sin^{-1}$ and $\tan^{-1}$): We have two formulas for geodesic distance:

$$d\left(\begin{bmatrix}\lambda_1 \\ \phi_1\end{bmatrix}, \begin{bmatrix}\lambda_2 \\ \phi_2\end{bmatrix}\right) = r\sin^{-1}\|n_1 \times n_2\| \tag{6.8}$$

$$= r\tan^{-1}\frac{\|n_1 \times n_2\|}{n_1 \cdot n_2}, \tag{6.9}$$

where n_1 and n_2 are given by Equations (6.5) and (6.6), respectively.

The cross product, $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$ of two vectors in $\mathbb{R}^3$ is computed as follows.

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} \tag{6.10}$$

6.1.3 Geodesic Distance in Python

To implement the `geodesic_distance` function in Python, we will use Equation (6.7). This approach makes use of the `angle` method which you have already implemented in `Vector.py`. We assume the radius of the earth in km is `r_earth = 6371.0`. Use Equations (6.5) and (6.6) to compute n_1 and n_2, then use the `angle` method to compute the angle between the two vectors.

```
def geodesic_distance(loc1, loc2) -> float:
    import math
    r_earth = 6371.0  # radius of earth in km
    phi1d, lambda1d = loc1  # angles in degrees
    phi2d, lambda2d = loc2

    phi1 = math.radians(phi1d)  # convert to radians
    phi2 = math.radians(phi2d)
    lambda1 = math.radians(lambda1d)
    lambda2 = math.radians(lambda2d)
    n1 = Vector(...)
    n2 = Vector(...)
    return r_earth * ...
```

6.1.4 String distance

There are two common ways to describe the distance between strings in a programming language such as Python. The first is called the `Levenshtein distance` (Definition 6.1.3 which counts the number of

edits (character insertions, deletions, replacement) a person would have to do to convert one string into the other. While the Levenshtein distance is an intuitive way to measure distance between strings, it is computationally expensive to calculate. For this reason, a second definition of distance is sometimes used, called the *longest-common-prefix distance* (Definition 6.1.4).

Both of these definitions do indeed define distances according to Definition 6.1.1. However, we omit the proofs.

Definition 6.1.3 (*Levenshtein Distance*)

Given two finite possibly empty strings, a, b:

$$lev(a,b) = \begin{cases} len(a) & \text{if } len(b) = 0 \\ len(b) & \text{if } len(a) = 0 \\ lev(a[1:], b[1:]) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} lev(a[1:], b) \\ lev(a, b[1:]) \\ lev(a[1:], b[1:]) \end{cases} & \text{otherwise} \end{cases}$$

To compute the Levenshtein distance in Python take a look at the following code. Notice that the code follows very closely Definition 6.1.3.

```
def lev(a, b) -> int:
    if len(a) == 0:
        return len(b)
    elif len(b) == 0:
        return len(a)
    elif a[0] == b[0]:
        return lev(a[1:], b[1:])
    else:
        return 1 + min(lev(a[1:], b),
                       lev(a, b[1:]),
                       lev(a[1:], b[1:]))
```

Definition 6.1.4 (*Longest Common Prefix Distance*)

Let p be the longest common prefix of strings a and b so that $a = p :: a_{\text{suffix}}$ and $b = p :: b_{\text{suffix}}$, where :: denotes string concatenation.

$$lcp(a, b) = len(a) + len(b) - 2 \times len(p)$$

To implement Definition 6.1.4 in Python, the code follows closely the formula for $lcp(a, b)$ in the definition. We have to implement a local function to compute $len(p)$ without actually computing p directly.

```
def lcp_distance(a, b) -> int:
  def len_longest_prefix(s1, s2):
    if s1 == s2:
      return len(s1)
    elif s1 == "":
      return 0
    elif s2 == "":
      return 0
    elif s1[0] == s2[0]:
      # compute 1 + lcp of the tails of the two strings
      return 1 + ...
    else:
      # the strings are different and have no common prefix
      return ...

  return len(a) + len(b) - 2 * len_longest_prefix(a,b)
```

6.2 Programming Exercises

In the homework exercises you will write a function that tests whether a given distance function makes a set into a metric space. You will be given a generator that produces randomly selected elements of the set. Your task is to verify each of the four distance axioms against a large number of randomly selected element pairs.

The key techniques are:

1. Property-based testing: verify a predicate holds for many random samples
2. Almost-equal comparison: use a small ε instead of exact equality
3. Manipulating Boolean values
4. `all`

1. Property based testing
2. Almost equal
3. Manipulating Boolean values
4. `all`

6.2.1 Classwork

1. Start `MetricSpace.py`

6.2.2 Homework

- Implement `MetricSpace.py`
  - `discrete_distance`
  - `manhattan_distance`
  - `geodesic_distance`
  - `is_metric_space`
- Test with `MetricSpace_test.py`

Chapter 7

Vector Spaces

Chapter Objectives

- Recall the definition of a real vector space (Chapter 3) and recognise that it subsumes the eight individual axioms found in many textbooks.
- Study examples of real vector spaces beyond $\mathbb{R}^n$ and $\mathbb{R}^{n\times m}$: spaces of polynomials and continuous functions.
- Identify subspaces and verify the subspace criterion.
- Define linear combinations, span, and linear independence, and determine whether a given set of vectors is linearly independent.

In Chapter 3 we defined a real vector space as an Abelian group equipped with an $\mathbb{R}$-scaling operation, and verified that $\mathbb{R}^{n\times m}$ is one. In this chapter we study vector spaces in depth: abstract examples beyond $\mathbb{R}^n$, subspaces, linear combinations, span, and linear independence.

7.1 Vector Space Formally

Recall from Chapter 3 (Definition 3.7.2) that a real vector space is an Abelian group $(V, +)$ equipped with an $\mathbb{R}$-scaling operation. The concept arises throughout mathematics, signal processing, control systems, and quantum mechanics.

This definition is intentionally terse. Many textbook treatments of vector spaces instead list eight individual axioms:

- four axioms for the Abelian group $(V, +)$: closure, associativity, identity, and inverse;
- four axioms for the scaling operation: $1v = v$, $(s+t)v = sv + tv$, $s(u+v) = su + sv$, and $(st)v = s(tv)$.

Our definition subsumes all eight by referring to the two structures — Abelian group and scaling operation — that we have already defined and studied. The definitions are equivalent.

In an entirely analogous way we may define a *complex vector space* as an Abelian group $(V, +)$ equipped with a $\mathbb{C}$-scaling operation, *i.e.* a map $\mathbb{C} \times V \to V$ satisfying the same four axioms with $\mathbb{C}$ in place of $\mathbb{R}$. Complex vector spaces arise naturally in signal processing, quantum mechanics, and control theory.

From the programming perspective, a real vector space involves two operations that a caller supplies — vector addition and scalar-vector multiplication — and two that are always fixed because the scalars are the real numbers: ordinary real addition and real multiplication. A function `is_vector_space` therefore only needs the vector-side operations as arguments; it can test the four scaling axioms using Python's built-in arithmetic on floats.

Definition 7.1.1 (*subspace*)

If V is a real vector space and $W \subset V$ is closed in the sense that

- $u \in W, \alpha \in \mathbb{R} \implies \alpha u \in W$
- $u, v \in W \implies u + v \in W$,

then W is referred to as a *subspace* of V.

Theorem 7.1.2 (*Subspaces are Vector Spaces*)

Every subspace of a real vector space is itself a real vector space.

A very common example of a real vector space is the set of n-dimensional vectors with components in $\mathbb{R}$. For example the set of real ordered 3-tuples forms a real vector space. In previous chapters you have already manipulated vectors from $\mathbb{R}^n$ and written Python functions to add, subtract, and scale these vectors.

We may easily extend $\mathbb{R}^n$ to $\mathbb{C}^n$, *i.e.* the set of all n-tuples whose components are complex numbers.

Example 7.1.3

The student may verify that $\mathbb{C}^n$ is a complex vector space.

7.2 Vector Spaces of Functions

There are other examples of vector spaces which are very different from $\mathbb{R}^n$, for example, the set of continuous functions on an interval is a vector space.

Definition 7.2.1 ($\mathcal{C}^0[a,b]$)

Let $\mathcal{C}^0[a,b]$ denote the set of continuous, real-valued functions on the interval $[a,b]$ where $a, b \in \mathbb{R}$.

Theorem 7.2.2 ($\mathcal{C}^0[a,b]$ ***is a vector space***)

$\mathcal{C}^0[a,b]$ is a vector space under the following operations

- If $f, g \in \mathcal{C}^0[a,b]$, then the sum is $f+g$. $f+g$ denotes the function whose value at x is $f(x)+g(x)$
- if $f \in \mathcal{C}^0[a,b]$ and $\alpha \in \mathbb{R}$, then the scalar product αf denotes the function whose value at x is $\alpha f(x)$.

Proof. omitted

We omit the proof here, but it would be a useful but tedious exercise to verify that $\mathcal{C}^0[a,b]$ is indeed a vector space.

Because of Theorem 7.1.2, every closed subset of $\mathcal{C}^0[a,b]$ is also a vector space. One such subset is called $\mathcal{C}^1[a,b]$.

Definition 7.2.3 ($\mathcal{C}^n[a,b]$)

If $n \in \mathbb{N}$, we define $\mathcal{C}^n[a,b]$ to mean the set of n-times differentiable functions on $[a,b]$. In particular, $\mathcal{C}^1[a,b]$ is the set of (1-time) differentiable functions on $[a,b]$.

Theorem 7.2.4

If $n \in \mathbb{N}$, then $\mathcal{C}^n[a,b]$ is a vector space.

Proof. We need only show that $\mathcal{C}^n[a,b] \subset \mathcal{C}^0[a,b]$, and that $\mathcal{C}^n[a,b]$ is closed under addition and scalar multiplication.

- If f is differentiable (n-times for $n > 0$) then it is necessarily continuous.
- Let f and g be differentiable, then $f+g$ is also differentiable and in fact
$$\frac{d^n}{dx^n}(f+g) = \frac{d^n}{dx^n}f + \frac{d^n}{dx^n}g\,.$$
- Let f be differentiable, and let $\alpha \in \mathbb{R}$. Then αf is also differentiable and in fact
$$\frac{d^n}{dx^n}(\alpha f) = \alpha\frac{d^n}{dx^n}f\,.$$

Theorem 7.2.5

Define $\mathcal{C}^n = \bigcup_{a,b\in\mathbb{R}} \mathcal{C}^n[a,b]$. $\mathcal{C}^n$ is a vector space.

Proof. omitted

7.3 Subspaces

One powerful tool we have of generating (or identifying) vector spaces is by taking subsets of existing (known) vector spaces and showing they are closed. We already know from Theorem 7.1.2 that closed subspaces are themselves vector spaces.

Theorem 7.3.1 (*Linear Combinations*)

Let V be a real vector space. Let $S \subset V$, with $S \neq \emptyset$. The set of linear combinations of S, *i.e.* span(S), is a real vector space.

Proof. We only need to show span(S) is a subspace of V. Then by Theorem 7.1.2 it is a real vector space.

- Closure under scalar multiplication: Let $s \in \mathbb{R}$. Let $u \in \text{span}(S)$. Show $su \in \text{span}(S)$.

  Choose $\alpha_1, \ldots, \alpha_m \in \mathbb{R}$ and $u_1, \ldots, u_m \in S$ such that $u = \sum_{i=1}^{m} \alpha_i u_i$. Now $su = s\sum_{i=1}^{m} \alpha_i u_i = \sum_{i=1}^{m} s\alpha_i u_i$. Thus su is a linear combination of $u_1, \ldots, u_m$; hence $su \in \text{span}(S)$.

- Closure under vector addition: Let $u, v \in \text{span}(S)$. Show $u + v \in \text{span}(S)$.

  Choose $\alpha_1, \ldots, \alpha_m \in \mathbb{R}$ and $u_1, \ldots, u_m \in S$ such that $u = \sum_{i=1}^{m} \alpha_i u_i$. Choose $\beta_1, \ldots, \beta_\ell \in \mathbb{R}$ and $v_1, \ldots, v_\ell \in S$ such that $v = \sum_{i=1}^{\ell} \beta_i v_i$. So $u + v = \sum_{i=1}^{m} \alpha_i u_i + \sum_{i=1}^{\ell} \beta_i v_i$. Thus $u + v$ is a linear combination of $u_1, \ldots, u_m, v_1, \ldots, v_\ell \in S$, which means $u + v \in \text{span}(S)$.

Theorem 7.3.2 (*Vector Space of Polynomials*)

The set $\mathbb{R}[x]$ of polynomials with real coefficients is a real vector space.

Proof. Let $S = \{1, x, x^2, x^3, \ldots\} \subset \mathcal{C}^n$. *I.e.*, $S = \{x^i \mid i \in \mathbb{N}\}$. Then $\mathbb{R}[x] = \text{span}(S)$. Now apply Theorem 7.3.1.

Example 7.3.3 shows how periodic waves of many sorts (even discontinuous functions) can be approximated arbitrarily well as sums of continuous functions, in particular as scaled sums (linear combinations) of sinusoids.

Example 7.3.3 (*Sinusoidal Signals*)

Let's consider the $\sin : [0, 2\pi] \to \mathbb{R}$ and $\cos : [0, 2\pi] \to \mathbb{R}$. Let $S = \{\sin n\theta \mid n \in \mathbb{N}\}$. Now, $\text{span}(S)$ is a vector space over $\mathbb{R}$. We know this because $S \subset \mathcal{C}^0[0, 2\pi]$ and by Theorem 7.3.1.

Now let's look at a couple of linear combinations of subsets of S. Take the subset $\{\sin x, \sin 2x, \ldots, \sin 20x\}$ and take the coefficients $\alpha_n = \frac{2(-1)^{n+1}}{n\pi}$, for $n = 1, 2, \ldots, 20$.

The first six individual terms of this sum can be seen in Figure 7.1 [top-left]. A plot of the function [top-right] $\sum_{n=1}^{6} \alpha_n \sin nx$, [bottom-left] $\sum_{n=1}^{12} \alpha_n \sin nx$, and [bottom-right] $\sum_{n=1}^{20} \alpha_n \sin nx$ are also shown.

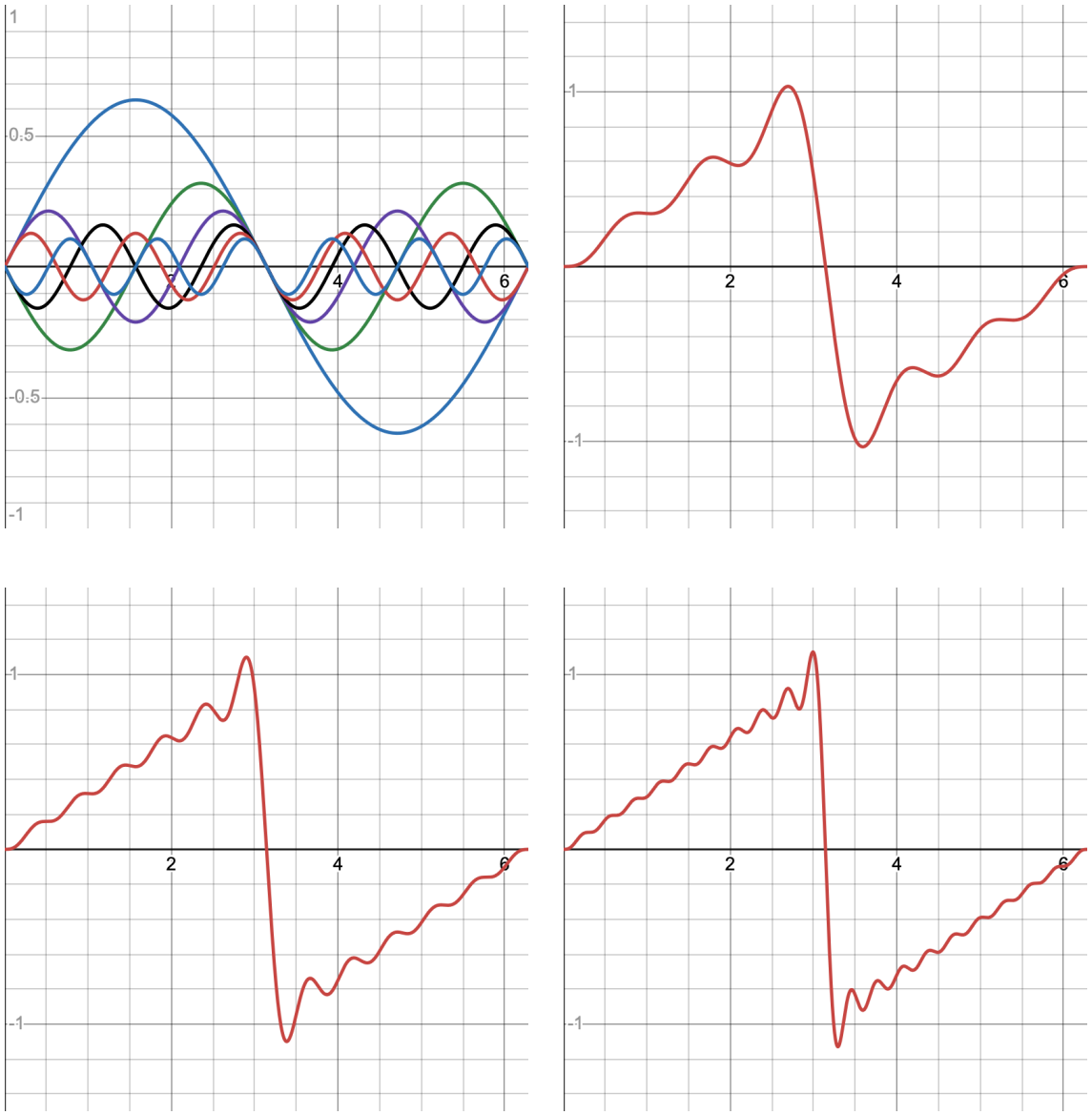


Figure 7.1: [top-left] Plot of first 6 isolated components of $\sum_{n=1}^{20} \alpha_n \sin nx$. [top-right] Plot of $\sum_{n=1}^{6} \alpha_n \sin nx$. [bottom-left] $\sum_{n=1}^{12} \alpha_n \sin nx$. [bottom-right] $\sum_{n=1}^{20} \alpha_n \sin nx$.

7.4 Programming Exercises

7.4.1 Classwork

1. Begin `VectorSpace.py`

7.4.2 Homework

- Partially implement `VectorSpace.py`
  - `is_vector_space(v_member, vv_add, sv_mult, v_identity, v_inverse, v_equivalent, v_gen, s_gen, iterations)`
    Returns `True` iff the supplied operations make $(V, +)$ an Abelian group and `sv_mult` an $\mathbb{R}$-scaling operation. The scalar generator `s_gen` should return floats; the function uses ordinary float arithmetic for the scalar axioms.
- Test with `VectorSpace_test.py`

Chapter 8

Geometry of Vector Spaces

Chapter Objectives

- Define a norm on a vector space and verify the four norm axioms for standard and non-standard examples.
- Define an inner product and derive norms, distances, and angles from it.
- State and apply the Cauchy-Schwarz inequality $|\langle u, v\rangle| \leq \|u\| \, \|v\|$.
- Compute the orthogonal projection of one vector onto another and decompose vectors into parallel and perpendicular components.

In this chapter we consider several concepts which you've already seen in $\mathbb{R}^n$ and extend them to general vector spaces. In section 7.2 we saw some examples of vector spaces. In particular, in this chapter, we will examine what it means for a vector to have a length, to measure the distance between two vectors, to measure the angle between two vectors, and to compute the projection of one vector onto

another. These concepts are intuitive for 2 and 3-dimensional real vector spaces (over $\mathbb{R}^n$). However, their meaning is more abstract over infinite dimensional vector spaces.[1]

8.1 Norm for a Vector Space

Some vector spaces are equipped with a way to measure the *length* of the vectors they contain. A function that measures length is called a *norm*.

Definition 8.1.1 (*norm*)

If V is a vector space over $\mathbb{R}$, then a function $\|\cdot\|$ is called a *norm* provided

1. (positive) $u \in V \implies \|u\| \geq 0$
2. (definite) $\|u\| = 0 \iff u = 0$
3. (scalability) $u \in V, \alpha \in \mathbb{R} \implies \|\alpha u\| = |\alpha|\|u\|$
4. (triangle inequality) $u, v \in V \implies \|u + v\| \leq \|u\| + \|v\|$

The following is an example of a vector space of functions, with a norm which *measures* the *length* of a function. This idea of *length* is abstract; it is just a function which obeys the axioms defining a norm.

Note in Example 8.1.2 that we define the norm of a function as

$$\|f\| = \sqrt{\int_0^{2\pi} f^2(t)dt}\,,$$

however in such a case it is also idiomatic to state the definition in

[1] We will discuss the dimension of a vector space in Section 9.3.

terms of the square of the norm:

$$\|f\|^2 = \int_0^{2\pi} f^2(t)dt\,.$$

When a norm is defined in this way, we accept that the norm is the positive number which fulfills this identity, as a norm is positive by definition.

Example 8.1.2 (*Normed Vector Space*)

The set, $\mathcal{C}^0[0, 2\pi]$ of real valued continuous functions on the closed interval $[0, 2\pi]$ is a vector space over $\mathbb{R}$, according to Theorem 7.2.2. We can define a norm on $\mathcal{C}^0[0, 2\pi]$ as follows. If $f : [0, 2\pi] \to \mathbb{R}$, let $\|f\| = \sqrt{\int_0^{2\pi} f^2(t)dt}$.

We may verify the axioms of Definition 8.1.1.

1. (positive) $\|f\|^2 = \int_0^{2\pi} f^2(t)dt$ is the integral of a continuous function which is non-negative, therefore its square root is 0 or positive.

2. (definite) If $f = 0$ then $\|f\|^2 = \int_0^{2\pi} 0^2 dt = 0$. Conversely, if $\|f\|^2 = 0 = \int_0^{2\pi} f(t)^2 dt$, then f being continuous must be zero.

3. (scalability) Let $\alpha \in \mathbb{R}$.

$$\begin{aligned}
\|\alpha f\| &= \sqrt{\int_0^{2\pi} (\alpha f(t))^2 dt} \\
&= \sqrt{\int_0^{2\pi} \alpha^2 f^2(t) dt} \\
&= \sqrt{\alpha^2 \int_0^{2\pi} f^2(t) dt} \\
&= |\alpha| \sqrt{\int_0^{2\pi} f^2(t) dt} \\
&= |\alpha| \|f\|
\end{aligned}$$

4. (triangle inequality) Let $f, g \in \mathcal{C}^0[0, 2\pi]$. To show this, we need the Schwarz inequality

$$\int_a^b f(t)g(t)dt \leq \sqrt{\int_a^b f^2(t)dt} \sqrt{\int_a^b g^2(t)dt}$$

$$
\begin{aligned}
\|f+g\|^2 &= \int_0^{2\pi} (f(t)+g(t))^2 dt \\
&= \int_0^{2\pi} \big(f^2(t)+g^2(t)+2f(t)g(t)\big) dt \\
&= \int_0^{2\pi} f^2(t)dt + \int_0^{2\pi} g^2(t)dt + \int_0^{2\pi} f(t)g(t)dt \\
&\leq \int_0^{2\pi} f^2(t)dt + 2\int_0^{2\pi} g^2(t)dt \\
&\qquad + 2\sqrt{\int_0^{2\pi} f^2(t)dt}\,\sqrt{\int_0^{2\pi} g^2(t)dt} \\
&= \left(\sqrt{\int_0^{2\pi} f^2(t)dt} + \sqrt{\int_0^{2\pi} g^2(t)dt}\right)^2 \\
&= (\|f\| + \|g\|)^2
\end{aligned}
$$

Taking the square root of both sides we have

$$\|f+g\| \leq \|f\| + \|g\|$$

Recall that we talked about distances and metric spaces, which are sets which allow measurement of distances between elements in Section 6.1. In Theorem 8.1.3 we show that any normed vector space is simultaneously a metric space, because the norm gives us a way to define a distance function compatible with the Definition 6.1.1 (page 177).

Theorem 8.1.3 (*A normed vector space is a metric space*)

Let V be a normed vector space over $\mathbb{R}$ If $d(u,v) : V \times V \to \mathbb{R}$ by

$$d(u,v) = \|u-v\|,$$

then V is a metric space with distance function d.

Proof. We must prove properties 1-4 of Definition 6.1.1.

1. (positive) Let $u, v \in V$ such that $u \neq v$.

$$\begin{aligned} d(u,v) &= \|u - v\| \\ &\geq 0 \end{aligned}$$

2. (definite) Let $v \in V$

$$\begin{aligned} d(v,v) &= \|v - v\| \\ &= \|0\| \\ &= 0 \end{aligned}$$

3. (symmetric) Let $u, v \in V$.

$$\begin{aligned} d(u,v) &= \|u - v\| \\ &= \|(-1)(v - u)\| \\ &= |-1|\|(v - u)\| \\ &= 1\|(v - u)\| \\ &= \|(v - u)\| \\ &= d(v,u) \end{aligned}$$

4. (triangle inequality) Let $u, v, w \in V$

$$\begin{aligned} d(u,w) &= \|u - w\| \\ &= \|u - v + v - w\| \\ &\leq \|u - v\| + \|v - w\| \\ &= d(u,v) + d(v,w) \end{aligned}$$

Example 8.1.4 (*Distance between* sin *and* cos)

As in Example 8.1.2, we consider the vector space $\mathcal{C}^0[0, 2\pi]$. We will find the distance between vectors u and v where $u = \sin$ and $v = \cos$, which as we see are functions (vectors) in $\mathcal{C}^0[0, 2\pi]$, as they are continuous functions on the closed interval $[0, 2\pi]$.

$$(d(u, v))^2 = (d(\sin, \cos))^2 = \|\sin - \cos\|^2$$

$$\begin{aligned}
\|\sin - \cos\|^2 &= \int_0^{2\pi} (\sin(t) - \cos(t))^2 dt \\
&= \int_0^{2\pi} \left(\underbrace{\sin^2(t) + \cos^2(t)}_{=\,1} - 2\sin(t)\cos(t) \right) dt \\
&= \int_0^{2\pi} (1 - 2\sin(t)\cos(t)) dt \\
&= \int_0^{2\pi} 1 dt - 2\int_0^{2\pi} \sin(t)\cos(t) dt \\
&= (2\pi - 0) - 2\int_0^{2\pi} \sin(t)\cos(t) dt
\end{aligned}$$

I.e., $d^2(\sin, \cos) = 2\pi - 2\int_0^{2\pi} \sin(t)\cos(t)dt$.

Now we must integrate $\sin(t)\cos(t)$.

$$\begin{aligned} d(\sin,\cos) &= \sqrt{2\pi - 2\left(\frac{1}{2}\right)\int_0^{2\pi} \sin(2t)dt} \\ &= \sqrt{2\pi - 2\left(\frac{1}{2}\right)\left(\frac{1}{2}\right)\cos(2t)\Big|_0^{2\pi}} \\ &= \sqrt{2\pi - \frac{1}{2}(\cos 4\pi - \cos 0)} \\ &= \sqrt{2\pi - \frac{1}{2}(0-0)} \\ &= \sqrt{2\pi} \end{aligned}$$

8.2 Unit Vectors

A unit vector (intuitively) is a vector of length one. Given vector, v there are two vectors parallel to v of unit length, in particular $\frac{v}{\|v\|}$ and $\frac{-v}{\|v\|}$.

Theorem 8.2.1 (*Unit Vector*)

Given vector v, the vectors $\frac{v}{\|v\|}$ and $\frac{-v}{\|v\|}$ have unit length.

Proof.

$$\begin{aligned}
\left\| \frac{v}{\|v\|} \right\| &= \left\| \frac{1}{\|v\|} v \right\| \\
&= \left| \frac{1}{\|v\|} \right| \|v\| \\
&= \frac{\|v\|}{\|v\|} = 1 \\
\left\| \frac{-v}{\|v\|} \right\| &= \left\| (-1) \left(\frac{v}{\|v\|} \right) \right\| \\
&= |-1| \underbrace{\left\| \frac{v}{\|v\|} \right\|}_{=1 \text{ from above}} = 1
\end{aligned}$$

Definition 8.2.2 (*normalize*)

To *normalize* a vector, v, is to compute a unit vector parallel to v in the same direction as v.

The Python method `v.normalize()` computes the unit vector in the direction of v.

```
class Vector:
  ...
  def normalize(self):
    n = self.norm()
    if n == 0:
        return None
    else:
        ...
```

Theorem 8.2.3 (*Scaling a normal vector*)

Let V be a vector space over K. Let $s \in K$, then $\frac{s}{\|v\|}v$ has length $|s|$.

Proof. Apply Theorem 8.2.1

8.3 What Is an Inner Product?

In Section 1.6 we defined the dot product (Definition 1.6.2) for vectors of the same dimension. Now, we will generalize this concept to work with arbitrary vector spaces.

Definition 8.3.1 (*inner product*)

If V is a vector space over field $\mathbb{R}$, and $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ is said to be an *inner product* provided:

1. (symmetric) $u, v \in V \implies \langle u, v \rangle = \langle v, u \rangle$
2. (positive-definite) $u, v \in V \implies \langle u, u \rangle \geq 0$ and $\langle u, u \rangle = 0$ if and only if $u = 0$.
3. (bilinear) $u, v, w \in V$ and $\alpha, \beta \in \mathbb{R} \implies \langle \alpha u + \beta v, w \rangle = \alpha \langle u, w \rangle + \beta \langle v, w \rangle$

8.4 Inner Product in $\mathbb{R}^n$

Recall in Sections 1.5 and 1.6, we defined a norm, distance, and dot product for $\mathbb{R}^n$. In those sections, we started with length, which lead to distance, which lead to dot product. However, in the more theoretical

approach, we typically start with an inner product, from which we induce a norm and thus a distance.

Theorem 8.4.1 (*Dot Product*)

The dot product defined in Definition 1.6.2 is an inner product.

The proof is straightforward.

Proof. Let $u, v, w \in \mathbb{R}^n$, in particular $u = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$, $v = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$, $w = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$. We must show the 3 axioms from Definition 8.3.1 hold.

1. (symmetric)

$$\langle u, v \rangle = \left\langle \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}, \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \right\rangle = \sum_{k=1}^{n} a_k b_k = \sum_{k=1}^{n} b_k a_k$$

$$= \left\langle \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \right\rangle = \langle v, u \rangle$$

2. (positive-definite) We handle positive and definite as two sub-cases.

- (positive)

$$\langle u, u \rangle = \sum_{k=1}^{n} a_k a_k = \sum_{k=1}^{n} a_k^2 \quad \geq 0$$

- (definite) Let $\langle u, u \rangle = 0$. We show $u = 0$ by contradiction. Suppose $u \neq 0$, in particular suppose $a_i \neq 0$ for some i, so $a_i^2 > 0$. Now

$$\begin{aligned} \langle u, u \rangle &= \sum_{k=0}^{n} a_k^2 \\ &= \underbrace{\sum_{k=0}^{i-1} a_k^2}_{\geq 0} + \overbrace{a_i^2}^{>0} + \underbrace{\sum_{k=i+1}^{n} a_k^2}_{\geq 0} \\ &> 0 \end{aligned}$$

This is a contradiction of the assumption that $\langle u, u \rangle = 0$. By contradiction, $u = 0$.

3. (bilinear) Let $\alpha, \beta \in \mathbb{R}$,

$$\begin{aligned}
\langle \alpha u + \beta v, w \rangle &= \left\langle \alpha \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \beta \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \right\rangle \\
&= \left\langle \begin{bmatrix} \alpha a_1 + \beta b_1 \\ \alpha a_2 + \beta b_2 \\ \vdots \\ \alpha a_n + \beta b_n \end{bmatrix}, \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \right\rangle \\
&= \sum_{k=1}^{n} (\alpha a_k + \beta b_k)(c_k) = \sum_{k=1}^{n} (\alpha a_k c_k + \beta b_k c_k) \\
&= \sum_{k=1}^{n} \alpha a_k c_k + \sum_{k=1}^{n} \beta b_k c_k = \alpha \sum_{k=1}^{n} a_k c_k + \beta \sum_{k=1}^{n} b_k c_k \\
&= \alpha \langle u, w \rangle + \beta \langle v, w \rangle
\end{aligned}$$

Example 8.4.2 (*Inner product on* $\mathcal{C}^0[0, 2\pi]$)

As in Example 8.1.2 consider the vector space $\mathcal{C}^0[0, 2\pi]$. We may define $\langle f, g \rangle = \int_0^{2\pi} f(t)g(t)dt$. Let's verify that this definition obeys the axioms in Definition 8.3.1. Let $f, g, h \in \mathcal{C}^0[0, 2\pi]$.

1. (symmetry)

$$\begin{aligned}\langle f, g\rangle &= \int_0^{2\pi} f(t)g(t)dt \\ &= \int_0^{2\pi} g(t)f(t)dt \\ &= \langle g, f\rangle\end{aligned}$$

2. (positive definite)

$$\begin{aligned}\langle f, f\rangle &= \int_0^{2\pi} f(t)f(t)dt \\ &= \int_0^{2\pi} f^2(t)dt \\ &\geq 0 \qquad\qquad \text{because } f^2 \geq 0\end{aligned}$$

Next, suppose $f(t) = 0$

$$\langle f, f\rangle = \int_0^{2\pi} (0)(0)dt = 0$$

Conversely, Suppose $\langle f, f\rangle = 0$.

$$0 = \langle f, f\rangle = \int_0^{2\pi} f^2(t)dt$$

Since $f^2(t) \geq 0$ and continuous, it must be 0.

3. (bilinearity)

$$\begin{aligned}\langle \alpha f + \beta g, h\rangle &= \int_0^{2\pi} (\alpha f(t) + \beta g(t))h(t)dt \\ &= \int_0^{2\pi} \alpha f(t)h(t)dt + \int_0^{2\pi} \beta g(t)h(t)dt \\ &= \alpha \int_0^{2\pi} f(t)h(t)dt + \beta \int_0^{2\pi} g(t)h(t)dt \\ &= \alpha \langle f, h\rangle + \beta \langle g, h\rangle\end{aligned}$$

Example 8.4.3 (*Compute inner product on* $\mathcal{C}^0[0, 2\pi]$)

Again we consider the vector space $\mathcal{C}^0[0, 2\pi]$ from Example 8.1.2. Since $\sin, \cos \in \mathcal{C}^0[0, 2\pi]$, we will compute $\langle \sin, \cos\rangle$.

$$\begin{aligned}\langle \sin, \cos\rangle &= \int_0^{2\pi} \sin(t)\cos(t)dt \\ &= \frac{1}{2}\int_0^{2\pi} \sin(2t)dt \\ &= \left(\frac{1}{2}\right)\left(\frac{1}{2}\right)\cos(2t)\Big|_0^{2\pi} \\ &= \frac{1}{4}(\cos 4\pi - \cos 0) \\ &= \frac{1}{4}(0 - 0) = 0\end{aligned}$$

8.5 Norm Induced by an Inner Product

In order to show that an inner product induces a norm, we typically invoke the Cauchy-Schwarz inequality, which we present and prove here

as a lemma. This lemma is a critical step in the proof of Theorem 8.5.3.

Lemma 8.5.1 (*Cauchy-Schwarz inequality*)

If $\langle \cdot, \cdot \rangle$ is an inner product for vector space V, then

$$|\langle u, v \rangle|^2 \leq |\langle u, u \rangle| \, |\langle v, v \rangle| .$$

This proof is inspired by stackexchange post `https://math.stackexchange.com/q/478404`, thanks to `walcher`, `https://math.stackexchange.com/users/89844/walcher`.

Proof. Let $\beta = \frac{\langle u,v \rangle}{\langle v,v \rangle}$.

$$\begin{aligned}
0 &\leq \langle u - \beta v, u - \beta v \rangle \\
&= \langle u, u - \beta v \rangle - \langle \beta v, u - \beta v \rangle \\
&= \langle u, u \rangle - \langle u, \beta v \rangle - \langle \beta v, u \rangle + \langle \beta v, \beta v \rangle \\
&= \langle u, u \rangle - \langle u, \beta v \rangle - \langle u, \beta v \rangle + \langle \beta v, \beta v \rangle \\
&= \langle u, u \rangle - 2 \langle u, \beta v \rangle + \langle \beta v, \beta v \rangle \\
&= \langle u, u \rangle - 2\beta \langle u, v \rangle + \beta^2 \langle v, v \rangle \\
&= \langle u, u \rangle - 2\frac{\langle u, v \rangle}{\langle v, v \rangle} \langle u, v \rangle + \left(\frac{\langle u, v \rangle}{\langle v, v \rangle} \right)^2 \langle v, v \rangle \\
&= \langle u, u \rangle - \frac{2 \langle u, v \rangle^2}{\langle v, v \rangle} + \frac{\langle u, v \rangle^2}{\langle v, v \rangle} \\
&= \langle u, u \rangle - \frac{\langle u, v \rangle^2}{\langle v, v \rangle}
\end{aligned}$$

$$0 \leq \langle u, u \rangle \langle v, v \rangle - \langle u, v \rangle^2 \implies \langle u, v \rangle^2 \leq \langle u, u \rangle \langle v, v \rangle$$

In the special case that $\langle u, u\rangle = \|u\|^2$, we can rewrite the Cauchy-Schwarz inequality as

$$|\langle u, v\rangle|^2 \leq \|u\|^2 \, \|v\|^2 , \tag{8.1}$$

or simply as

$$|\langle u, v\rangle| \leq \|u\| \, \|v\| . \tag{8.2}$$

Example 8.5.2 (*Cauchy-Schwarz inequality*)

Show the Cauchy-Schwarz inequality holds for vectors $u = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ and $v = \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix}$. First we compute $\langle u, v\rangle$.

$$\begin{aligned} \langle u, v\rangle &= \left\langle \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix} \right\rangle \\ &= (1)(-1) + (2)(2) + (3)(-1) \\ &= -1 + 4 - 3 = -2 \end{aligned}$$

$$|\langle u, v\rangle| = |-2| = 2$$

Now we compute $\|u\| \, \|v\|$.

$$\begin{aligned} \|u\| \, \|v\| &= \left\| \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right\| \left\| \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix} \right\| \\ &= \left(\sqrt{1^2 + 2^2 + 3^2}\right)\left(\sqrt{(-1)^2 + 2^2 + (-1)^2}\right) \\ &= \left(\sqrt{1 + 4 + 9}\right)\left(\sqrt{1 + 4 + 1}\right) = \left(\sqrt{14}\right)\left(\sqrt{6}\right) \\ &= \sqrt{(2)(7)(2)(3)} = 2\sqrt{21} \end{aligned}$$

Since $\sqrt{21} > 1$, we have $|\langle u, v\rangle| = 2 < 2\sqrt{21} = \|u\|\ \|v\|$.

Theorem 8.5.3 (*Norm induced by an inner product*)

If V is a vector space over $\mathbb{R}$ and $\langle \cdot, \cdot \rangle$ be an inner product for V, then $\sqrt{\langle u, u\rangle}$ defines a norm.

Proof. Let V be a vector space equipped with an inner product, and let

$$\|u\|^2 = \langle u, u\rangle \ .$$

We must show 1-4 from Definition 8.1.1.

1. (positive) Let $u \in V$. $\|u\|^2 = \langle u, u\rangle \geq 0$ by Definition 8.3.1, so $\|u\| \geq 0$.
2. (definite)
   - ($\implies$) Let $u \in V$ such that $0 = \|u\|$. By assumption $\|u\| = \sqrt{\langle u, u\rangle}$ and $\sqrt{\langle u, u\rangle} = 0 \implies \langle u, u\rangle = 0 \implies u = 0$.
   - ($\impliedby$) Let $u = 0$, so $\sqrt{\langle u, u\rangle} = \sqrt{\langle 0, 0\rangle} = \sqrt{0} = 0$.
3. (scalability) Let $u \in V$ and $\alpha \in \mathbb{R}$.

$$\begin{aligned} \|\alpha u\| &= \sqrt{\langle \alpha u, \alpha u\rangle} \\ &= \sqrt{\alpha \langle \alpha u, u\rangle} \\ &= \sqrt{\alpha^2 \langle u, u\rangle} \\ &= |\alpha| \sqrt{\langle u, u\rangle} \\ &= |\alpha| \|u\| \end{aligned}$$

4. (triangle inequality) Let $u, v \in V$:

$$\begin{aligned}
\|u+v\|^2 &= \langle u+v, u+v\rangle \\
&= \langle u, u+v\rangle + \langle v, u+v\rangle \\
&= \langle u, u\rangle + \langle u, v\rangle + \langle v, u\rangle + \langle v, v\rangle \\
&= \langle u, u\rangle + \langle u, v\rangle + \langle u, v\rangle + \langle v, v\rangle \\
&= \|u\|^2 + 2\langle u, v\rangle + \|v\|^2 \\
&\leq \|u\|^2 + 2|\langle u, v\rangle| + \|v\|^2 \\
&\leq \|u\|^2 + 2\|u\|\|v\| + \|v\|^2 && \text{by Lemma 8.5.1} \\
&= (\|u\| + \|v\|)^2 \\
\|u+v\| &\leq \|u\| + \|v\|
\end{aligned}$$

Corollary 8.5.4

The dot product induces a norm.

$$\|v\| = \sqrt{v \cdot v}$$

Proof. Simple application of Theorems 8.4.1 and 8.5.3

8.6 Inner Product and Matrix Multiplication

One way to think of a dot product is as a product of matrices. To see this, let $u = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$, $v = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$, then

$$u \cdot v = \sum_{i=1}^{n} a_i b_i \, .$$

Now consider the product of an $n \times 1$ matrix with a $1 \times n$ matrix.

$$u^T v = \underbrace{\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}^T}_{(1\times n)^T} \times \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}}_{n\times 1} = \underbrace{\begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}}_{n\times 1} \times \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}}_{n\times 1} = \underbrace{\begin{bmatrix} \sum_{i=1}^{n} a_i b_i \end{bmatrix}}_{1\times 1} = \begin{bmatrix} u \cdot v \end{bmatrix} \, .$$

Now let's consider the product of two $n \times n$ matrices. Let $A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ where a_i is a row vector and $B = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix}$ where b_i is a column vector.

Now we can express the product in terms of dot products

$$AB = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix}$$

$$= \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \cdots & a_1 \cdot b_n \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \cdots & a_2 \cdot b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \cdots & a_n \cdot b_n \end{bmatrix}$$

$$(AB)_{i,j} = a_i \cdot b_j$$

A comment about the notation. If we view u and v simply as n-dimensional vectors, then $u \cdot v$ denotes the dot product. However if we view u and v as a $n \times 1$ matrices, then $u^T v$ computes the same value but rather than computing a number computes a 1×1 matrix. Therfore, it is common to use the notation $u^T v$ as an alternative notation for the dot product.

Theorem 8.6.1 ($A^T A$ *is Symmetric*)

If A is a square matrix, then $A^T A$ and AA^T are symmetric.

Proof. Suppose A has size $n \times n$. Denote the column vectors of A as $v_1, v_2, \ldots v_n$. So we can denote $A = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$. Using similar notation we can denote $A^T = \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix}$. *I.e.*, row i of A^T is the transpose of column i of A.

If we compute the matrix product, A^TA, we have

$$A^TA = \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix} \times \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$$
$$= \begin{bmatrix} v_1^Tv_1 & v_1^Tv_2 & \cdots & v_1^Tv_n \\ v_2^Tv_1 & v_2^Tv_2 & \cdots & v_2^Tv_n \\ \vdots & \vdots & \ddots & \vdots \\ v_n^Tv_1 & v_n^Tv_2 & \cdots & v_n^Tv_n \end{bmatrix}$$

But since $v_i^Tv_j = \langle v_i, v_j \rangle$ we have

$$A^TA = \begin{bmatrix} \langle v_1, v_1 \rangle & \langle v_1, v_2 \rangle & \cdots & \langle v_1, v_n \rangle \\ \langle v_2, v_1 \rangle & \langle v_2, v_2 \rangle & \cdots & \langle v_2, v_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle v_n, v_1 \rangle & \langle v_n, v_2 \rangle & \cdots & \langle v_n, v_n \rangle \end{bmatrix}$$

Finally, since $\langle v_j, v_i \rangle = \langle v_i, v_j \rangle$ we see that the matrix A^TA is symmetric.

$$A^TA = \begin{bmatrix} \langle v_1, v_1 \rangle & \langle v_1, v_2 \rangle & \cdots & \langle v_1, v_n \rangle \\ \langle v_1, v_2 \rangle & \langle v_2, v_2 \rangle & \cdots & \langle v_2, v_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle v_1, v_n \rangle & \langle v_2, v_n \rangle & \cdots & \langle v_n, v_n \rangle \end{bmatrix} \tag{8.3}$$

Now what about AA^T? Let $B = A^T$ and we have $B^T = A$. So $AA^T = B^TB$, which we know is symmetric from Equation 8.3.

Why is Equation (8.3) interesting? Because, as will be seen in Section 9.8, if we could somehow select $v_1 \ldots v_n$ such that $\langle v_i, v_j \rangle = 0$

when $i \neq j$, then we would have 0 off the diagonal. Furthermore, recall that and $\langle v_i, v_i \rangle = \|v_i\|^2$, so we would have $\|v_i\|^2$ on the diagonal.

$$A^T A = \begin{bmatrix} \|v_1\|^2 & 0 & \cdots & 0 \\ 0 & \|v_2\|^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \|v_n\|^2 \end{bmatrix}$$

Additionally, if we could choose $v_1 \dots v_n$ to be unit vectors, then we would have $\|v_i\|^2 = 1$, and thus we would have

$$\begin{aligned} A^T A &= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \\ &= I \\ A^T &= A^{-1} \end{aligned} \tag{8.4}$$

CAREFUL!, Equation (8.4) is not true in general. Certainly it is not true that $A^T = A^{-1}$ for all square matrices; rather if we can assure that $\langle v_i, v_j \rangle = 0$ for $i \neq j$, and also assure that $\|v_i\| = 1$, then we can compute the inverse trivially.

8.7 Angle Induced by Inner Product

What happens when we apply the Cauchy-Schwarz (Lemma 8.5.1) to two distinct vectors, each with positive length?

$$|\langle u, v \rangle| \leq \|u\| \|v\| \tag{8.5}$$

$$\frac{|\langle u, v \rangle|}{\|u\| \|v\|} \leq 1 \tag{8.6}$$

Since the norm of a vector is non-negative, inequality (8.6) leads to

$$-1 \leq \frac{\langle u, v\rangle}{\|u\|\|v\|} \leq 1\,. \tag{8.7}$$

Definition 8.7.1 (*Angle between vectors*)

The quantity $\frac{\langle u,v\rangle}{\|u\|\|v\|}$ is bounded between 1 and -1, so there is some angle $\theta_{u,v}$ which depends on u and v for which

$$\cos\theta_{u,v} = \frac{\langle u, v\rangle}{\|u\|\|v\|}\,. \tag{8.8}$$

This angle is defined as the *angle between* u and v.

In $\mathbb{R}^n$ this corresponds to the physical angle swept out when rotating one vector into another. However, more generally, Equation (8.8), gives us an abstract way to measure angles *any* vector space as long as the vector space is equipped with a norm an inner product which are related by Theorem 8.5.3.

In Section 8.9, we will need to project one vector onto another. This means we'll want to compute a vector of some desired length, but in the direction of some other given vector. Before talking more about projections in, we first stop to consider how to construct a vector in the same direction as a given vector, but with a desired length.

Theorem 8.7.2 (*Vector in same direction*)

Let V be an inner product space over K, and $v \in V$, $s \in K$, with $s > 0$ (or $s < 0$). Then $\frac{s}{\|v\|}v$ is a vector of length $|s|$ in the same direction (or opposite direction) as v.

Proof. $\frac{s}{\|v\|}v$ has length $|s|$ was shown in Theorem 8.2.3. To show

v and $\frac{s}{\|v\|}v$ are the same direction we compute the angle between the vectors.

$$\cos\theta = \frac{\left\langle v, \frac{s}{\|v\|}v \right\rangle}{\|v\| \left\| \frac{s}{\|v\|}v \right\|} = \frac{\left\langle v, \frac{s}{\|v\|}v \right\rangle}{\left| \frac{s}{\|v\|} \right| \|v\| \ \|v\|}$$
$$= \frac{\frac{s}{\|v\|}\langle v, v\rangle}{\left| \frac{s}{\|v\|} \right| \langle v, v\rangle} = \frac{s}{|s|} \in \{1, -1\}$$

Thus $\cos\theta = 1$ if $s > 0$ and $\cos\theta = -1$ if $s < 0$; meaning the angle is either 0 (same direction) or π (opposite direction).

Example 8.7.3 (*Angle between two vectors in* $\mathbb{R}^2$)

Compute the angle between the vectors u and v where

$u = \begin{bmatrix} -\sqrt{3} \\ -1 \end{bmatrix}$ and $v = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$.

$$\cos\theta = \frac{\langle u, v\rangle}{\|u\|\|v\|} = \frac{\left\langle \begin{bmatrix} -\sqrt{3} \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\rangle}{\left\| \begin{bmatrix} -\sqrt{3} \\ -1 \end{bmatrix} \right\| \left\| \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\|}$$

$$= \frac{(-\sqrt{3})(0) + (-1)(-1)}{\left(\sqrt{\left(-\sqrt{3}\right)^2 + (-1)^2}\right)\left(\sqrt{0^2 + (-1)^2}\right)}$$

$$= \frac{1}{\sqrt{3+1}\sqrt{0+1}} = \frac{1}{\sqrt{4}}$$

$$\cos\theta = \frac{1}{2}$$

$$\theta = \cos^{-1}\left(\frac{1}{2}\right) = \frac{\pi}{3} = 60^\circ$$

In Example 8.7.3 we computed the angle between two 2-dimension vectors. This is the angle you would measure with a protractor. In Example 8.7.4 we compute the angle between two vectors in a vector space of functions. The physical meaning of angle in this case is less clear.

Example 8.7.4 (*Angle between* sin *and* cos)

Let's compute the angle between u and v where $u = \sin$ and $v = \cos$ considering them vectors in the vector space $\mathcal{C}^0[0, 2\pi]$ from Example 8.1.2. In Example 8.4.3 we have already computed

$$\langle \sin, \cos \rangle = 0$$

To compute $\frac{\langle \sin,\cos\rangle}{\|\sin\|\ \|\cos\|}$ we will need to use the fact that $\|\cos\| \neq 0$ and $\|\sin\| \neq 0$.

$$\begin{aligned}
\|\sin\|^2 &= \int_0^{2\pi} \sin^2(t)dt = \int_0^{2\pi} \frac{1}{2}(1-\cos 2t)dt \\
&= \int_0^{2\pi} \frac{1}{2}dt - \frac{1}{2}\int_0^{2\pi} \cos 2t dt \\
&= \frac{1}{2}t\Big|_0^{2\pi} - \frac{1}{4}\cos 2t\Big|_0^{2\pi} \\
&= \frac{1}{2}(2\pi - 0) - \frac{1}{4}(0-0) \\
&= \pi \\
\|\sin\| &= \sqrt{\pi}
\end{aligned}$$

$$\begin{aligned}
\|\cos\|^2 &= \int_0^{2\pi} \cos^2(t)dt = \int_0^{2\pi} \big(1-\sin^2(t)\big)dt \\
&= \int_0^{2\pi} 1dt - \underbrace{\int_0^{2\pi} \sin^2(t)dt}_{=\ \pi \text{ from above}} \\
&= t\Big|_0^{2\pi} - \pi = 2\pi - 0 - \pi = \pi \\
\|\cos\| &= \sqrt{\pi}
\end{aligned}$$

$$\cos\theta = \frac{\langle \sin,\cos\rangle}{\|\sin\|\ \|\cos\|} = \frac{0}{\sqrt{\pi}\sqrt{\pi}} = 0$$

$$\theta = \cos^{-1} 0 = \frac{\pi}{2} = 90^\circ$$

Example 8.7.4 shows the *angle* between the sin and cos functions is $\frac{\pi}{2} = 90^\circ$. This is not a coincidence; it is related to the fact that

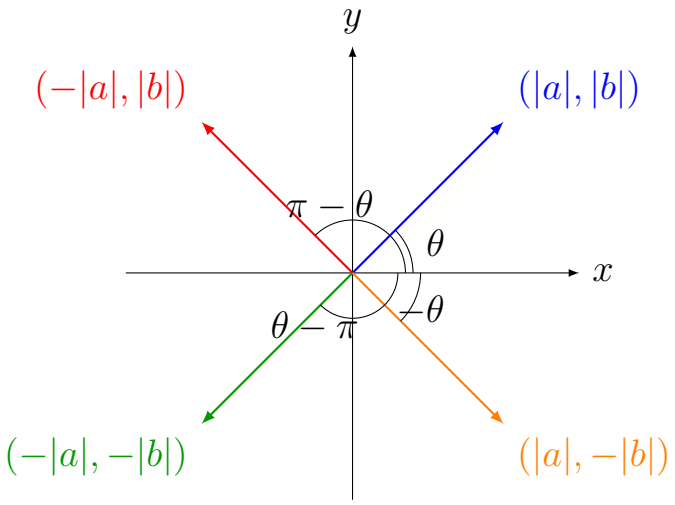


Figure 8.1: Illustration of `atan2(...)`

$\sin(t) = \cos\left(t - \frac{\pi}{2}\right)$.

8.8 Correct Sign of the Angle

There is a limitation of Definition 8.7.1. Notice that there are two angles between vectors u and v; namely The angle from u to v and also the angle from v to u. If θ is the angle from u to v, then $-\theta$ (equivalently $2\pi - \theta$) is the angle from v to u. However, $\cos\theta = \cos -\theta = \cos(2\pi - \theta)$. Moreover, $\langle u, v\rangle = \langle v, u\rangle$, as we know the inner product is symmetric (Definition 8.3.1. Thus we cannot determine using Equation 8.8 whether u is clockwise or counterclockwise from v.

To determine the orientation of the angle between two vectors in 2-dimensions, $\mathbb{R}^2$, we can use the determinant of the matrix formed by the column vectors. Supposing $u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$, and $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$.

$$\det\left(\begin{bmatrix} u & v \end{bmatrix}\right) = \det\left(\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix}\right) = u_1 v_2 - u_2 v_1$$

If this quantity is positive, then v is counterclockwise from u. If this quantity is negative, then v is clockwise from u. If the quantity is 0, then u and v are colinear.

Example 8.8.1 (*Sign of Angle*)

Let's consider again the angle between the vectors from Example 8.7.3:
$u = \begin{bmatrix} -\sqrt{3} \\ -1 \end{bmatrix}$ and $v = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$.

$$\begin{aligned} \det([u\ v]) &= \det\left(\begin{bmatrix} -\sqrt{3} & 0 \\ -1 & -1 \end{bmatrix}\right) \\ &= (-\sqrt{3})(-1) - (-1)(0) \\ &= \sqrt{3} > 0 \end{aligned}$$

The fact that this determinant is positive tells us that the angle 60° computed in Example 8.7.3 has the correct sign.

Note that this technique in Example 8.8.1 only works because we can compute $\det([u\ v])$ as $u, v \in \mathbb{R}^2$. If $v, u \in \mathbb{R}^3$ we would not be able to compute $\det\left(\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \\ u_3 & v_3 \end{bmatrix}\right)$, as the determinant is not defined for a 3×2 matrix.

In many programming languages, there is a math function named `atan2` which computes the signed inverse tangent of the quotient of its two arguments. In Python `atan2(a,b)` returns the angle in a right triangle whose opposite and adjacent sides are a and b respectively, and whose hypotenuse is $\sqrt{a^2 + b^2}$. Recall $\tan\theta = \frac{a}{b}$. However, this quotient loses information because $\frac{a}{b} = \frac{-a}{-b}$. The `atan2` function takes the signs of its argument into account.

Figure 8.1 shows the output of the `atan2` function. The correct an-

gle (with the correct sign) can be computed by $\tan^{-1}\left(\frac{\det([u,v])}{\langle u,v\rangle}\right)$, where `atan2(a,b)` is used to compute $\tan^{-1}\left(\frac{a}{b}\right)$.

Example 8.8.2 (*Compute angle using* $\tan^{-1}$)

Let's take again the vectors from Example 8.8.1: $u = \begin{bmatrix} -\sqrt{3} \\ -1 \end{bmatrix}$ and $v = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$.

$$\begin{aligned} \tan\theta &= \frac{\det([u,v])}{\langle u,v\rangle} \\ &= \frac{(-\sqrt{3})(-1)-(-1)(0)}{(-\sqrt{3})(0)+(-1)(-1)} \\ &= \sqrt{3} \\ \theta &= \frac{\pi}{3} = 60^\circ \end{aligned}$$

8.9 Projections

In Section 9.9 we will present a procedure for generating an orthonormal basis given a set of sufficiently many linearly independent vectors. However, we first need a projection. In order to talk about projection, we must talk about direction, and angle; thus we must be working in a vector space with an inner product.

8.9.1 Intuition of Projections

Figure 8.2 shows an intuition of what we mean by projection. In the figure we construct the shadow of vector v orthogonally onto vector u. Notice that this *projection vector* is parallel to u, but its length is not a function of u. A longer or shorter vector u would result in the same

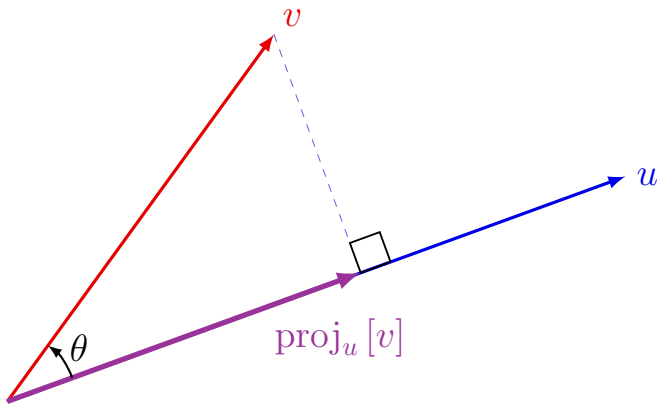


Figure 8.2: Illustration of $\text{proj}_u[v]$

projection of v. We can compute the length of this projection using simple trigonometry

$$\cos\theta = \frac{\|\text{proj}_u[v]\|}{\|v\|}$$
$$\|\text{proj}_u[v]\| = \|v\|\cos\theta$$

Now, we wish to construct a vector in the direction of u but with length $\|v\|\cos\theta$. So we scale the unit vector parallel to u by $\|v\|\cos\theta$, as shown in Figure 8.3.

Thus with the help of Equation (8.8) (page 218), we derive Equa-

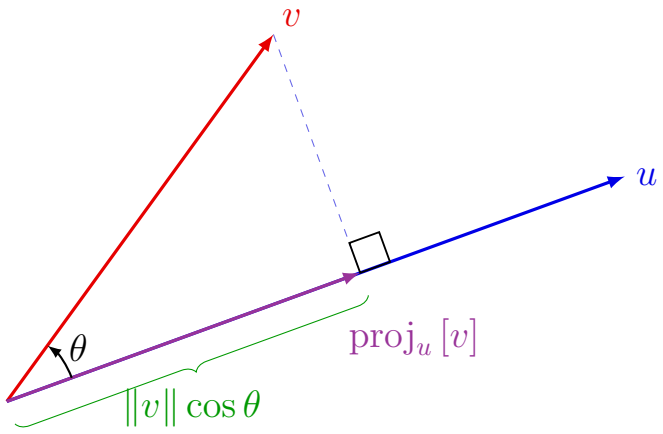


Figure 8.3: Illustration of $\|\text{proj}_u[v]\|$

tion 8.9 which represents the projection vector, $\text{proj}_u[v]$.

$$\begin{aligned}
\text{proj}_u[v] &= (\|v\|\cos\theta)\frac{u}{\|u\|} \\
&= \frac{\|v\|}{\|u\|}u\cos\theta \\
&= \frac{\|v\|}{\|u\|}u\frac{\langle u,v\rangle}{\|u\|\|v\|} && \text{by (8.8)} \\
&= \frac{\langle u,v\rangle}{\|u\|^2}u && (8.9)
\end{aligned}$$

8.9.2 Definition of Projection

Equation (8.9) motivates Definition 8.9.1.

Definition 8.9.1 (*projection*)

Given two vectors u, v, the *projection* of v onto u is the vector, p, having length $\frac{|\langle u,v\rangle|}{\|u\|}$ but parallel to u (if $\langle u,v\rangle > 0$) or parallel

to $-u$ (if $\langle u, v\rangle < 0$). By parallel we mean the angle between the vectors is 0.

We denote the projection of v onto u as $\text{proj}_u[v]$.

Now, we prove, formally, what we motivated geometrically in 2 dimensions in Equation (8.9).

Theorem 8.9.2 (*Projections*)

If u and v are elements in a vector space whose norm is induced from an inner product as in Theorem 8.5.3, then

$$\text{proj}_u[v] = \frac{\langle u, v\rangle}{\langle u, u\rangle} u = \frac{\langle u, v\rangle}{\|u\|^2} u$$

Proof. Let u, v be vectors in vector space V having an inner product induced norm. We must prove the u and $\text{proj}_u[v]$ are parallel, and that $\|\text{proj}_u[v]\| = |\langle u, v\rangle|$.

- Show $\|\text{proj}_u[v]\| = \frac{|\langle u,v\rangle|}{\|u\|}$.

$$\begin{aligned}\|\text{proj}_u[v]\|^2 &= \langle \text{proj}_u[v], \text{proj}_u[v]\rangle = \left\langle \frac{\langle u, v\rangle}{\|u\|^2} u, \frac{\langle u, v\rangle}{\|u\|^2} u \right\rangle \\ &= \left(\frac{\langle u, v\rangle}{\|u\|^2}\right)^2 \langle u, u\rangle = \left(\frac{\langle u, v\rangle}{\|u\|^2}\right)^2 \|u\|^2 = \frac{\langle u, v\rangle^2}{\|u\|^2}\end{aligned}$$

  Thus we have $\|\text{proj}_u[v]\| = \frac{|\langle u,v\rangle|}{\|u\|}$.

- Show $\text{proj}_u[v]$ is parallel/anti-parallel with u. This means

the angle between the vectors is 0 or π.

$$\begin{aligned}
\cos\theta &= \frac{\langle u, \operatorname{proj}_u[v]\rangle}{\|u\|\|\operatorname{proj}_u[v]\|} \\
&= \frac{\left\langle u, \frac{\langle u,v\rangle}{\|u\|^2}u\right\rangle}{\|u\|\left\|\frac{\langle u,v\rangle}{\|u\|^2}u\right\|} = \frac{\frac{\langle u,v\rangle}{\|u\|}\langle u,u\rangle}{\left|\frac{\langle u,v\rangle}{\|u\|}\right|\|u\|\|u\|} \\
&= \frac{\frac{\langle u,v\rangle}{\|u\|^2}\|u\|^2}{\left|\frac{\langle u,v\rangle}{\|u\|^2}\right|\|u\|^2} = \frac{\langle u,v\rangle}{|\langle u,v\rangle|} \\
|\cos\theta| &= 1
\end{aligned}$$

Thus either $\cos\theta = 1$ and $\theta = 0$, or $\cos\theta = -1$ and $\theta = \pi$.

Example 8.9.3 (*Computing the projection*)

Compute $\operatorname{proj}_u[v]$, where $v = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$ and $u = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}$.

Now we compute $\operatorname{proj}_u[v]$.

$$
\begin{aligned}
\operatorname{proj}_u[v] &= \frac{\langle u, v\rangle}{\|u\|^2} u \\
&= \frac{\left\langle \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 6 \\ 8 \end{bmatrix} \right\rangle}{\left\| \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \right\|^2} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \\
&= \frac{\frac{6}{\sqrt{2}} - \frac{8}{\sqrt{2}}}{\left(\frac{1}{\sqrt{2}}\right)^2 + \left(\frac{-1}{\sqrt{2}}\right)^2} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \\
&= \frac{\frac{-2}{\sqrt{2}}}{\frac{1}{2} + \frac{1}{2}} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} = \frac{-\sqrt{2}}{1} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \\
&= -\sqrt{2} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}
\end{aligned}
$$

The formula for the projection is simpler if we project onto a unit vector.

Corollary 8.9.4

If e is a unit vector, then

$$\operatorname{proj}_e[v] = \langle e, v\rangle\, e\,.$$

Proof.

$$\begin{aligned}\operatorname{proj}_e[v] &= \frac{\langle e, v\rangle}{\|e\|^2} e \\ &= \frac{\langle e, v\rangle}{1} e \\ &= \langle e, v\rangle\, e\end{aligned}$$

Example 8.9.5 (*Projecting onto a unit vector*)

Compute $\operatorname{proj}_e[v]$, where $v = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$ and $e = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}$. Recall that $\|e\| = 1$.

$$\begin{aligned}\|e\| &= \left\| \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \right\| = \sqrt{\left(\frac{1}{\sqrt{2}}\right)^2 + \left(\frac{-1}{\sqrt{2}}\right)^2} \\ &= \sqrt{\frac{1}{2} + \frac{1}{2}} = 1\end{aligned}$$

Now we compute $\operatorname{proj}_e[v]$.

$$\begin{aligned}
\text{proj}_e\,[v] &= \langle e, v\rangle\, e \\
&= \left\langle \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 6 \\ 8 \end{bmatrix} \right\rangle \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \\
&= \left(\frac{6}{\sqrt{2}} - \frac{8}{\sqrt{2}} \right) \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \\
&= \frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}} (6-8) \begin{bmatrix} 1 \\ -1 \end{bmatrix} \\
&= \frac{1}{2}(2) \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}
\end{aligned}$$

We see in the proof of Theorem 8.9.2, that if the angle between u and v is acute (between $-90°$ and $90°$), then the projection of v onto u is in the same direction as u. However, if the angle between u and v is obtuse, then the projection points in exactly the opposite direction as u.

8.9.3 Construction of Orthogonal Vectors

Given vectors u and v which are not colinear, we can construct a vector orthogonal to u simply by subtracting $\text{proj}_u\,[v]$.

Theorem 8.9.6

Given u and v (not colinear)

$$\langle u, v - \text{proj}_u\,[v]\rangle = 0$$

Proof.

$$\begin{aligned}
\langle u, v - \operatorname{proj}_u[v] \rangle &= \langle u, v \rangle - \langle u, \operatorname{proj}_u[v] \rangle \\
&= \langle u, v \rangle - \left\langle u, \frac{\langle u, v \rangle}{\|u\|^2} u \right\rangle \\
&= \langle u, v \rangle - \frac{\langle u, v \rangle}{\|u\|^2} \langle u, u \rangle \\
&= \langle u, v \rangle - \frac{\langle u, v \rangle}{\|u\|^2} \|u\|^2 \\
&= \langle u, v \rangle - \langle u, v \rangle = 0
\end{aligned}$$

Theorem 8.9.7

The projection of v onto u is the same as the projection of v onto any vector colinear to u. *I.e.*, $s \neq 0 \implies \operatorname{proj}_u[v] = \operatorname{proj}_{su}[v]$.

Proof.

$$\begin{aligned}
\operatorname{proj}_{su}[v] &= \frac{\langle su, v \rangle}{\|su\|^2} su \\
&= \frac{s^2}{|s|^2} \frac{\langle u, v \rangle}{\|u\|^2} u \\
&= \frac{\langle u, v \rangle}{\|u\|^2} u \\
&= \operatorname{proj}_u[v]
\end{aligned}$$

8.9.4 Idempotency of Projections

One might wonder why the projection (Definition 8.9.1) is defined such that $\|\text{proj}_u[v]\| = |\langle u, v\rangle|$. The reason is so that the projection operator is idempotent. *I.e.*, applying the projection a second time (and thus arbitrarily many times) achieves the same result as simply applying it once. If P is a projection operator, then $P^2 = P$.

Theorem 8.9.8 (*Projections are Idempotent*)

$$\text{proj}_u[\text{proj}_u[v]] = \text{proj}_u[v].$$

Proof. Let V be an inner product space whose norm is induced by the inner product, and let $u, v \in V$.

$$\begin{aligned}
\text{proj}_u[\text{proj}_u[v]] &= \frac{\left\langle u, \frac{\langle u,v\rangle}{\|u\|^2}u\right\rangle}{\|u\|^2}u = \frac{\frac{\langle u,v\rangle}{\|u\|^2}\langle u,u\rangle}{\|u\|^2}u \\
&= \frac{\frac{\langle u,v\rangle}{\|u\|^2}\|u\|^2}{\|u\|^2}u = \frac{\langle u,v\rangle}{\|u\|^2}u \\
&= \text{proj}_u[v]
\end{aligned}$$

8.9.5 Linearity of the Projection Operator

Theorem 8.9.9 (*Linearity of projection*)

The projection operator is linear.

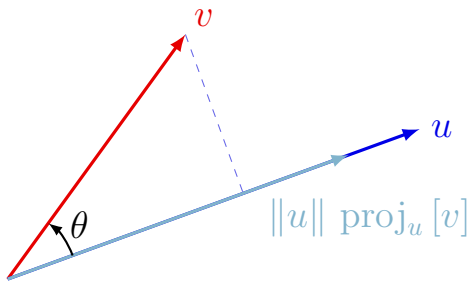


Figure 8.4: Illustration of $\|u\|\ \mathrm{proj}_u\,[v]$

Proof. We will show

$$\mathrm{proj}_u\,[\alpha v + w] = \alpha\ \mathrm{proj}_u\,[v] + \mathrm{proj}_u\,[w]\,.$$

$$\begin{aligned}
\mathrm{proj}_u\,[\alpha v + w] &= \frac{\langle u, \alpha v + w\rangle}{\|u\|^2} u && \text{by Theorem 8.9.2} \\
&= \frac{\alpha\,\langle u, v\rangle + \langle u, w\rangle}{\|u\|^2} u && \text{by linearity of the inner product} \\
&= \alpha \mathrm{proj}_u\,[v] + \mathrm{proj}_u\,[w] && \text{by Theorem 8.9.2}
\end{aligned}$$

8.9.6 Alternate Interpretation of Inner Product

Figure 8.4 shows a different way to think about inner product. If we consider the projection vector $\mathrm{proj}_u\,[v]$ in Figure 8.2, its direction is determined by u, but its length is independent of u. If we construct a new vector which is $\mathrm{proj}_u\,[v]$ scaled by the length of u, we see the vector $\|u\|\ \mathrm{proj}_u\,[v]$ as shown in Figure 8.4. What is the length of this vector?

$$\left\| \overbrace{\|u\| \operatorname{proj}_u [v]}^{\text{scaled projection}} \right\| = \|u\| \, \|\operatorname{proj}_u [v]\| \tag{8.10}$$

$$= \|u\| \left(\frac{|\langle u, v \rangle|}{\|u\|} \right) \qquad \text{by Definition 8.9.1}$$

$$= |\langle u, v \rangle| \tag{8.11}$$

From Equation (8.11), we can think of the inner product $\langle u, v \rangle$ as the length of the projection of v onto u scaled by the length of u.

8.10 Programming Exercises

8.10.1 Classwork

1. Test Cauchy-Schwarz inequality

8.10.2 Homework

- Partially implement `VectorSpace.py`
  - `is_norm`
  - `is_inner_product`
- Implement `Vector.py`
  - `angle`
- Test with
  - `Vector_norm_test.py`

- Vector_inner_product_test.py
- Vector_angle_test.py

Chapter 9

Basis

Chapter Objectives

- Define basis and dimension of a vector space; determine whether a given set of vectors is a basis.
- Express a vector as a linear combination of basis vectors and find its coordinates in a given basis.
- Convert between coordinate representations under a change of basis.
- Apply the Gram-Schmidt process to transform any basis into an orthonormal basis.

In this chapter we introduce the concept of a *basis*, Section 9.5. However, in order to talk about the basis, we first review some of the concepts introduced in earlier chapters such as linear independence and span. The size of the basis (cardinality) is called the dimension of the vector space in question.

Once *basis* has been defined we be able to talk about *coordinate systems*. Different bases can be used to characterize the same space, but with different coordinates for the vectors therein. We discuss in

Section 9.5.2 how to convert coordinates between different bases.

Next, in Section 9.6, we discuss orthonormality including orthonormal vectors and in particular, orthonormal bases, extracting coordinates from vectors which are linear combinations of orthonormal bases.

Finally, we discuss, in Section 9.9, the Gram-Schmidt Process which is an algorithm for constructing an orthonormal basis from any given basis.

9.1 Linear Combinations (Revisited)

Definition 9.1.1 (*column matrix*)

If $U = [u_1, u_2, \ldots, u_n]$ is a tuple of vectors, each of dimension m, then the *column matrix* of U is the $m \times n$ matrix formed by adjoining the vectors as columns.

We may also say: "The column matrix of $u_1, u_2, \ldots, u_n$".

Example 9.1.2 (*Column matrix*)

Let $u_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, $u_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$, and $u_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

The column matrix of u_1, u_2, u_3 is $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$.

The column matrix may also be denoted $\begin{bmatrix} u_1 & u_2 & \ldots & u_n \end{bmatrix}$.

We discussed linear combinations in Section 5.4; we recall the definition here.

Definition 9.1.3 (*linear combination*)

Given a set of vectors, $U = \{u_1, u_2, \ldots, u_n\}$, each of dimension n, and scalars $\alpha_1, \alpha_2, \ldots, \alpha_n$. The sum,

$$\sum_{k=1}^{n} \alpha_k u_k = \alpha_1 u_1 + \alpha_2 u_2 + \cdots + \alpha_n u_n \,,$$

is said to be a *linear combination* of U.

We may also denote the same linear combination as Ax where A is the column matrix of $u_1, u_2, \ldots, u_n$, and $x = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$.

$$\sum_{k=1}^{n} \alpha_k u_k = A \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

9.2 Span (Revisited)

We already talked about the *span* in Section 5.4 Definition 5.4.2. Here, we restate the definition in terms of vector spaces.

The vector spaces $\mathbb{R}^n$ have a special property that every element in the space can be expressed as a linear combination of well-chosen set of n-vectors. For example, every element in R^2 can be written as $\alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, for some real numbers, α and β. However, the set $\left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$, is not unique in this regard; the same vector space can

also be generated as all the linear combinations of $\left\{\begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}\right\}$, or of $\left\{\begin{bmatrix}\sqrt{2}\\0\end{bmatrix}, \begin{bmatrix}0\\\pi\end{bmatrix}\right\}$.

Definition 9.2.1 (*span*)

If V is a vector space, and $B \subseteq V$ is a set of vectors, then we define $\text{span}(B)$ to be the set of all linear combinations of finite subsets of B.

Let's look at an example of a span of an infinite set.

Example 9.2.2 (*Span of a set of sinusoid functions*)

Suppose $B = \{t \mapsto \sin(n\pi t) \mid n \in \mathbb{N}\}$. What is the $\text{span}(B)$? We must first understand which elements are in B. B is an infinite set of all functions such as:

$$\sin(\pi t), \sin(2\pi t), \sin(3\pi t), \ldots .$$

Every element of $\text{span}(B)$ is a linear combination of a finite subset of B, and the set $\text{span}(B)$ is exactly the set of such linear combinations.

For example, the following functions are in $\text{span}(B)$:

$$\begin{aligned} f_1(t) &= 0\sin(\pi t) \\ f_2(t) &= 4\sin(\pi t) - 3\sin(2\pi t) \\ f_3(t) &= 4.3\sin(2\pi t) - \frac{12}{91}\sin(5\pi t) + \sqrt{3}\sin(732\pi t) \end{aligned}$$

It is important to note that we can talk about the span of a finite set or the span of an infinite set. In Example 9.2.2, the set B is infinite.

However, a linear combination is a *finite sum*. In particular

$$f(t) = \sum_{k=1}^{\infty} \frac{1}{k!} \sin(k\pi t)$$

is not a linear combination. Why? Because it is not really a sum. Rather, it is a limit:

$$f(t) = \lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k!} \sin(k\pi t)\,.$$

For every choice of $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, the sum $\sum_{k=1}^{n} \alpha_k \sin(k\pi t)$ is a well-defined function. However, for certain choices of $(\alpha_1, \alpha_2, \ldots,)$, the infinite sum

$$f(t) = \sum_{k=1}^{\infty} \alpha_k \sin(k\pi t) = \lim_{n\to\infty} \sum_{k=1}^{n} \alpha_k \sin(k\pi t)\,.$$

may not even converge.

Example 9.2.3 (*Divergent infinite sum*)

As a trivial case, let $\alpha_k = e^k$, certainly the sum diverges.

$$f(t) = \sum_{k=1}^{\infty} e^k \sin(k\pi t) = \lim_{n\to\infty} \sum_{k=1}^{n} e^k \sin(k\pi t)\,.$$

Theorem 9.2.4 (*Span of finite set*)

If B is a finite, non-empty set, then span(B) is simply the set of linear combinations of B.

Proof. Every linear combination of a subset of B is also a linear

combination of B. Let $U \subset B$. Since $|B| < \infty$ we have $|U| \leq |B|$.

1. Suppose $|U| = |B|$. Then $U = B$. So every linear combination of U is a linear combination of B.
2. Suppose $|U| < |B|$. Denote $U = \{u_1, u_2, \ldots, u_n\}$. Let $W = B \setminus U = \{w_1, w_2, \ldots, w_m\}$. Let $\sum_{k=1}^{n} \alpha_k u_k$ be a linear combination of U. Then $\sum_{k=1}^{n} \alpha_k u_k + \sum_{k=1}^{m} (0)(w_k)$ is a linear combination of B.

9.3 Dimension

Definition 9.3.1 (*dimension*)

If a vector space V is spanned by (is equal to the span of) some finite set B, then V is said to have *finite dimension*.

If the vector space has finite dimension, then the cardinality of a smallest set B which spans V is called the *dimension* of V.

If the vector space does not have finite dimension, then it is said to be *infinite dimensional*.

By *smallest set* we mean the following: B is a smallest set which spans V, then if B' also spans V, then B and B' have the same cardinality or B' has larger cardinality of B. *I.e.*, $|B| \leq |B'|$.

Theorem 9.3.2

Let V be a vector space. Let B and C be finite sets which span V such that $|B| < |C|$. Then some proper subset of C also spans V.

Definition 9.3.3 (*Subspace*)

If V and U are vector spaces (not necessarily distinct) then we refer to U as a *subspace* of V provided $U \subseteq V$.

Theorem 9.3.4 (*The span is a vector space*)

If V is a vector space, and B is a subset of V, then $\text{span}(B)$ is a subspace of V.

9.4 Linear Independence (Revisited)

Recall the definition of *linearly independent*, Definition 5.4.3 from page 148. We repeat the definition here.

Definition 9.4.1 (*linearly independent*)

A set, U, of vectors is said to be *linearly dependent* if some $u \in U$, can be written as a linear combination of some (necessarily finite) subset of $U \setminus \{u\}$; *i.e.*, if there exists $\alpha_1, \alpha_2, \ldots, \alpha_n$, and $\{u_1, u_2, \ldots, u_n\} \subset U \setminus \{u\}$ such that

$$u = \sum_{k=1}^{n} \alpha_k u_k \, .$$

If U is not linearly dependent, it is called *linearly independent*.

In much of the literature, Theorem 9.4.2 is taken as the definition of linear independence. We prove it here as an equivalent statement.

Theorem 9.4.2 (*Alternate definition of Linearly Independent*)

Let V be a vector space, and let $U = \{u_1, u_2, \ldots, u_n\} \subset V$ be a set of vectors.

U is linearly independent if and only if

$$\sum_{i=1}^{n} \alpha_i u_i = 0 \implies \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{9.1}$$

Proof. Let V and U be as above.

First, we prove ($\impliedby$): Suppose Equation (9.1).

We wish to show U is linearly independent. For purpose of contradiction, suppose U is linearly dependent. Let $u_k \in U$, and choose $(\alpha_i)_{i \neq k}$, for which $u_k = \sum_{i \neq k} \alpha_i u_i$. Let $\alpha_k = -1$. Now,

$$\begin{aligned} \sum_{i=1}^{n} \alpha_i u_i &= \alpha_k u_k + \sum_{i \neq k} \alpha_i u_i \\ &= (-1) u_k + \sum_{i \neq k} \alpha_i u_i \\ &= (-1) u_k + u_k = 0, \end{aligned}$$

which contradicts Equation (9.1). Thus, U is linearly independent.

Now, we prove ($\implies$): Suppose U is linearly independent, and let $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$ such that $\sum_{i=1}^{n} \alpha_i u_i = 0$. We want to show $\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$. For purpose of contradiction, suppose $\alpha_k \neq 0$ for

some k.

$$\begin{aligned}
\sum_{i=1}^{n} \alpha_i u_i &= 0 \\
\alpha_k u_k + \sum_{i \neq k} \alpha_i u_i &= 0 \\
\sum_{i \neq k} \alpha_i u_i &= -\alpha_k u_k \\
\sum_{i \neq k} \left(\frac{\alpha_i}{-\alpha_k} \right) u_i &= u_k,
\end{aligned}$$

which means (by Definition 9.4.1) that U is linearly dependent—contradicting our assumption that U is linearly independent.

Theorem 9.4.3 (*Invertible implies Linearly Independent*)

Suppose V is a vector space of n-vectors of the form $\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ and $U = \{u_1, u_2, \ldots, u_n\} \subset V$ is a set of vectors. Let A be the $n \times n$ column matrix of $u_1 \ u_2 \ \ldots \ u_n$.
If A is invertible, then U is linearly independent.

Proof. Let V, U and A be defined as above. Let $b \in V$ and let $x = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} \in V$ such that $Ax = b$.

Note that $\sum_{i=1}^{n} \alpha_i u_i = Ax$.

Let A be invertible, show columns are linearly independent.

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

In particular, $b = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \implies x = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$. *I.e.*., $Ax = 0 \implies x = 0$, thus U is linearly independent.

Theorem 9.4.4 (*Converse of Theorem 9.4.3*)

Suppose V is a vector space of n-vectors of the form $\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ and $U = \{u_1, u_2, \ldots, u_n\} \subset V$ is a set of vectors. Let A be the $n \times n$ matrix formed by the columns $[u_1\ u_2\ \ldots\ u_n]$.
If U is linearly independent, then A is invertible.

We do not prove Theorem 9.4.4 as the proof requires some results we have not presented in this course.

9.5 Basis

There are many possible equivalent definitions of *basis* in the literature. Here is one definition in particular. We use *linearly independence* to define *basis*.

Definition 9.5.1 (*basis*)

If V is a vector space, and $B \subset V$ is linearly independent, and $V = \text{span}(B)$, then B is called a *basis* of V.

Definition 9.5.1 can be thought of as saying: a basis is a minimal set, B, which generates the vector space in question via linear combinations. It is *minimal* in the sense that every subset of B fails to be a basis.

Example 9.5.2 (*Basis for vector space of polynomials*)

Theorem 7.3.2 claimed that the set of polynomials with coefficients in field $\mathbb{R}$ is a vector space. The set $B = \{x^n \mid n \in \mathbb{N}\}$, is a basis for $\mathbb{R}[x]$.

The fact that B is linearly independent should be clear. No x^m can be written as a linear combination of powers of x other than x^m.

If p is a polynomial of degree n, then there exist unique choices for α_0, α_1, $\ldots \alpha_n$, each in $\mathbb{R}$, such that $p(x) = \sum_{j=0}^{n} \alpha_j x^j$. Thus every polynomial is a linear combination of B, and every linear combination of B is a polynomial; hence $\mathbb{R}[x] = \text{span}(B)$.

In Example 9.5.2 we identified a set B which is a basis for $\mathbb{R}[x]$. We commented before that every subset of a basis fails to be a basis. For example, let $B' \subsetneq B$, and let $x^m \in B \setminus B'$. Then it is clear that since $x^m \notin B'$ then no linear combination of B' will equal x^m, even though $x^m \in \text{span}(\mathbb{R}[x])$. Consequently, B' cannot possibly be a basis for $\mathbb{R}[x]$.

Example 9.5.3 (*Basis for* $\mathbb{R}^3$)

Let's consider the three sets

- $B_1 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \right\}$.
  B_1 is not a basis because it is not linearly independent. In particular
  $$\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$
- $B_2 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}$.
  B_2 is a basis because it is linearly independent and spans $\mathbb{R}^3$. In particular, given any vector $b \in \mathbb{R}^3$ we can find $\alpha_1, \alpha_2, \alpha_3 \in \mathbb{R}$ such that
  $$\alpha_1 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + \alpha_3 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = b$$
  We know this because
  $$\det\left(\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \right) = 1 \neq 0\,.$$
- $B_3 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right\}$
  B_3 is not a basis because it does not span $\mathbb{R}^3$. In particular there are no $\alpha_1, \alpha_2 \in \mathbb{R}$ such that $\alpha_1 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.

Theorem 9.5.4

If V is a vector space with dimension $d < \infty$, then every linearly independent subset, $B \subset V$, is a basis if and only if $|B| = d$.

According to Theorem 9.5.4, every basis of $\mathbb{R}^3$ has exactly three elements. If we look at Example 9.5.3 we confirm that B_2 is a basis and $|B_2| = 3$, whereas B_1 and B_3 fail to be bases, a fact that is confirmed by the fact that $|B_1| = 4 \neq 3$ and $|B_3| = 2 \neq 3$.

Theorem 9.5.5 (*Existence of a Basis*)

Every vector space indeed has a basis.

Proof. Beyond the scope of this course

The claim that every vector space has a basis is somewhat obscure. In some cases the basis is a finite set (*e.g.* $\mathbb{R}^n$). In some cases the basis is a countably infinite set (*e.g.* $\mathbb{R}[x]$). However, in some cases the basis is an uncountable set (*e.g.* $\mathcal{C}^0[0, 2\pi]$ from Example 8.4.2).

9.5.1 Coordinates

In the following sections, unless otherwise indicated, we will only consider vector spaces which have a finite basis, and thus are finite dimensional vector spaces.

In a (finite dimensional) vector space, any element, v, of a vector space can be uniquely identified by its *coordinates* with respect to any given basis.

Definition 9.5.6 (*coordinates*)

If $B = \{v_1, v_2, \ldots, v_n\}$ is a basis for vector space, V, and

$a = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} \in \mathbb{R}^n$ such that $v = \sum_{k=1}^{n} \alpha_k v_k$, then a is said to be the *coordinate vector* of v with respect to basis B.

For Examples 9.5.7 and 9.7.3, recall that we can compute the inverse of a 2×2 matrix (if it exists) by inspection, see Theorem 4.8.1.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Example 9.5.7 (*Compute coordinates w.r.t. a basis*)

Suppose we have basis $B = \{v_1, v_2\} = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$, and let $v = 2v_1 + 3v_2 = 2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 3 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$.

Now turn it around. Given $\begin{bmatrix} 3 \\ 5 \end{bmatrix}$ how can we reproduce the coordinate vector $\begin{bmatrix} 2 \\ 3 \end{bmatrix}$?

To do this we need to solve the equation:

$$\begin{aligned} c_1 v_1 + c_2 v_2 &= \begin{bmatrix} 3 \\ 5 \end{bmatrix} \\ c_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + c_2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} &= \begin{bmatrix} 3 \\ 5 \end{bmatrix} \end{aligned} \tag{9.2}$$

We can write Equation (9.2) as a matrix equation.

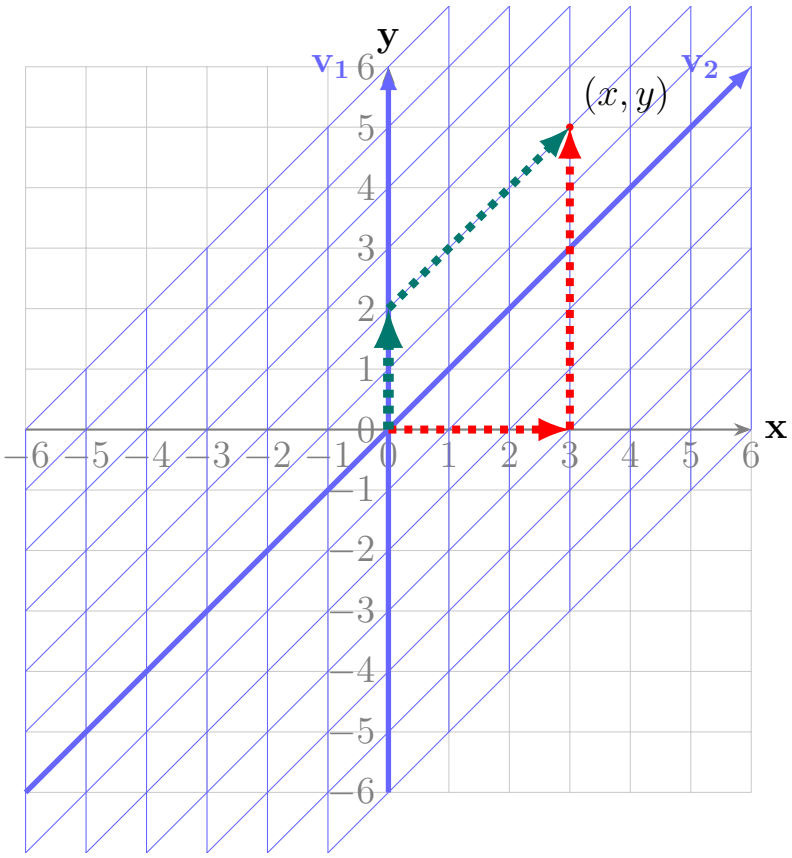


Figure 9.1: Illustration of Coordinate Computation for Example 9.5.7.
$\begin{bmatrix} x \\ y \end{bmatrix} = 2v_1 + 3v_2 = 3\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 5\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ where $v_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ $v_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 5 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 5 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

The coordinates computed in Example 9.5.7 are illustrated in Figure 9.1. We see that the vector indicated as (x, y) in the figure, can be thought of as $3\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 5\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ (on the grey grid) or equivalently as $2v_1 + 3v_2 = 2\begin{bmatrix} 0 \\ 1 \end{bmatrix} + 3\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ (on the blue grid).

9.5.2 Change of Basis

In Section 9.5.1 we talked about how to compute the coordinates of a known vector with respect to a given basis. The problem we address in this section, is similar but slightly more general. Suppose we know the coordinates of a vector in one coordinate system (with respect to one basis) and we wish to represent the vector in another coordinate system (with respect to a different basis). How do we convert between coordinate systems.

We start with a vector $v = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ and compute its coordinates to basis, B_1, $\{v_1, v_2\}$, and then the coordinates with respect to a second basis, B_2, $\{u_1, u_2\}$.

$$v_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \qquad v_2 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{9.3}$$

$$u_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \qquad u_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{9.4}$$

We represent each basis as a 2×2 matrix of column vectors.

$$B_1 = \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \tag{9.5}$$

$$B_2 = \begin{bmatrix} u_1 & u_2 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix} \tag{9.6}$$

Notice

$$\det(B_1) = \left(\frac{1}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\right) + \left(\frac{1}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\right) = \frac{1}{2} + \frac{1}{2} = 1\,.$$

So $B_1^{-1} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$.

To find the coordinates with respect to B_1, we solve for $\begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$.

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

$$\begin{aligned} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} &= \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ &= \begin{bmatrix} \frac{7}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \end{aligned}$$

meaning that the vector $\begin{bmatrix} 3 \\ 4 \end{bmatrix}$ has coordinates $\begin{bmatrix} \frac{7}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \approx \begin{bmatrix} 4.950 \\ 0.707 \end{bmatrix}$ with respect to basis $B_1 = \begin{bmatrix} v_1 & v_2 \end{bmatrix}$, as shown in Figure 9.2 (top). In the figure we see the point $\begin{bmatrix} 3 \\ 4 \end{bmatrix}$ in the grey coordinate system is equivalent to the point $\begin{bmatrix} 4.950 \\ 0.707 \end{bmatrix}$ in the blue coordinate system.

Now the question we ask is, given $\begin{bmatrix} \frac{7}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$ in the blue coordinate system B_1 how can we find its coordinates in coordinate system B_2? Clearly we could first find the coordinates in the standard coordinate system $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, then repeat the process shown above to convert to coordinate system B_2, as shown in Figure 9.2 (bottom).

Since each of the transformations in question is invertible, we should be able to convert from B_1 to B_2 directly with a single matrix multiplication, as shown in Figure 9.3. And in fact we can as is shown in Theorem 9.5.8.

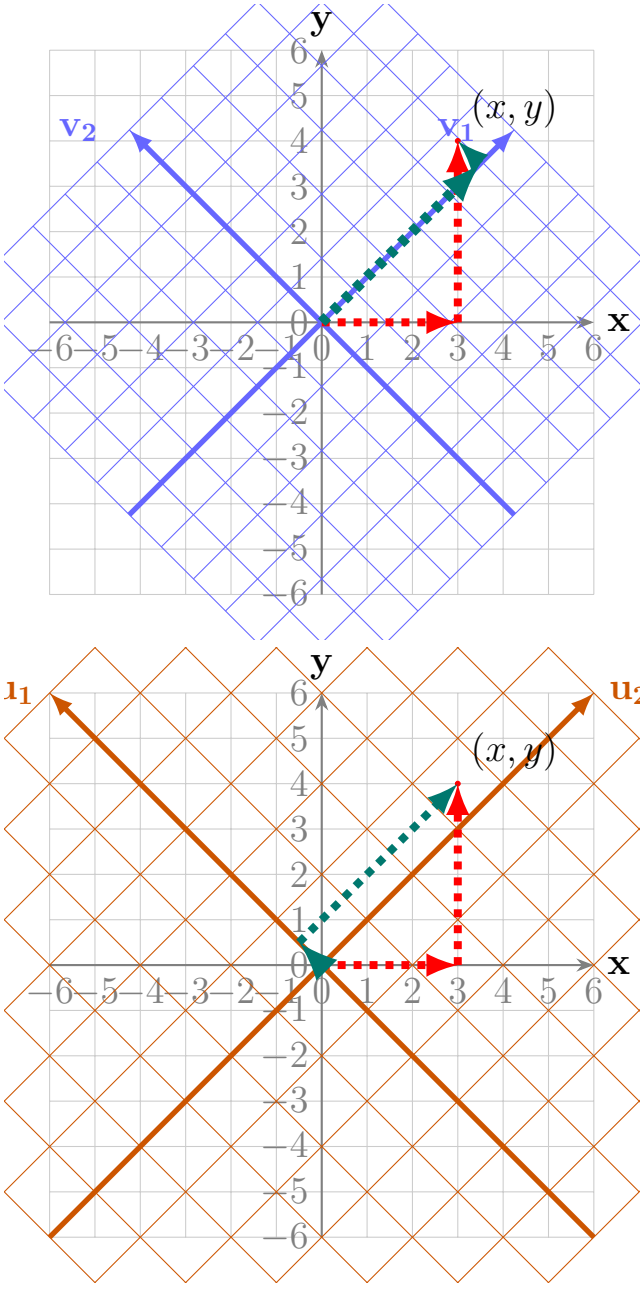


Figure 9.2: Illustration of Coordinate Computations: B_1 (top), B_2 (bottom)

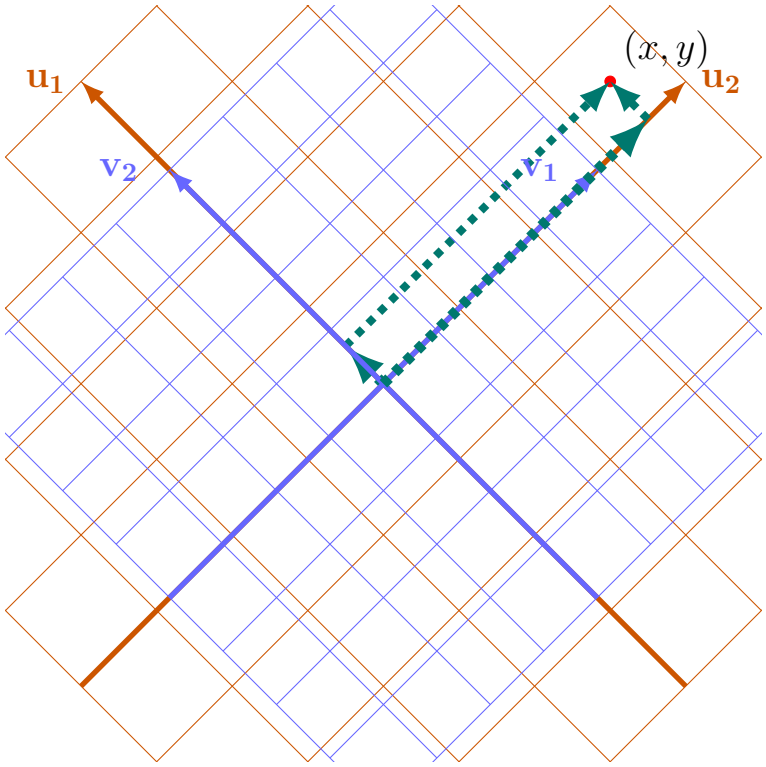


Figure 9.3: Illustration of Coordinate Computations: $B_1 \to B_2$

Theorem 9.5.8 specifies how to change coordinates between two coordinate systems given the bases for the two coordinate systems.

Theorem 9.5.8 (*Change of basis*)

If $B_1 = \begin{bmatrix} v_1 & v_2 \end{bmatrix}$ and $B_2 = \begin{bmatrix} u_1 & u_2 \end{bmatrix}$ represent two bases of $\mathbb{R}^n$, and $\begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix}$ is a coordinate vector of v with respect to basis B_1, then to change coordinate systems from B_1 to B_2; *i.e.* to express v in terms of coordinate system B_2:

$$\underbrace{\begin{bmatrix} c_{21} \\ c_{22} \\ \vdots \\ c_{2n} \end{bmatrix}}_{\text{Coordinates w.r.t. } B_2} = B_2^{-1} B_1 \underbrace{\begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix}}_{\text{Coordinates w.r.t. } B_1} \tag{9.7}$$

Proof. We have

$$B_1 \begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix} = v \qquad B_2 \begin{bmatrix} c_{21} \\ c_{22} \\ \vdots \\ c_{2n} \end{bmatrix} = v$$

We equate the two left-hand sides and solve for $\begin{bmatrix} c_{21} \\ c_{22} \\ \vdots \\ c_{2n} \end{bmatrix}$.

$$B_1 \begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix} = B_2 \begin{bmatrix} c_{21} \\ c_{22} \\ \vdots \\ c_{2n} \end{bmatrix}$$

$$B_2^{-1} B_1 \begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix} = B_2^{-1} B_2 \begin{bmatrix} c_{21} \\ c_{22} \\ \vdots \\ c_{2n} \end{bmatrix} = \begin{bmatrix} c_{21} \\ c_{22} \\ \vdots \\ c_{2n} \end{bmatrix} \tag{9.8}$$

Example 9.5.9 addresses the question: Given the coordinates with respect to one basis, can we compute the coordinates with respect to a second basis?

Example 9.5.9 (*Change of coordinate systems*)

Given v_1, v_2, u_1, u_2 as in (9.3) and (9.4), and given that vector v has coordinates, $\begin{bmatrix} c_{11} \\ c_{12} \end{bmatrix} = \begin{bmatrix} \frac{7}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$, with respect to basis $B_1 = \begin{bmatrix} v_1 & v_2 \end{bmatrix}$, what are the coordinates, $\begin{bmatrix} c_{21} \\ c_{22} \end{bmatrix}$, of v with respect to basis $B_2 = \begin{bmatrix} u_1 & u_2 \end{bmatrix}$?

The meaning of (9.8) is that given any coordinate vector with respect to basis, $\begin{bmatrix} v_1 & v_2 \end{bmatrix}$, we may multiply on the left by $B_2^{-1} B_1$ to obtain the coordinates of the same vector with respect to basis, $\begin{bmatrix} u_1 & u_2 \end{bmatrix}$.

First we compute $B_2^{-1}B_1$, noting that $\det\{B_2\} = (-1)(1) - (1)(1) = -2$.

$$B_2^{-1}B_1 = \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \qquad \text{by (9.5) and (9.6)}$$

$$= \frac{1}{-2} \begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \qquad \text{by Th. 4.8.1}$$

$$= -\frac{1}{2} \begin{bmatrix} \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$= -\frac{1}{2} \begin{bmatrix} 0 & -\frac{2}{\sqrt{2}} \\ -\frac{2}{\sqrt{2}} & 0 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix}$$

Knowing $B_2^{-1}B_1$, we compute $\begin{bmatrix} c_{21} \\ c_{22} \end{bmatrix}$ from $\begin{bmatrix} c_{11} \\ c_{12} \end{bmatrix}$.

$$\begin{bmatrix} c_{21} \\ c_{22} \end{bmatrix} = B_2^{-1}B_1 \begin{bmatrix} c_{11} \\ c_{12} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} \begin{bmatrix} \frac{7}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \cdot \frac{7}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ \frac{7}{2} \end{bmatrix}$$

We can verify this value of $\begin{bmatrix} c_{21} \\ c_{22} \end{bmatrix}$

$$B_2 \begin{bmatrix} c_{21} \\ c_{22} \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ \frac{7}{2} \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} + \frac{7}{2} \\ \frac{1}{2} + \frac{7}{2} \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

9.5.3 Complexity of Change of Basis

Notice in Theorem 9.5.8 that we must compute $(B_2)^{-1}B_1$ which means we must compute the inverse of B_2. As we will see later, (Theorem 9.8.2) if basis B_2 is orthonormal, then $(B_2)^{-1} = (B_2)^T$ so computing the inverse is trivial.

In Theorem 9.5.8 we see a computation: $B_2^{-1}B_1 \begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix}$. This computation can be done in two ways, which are mathematically equivalent, but computationally different.

$$\left(B_2^{-1}B_1\right) \begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix} = B_2^{-1} \left(B_1 \begin{bmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1n} \end{bmatrix} \right) \tag{9.9}$$

The complexity multiplying two $n \times n$ matrices (discussed in Sections 3.3 and 3.8.6) is $O\left(n^3\right)$ while the complexity of multiplying an $n \times n$ matrix by an n-vector is only $O\left(n^2\right)$.

This imbalance of complexity is significant and important. To compute the left-hand side of (9.9) requires $O\left(n^3\right) + O\left(n^2\right)$ of work while computing the right-hand side of (9.9) only requires $2O\left(n^2\right)$ of work. If the task is to compute a small number of coordinates (less than n) then it is better to use the right-hand side of (9.9). However, if the task is to compute a large number of coordinate transformations, it is better to precompute $B_2^{-1}B_1$, then reuse that matrix applying it to a given sequence of coordinates.

9.6 Orthonormal Vectors

If a vector space is equipped with an inner product, then Section 8.7 explains how we can define an angle between non-zero vectors. If this angle is $\frac{\pi}{2}$, then we say the non-zero vectors, u and v are orthogonal; consequently, then we have

$$\begin{aligned} \cos\frac{\pi}{2} &= \frac{\langle u, v\rangle}{\|u\|\|v\|} \\ &= 0 \\ \implies \langle u, v\rangle &= 0 \end{aligned}$$

Definition 9.6.1 (*orthogonal vectors*)

If u, v are non-zero vectors in inner product space, V, and $\langle u, v\rangle = 0$, then u and v are said to be *orthogonal*.

Definition 9.6.2 (*normal vector*)

A vector, u, is said to be a *normal vector*, if $\|u\| = 1$.

Furthermore, a vector $v \neq 0$ can be *normalized* by scaling it by the inverse of its norm: $\frac{1}{\|v\|}v$, because $\frac{1}{\|v\|}v$ is a normal vector.

Definition 9.6.3 (*orthonormal*)

A set of vectors $\{v_1, v_2, \ldots, v_n\}$ is said to be *orthonormal* if

1. $\|v_i\| = 1$ for $i = 1, 2, \ldots, n$
2. $\langle v_i, v_j\rangle = 0$ for $i \neq j$.

I.e., A set of vectors is orthonormal, if the vectors are normal and pairwise orthogonal.

Theorem 9.6.4

If vector space, V, has finite dimension, d, and $B \subset V$ is orthonormal with cardinality d, then B is a basis of V.

Theorem 9.6.4 says if a vector space has dimension, d, then each orthonormal set of size d is a basis.

Example 9.6.5 (*Standard basis is orthonormal*)

Verify that the standard basis is orthonormal. $B = \{v_1, v_2\}$ where $v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $v_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

$$\begin{aligned} \|v_1\| &= \sqrt{1^2 + 0^2} = \sqrt{1} = 1 \\ \|v_2\| &= \sqrt{0^2 + 1^2} = \sqrt{1} = 1 \\ \langle v_1, v_2 \rangle &= (1)(0) + (0)(1) = 0 + 0 = 0 \end{aligned}$$

Example 9.6.6 (*A non-orthonormal basis*)

Verify that the basis, $B = \{v_1, v_2\}$, is not orthonormal, where $v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $v_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

$$\begin{aligned} \|v_1\| &= \sqrt{1^2 + 0^2} = \sqrt{1} = 1 \\ \|v_2\| &= \sqrt{1^2 + 1^2} = \sqrt{2} \\ \langle v_1, v_2 \rangle &= (1)(1) + (0)(1) = 1 + 0 = 1 \end{aligned}$$

Example 9.6.7 (*Standard basis rotated* 45° *is orthonormal.*)

If we rotate the standard basis $45^\circ = \frac{\pi}{4}$ counterclockwise, we obtain another orthonormal basis. Verify that the basis $B = \{v_1, v_2\}$ is

orthonormal, where $v_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$ and $v_2 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

$$\|v_1\| = \sqrt{\left(\frac{1}{\sqrt{2}}\right)^2 + \left(\frac{1}{\sqrt{2}}\right)^2} = \sqrt{\frac{1}{2} + \frac{1}{2}} = 1$$

$$\|v_2\| = \sqrt{\left(-\frac{1}{\sqrt{2}}\right)^2 + \left(\frac{1}{\sqrt{2}}\right)^2} = \sqrt{\frac{1}{2} + \frac{1}{2}} = 1$$

$$\langle v_1, v_2 \rangle = \left(\frac{1}{\sqrt{2}}\right)\left(-\frac{1}{\sqrt{2}}\right) + \left(\frac{1}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\right) = -\frac{1}{2} + \frac{1}{2} = 0$$

9.7 Orthonormal Basis

In Example 9.5.7 we computed the coordinate of a given vector with respect to a given basis. If the basis is orthonormal, it is easier to find the coordinate vector.

Theorem 9.7.1 (*Coordinates with orthonormal basis*)

If $B = \{v_1, v_2, \ldots, v_n\}$ is an orthonormal basis, then we can extract the coordinate vector of v as follows.

$$c_k = \langle v_k, v \rangle \tag{9.10}$$

Thus we have $v = \sum_{k=1}^{n} \langle v_k, v \rangle \, v_k$.

Proof. Let $\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$ be the coordinates of v with respect to basis B. 🎓

$$v = c_1 v_1 + c_2 v_2 + \cdots + c_n v_n = \sum_{i=1}^{n} c_i v_i$$

$$\langle v_k, v \rangle = \left\langle v_k, \sum_{i=1}^{n} c_i v_i \right\rangle = \sum_{i=1}^{n} \langle v_k, c_i v_i \rangle = \sum_{i=1}^{n} c_i \langle v_k, v_i \rangle$$

$$= \sum_{i=1}^{k-1} c_i \underbrace{\langle v_k, v_i \rangle}_{=0} + c_k \underbrace{\langle v_k, v_k \rangle}_{=1} + \sum_{i=k+1}^{n} c_i \underbrace{\langle v_k, v_i \rangle}_{=0} = c_k$$

Equation (9.10) is easy to see for the *standard basis* $B = \{v_1, v_2\}$, where $v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $v_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

Example 9.7.2 (*Coordinates in the standard basis*)

Let $v = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$. We wish to compute the coordinates of v with respect to the standard basis

$$B = \{v_1, v_2\} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}.$$

We can recuperate 3 and 4 individually as follows.

$$\langle v, v_1 \rangle = \left\langle \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\rangle = 3 \times 1 + 4 \times 0 = 3 \tag{9.11}$$

$$\langle v, v_2 \rangle = \left\langle \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\rangle = 3 \times 0 + 4 \times 1 = 4\,. \tag{9.12}$$

We can verify (9.11) and (9.12):

$$\begin{aligned} v &= \langle v_1, v \rangle\, v_1 + \langle v_2, v \rangle\, v_2 \\ &= 3v_1 + 4v_2 \\ &= 3 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 4 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 3 \\ 4 \end{bmatrix} \end{aligned}$$

Example 9.7.2 may not seem important, but we can carry out the same process on any orthonormal basis. In Example 9.7.3 we look at extracting the coordinates of a given vector with respect to a non-standard basis,

$$\left\{ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \right\}.$$

Example 9.7.3 (*Coordinates by matrix inverse.*)

As in Example 9.7.2, let $v = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$. Find the coordinates of v in the basis $B = \{v_1, v_2\}$, where $v_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$ and $v_2 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$. *I.e..*,

find c_1, c_2 such that $v = c_1 v_1 + c_2 v_2 = c_1 \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} + c_2 \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$.

We can solve the equation:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \tag{9.13}$$

using any technique we've learned so far:

- Gaussian elimination with back-substitution
- Gauss-Jordan elimination
- Inverse matrix.

We will use Theorem 4.8.1 to invert the 2×2 matrix by inspection, using the fact that $\det\left(\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}\right) = 1$

$$\begin{aligned} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} &= \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ &= \frac{1}{1} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ &= \begin{bmatrix} \frac{3}{\sqrt{2}} + \frac{4}{\sqrt{2}} \\ \frac{-3}{\sqrt{2}} + \frac{4}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{7}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \end{aligned}$$

In the following example, we again compute the coordinates of the

vector $\begin{bmatrix}3\\4\end{bmatrix}$ in the basis $B = \left\{ \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}, \begin{bmatrix}-\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix} \right\}$, but we exploit that fact that we know the basis is orthonormal.

Example 9.7.4 (*Coordinates by inner product*)

Again, let B be as in Example 9.7.2. We have shown that $B = \{v_1, v_2\}$ is orthonormal (Example 9.6.7) so we can use the inner product to compute c_1 and c_2, where $v_1 = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}$ and $v_2 = \begin{bmatrix}-\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}$.

$$
\begin{aligned}
c_1 = \langle v, v_1 \rangle &= \left\langle \begin{bmatrix}3\\4\end{bmatrix}, \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix} \right\rangle && \text{by Theorem 9.7.1} \\
&= \frac{3}{\sqrt{2}} + \frac{4}{\sqrt{2}} = \frac{7}{\sqrt{2}} \\
c_2 = \langle v, v_2 \rangle &= \left\langle \begin{bmatrix}3\\4\end{bmatrix}, \begin{bmatrix}-\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix} \right\rangle && \text{by Theorem 9.7.1} \\
&= \frac{-3}{\sqrt{2}} + \frac{4}{\sqrt{2}} = \frac{1}{\sqrt{2}}
\end{aligned}
$$

We see that $\begin{bmatrix}c_1\\c_2\end{bmatrix} = \begin{bmatrix}\frac{7}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}$ agrees with the results we obtained in Example 9.7.3

Next we look at a sequence of examples using a 3×3 matrix. We verify that a given set, B, of three vectors is indeed a basis, and in fact is an orthonormal basis.

$$B = \left\{ \underbrace{\frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{=v_1}, \underbrace{\frac{1}{\sqrt{6}} \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}}_{=v_2}, \underbrace{\frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}}_{=v_3} \right\} . \tag{9.14}$$

Example 9.7.5 (*Show Linear Independence*)

We could verify that B is a basis by verifying that v_1, v_2, and v_3 are pairwise orthogonal

$$\langle v_1, v_2 \rangle = \langle v_2, v_3 \rangle = \langle v_1, v_3 \rangle = 0 \, .$$

Orthogonal vectors are necessarily linearly independent. However, we could verify linear independence by computing the determinant of the matrix form by the column vectors. If the determinant is not 0, then the vectors are linear independent.

$$\begin{aligned}
\det(B) &= \det\left(\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \right) \\
&= \frac{1}{\sqrt{3}} \det\left(\begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \right) - \frac{-2}{\sqrt{6}} \det\left(\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \right) + 0 \\
&= \frac{(\frac{1}{\sqrt{6}})(\frac{-1}{\sqrt{2}}) - (\frac{1}{\sqrt{6}})(\frac{1}{\sqrt{2}})}{\sqrt{3}} + \frac{2\Big((\frac{1}{\sqrt{3}})(\frac{-1}{\sqrt{2}}) - (\frac{1}{\sqrt{3}})(\frac{1}{\sqrt{2}})\Big)}{\sqrt{6}} \\
&= \frac{-2}{6} + \frac{-4}{6} = \frac{-6}{6} = -1
\end{aligned}$$

This computation of $\det(B)$ is not a problem when done programmatically, but when computing it by hand a shortcut is rec-

ommended exploiting the identity

$$\det(\alpha B) = (\alpha^{dim(B)}) \det(B)\,.$$

$$\begin{aligned}
\det(B) &= \det\left(\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix}\right) \\
(\sqrt{6})^3 \ \det(B) &= \det\left(\sqrt{6}\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix}\right) \\
6\sqrt{6} &= \det\left(\begin{bmatrix} \sqrt{2} & -2 & 0 \\ \sqrt{2} & 1 & \sqrt{3} \\ \sqrt{2} & 1 & -\sqrt{3} \end{bmatrix}\right) \\
&= \sqrt{2}\det\left(\begin{bmatrix} 1 & \sqrt{3} \\ 1 & -\sqrt{3} \end{bmatrix}\right) + 2\det\left(\begin{bmatrix} \sqrt{2} & \sqrt{3} \\ \sqrt{2} & -\sqrt{3} \end{bmatrix}\right) \\
&= \sqrt{2}\Big(-\sqrt{3} - \sqrt{3}\Big) + 2\Big(-\sqrt{6} - \sqrt{6}\Big) \\
&= -2\sqrt{6} - 4\sqrt{6} = -6\sqrt{6} \\
\det(B) &= \frac{6\sqrt{6}}{-6\sqrt{6}} = -1
\end{aligned}$$

Example 9.7.6 (*Show Orthogonality*)

Now we show that the vectors are pairwise orthogonal by computing $\langle v_i, v_j \rangle$ for all $i \neq j$.

$$\begin{aligned}
\langle v_1, v_2 \rangle &= \left\langle \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \frac{1}{\sqrt{6}} \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \right\rangle \\
&= \left(\frac{1}{\sqrt{3}}\right)\left(\frac{1}{\sqrt{6}}\right) \left\langle \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \right\rangle \\
&= \left(\frac{1}{\sqrt{3}}\right)\left(\frac{1}{\sqrt{6}}\right)((1)(-2) + (1)(1) + (1)(1)) \\
&= \left(\frac{1}{\sqrt{3}}\right)\left(\frac{1}{\sqrt{6}}\right)(-2 + 1 + 1) = 0
\end{aligned}$$

In like manner, the student may verify: $\langle v_2, v_3 \rangle = \langle v_1, v_3 \rangle = 0$.

Example 9.7.7 (*Show Normality*)

We may verify that each vector is normal by showing that $\|v_1\| = \|v_2\| = \|v_3\| = 1$.

$$\begin{aligned}
\|v_1\| &= \left\| \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\| = \left\| \begin{bmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix} \right\| \\
&= \sqrt{\left(\frac{1}{\sqrt{3}}\right)^2 + \left(\frac{1}{\sqrt{3}}\right)^2 + \left(\frac{1}{\sqrt{3}}\right)^2} \\
&= \sqrt{\frac{1}{3} + \frac{1}{3} + \frac{1}{3}} = \sqrt{1} = 1
\end{aligned}$$

We could also compute $\|v_1\|$ as follows:

$$\|v_1\| = \left\| \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\| = \left| \frac{1}{\sqrt{3}} \right| \left\| \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\|$$
$$= \frac{1}{\sqrt{3}} \sqrt{1^2 + 1^2 + 1^2} = \frac{1}{\sqrt{3}} \sqrt{3} = 1$$

In like manner, the student may verify: $\|v_2\| = \|v_3\| = 1$.

Example 9.7.8 (*Coordinates in a* 3×3 *orthonormal basis*)

Let B be defined as in Equation (9.14).

Find the coordinates of $v = \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}$ with respect to basis, B.

Find $\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$ such that $v = \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix} = c_1 v_1 + c_2 v_2 + c_3 v_3$ as follows:

$$
\begin{aligned}
c_1 &= \langle v, v_1 \rangle \\
&= \left\langle \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}, \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\rangle = \frac{1}{\sqrt{3}} \left\langle \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\rangle \\
&= \frac{1}{\sqrt{3}}((1)(1) + (2)(1) + (-3)(1)) = 0
\end{aligned}
$$

$$
\begin{aligned}
c_2 &= \langle v, v_2 \rangle \\
&= \left\langle \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}, \frac{1}{\sqrt{6}} \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \right\rangle = \frac{1}{\sqrt{6}} \left\langle \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}, \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \right\rangle \\
&= \frac{1}{\sqrt{6}}((1)(-2) + (2)(1) + (-3)(1)) = \frac{-3}{\sqrt{6}}
\end{aligned}
$$

$$
\begin{aligned}
c_3 &= \langle v, v_3 \rangle \\
&= \left\langle \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}, \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \right\rangle = \frac{1}{\sqrt{2}} \left\langle \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \right\rangle \\
&= \frac{1}{\sqrt{2}}((1)(0) + (2)(1) + (-3)(-1)) = \frac{5}{\sqrt{2}}
\end{aligned}
$$

We may verify this computation by assuring that $B \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = v$.

$$B \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 \\ \frac{-3}{\sqrt{6}} \\ \frac{5}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 0 + \frac{-2}{\sqrt{6}}\frac{-3}{\sqrt{6}} + 0 \\ 0 + \frac{1}{\sqrt{6}}\frac{-3}{\sqrt{6}} + \frac{1}{\sqrt{2}}\frac{5}{\sqrt{2}} \\ 0 + \frac{1}{\sqrt{6}}\frac{-3}{\sqrt{6}} - \frac{1}{\sqrt{2}}\frac{5}{\sqrt{2}} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{6}{6} \\ \frac{-3}{6} + \frac{5}{2} \\ \frac{-3}{6} - \frac{5}{2} \end{bmatrix} = \begin{bmatrix} \frac{6}{6} \\ \frac{-1}{2} + \frac{5}{2} \\ \frac{-1}{2} - \frac{5}{2} \end{bmatrix} = \begin{bmatrix} \frac{6}{6} \\ \frac{4}{2} \\ \frac{-6}{2} \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix} = v$$

9.8 Orthogonal Matrix

Definition 9.8.1 (*orthogonal matrix*)

If $B = \{v_1, v_2, \ldots, v_n\}$ is an orthonormal basis of $\mathbb{R}^n$, the matrix formed by its columns is called an *orthogonal matrix*.

As an example, look again at the basis, B, be as in Equation (9.14). What is the inverse of A if

$$A = \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix} ? \tag{9.15}$$

It turns out that if a matrix is orthogonal, then its inverse and its transpose are the same.

Theorem 9.8.2 (*Inverse of Orthogonal Matrix*)

If A is an orthogonal matrix, then

$$A^{-1} = A^T$$

Proof. We will show that $A^TA = I$, by which we know that A^T and A are inverses of each other. As in Theorem 8.6.1, let $A = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$. We know by Equation 8.3 that

$$A^TA = \begin{bmatrix} \langle v_1, v_1 \rangle & \langle v_1, v_2 \rangle & \cdots & \langle v_1, v_n \rangle \\ \langle v_1, v_2 \rangle & \langle v_2, v_2 \rangle & \cdots & \langle v_2, v_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle v_1, v_n \rangle & \langle v_2, v_n \rangle & \cdots & \langle v_n, v_n \rangle \end{bmatrix}. \tag{9.16}$$

Since A is orthogonal, we know that v_i and v_j are orthogonal when $i \neq j$, so $\langle v_i, v_j \rangle = 0$. Furthermore we know that $\langle v_i, v_i \rangle = \|v_i\|^2 = 1$. So we have 1 along the diagonal, and 0 in every non-diagonal entry of A^TA. *I.e.*, $A^TA = I$.

Since $A^TA = I$, then $A^T = A^{-1}$.

Example 9.8.3 (*Product of orthogonal matrix with its transpose*)

Let's multiply $A = \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$ by its transpose.

$$A\,A^T = \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \times \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ \frac{-2}{\sqrt{6}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Example 9.8.4 (*Inverse of* 2×2 *orthogonal matrix*)

Recall in the Example 9.7.3, we computed the inverse of

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

We computed the inverse using Theorem 4.8.1 which says:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

However, since we know that $\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$ is an orthogonal matrix, we know its inverse is simply its transpose.

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}^{T} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

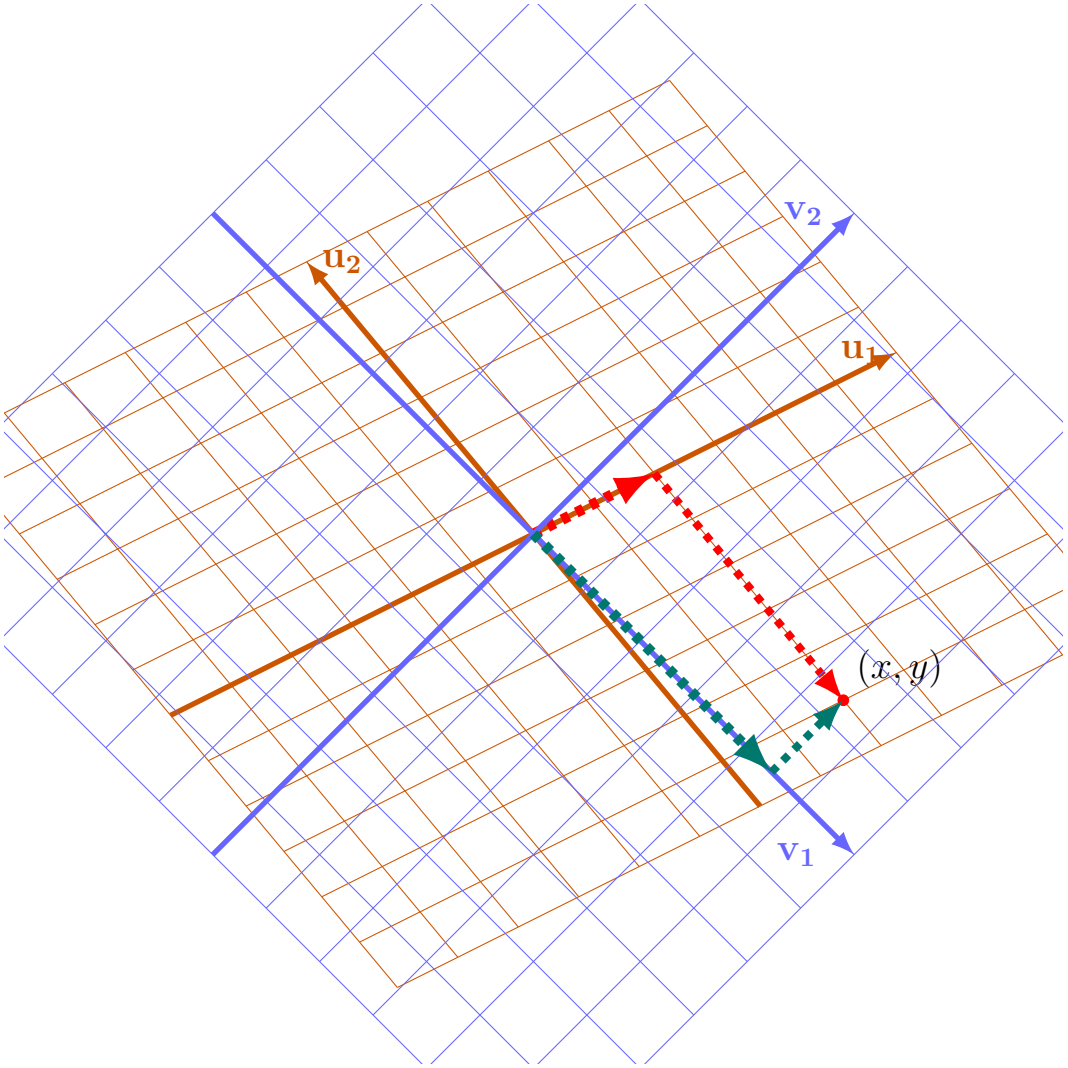


Figure 9.4: Illustration of Coordinate Computation from a non-orthonormal basis to orthonormal basis

Example 9.8.5 (*Change of Coordinates to an Orthonormal Basis*)

Let's suppose B_1 is the basis $\{u_1, u_2\}$ where $u_1 = \begin{bmatrix} 0.8 \\ 0.4 \end{bmatrix}$, $u_2 = \begin{bmatrix} -0.5 \\ 0.6 \end{bmatrix}$. Furthermore, suppose B_2 is the basis $\{v_1, v_2\}$ where $v_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}$, $v_2 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$. $\{v_1, v_2\}$ is already shown to be orthonormal. Suppose (x, y) is a point with coordinates $p_1 = \begin{bmatrix} 2 \\ -5 \end{bmatrix}$ in coordinate system B_1. What are the coordinates of p in coordinate system B_2?

The point $p = (x, y)$ is illustrated in Figure 9.4.

According to Theorem 9.5.8, $p_2 = B_2^{-1} B_1 p_1$, where

$$B_1 = \begin{bmatrix} u_1 & u_2 \end{bmatrix} = \begin{bmatrix} 0.8 & -0.5 \\ 0.4 & 0.6 \end{bmatrix}$$

$$B_2 = \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Since B_2 is orthogonal, then $B_2^{-1} = B_2^T = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$.

Thus

$$B_2^{-1}B_1 = B_2^T B_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0.8 & -0.5 \\ 0.4 & 0.6 \end{bmatrix}$$
$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0.8 & -0.5 \\ 0.4 & 0.6 \end{bmatrix}$$
$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 0.8-0.4 & -0.5-0.6 \\ 0.8+0.4 & -0.5+0.6 \end{bmatrix}$$
$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 0.4 & -1.1 \\ 1.2 & 0.1 \end{bmatrix}$$

$$B_2^{-1}B_1 p_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 0.4 & -1.1 \\ 1.2 & 0.1 \end{bmatrix} \begin{bmatrix} 2 \\ -5 \end{bmatrix}$$
$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 6.3 \\ 1.9 \end{bmatrix} \approx \begin{bmatrix} 4.455 \\ 1.343 \end{bmatrix}$$

We provide an alternate proof of Theorem 9.7.1.

Theorem 9.8.6 (*Coordinates with orthonormal basis (revisited)*)

If $B = \{v_1, v_2, \ldots, v_n\}$ is an orthonormal basis, then we can extract the coordinate vector of v as follows.

$$c_k = \langle v_k, v \rangle$$

Proof. Let $\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$ be the coordinates of v with respect to basis B.

Let matrix $B = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$ be the matrix whose ith column is v_i. Notice that B is an orthonormal matrix.

$$
\begin{aligned}
v &= c_1 v_1 + c_2 v_2 + \cdots + c_n v_n \\
&= \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = B \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}
\end{aligned}
$$

$$
\underbrace{B^{-1}}_{=B^T} v = \underbrace{B^{-1} B}_{=I} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}
$$

$$
B^T v = \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix} v = \begin{bmatrix} v_1^T v \\ v_2^T v \\ \vdots \\ v_n^T v \end{bmatrix} = \begin{bmatrix} \langle v_1, v \rangle \\ \langle v_2, v \rangle \\ \vdots \\ \langle v_n, v \rangle \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}
$$

This means c_k equals the kth row of B^T times the column v. But the kth row of B^T is the kth column of B or simply v_k. Thus

$$
c_k = \langle v_k, v \rangle \ .
$$

9.9 Gram-Schmidt Process

We have seen that any subset of d linearly independent vectors is a basis for the given vector space. But the vectors may not be orthonormal. The Gram-Schmidt process is a way of *converting* an arbitrary basis to an orthonormal basis.

The Gram-Schmidt process depends on the definition of *projection* (Definition 8.9.1) and the formula given in Theorem 8.9.2.

$$\text{proj}_u[v] = \frac{\langle u, v\rangle}{\|u\|^2} u\,. \tag{9.17}$$

In Python, if you want to compute $\text{proj}_u[v]$, it is sub-optimal to use the Equation 9.17 because $\|u\|$ is computed as $\sqrt{\langle u, u\rangle}$; *i.e.*, you'd like your code to avoid redundantly computing $\left(\sqrt{\langle u, u\rangle}\right)^2$. It is better to implement the Python `proj` function using Equation 9.18

$$\text{proj}_u[v] = \frac{\langle u, v\rangle}{\langle u, u\rangle} u\,. \tag{9.18}$$

As shown in Figure 9.5 if we are given two vectors v_1, v_2 we can generate a vector, u_2, which is orthogonal to v_1 and is a linear combination of v_1, v_2. Looking at the figure we can derive Equation (9.19).

$$u_2 = v_2 - \text{proj}_{v_1}[v_2] \tag{9.19}$$

There are several ways to be convinced of Equation (9.19). One way is just to look at the proof in Theorem 8.9.6

9.9.1 Algebraic Motivation of Gram Schmidt

It may seem that Theorem 8.9.6 is unmotivated. Perhaps a better (or at least alternative) way to understand Equation (9.19) is as follows.

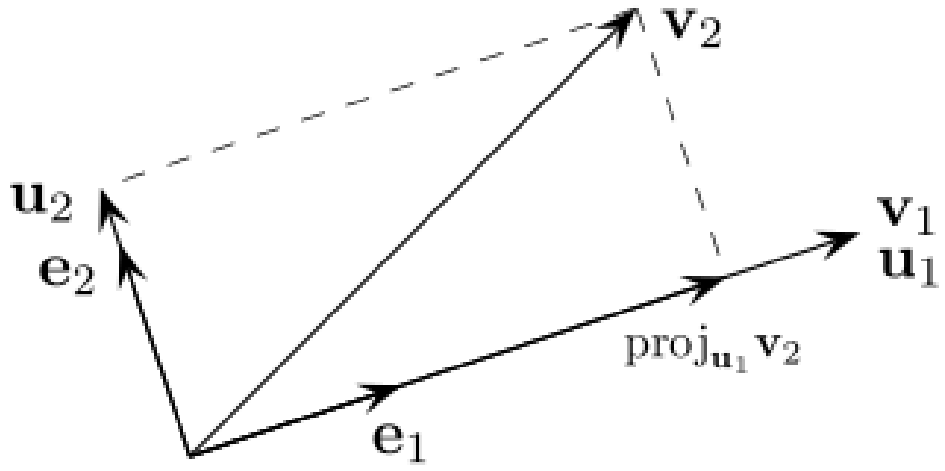


Figure 9.5: Generating an orthonormal vector.

Let $u_1 = v_1$, and require that $\alpha, \beta \in \mathbb{R}$.

$$u_2 = \alpha v_2 + \beta u_1 \tag{9.20}$$
$$0 = \langle u_1, u_2 \rangle \tag{9.21}$$

Solve for α and β given the Equations (9.20) and (9.21).

$$\begin{aligned} 0 &= \langle u_1, u_2 \rangle \\ &= \langle u_1, \alpha v_2 + \beta u_1 \rangle \\ &= \langle u_1, \alpha v_2 \rangle + \langle u_1, \beta u_1 \rangle \\ &= \alpha \langle u_1, v_2 \rangle + \beta \langle u_1, u_1 \rangle \\ \beta &= \frac{-\langle u_1, v_2 \rangle}{\alpha \langle u_1, u_1 \rangle} \\ \beta u_1 &= \frac{-1}{\alpha} \frac{\langle u_1, v_2 \rangle}{\langle u_1, u_1 \rangle} u_1 = \frac{-1}{\alpha} \text{proj}_{u_1}[v_2] \end{aligned} \tag{9.22}$$

We see from Equation (9.22) that the length of vector v_2 plays

no role in the value of α. So we are free to chose $\alpha = 1$, so that Equation (9.20) becomes exactly Equation (9.19).

$$\begin{aligned} u_2 &= \alpha v_2 + \beta u_1 \\ &= v_2 - \text{proj}_{u_1}[v_2] \end{aligned}$$

9.9.2 Geometric Motivation for Gram Schmidt

Another way to understand Equation (9.19) is to look closely at Figure 9.5. We are given v_1 and v_2, and we want to derive orthogonal vectors u_1 and u_2; *i.e.*, $\langle u_1, u_2 \rangle = 0$. Geometrically, in the figure, v_2 is the sum of the two vectors $\text{proj}_{v_1}[v_2]$ and u_2. *I.e.*,

$$v_2 = u_2 + \text{proj}_{v_1}[v_2] \tag{9.23}$$

We simply define $u_1 = v_1$. But what is u_2? It is the vector we are trying to define. We need to define u_2 so that Equation (9.23) is satisfied, and so that simultaneously $\langle u_1, u_2 \rangle = 0$. So, if we simply take Equation (9.23) as the definition of u_2 and solve for u_2 we get $u_2 = v_2 - \text{proj}_{v_1}[v_2]$ which is Equation (9.19).

It remains to be shown, however, that $\langle u_1, u_2 \rangle = 0$. The orthogonality of u_1 and u_2 is guaranteed by Theorem 8.9.6.

9.9.3 Gram Schmidt Formally

It turns out we can repeat this process alluded to in Sections 9.9.1 and 9.9.2, not only for two given vectors, but for any finite number of given vectors $v_1, v_2, \ldots, v_n$.

Theorem 9.9.1 (*Gram-Schmidt*)

Let V be a vector space of dimension n equipped with an inner product and a norm induced by the inner product. Let $B = \{v_1, v_2, \ldots, v_n\}$ be a basis (not necessarily orthonormal). Then $U = \{u_1, u_2, \ldots, u_n\}$ is an orthogonal basis, if

$$
\begin{aligned}
u_1 &= v_1 \\
u_2 &= v_2 - \operatorname{proj}_{u_1}[v_2] \\
u_3 &= v_3 - \operatorname{proj}_{u_1}[v_3] - \operatorname{proj}_{u_2}[v_3] \\
u_4 &= v_4 - \operatorname{proj}_{u_1}[v_4] - \operatorname{proj}_{u_2}[v_4] - \operatorname{proj}_{u_3}[v_4] \\
&\vdots \\
u_i &= v_i - \sum_{k=1}^{i-1} \operatorname{proj}_{u_k}[v_i] \\
&\vdots \\
u_n &= v_n - \sum_{k=1}^{n-1} \operatorname{proj}_{u_k}[v_n]
\end{aligned}
$$

Moreover $E = \{e_1, e_2, \ldots, e_n\}$ is an orthonormal basis, if

$$e_k = \frac{u_k}{\|u_k\|}$$

Proof. The basis E is clearly normal, because by construction, $e_k = \frac{u_k}{\|u_k\|}$. It thus suffices to show that any two elements of the basis are orthogonal. Take e_i, e_j, $i \neq j$ and show $\langle e_i, e_j \rangle = 0$. Without loss of generality assume $0 < i < j \leq n$.

$$\langle e_i, e_j \rangle = \left\langle \frac{u_i}{\|u_i\|}, \frac{u_j}{\|u_j\|} \right\rangle = \frac{\langle u_i, u_j \rangle}{\|u_i\| \ \|u_j\|}$$

This means that $\langle e_i, e_j \rangle = 0$ whenever $\langle u_1, u_2 \rangle = 0$.
Thus we show that $\langle u_i, u_j \rangle = 0$ by induction.

- **Base case**: Show $\langle u_1, u_2 \rangle = 0$
  Proven in Theorem 8.9.6.
- **Inductive case**: Assume $\langle u_i, u_j \rangle = 0$ if and only if $0 \leq i < j < n$. Take $0 < \ell < n$. Show $\langle u_\ell, u_n \rangle = 0$.

$$\begin{aligned}
\langle u_\ell, u_n \rangle &= \left\langle u_\ell, v_n - \sum_{k=1}^{n-1} \text{proj}_{u_k}[v_n] \right\rangle \\
&= \left\langle u_\ell, v_n - \sum_{k=1}^{n-1} \frac{\langle v_n, u_k \rangle}{\langle u_k, u_k \rangle} u_k \right\rangle \\
&= \langle u_\ell, v_n \rangle - \left\langle u_\ell, \sum_{k=1}^{n-1} \frac{\langle v_n, u_k \rangle}{\langle u_k, u_k \rangle} u_k \right\rangle \\
&= \langle u_\ell, v_n \rangle - \sum_{k=1}^{n-1} \left(\frac{\langle v_n, u_k \rangle}{\langle u_k, u_k \rangle} \underbrace{\langle u_\ell, u_k \rangle}_{=0 \text{ when } k \neq \ell} \right) \\
&= \langle u_\ell, v_n \rangle - \frac{\langle v_n, u_\ell \rangle}{\langle u_\ell, u_\ell \rangle} \langle u_\ell, u_\ell \rangle \\
&= \langle u_\ell, v_n \rangle - \langle v_n, u_\ell \rangle = 0
\end{aligned}$$

The algorithm proposed in Theorem 9.9.1 allows us to compute $u_1, u_2, \ldots, u_n$ from $v_1, v_2, \ldots, v_n$, by subtracting off projections of previously computed values of u_k. However, we can simplify the Gram-

Schmidt process slightly by applying Corollary 8.9.4 and computing the projection onto a unit vector. Let $e = \frac{u}{\|u\|}$, then

$$\text{proj}_e[v] = \langle e, v \rangle e\,.$$

We can now compute u_1, e_1, u_2, e_2, …, u_n, e_n as follows:

$$\begin{array}{rcl|rcl}
u_1 &=& v_1 & e_1 &=& \frac{u_1}{\|u_1\|} \\
u_2 &=& v_2 - \langle e_1, v_2 \rangle e_1 & e_2 &=& \frac{u_2}{\|u_2\|} \\
u_3 &=& v_3 - \langle e_1, v_3 \rangle e_1 - \langle e_2, v_3 \rangle e_2 & e_3 &=& \frac{u_3}{\|u_3\|} \\
u_4 &=& v_4 - \langle e_1, v_4 \rangle e_1 - \langle e_2, v_4 \rangle e_2 - \langle e_3, v_4 \rangle e_3 & e_4 &=& \frac{u_4}{\|u_4\|} \\
\dots & & & \dots & & \\
u_i &=& v_i - \sum_{k=1}^{i-1} \langle e_k, v_i \rangle e_k & e_i &=& \frac{u_i}{\|u_i\|} \\
\dots & & & \dots & & \\
u_n &=& v_n - \sum_{k=1}^{n-1} \langle e_k, v_n \rangle e_k & e_n &=& \frac{u_n}{\|u_n\|}
\end{array}$$

One additional optimization which makes the Python code simpler happens when we notice that u_k is only used to compute e_k. *I.e.*, e_k is the normalized version of u_k. But in Python we have a method on `Vector` named `normalize`, which lets us compute e_k without ever assigning to u_k. The formula for e_i is basically (pseudocode)

$$e_i = \left(v_i - \sum_{k=1}^{i-1} \langle e_k, v_i \rangle e_k \right).normalize()$$

Attention: the lower limit of the summation $\sum_{k=1}^{i-1}$ is $k = 1$ in the mathematical representation, but is $k = 0$ in the Python code.

Suppose the `Vector`, `v`, has dimension n, i.e., `v[0]`, `v[1]`, …, `v[n-1]`. then we can compute `e` as follows.

```
n = v.dim
```

```
    zero = Vector.zero(n)
    e = [zero] * n
    for i in range(n):
      e[i] = (v[i] - sum((e[k].scale(e[k].inner_product(v[i]))
                          for k in range(i)),
                         zero)
              ).normalize()
```

9.10 Homework

- Finish implementation Vector.py
  - is_orthogonal
- Finish implementation VectorSpace.py
  - gram_schmidt
  - find_coordinates
  - change_coordinates
- Test with
  - orthogonal_test.py
  - gram_schmidt_test.py
  - coordinates_test.py

Chapter 10

Eigenvalues

Chapter Objectives

- Define eigenvalues and eigenvectors of a square matrix and explain their geometric meaning as scaling directions.
- Compute eigenvalues by solving the characteristic equation $\det(A - \lambda I) = 0$.
- Find a basis of eigenvectors for each eigenspace.
- Diagonalize a matrix $A = PDP^{-1}$ when possible and use diagonalization to compute A^n in closed form.

10.1 Motivation

Recall in Section 3.8.5 we discussed an algorithm called *fast-power*, to compute a power of a matrix such as M^n which uses m multiplications where $\log_2 n \leq m \leq 2 \log_2 n$. The fast-power algorithm applies not only to matrices but to any monoid. In particular the set of $n \times n$ matrices is a monoid under matrix multiplication.

How many floating-point computations are needed to compute M^n using naive matrix multiplication? If the matrix has dimension $d \times d$, then a single naive matrix multiplication requires approximately d^3 floating point multiplications. *I.e.*, in the worst case to compute M^n requires $2d^3 \log_2 n$ floating point multiplications.

In Section 3.8.7 we discussed the Strassen algorithm which reduces the complexity of matrix multiplication from n^3 to approximately $n^2 \log_{2.7} n$. But overhead in the algorithm makes it useless for sizes of matrices below $2^{10} \times 2^{10}$.

The technique of fast-power works well if we only need to compute M^n. However, if we actually need to compute M^0, M^1, M^2, M^3, ..., M^n, then the naive approach of $M^n = M \times M^{n-1}$ is more efficient.

Example 10.1.1 (*Approximate* e^M *for a square matrix* M)

Recall that for real numbers, we can approximate the exponential by computing sufficiently many terms of the infinite series

$$e^x = \sum_{k=0}^{\infty} \frac{1}{k!} x^k \,.$$

If M is an $d \times d$ matrix, then we can also compute

$$e^M = \sum_{k=0}^{\infty} \frac{1}{k!} M^k \,, \tag{10.1}$$

in which case we indeed need to compute each of M^0, M^1, M^2, M^3, ..., M^n for some sufficiently large value of n so that

$$\left\| \left(\sum_{k=0}^{n-1} \frac{1}{k!} M^k \right) - \left(\sum_{k=0}^{n} \frac{1}{k!} M^k \right) \right\| = \frac{\|M^n\|}{n!} < \epsilon \,.$$

Similar to Example 10.1.1 we can also compute

$$\sin M = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} M^{2k+1} \tag{10.2}$$

$$\cos M = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} M^{2k} \tag{10.3}$$

$$\tag{10.4}$$

Using Equations (10.2) and (10.3) we can verify many familiar trig identities such as

$$\sin^2 M + \cos^2 M = I \,,$$

where I is the identity matrix, and

$$\cos(2M) = (2\cos^2 M) - I \,.$$

In this chapter we present a concept called eigenvalue (and eigenvector) which has many applications in mathematics, science, and engineering. One application in computer science is that it allows us, in some circumstances, to reduce even more the work and time needed to compute M^n. In summary if we can decompose M as $M = PDP^{-1}$ where D is a diagonal matrix, then we can compute M^n as

$$\begin{aligned} M^n &= \left(PDP^{-1}\right)^n \\ &= PD^nP^{-1} \end{aligned} \tag{10.5}$$

To understand Equation (10.5), we observe the following pattern.

$$
\begin{aligned}
M^2 &= \big(PDP^{-1}\big)\big(PDP^{-1}\big) \\
&= (PD)\underbrace{\big(P^{-1}P\big)}_{=I}\big(DP^{-1}\big) \\
&= (PD)I\big(DP^{-1}\big) \\
&= PD^2P^{-1}
\end{aligned}
$$

$$
\begin{aligned}
M^3 &= \big(PDP^{-1}\big)\big(PDP^{-1}\big)\big(PDP^{-1}\big) \\
&= PD\underbrace{\big(P^{-1}P\big)}_{=I}D\underbrace{\big(P^{-1}P\big)}_{=I}DP^{-1} \\
&= PDI(D)IDP^{-1} \\
&= PD^3P^{-1}
\end{aligned}
$$

$$
\begin{aligned}
M^4 &= \big(PDP^{-1}\big)\big(PDP^{-1}\big)\big(PDP^{-1}\big)\big(PDP^{-1}\big) \\
&= PD\big(P^{-1}P\big)D\big(P^{-1}P\big)D\big(P^{-1}P\big)DP^{-1} \\
&= PDIDIDIDP^{-1} \\
&= PD^4P^{-1}
\end{aligned}
$$

$$
\begin{aligned}
M^n &= \overbrace{\big(PDP^{-1}\big)\big(PDP^{-1}\big)\cdots\big(PDP^{-1}\big)}^{n \text{ times}} \\
&= PD\overbrace{\big(P^{-1}P\big)D}^{n-1 \text{ times}}\cdots P^{-1} \\
&= PD\overbrace{ID}^{n-1 \text{ times}}\cdots P^{-1} \\
&= PD^nP^{-1}
\end{aligned}
$$

Computing a power of a diagonal $d \times d$ matrix requires only d-many floating point multiplications.

$$D^n = \begin{bmatrix} a_1 & 0 & \cdots & 0 \\ 0 & a_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_d \end{bmatrix}^n = \begin{bmatrix} a_1^n & 0 & \cdots & 0 \\ 0 & a_2^n & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_d^n \end{bmatrix}$$

If the decomposition, $M = PDP^{-1}$, is known, then we can reduce the number of floating point multiplications from $2d^3 \log_2 n$ to $2d^3 + d$. This might appear to be a huge savings, but finding P and its inverse are themselves compute intensive. In this chapter, we discuss how to find P (and consequently P^{-1}) and D.

Example 10.1.2 (*Computing a matrix to a power*)

Suppose, $M = \frac{1}{8}\begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix}$. We observe that $M = PDP^{-1}$ where $D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & -5 & 0 \\ 0 & 0 & 8 \end{bmatrix}$, and $P = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix}$

Furthermore, we can compute $P^{-1} = \frac{1}{16}\begin{bmatrix} 2 & 7 & -3 \\ 4 & -2 & -6 \\ 2 & -1 & 5 \end{bmatrix}$.

We now compute M^4? Since

$$D^4 = \begin{bmatrix} 2^4 & 0 & 0 \\ 0 & (-5)^4 & 0 \\ 0 & 0 & 8^4 \end{bmatrix} = \begin{bmatrix} 16 & 0 & 0 \\ 0 & 625 & 0 \\ 0 & 0 & 4096 \end{bmatrix},$$

we have

$$\begin{aligned} M^4 &= \left(PDP^{-1}\right)^4 \\ &= PD^4P^{-1} \\ &= \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} 16 & 0 & 0 \\ 0 & 625 & 0 \\ 0 & 0 & 4096 \end{bmatrix} \frac{1}{16} \begin{bmatrix} 2 & 7 & -3 \\ 4 & -2 & -6 \\ 2 & -1 & 5 \end{bmatrix} \\ &= \frac{1}{8} \begin{bmatrix} 14804 & -7338 & 26946 \\ -1218 & 737 & 1827 \\ 6942 & -3471 & 22355 \end{bmatrix}. \end{aligned}$$

In summary, we computed the 4th power of 3 integers, and we performed two matrix multiplications.

Example 10.1.3 (*Compute exponential of a matrix*)

Compute e^M where $M = \frac{1}{8}\begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix}$ as in Example 10.1.2. Recall $M = PDP^{-1}$ where $P = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix}$, $P^{-1} = \frac{1}{16}\begin{bmatrix} 2 & 7 & -3 \\ 4 & -2 & -6 \\ 2 & -1 & 5 \end{bmatrix}$, and $D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & -5 & 0 \\ 0 & 0 & 8 \end{bmatrix}$.

Starting with Equation (10.1) we have

$$
\begin{aligned}
e^M &= \sum_{k=0}^{\infty} \frac{1}{k!} M^k = \sum_{k=0}^{\infty} \frac{1}{k!} \left(PDP^{-1}\right)^k \\
&= \sum_{k=0}^{\infty} \frac{1}{k!} \left(PD^kP^{-1}\right) = P\left(\sum_{k=0}^{\infty} \frac{1}{k!} D^k\right) P^{-1} \\
&= P\left(\sum_{k=0}^{\infty} \frac{1}{k!} \begin{bmatrix} 2 & 0 & 0 \\ 0 & -5 & 0 \\ 0 & 0 & 8 \end{bmatrix}^k\right) P^{-1} \\
&= P \begin{bmatrix} \sum_{k=0}^{\infty} \frac{2^k}{k!} & 0 & 0 \\ 0 & \sum_{k=0}^{\infty} \frac{(-5)^k}{k!} & 0 \\ 0 & 0 & \sum_{k=0}^{\infty} \frac{8^k}{k!} \end{bmatrix} P^{-1} \\
&= P \begin{bmatrix} e^2 & 0 & 0 \\ 0 & e^{-5} & 0 \\ 0 & 0 & e^8 \end{bmatrix} P^{-1} \\
&= \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} e^2 & 0 & 0 \\ 0 & e^{-5} & 0 \\ 0 & 0 & e^8 \end{bmatrix} \frac{1}{16} \begin{bmatrix} 2 & 7 & -3 \\ 4 & -2 & -6 \\ 2 & -1 & 5 \end{bmatrix} \\
&= \frac{1}{16} \begin{bmatrix} 2e^2 + 2e^{-5} & 7e^2 - 2e^{-5} - 3e^8 & -3e^2 + 2e^8 \\ 4e^2 - 2e^{-5} & 14e^2 + e^{-5} & -6e^2 \\ -2e^{-5} & e^{-5} - 2e^8 & 4e^8 \end{bmatrix}
\end{aligned}
$$

10.2 Matrix of Polynomials

What does it mean to multiply a polynomials by a matrix or a matrix by a polynomial? There are two potential meanings. Suppose

$p(x) = 3x^2 - 1$, and suppose $M = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix}$. We might interpret $p(x) \times M$ as a polynomial with matrix coefficients as in Equation (10.6), or we might interpret it as a matrix with polynomial entries as in Equation (10.8).

First, we view $p(x) \times M$ as an exotic polynomial.

$$\begin{aligned} p(x) \times M &= \left(3x^2 - 1\right) \times \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix} \\ &= 3 \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix} x^2 - \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix} && (10.6) \\ &= \begin{bmatrix} 3 & 6 & 9 \\ 6 & -3 & 0 \\ 0 & -3 & 6 \end{bmatrix} x^2 + \begin{bmatrix} -1 & -2 & -3 \\ -2 & 1 & 0 \\ 0 & 1 & -2 \end{bmatrix} && (10.7) \end{aligned}$$

Next, we view $p(x) \times M$ as an exotic matrix.

$$\begin{aligned} p(x) \times M &= \left(3x^2 - 1\right) \times \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix} \\ &= \begin{bmatrix} 1\left(3x^2 - 1\right) & 2\left(3x^2 - 1\right) & 3\left(3x^2 - 1\right) \\ 2\left(3x^2 - 1\right) & -1\left(3x^2 - 1\right) & 0\left(3x^2 - 1\right) \\ 0\left(3x^2 - 1\right) & -1\left(3x^2 - 1\right) & 2\left(3x^2 - 1\right) \end{bmatrix} && (10.8) \\ &= \begin{bmatrix} 3x^2 - 1 & 6x^2 - 2 & 9x^2 - 3 \\ 6x^2 - 2 & -3x^2 + 1 & 0 \\ 0 & -3x^2 + 1 & 6x^2 - 2 \end{bmatrix} && (10.9) \end{aligned}$$

Semantically (mathematically), the values in Equations (10.7) and (10.9) are the same. If we look closely at Equation (10.7) we can interpret it as a matrix, $\begin{bmatrix} 3 & 6 & 9 \\ 6 & -3 & 0 \\ 0 & -3 & 6 \end{bmatrix}$, times a scalar x^2, plus a matrix, $\begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \\ 0 & -1 & 2 \end{bmatrix}$. If we compute the scaled matrix and then perform the matrix addition, then we obtain exactly the matrix in Equation (10.9).

Let's consider how we would represent these matrices and polynomials in Python. Equation (10.7) is an object of type `Polynomial` whose coefficients are of type `SqMatrix` whose entries are of type `int`. You might interpret this type as `Polynomial[SqMatrix[int]]`. However, Equation (10.9) is an object of type `SqMatrix` whose entries are of type `Polynomial` whose coefficients are in turn of type `int`. You might interpret this type as `SqMatrix[Polynomial[int]]`. Syntactically the types `Polynomial[SqMatrix[int]]` and `SqMatrix[Polynomial[int]]` are very different; their respective structures are different. However, semantically and mathematically the types are the same. *I.e.*, there is an *isomorphism* from one to the other. In computer science, such an isomorphism is called a `functor`.

How does this disparity between `Polynomial[SqMatrix[int]]` and `SqMatrix[Polynomial[int]]` effect you, the programmer? It is important because the `Polynomial` class has a method `def __mul__(self:Polynomial, other):` which needs to know how to multiply by an `other` of type `Matrix`. Likewise, the `Matrix` class has a method `def __mul__(self:Matrix, other):` which needs to know how to multiply by an `other` of type `Polynomial`. Do you want $p * M$ and $M * p$ to return objects which are equal to each other, or are you happy that they return objects which are mathematically equal but

which the programming language considers unrelated?

```
class Polynomial:
  ...
  def __mul__(self, other):
    if isinstance(other, Matrix):
       ...

class Matrix:
  ...
  def __mul__(self, other):
    if isinstance(other, Polynomial):
       ...
```

10.3 Defining Eigenvalues

Recall that we would like to express a square matrix, M as a decomposition such as $M = PDP^{-1}$, where D is a (square) diagonal matrix. In this section we address how to compute the diagonal elements of D given matrix M. In Section 10.5 we will consider the problem of finding P.

Definition 10.3.1 (*Eigenvalues*)

If M is a $d \times d$ matrix and $v \neq 0$ is a d-vector, then λ is called an *eigenvalue* of M if $Mv = \lambda v$. In such a case v is called an *eigenvector* corresponding to λ.

A matrix M has up to d-many eigenvalues which may be zero, real ($\in \mathbb{R}$) or complex ($\in \mathbb{C}$). In the case that the matrix has fewer than d eigenvalues then there are multiple, linearly independent vectors, $v_1, v_2, \ldots, v_m$ such that $Mv_1 = \lambda v_1$, $Mv_2 = \lambda v_2$, $\ldots$, $Mv_m = \lambda v_m$. In such a case we say the eigenvalue λ has multiplicity m. The sum of the multiplicities of all the eigenvalues is always d (the dimension of M).

Computationally repeated, zero-valued, and complex-valued eigenvalues are problematic; thus we will concentrate on situations where

the eigenvalues are real, non-zero, and distinct.

10.4 Computing Eigenvalues

To compute the eigenvalues, we will find the roots of the so-called *characteristic polynomial*.

Recall that if λ is the eigenvalue corresponding to the eigenvector v, then by definition, $Mv = \lambda v$. This equation is equivalent to:

$$\begin{aligned} 0 &= Mv - \lambda v \\ &= Mv - \lambda I v \\ &= (M - \lambda I)v \end{aligned}$$

We know that $(M - \lambda I)v = 0$ has only the zero solution when $\det(M - \lambda I) \neq 0$. *I.e.*, if $\det(M - \lambda I) \neq 0$ then $(M - \lambda I)v = 0 \implies v = 0$.

Thus the only way to have non-zero solutions of $(M - \lambda I)v = 0$, it must be true that $\det(M - \lambda I) = 0$. Thus to compute the eigenvalues, we compute the determinant of $(M - \lambda I)$ and ask when is it zero.

Example 10.4.1 (*Computing the Characteristic Equation*)

Let's consider the matrix:

$$M = \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix}$$

$$\begin{aligned} \det(M - \lambda I) &= \det\left(\begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \\ &= \det\left(\begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix} - \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \right) \\ &= \det\left(\begin{bmatrix} 6-\lambda & 5 & 87 \\ 14 & 9-\lambda & -21 \\ 26 & -13 & 25-\lambda \end{bmatrix} \right) \end{aligned}$$

We can use the Laplacian expansion (from Section 4.4) to compute the determinant.

$$\begin{aligned} 0 &= (6-\lambda) \underbrace{\begin{vmatrix} 9-\lambda & -21 \\ -13 & 25-\lambda \end{vmatrix}}_{=d_1} - 5 \underbrace{\begin{vmatrix} 14 & -21 \\ 26 & 25-\lambda \end{vmatrix}}_{=d_2} + 87 \underbrace{\begin{vmatrix} 14 & 9-\lambda \\ 26 & -13 \end{vmatrix}}_{=d_3} \\ &= (6-\lambda)d_1 - 5d_2 + 87d_3 \end{aligned}$$

$$
\begin{aligned}
d_1 &= (9-\lambda)(25-\lambda) - (-13)(-21) \\
&= -48 - 34\lambda + 1\lambda^2 \\
(6-\lambda)d_1 &= (6-\lambda)\big(-48 - 34\lambda + 1\lambda^2\big) \\
&= -288 - 156\lambda + 40\lambda^2 - 1\lambda^3 \\
d_2 &= (14)(25-\lambda) - (26)(-21) \\
&= 896 - 14\lambda \\
-5d_2 &= -5(896 - 14\lambda) \\
&= -4480 + 70\lambda \\
d_3 &= (14)(-13) - (26)(9-\lambda) \\
&= -416 + 26\lambda \\
87d_3 &= 87(-416 + 28\lambda) \\
&= -36192 + 2262\lambda
\end{aligned}
$$

We sum the three intermediate terms.

$$
\begin{aligned}
0 &= (-288 - 156\lambda + 40\lambda^2 - 1\lambda^3) \\
&\quad + (-4480 + 70\lambda) \\
&\quad + (-36192 + 2262\lambda) \\
&= -40960 + 2176\lambda + 40\lambda^2 - 1\lambda^3
\end{aligned} \tag{10.10}
$$

Finally, we have a cubic polynomial in λ.

In the example we have derived the characteristic polynomial, Equation (10.10). This cubic polynomial has three roots, which are $\{\lambda_1, \lambda_2, \lambda_3\} = \{-40, 16, 64\}$.

10.5 Computing Eigenvectors

Suppose M is a $d \times d$ matrix with eigenvalues $\{\lambda_1, \lambda_2, \cdots, \lambda_d\}$.

10.5.1 Computing Eigenvectors Explicitly

If a $d \times d$ matrix, A, has distinct, non-zero, eigenvalues $\{\lambda_1, \lambda_2, \ldots, \lambda_d\}$, we can compute the k-th eigenvector, corresponding to λ_k as follows. First we multiply together (in any order) all the matrices $A - \lambda_i I$ except $A - \lambda_k I$.

$$\mathfrak{P}_k = \prod_{\substack{1 \le k \le d \\ i \ne k}} (M - \lambda_i I) \tag{10.11}$$

This matrix will have a determinant of zero, and sometimes the matrix will have a column of zeros such as,

$$\begin{bmatrix} 1 & 2 & 0 & -1 \\ 2 & 4 & 0 & -2 \\ -1 & -2 & 0 & 1 \\ -1 & -2 & 0 & 1 \end{bmatrix}.$$

In other cases the matrix will have repeated rows or repeated columns or a row of zeros.

To find the k-th eigenvector, we simply choose one of the non-zero columns. By convention, we will agree to choose a non-zero column vector with the minimum norm. In the case shown here two vectors minimize the norm: $\begin{bmatrix} 1 \\ 2 \\ -1 \\ -1 \end{bmatrix}$ and $\begin{bmatrix} -1 \\ -2 \\ 1 \\ 1 \end{bmatrix}$. We can choose either of these.

Example 10.5.1 (*Computing Eigenvectors*)

Recall in Example 10.4.1, we computed the eigenvalues of

$$M = \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix}$$

to be

$$\begin{aligned} \lambda_1 &= -40 \\ \lambda_2 &= 16 \\ \lambda_3 &= 64 \end{aligned}$$

Let's compute the eigenvectors. We need to compute $M - \lambda_i$ for $i = 1, 2, 3$.

$$M_1 = M - \lambda_1 I = M + 40I = \begin{bmatrix} 46 & 5 & 87 \\ 14 & 49 & -21 \\ 26 & -13 & 65 \end{bmatrix}$$

$$M_2 = M - \lambda_2 I = M - 16I = \begin{bmatrix} -10 & 5 & 87 \\ 14 & -7 & -21 \\ 26 & -13 & 9 \end{bmatrix}$$

$$M_3 = M - \lambda_3 I = M - 64I = \begin{bmatrix} -58 & 5 & 87 \\ 14 & -55 & -21 \\ 26 & -13 & -39 \end{bmatrix}$$

Next we compute three products.

$$\begin{aligned}
\mathfrak{P}_1 &= M_2 M_3 \\
&= (M - \lambda_2 I)(M - \lambda_3 I) \\
&= \begin{bmatrix} -10 & 5 & 87 \\ 14 & -7 & -21 \\ 26 & -13 & 9 \end{bmatrix} \begin{bmatrix} -58 & 5 & 87 \\ 14 & -55 & -21 \\ 26 & -13 & -39 \end{bmatrix} \\
&= \begin{bmatrix} 2912 & -1456 & -4368 \\ -1456 & 728 & 2184 \\ -1456 & 728 & 2184 \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\mathfrak{P}_2 &= M_1 M_3 \\
&= (M - \lambda_1 I)(M - \lambda_3 I) \\
&= \begin{bmatrix} 46 & 5 & 87 \\ 14 & 49 & -21 \\ 26 & -13 & 65 \end{bmatrix} \begin{bmatrix} -58 & 5 & 87 \\ 14 & -55 & -21 \\ 26 & -13 & -39 \end{bmatrix} \\
&= \begin{bmatrix} -336 & -1176 & 504 \\ -672 & -2352 & 1008 \\ 0 & 0 & 0 \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\mathfrak{P}_3 &= M_1 M_2 \\
&= (M - \lambda_1 I)(M - \lambda_2 I) \\
&= \begin{bmatrix} 46 & 5 & 87 \\ 14 & 49 & -21 \\ 26 & -13 & 65 \end{bmatrix} \begin{bmatrix} -10 & 5 & 87 \\ 14 & -7 & -21 \\ 26 & -13 & 9 \end{bmatrix} \\
&= \begin{bmatrix} 1872 & -936 & 4680 \\ 0 & 0 & 0 \\ 1248 & -624 & 3120 \end{bmatrix}
\end{aligned}$$

Now to find an eigen vector associated with $\lambda_1 = -40$, we select any non-zero column of $\mathfrak{P}_1$. For $\lambda_2 = 16$, we select a column from $\mathfrak{P}_2$. For $\lambda_3 = 64$, we select a column from $\mathfrak{P}_3$.

For simplicity we choose here the left-most columns of each:

$$v_1 = \begin{bmatrix} 2912 \\ -1456 \\ -1456 \end{bmatrix}, \quad v_2 = \begin{bmatrix} -336 \\ -672 \\ 0 \end{bmatrix}, \quad v_3 = \begin{bmatrix} 1872 \\ 0 \\ 1248 \end{bmatrix}$$

Any non-zero multiple of an eigenvector is also an eigenvector. Thus we are also allowed to scale v_1, v_2 and v_3 (from Example 10.5.1) if desired:

$$\frac{1}{1456} v_1 = \begin{bmatrix} 2 \\ -1 \\ -1 \end{bmatrix}$$

$$\frac{-1}{336} v_2 = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$$

$$\frac{1}{624} v_3 = \begin{bmatrix} 3 \\ 0 \\ 2 \end{bmatrix},$$

or even normalize them:

$$\frac{v_1}{\|v_1\|} \approx \begin{bmatrix} 0.8165 \\ -0.4082 \\ -0.4082 \end{bmatrix} \tag{10.12}$$

$$\frac{v_2}{\|v_2\|} \approx \begin{bmatrix} 0.4472 \\ 0.8944 \\ 0 \end{bmatrix} \tag{10.13}$$

$$\frac{v_3}{\|v_3\|} \approx \begin{bmatrix} 0.8321 \\ 0 \\ 0.5547 \end{bmatrix} \tag{10.14}$$

10.5.2 Eigenvectors as Fixed-Point

There are many algorithms for finding (or approximating) eigenvectors for M. One method is to take any non-zero vector, w_0, and compute the sequence:

$$\begin{aligned} w_1 &= \frac{Mw_0}{\|Mw_0\|} \\ w_2 &= \frac{Mw_1}{\|Mw_1\|} \\ w_3 &= \frac{Mw_2}{\|Mw_2\|} \\ &\vdots \quad \vdots \\ w_n &= \frac{Mw_{n-1}}{\|Mw_{n-1}\|} \end{aligned}$$

This sequence converges to the eigenvector closest to w_0. Unfortunately, the sequence sometimes diverges, or converges to the same eigenvector for different initial choice of w_0.

If we try to use this method to find d-many eigenvalues of a $d \times d$ matrix, then we have to be lucky in choosing good starting vectors. However, as we've seen, the matrix might not have d-many distinct, real, non-zero eigenvalues, so we'd be searching in vain.

Example 10.5.2 (*Eigenvector by iteration*)

Take $M = \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix}$, and chose $w_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$. The sequence proceeds as follows. First we compute w_1.

$$w_1 = \frac{Mw_0}{\|Mw_0\|}$$

$$Mw_0 \approx \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 6 \\ 14 \\ 26 \end{bmatrix}$$

$$\|Mw_0\| = \sqrt{6^2 + 14^2 + 26^2} = \sqrt{908}$$

$$w_1 = \frac{Mw_0}{\|Mw_0\|} = \frac{1}{\sqrt{908}} \begin{bmatrix} 6 \\ 14 \\ 26 \end{bmatrix} \approx \begin{bmatrix} 0.1991 \\ 0.4646 \\ 0.8628 \end{bmatrix}$$

Next, we compute w_2.

$$w_2 = \frac{Mw_1}{\|Mw_1\|}$$

$$Mw_1 \approx \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix} \begin{bmatrix} 0.1991 \\ 0.4646 \\ 0.8628 \end{bmatrix} \approx \begin{bmatrix} 78.5848 \\ -11.1505 \\ 20.7081 \end{bmatrix}$$

$$\|Mw_1\| = 82.0289$$

$$w_2 = \frac{Mw_1}{\|Mw_1\|} \approx \frac{1}{82.0289} \begin{bmatrix} 78.5848 \\ -11.1505 \\ 20.7081 \end{bmatrix} \approx \begin{bmatrix} 0.9580 \\ -0.1359 \\ 0.2524 \end{bmatrix}$$

If we continue this iteration we have $w_5 \approx \begin{bmatrix} 0.7658 \\ 0.0563 \\ 0.6405 \end{bmatrix}$, $w_{10} \approx \begin{bmatrix} 0.8374 \\ -0.0050 \\ 0.5466 \end{bmatrix}$, $w_{15} \approx \begin{bmatrix} 0.8315 \\ 0.0005 \\ 0.5555 \end{bmatrix}$, and finally $w_{20} \approx \begin{bmatrix} 0.8320 \\ 0 \\ 0.5546 \end{bmatrix}$.

Compare the value of w_{20} with Equations 10.12 through 10.14 in Section 10.5.1.

Example 10.5.2 illustrates how to find an approximation for one of the eigenvectors, but how can we approximate the eigenvalue associated with the eigenvector? Since we know $Mv = \lambda v$, thus $\|Mv\| = \|\lambda v\| = |\lambda| \, \|v\|$. We can solve for $|\lambda| = \frac{\|Mv\|}{\|v\|}$. If we compute this ratio for $v = w_{20}$.

Example 10.5.3 (*Magnitude of an eigenvalue*)

$$|\lambda| = \frac{\|Mw_{20}\|}{\|w_{20}\|}$$
$$\approx \frac{\left\| \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix} \begin{bmatrix} 0.8320 \\ 0 \\ 0.5546 \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0.8320 \\ 0 \\ 0.5546 \end{bmatrix} \right\|} \approx 64.0084$$

In Example 10.5.3 we found $|\lambda| = 64$, which means $\lambda = 64$ or $\lambda = -64$, but how can we know whether λ is positive or negative? To decide whether $\lambda = 64$ or $\lambda = -64$ we can look at the angle between Mv and v.

Example 10.5.4 (*Sign of eigenvalue*)

$$\cos\theta = \frac{\langle v, Mv \rangle}{\|v\|\|Mv\|} \approx \frac{\left\langle \begin{bmatrix} 0.8320 \\ 0 \\ 0.5546 \end{bmatrix}, \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix} \begin{bmatrix} 0.8320 \\ 0 \\ 0.5546 \end{bmatrix} \right\rangle}{\|v\|\|Mv\|}$$
$$\approx \frac{\left\langle \begin{bmatrix} 0.8320 \\ 0 \\ 0.5546 \end{bmatrix}, \begin{bmatrix} 53.2614 \\ -0.0029 \\ 35.5008 \end{bmatrix} \right\rangle}{\left\| \begin{bmatrix} 0.8320 \\ 0 \\ 0.5546 \end{bmatrix} \right\| \left\| \begin{bmatrix} 53.2614 \\ -0.0029 \\ 35.5008 \end{bmatrix} \right\|} \approx \frac{64.0085}{64.0085} = 1$$

$$\theta = 0^\circ$$

Since the angle is $\theta = 0^\circ$, then we take $\lambda = 64.0084$. If the

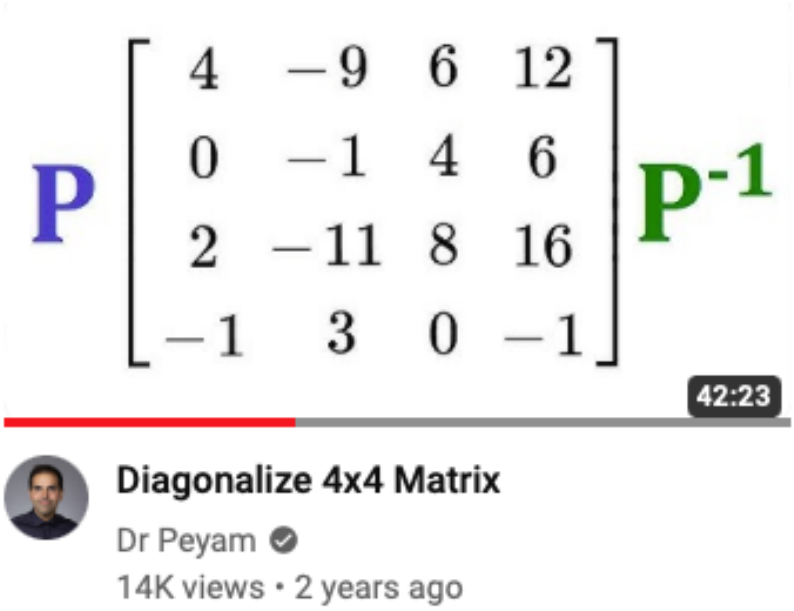


Figure 10.1: Dr Peyam diagonalizes a 4×4 matrix in 42 minutes

angle had been 180° then we'd take $\lambda = -64.0084$.

10.6 Diagonalization

In this section we will combine Sections 10.4 and 10.5.

We can diagonalize a 2×2 matrix or a 3×3, but for larger sizes, diagonalizing a matrix is time consuming and error prone. For example, in the YouTube video
Dr Peyam, `https://www.youtube.com/@drpeyam`, diagonalizes a 4×4 matrix by hand. He needs 42 minutes to finish.

Given a square matrix M, we'd like to find a diagonal matrix D and an invertible matrix P such that

$$M = PDP^{-1}.$$

Sometimes such a *decomposition* is possible, and sometimes it is not. In this section we will concentrate on cases where it is possible. In particular we will focus on cases where the eigenvalues are distinct, non-zero, and real (as opposed to complex).

We would like to construct a method named `diagonalize` in the `SqMatrix` class.

```
def diagonalize(self, epsilon):
  ...
  return p, d, p_inv
```

To compute the matrix, D, we simply create a diagonal matrix consisting of the eigenvalues.

Example 10.6.1 (*Diagonalization*)

Recall in Example 10.4.1, we computed the eigenvalues of

$$M = \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix}$$

to be $\lambda_1 = -40$, $\lambda_2 = 16$, and $\lambda_3 = 64$. With these values, we can construct:

$$D = \begin{bmatrix} -40 & 0 & 0 \\ 0 & 16 & 0 \\ 0 & 0 & 64 \end{bmatrix}$$

Now we wish to compute the matrix P for which $M = PDP^{-1}$. To construct P we simply take the eigenvectors we computed in Example 10.5.1, and copy them into the columns of P.

$$P = \begin{bmatrix} 2 & 1 & 3 \\ -1 & 2 & 0 \\ -1 & 0 & 2 \end{bmatrix}$$

Be careful when constructing P, that the order of the eigenvectors must correspond exactly to the order of the eigenvalues

in D.

We may verify the diagonalization from Example 10.6.1.

Example 10.6.2 (*Verify Diagonalization*)

We compute the determinant of P to be 16, and the inverse of P to be

$$P^{-1} = \frac{1}{16}\begin{bmatrix} 4 & -2 & -6 \\ 2 & 7 & -3 \\ 2 & -1 & 5 \end{bmatrix}$$

Now we can compute:

$$\begin{aligned} PDP^{-1} &= \begin{bmatrix} 2 & 1 & 3 \\ -1 & 2 & 0 \\ -1 & 0 & 2 \end{bmatrix}\begin{bmatrix} -40 & 0 & 0 \\ 0 & 16 & 0 \\ 0 & 0 & 64 \end{bmatrix}\frac{1}{16}\begin{bmatrix} 4 & -2 & -6 \\ 2 & 7 & -3 \\ 2 & -1 & 5 \end{bmatrix} \\ &= \begin{bmatrix} 6 & 5 & 87 \\ 14 & 9 & -21 \\ 26 & -13 & 25 \end{bmatrix} = M \end{aligned}$$

10.7 Application of Eigenvalues

A graph is a set of connections between vertices.

We can represent a graph as a matrix.

If the graph has undirected edges, then the matrix is symmetric.

If the graph if disconnected, then there is a zero eigenvalue.

10.8 Programming Exercises

10.8.1 Classwork

In the first semester Algebra-I course, you implemented functions to manipulate polynomials in a non-object-oriented way. You will find in the `algebra/` directory this Algebra-I code refactored and wrapped in a class named `Polynomial`. You should familiarize yourself with this code, but you don't need to modify it. You may use it as-is. The `Polynomial` class has overridden the arithmetic operators allowing you to add two polynomials using `u + v`, multiply two polynomials using `u * v`, and scale a polynomial by a number using `u * s` (`u` and `v` are polynomials, and `s` is a number.

Now suppose `p` is a `Polynomial` and `m` is a `Matrix` (or `SqMatrix`), what should `p * m` and `m * p` do?

There is code in `Polynomial.py` which interprets `p * m` as returning a new `Matrix` by multiplying each entry in the matrix by the `Polynomial`, `p`. However, the code does not make any assumptions about the type of the contents of the matrix, rather such element-wise multiplication is done using `self.scale(other[r][c])` where `self` is the `Polynomial` and `other` is a `Matrix`.

```
def __mul__(self, other):
  from matrix.Matrix import Matrix
  if isinstance(other, int) or isinstance(other, float):
    return self.scale(other)
  if isinstance(other, Polynomial):
    return Polynomial(poly_mult(self.coefs, other.coefs))
  if isinstance(other, Matrix):
    return Matrix.tabulate(other.rows, other.cols,
                           lambda r, c: self.scale(other[r][c]))
  else:
    raise RuntimeError(f"cannot multiply {self} by {other}")
```

10.8.2 Homework

Work in Progress

The programming exercises for this chapter are not yet complete. Several prerequisites are missing or need redesign before students can finish this chapter:

- `Polynomial.py` has not yet been provided. Note that the filename must be `Polynomial.py` (capital P) to match the convention used throughout the codebase (`Matrix.py`, `Vector.py`, etc.) and so that `from Polynomial import ...` works on case-sensitive systems (Linux). macOS filesystems are case-insensitive by default, so git may silently treat a rename to `polynomial.py` as a no-op; run `git config core.ignorecase false` in the repository to prevent this.
- `eigenvalues_test.py` has not yet been written.
- Computing the characteristic polynomial via `laplacian_expansion` on a polynomial matrix would work correctly, but Laplacian expansion is $O(n!)$ and impractical for any matrix larger than 3×3.
- The fast alternative—Gauss-Jordan elimination on the polynomial matrix $M - \lambda I$—requires a pivot-selection strategy suited to polynomials (e.g., minimizing degree growth), not the `abs`-based comparison currently hard-coded in `find_pivot_row`. Completing this chapter properly therefore requires refactoring `find_pivot_row` to accept a pluggable comparator, so that the same elimination code can be reused for floats, rationals, and polynomials.

This chapter will be updated in a future version of the course.

- Review `Polynomial.py`
- Implement `SqMatrix.py`
  - The method `laplacian_expansion` was originally implemented assuming the matrix entries were all numbers, `int` or `float`. You must refactor `laplacian_expansion` so that it is able to compute the determinant of a matrix of polynomials, while still working in a backward compatible way.
  - `characteristic_polynomial`
  - `eigenvalues`
  - `eigenvectors`
  - `eigen`
  - `diagonalize`
- Test with:
  - `eigenvalues_test.py`

Appendix A

Gauss-Jordan Elimination

Chapter Objectives

- Understand Gauss-Jordan elimination as a classical alternative to Gaussian elimination with back-substitution.
- Compute the inverse of a matrix using GJE on the augmented matrix $[A \mid I]$.
- Derive the determinant as a byproduct of the GJE process.

Chapter 5 covers the primary algorithm for solving linear systems and computing inverses. This chapter presents *Gauss-Jordan elimination* (GJE), a classical alternate approach that normalizes each pivot to 1 at each step rather than carrying the pivot through subsequent rows. All sections are enrichment.

A.1 Example of GJE

Let's take up at Equation (5.20) where we left off.

Example A.1.1 (*GJE*)

First we transform the matrix to have 1 on the diagonal and zeros above the diagonal in the right-most column.

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 0 & 0 & 2 & 19 \end{array}\right]$$

$$\frac{1}{2}R_3 \to R_3 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 0 & 0 & 1 & \frac{19}{2} \end{array}\right]$$

$$-2R_3 + R_2 \to R_2 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 0 & -14 \\ 0 & 0 & 1 & \frac{19}{2} \end{array}\right]$$

$$-3R_3 + R_1 \to R_1 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 0 & -\frac{47}{2} \\ 0 & 4 & 0 & -14 \\ 0 & 0 & 1 & \frac{19}{2} \end{array}\right]$$

Next, we transform the matrix to have 1 on the diagonal and zeros above the diagonal in column 2—traversing the columns from right to left.

$$\frac{1}{4}R_2 \rightarrow R_2 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 0 & -\frac{47}{2} \\ 0 & 1 & 0 & -\frac{7}{2} \\ 0 & 0 & 1 & \frac{19}{2} \end{array}\right]$$

$$-2R_2 + R_1 \rightarrow R_1 \Rightarrow \left[\begin{array}{ccc|c} 1 & 0 & 0 & -\frac{33}{2} \\ 0 & 1 & 0 & -\frac{7}{2} \\ 0 & 0 & 1 & \frac{19}{2} \end{array}\right]$$

In the case of simple Gaussian elimination we had to perform backward substitution. However, in the case of GJE, we can read the answer out of the right-most column of the matrix. We see that Equations (5.21) and (A.1) agree.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{-33}{2} \\ \frac{-7}{2} \\ \frac{19}{2} \end{bmatrix} \tag{A.1}$$

Programmatically it is easier to combine the two phases of zeroing-out below and above the diagonal into a single pass. First zero out the first column, except for the diagonal element in the first column first row, then normalize the row if necessary. The procedure on the first column is exactly the same as shown in Section 5.6.

Example A.1.2 (*Programmatic GJE*)

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ -1 & 2 & -1 & 0 \\ 2 & 1 & 5 & 11 \end{array}\right]$$

$$R_1 + R_2 \to R_2 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 2 & 1 & 5 & 11 \end{array}\right]$$

$$2R_1 - R_3 \to R_3 \Rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 0 & 3 & 1 & -1 \end{array}\right]$$

Second, we zero out the off-diagonal elements in the 2nd column, traversing the columns from left to right.

$$4R_1 - 2R_2 \to R_1 \Rightarrow \left[\begin{array}{ccc|c} 4 & 0 & 8 & 10 \\ 0 & 4 & 2 & 5 \\ 0 & 3 & 1 & -1 \end{array}\right]$$

$$4R_3 - 3R_2 \to R_3 \Rightarrow \left[\begin{array}{ccc|c} 4 & 0 & 8 & 10 \\ 0 & 4 & 2 & 5 \\ 0 & 0 & -2 & -19 \end{array}\right]$$

Next, we zero out the off-diagonal elements of column 3.

$$2R_1 + 8R_3 \to R_1 \Rightarrow \left[\begin{array}{ccc|c} 8 & 0 & 0 & -132 \\ 0 & 4 & 2 & 5 \\ 0 & 0 & -2 & -19 \end{array}\right]$$

$$R_2 + R_3 \to R_2 \Rightarrow \left[\begin{array}{ccc|c} 8 & 0 & 0 & -132 \\ 0 & 4 & 0 & -14 \\ 0 & 0 & -2 & -19 \end{array}\right]$$

Finally, we can normalize the rows, scaling each row by its leading element.

$$\frac{1}{8}R_1 \to R_1 \Rightarrow \left[\begin{array}{ccc|c} 1 & 0 & 0 & -\frac{33}{2} \\ 0 & 4 & 0 & -14 \\ 0 & 0 & -2 & -19 \end{array}\right] \tag{A.2}$$

$$\frac{1}{4}R_2 \to R_2 \Rightarrow \left[\begin{array}{ccc|c} 1 & 0 & 0 & -\frac{33}{2} \\ 0 & 1 & 0 & -\frac{7}{2} \\ 0 & 0 & -2 & -19 \end{array}\right] \tag{A.3}$$

$$\frac{1}{-2}R_3 \to R_3 \Rightarrow \left[\begin{array}{ccc|c} 1 & 0 & 0 & -\frac{33}{2} \\ 0 & 1 & 0 & -\frac{7}{2} \\ 0 & 0 & 1 & \frac{19}{2} \end{array}\right] \tag{A.4}$$

A.2 Pivoting and Numerical Stability

General pivot strategies (rational entries, polynomial entries, design of `find_pivot_row`, and condition number) are discussed in Section 5.8 of Chapter 5. The following example shows a complete 4×4 GJE computation with pivoting.

Example A.2.1 (*GJE with Pivot*)

We begin with the matrix

$$M = \left[\begin{array}{cccc|c} 2.5 & 2.5 & 2.5 & 0.0 & 1 \\ 2.5 & -5.0 & -5.0 & 2.5 & 2 \\ -7.5 & 2.5 & 2.5 & 0.0 & -1 \\ 1.0 & 2.0 & 1.0 & -1.0 & 0 \end{array}\right]$$

First we begin by swapping row 1 with row 3.

$$R_1 \leftrightarrow R_3 \Rightarrow \left[\begin{array}{cccc|c} -7.5 & 2.5 & 2.5 & 0.0 & -1 \\ 2.5 & -5.0 & -5.0 & 2.5 & 2 \\ 2.5 & 2.5 & 2.5 & 0.0 & 1 \\ 1.0 & 2.0 & 1.0 & -1.0 & 0 \end{array}\right]$$

Now we normalize row 1 by scaling it by $\frac{1}{M_{1,1}} = \frac{1}{-7.5}$.

$$\frac{1}{-7.5}R_1 \rightarrow R_1 \Rightarrow \left[\begin{array}{cccc|c} 1.0 & -0.3333 & -0.3333 & -0.0 & 0.1333 \\ 2.5 & -5.0 & -5.0 & 2.5 & 2 \\ 2.5 & 2.5 & 2.5 & 0.0 & 1 \\ 1.0 & 2.0 & 1.0 & -1.0 & 0 \end{array}\right]$$

Now we can zero-out all the entries in column 1 other than the diagonal element $M_{1,1}$ with the operations:

$$\begin{array}{l} -2.5R_1 + R_2 \rightarrow R_2 \\ -2.5R_1 + R_3 \rightarrow R_3 \Rightarrow \\ -1.0R_1 + R_4 \rightarrow R_4 \end{array} \left[\begin{array}{cccc|c} 1.0 & -0.333 & -0.333 & -0.0 & 0.133 \\ 0.0 & -4.167 & -4.167 & 2.5 & 1.667 \\ 0.0 & 3.333 & 3.333 & 0.0 & 0.667 \\ 0.0 & 2.335 & 1.333 & -1.0 & -0.133 \end{array}\right]$$

Now, we want to convert the second column to $\begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}$. So we look for a pivot element in column 2. The entry with the largest absolute value is -4.1667 which already on the diagonal, so no pivoting is necessary. We can simply scale row 2 by $\frac{1}{-4.1667}$, and then zero out the non-diagonal elements.

$$\frac{1}{-4.1667}R_2 \to R_2 \Rightarrow \left[\begin{array}{cccc|c} 1.0 & -0.333 & -0.333 & -0.0 & 0.133 \\ -0.0 & 1.0 & 1.0 & -0.6 & -0.4 \\ 0.0 & 3.333 & 3.333 & 0.0 & 0.667 \\ 0.0 & 2.335 & 1.333 & -1.0 & -0.133 \end{array}\right]$$

$$\begin{array}{c} 0.333R_2 + R_1 \to R_1 \\ -3.333R_2 + R_3 \to R_3 \Rightarrow \\ -2.335R_2 + R_4 \to R_4 \end{array} \left[\begin{array}{cccc|c} 1.0 & 0.0 & 0.0 & -0.198 & 0.0 \\ -0.0 & 1.0 & 1.0 & -0.6 & -0.4 \\ 0.0 & 0.0 & 0.0 & 1.998 & 2.0 \\ 0.0 & 0.0 & -1.002 & 0.401 & 0.802 \end{array}\right]$$

Next, we want to convert the third column to $\begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \\ 0.0 \end{bmatrix}$. So we again look for a pivot element, which is -1.002. We pivot by swapping $R_3 \leftrightarrow R_4$. Then we can perform row operations to zero out the non-diagonal entries of R_3.

$$R_3 \leftrightarrow R_4 \Rightarrow \left[\begin{array}{cccc|c} 1.0 & 0.0 & 0.0 & -0.198 & 0.0 \\ -0.0 & 1.0 & 1.0 & -0.6 & -0.4 \\ 0.0 & 0.0 & -1.002 & 0.401 & 0.802 \\ 0.0 & 0.0 & 0.0 & 1.998 & 2.0 \end{array}\right]$$

$$\frac{1}{-1.002}R_3 \to R_3 \Rightarrow \left[\begin{array}{cccc|c} 1.0 & 0.0 & 0.0 & -0.198 & 0.0 \\ -0.0 & 1.0 & 1.0 & -0.6 & -0.4 \\ -0.0 & -0.0 & 1.0 & -0.4 & -0.799 \\ 0.0 & 0.0 & 0.0 & 1.998 & 2.0 \end{array}\right]$$

$$\begin{array}{c} 0.0R_3 + R_1 \to R_1 \\ -1.0R_3 + R_2 \to R_2 \Rightarrow \\ 0.0R_3 + R_4 \to R_4 \end{array} \left[\begin{array}{cccc|c} 1.0 & 0.0 & 0.0 & -0.198 & 0.0 \\ 0.0 & 1.0 & 0.0 & -0.196 & 0.399 \\ -0.0 & -0.0 & 1.0 & -0.4 & -0.799 \\ 0.0 & 0.0 & 0.0 & 1.998 & 2.0 \end{array}\right]$$

Finally, we want to convert column 4 to $\begin{bmatrix}0.0\\0.0\\0.0\\1.0\end{bmatrix}$. Since there are no rows below row 4, there is no need to pivot. We can simply normalize and then zero out the entries in column 4 rows 1 through 3.

$$\tfrac{1}{1.998}R_4 \to R_4 \Rightarrow \left[\begin{array}{cccc|c} 1.0 & 0.0 & 0.0 & -0.198 & 0.0 \\ 0.0 & 1.0 & 0.0 & -0.196 & 0.399 \\ -0.0 & -0.0 & 1.0 & -0.4 & -0.799 \\ 0.0 & 0.0 & 0.0 & 1.0 & 1.002 \end{array}\right]$$

$$\begin{array}{c} 0.198R_4 + R_1 \to R_1 \\ 0.196R_4 + R_2 \to R_2 \Rightarrow \\ 0.4R_4 + R_3 \to R_3 \end{array} \left[\begin{array}{cccc|c} 1.0 & 0.0 & 0.0 & 0.0 & 0.204 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.599 \\ 0.0 & 0.0 & 1.0 & 0.0 & -0.398 \\ 0.0 & 0.0 & 0.0 & 1.0 & 1.002 \end{array}\right]$$

Thus we have the solution vector in the right-most column: $\begin{bmatrix}0.204\\0.599\\-0.398\\1.001\end{bmatrix}$.

In Example A.2.1, we know that at each step after zeroing out a column we should have the column vectors, $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, and $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$. However, in the examples above there are some cases of rounding error evidenced by the presence of -0.0. It is reasonable to replace the

column in question with the expected result but only after performing the row operations. *E.g.*, after finishing column 1, replace column 1 explicitly with $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$; after finishing column 2, replace column 2 explicitly with $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ etc. The Python function `replace_col` can be used to do such column replacement.

A.3 GJE in Python

GJE in Python is very similar code to simple Gaussian elimination as seen in Section 5.7. The main difference is that simple Gaussian elimination forces zeros only below the diagonal to obtain row echelon form, GJE forces zero below and also above the diagonal. The method responsible for these zeros is named `make_unit_diagonal`.

As a reminder we start with an $n \times n$ matrix, `a` and an n-vector `b`. We create the augmented matrix using the Python code `m = a.adjoin_col(b)` which looks like the following.

$$\left[\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{array}\right]$$

Whereas simple Gaussian elimination performs row operations to obtain a matrix such as

$$\left[\begin{array}{cccc|c} \boxed{a'_{1,1}} & a'_{1,2} & \cdots & a'_{1,n} & b'_1 \\ 0 & \boxed{a'_{2,2}} & \cdots & a'_{2,n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \boxed{a'_{n,n}} & b'_n \end{array}\right]$$

from which we perform back-substitution. In the case of GJE we, instead, perform row operations to obtain the identity matrix on the left, and a column representing the solution vector on the right.

$$\left[\begin{array}{cccc|c} \boxed{1} & 0 & \cdots & 0 & b'_1 \\ 0 & \boxed{1} & \cdots & 0 & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \boxed{1} & b'_n \end{array}\right]$$

The method `gauss_jordan_elimination` is implemented in the class `SqMatrix`, because the algorithm only works for a given square matrix. However, the `make_unit_diagonal` method works for any matrix, assuming it has at least as many columns as rows.

The `make_unit_diagonal` returns a tuple of two values. In case a solution was found (because A turned out to be invertible), the two values are the augmented matrix with the identity on the left, and the vector on the right which is the solution of $Ax = b$, and the determinant. However, in the case that A was discovered to be singular (no inverse exists), then the tuple `(None, 0)` is returned.

The version of `make_unit_diagonal` is a bit more elaborate than the one we show here. For the moment, let's look at a simpler version of `make_unit_diagonal` which *does not* compute the determinant.

The code below starts by setting `det = 1`, and ends with `return m, det`. However, this code excerpt omits the modifications of `det`. These modifications are explained in Section A.7.

The code for `make_unit_diagonal` iterates k along the diagonal: `for k in range(0, m.rows-1)`. With each iteration of k, the code forces 1 onto the diagonal at row k, column k: `m = m.scale_row(1 / m[k][k], k)`, and forces 0 into every entry in the column except the diagonal. The zeros are accomplished by the loop `for r in range(0, self.rows)` but skips row $r = k$.

```
class Matrix:

  def make_unit_diagonal(m):
    det = 1
    for k in range(0, m.rows - 1):
      pivot = m.find_pivot_row(k)
      if pivot is None:
        return None, 0
      else:
        ...

      m = m.scale_row(1 / m[k][k], k)
      for r in range(0, m.rows):
        if r != k:
          m = ...

    return m, det
```

Now the question is which row operation should be employed to force a zero into row r, column k? Suppose the matrix in question looks like the following: (recall the value at row k, column k is guaranteed to be 1 at this stage. If $k < r$, then the entries to the left of the diagonal are all 0, as seen here.

$$\begin{array}{r} \\ \\ \text{Row k:} \\ \\ \text{Row r:} \\ \\ \end{array} \begin{bmatrix} & & & \textbf{Col } k & & \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & \boxed{1} & \cdots & b'_k \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a'_{r,k} & \cdots & b'_r \\ \vdots & & \vdots & \vdots & & \vdots \end{bmatrix}$$

However, if $r < k$, then the values on row r to the left of the diagonal are all 0 except one of the elements (the element on the diagonal on row r) is 1.

$$\begin{array}{r} \\ \text{Row r:} \\ \\ \text{Row k:} \\ \\ \end{array} \begin{array}{c} \\ \vdots \\ a'_{r,0} \\ \vdots \\ 0 \\ \vdots \end{array} \left[\begin{array}{ccccccc} & & & \textbf{Col } k & & & \\ & & \vdots & \vdots & & & \vdots \\ \cdots & & a'_{r,r-1} & a'_{r,k} & \cdots & & b'_r \\ & & \vdots & \vdots & & & \vdots \\ \cdots & & 0 & \boxed{1} & \cdots & & b'_k \\ & & \vdots & \vdots & & & \vdots \end{array}\right]$$

We must make sure that the row operation does not destroy $a'_{r,0} \cdots a'_{r,r-1}$. The row operation $-a'_{r,k}R_k + R_r \to R_r$ is the correct one. This corresponds to the Python code:

```
m = m.row_operation(-m[r][k], k, 1, r)
```

Notice that if $j < k$ then $a'_{k,j} = 0$, (row k has all 0 element to the left of column k) so the value that gets put into row r, column j will be exactly $a'_{r,j}$ itself.

$$\begin{aligned} a'_{r,j} &\leftarrow (-a'_{r,k})(a'_{k,j}) + a'_{r,j} \\ &= (-a'_{r,k})(0) + a'_{r,j} \\ &= 0 + a'_{r,j} \\ &= a'_{r,j} \end{aligned}$$

A.4 Matrix Inversion

The 2×2 inverse formula is given in Theorem 4.8.1. For larger matrices, the inverse can be computed as a side effect of GJE. To do

this, simply apply the same set of elementary row operations which was made on the coefficients matrix this time to the identity matrix. In Section A.7 we applied the following set of row operations to

$$A = \begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix}$$

1. $R_1 + R_2 \to R_2$
2. $2R_1 - R_3 \to R_3$
3. $-2R_2 + 4R_1 \to R_1$
4. $-3R_2 + 4R_3 \to R_3$
5. $8R_3 + 2R_1 \to R_1$
6. $R_3 + R_2 \to R_2$
7. $\frac{1}{8}R_1 \to R_1$
8. $\frac{1}{4}R_2 \to R_2$
9. $-\frac{1}{2}R_3 \to R_3$

However, if we apply these same operations, in the same order to the identity matrix, $I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, the identity will be transformed into the inverse.

Example A.4.1 (*Inverse Matrix using Row Operations*)

Here is the computation:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{matrix} R_1 + R_2 \to R_2 \\ 2R_1 - R_3 \to R_3 \end{matrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & -1 \end{bmatrix}$$

$$\begin{matrix} -2R_2 + 4R_1 \to R_1 \\ -3R_2 + 4R_3 \to R_3 \end{matrix} \Rightarrow \begin{bmatrix} 2 & -2 & 0 \\ 1 & 1 & 0 \\ 5 & -3 & -4 \end{bmatrix}$$

$$\begin{matrix} 8R_3 + 2R_1 \to R_1 \\ R_3 + R_2 \to R_2 \end{matrix} \Rightarrow \begin{bmatrix} 44 & -28 & -32 \\ 6 & -2 & -4 \\ 5 & -3 & -4 \end{bmatrix}$$

$$\begin{matrix} \frac{1}{8}R_1 \to R_1 \\ \frac{1}{4}R_2 \to R_2 \\ -\frac{1}{2}R_3 \to R_3 \end{matrix} \Rightarrow \begin{bmatrix} \frac{11}{2} & -\frac{7}{2} & -4 \\ \frac{3}{2} & -\frac{1}{2} & -1 \\ -\frac{5}{2} & \frac{3}{2} & 2 \end{bmatrix}$$

We can verify that we have indeed computed the inverse by multiplying to get the identity.

Example A.4.2 (*Verifying the Inverse*)

$$\begin{bmatrix} \frac{11}{2} & -\frac{7}{2} & -4 \\ \frac{3}{2} & -\frac{1}{2} & -1 \\ -\frac{5}{2} & \frac{3}{2} & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix} \times \begin{bmatrix} \frac{11}{2} & -\frac{7}{2} & -4 \\ \frac{3}{2} & -\frac{1}{2} & -1 \\ -\frac{5}{2} & \frac{3}{2} & 2 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I$$

Programmatically it may be easier to perform the row operations on the matrix of coefficients and the identity simultaneously, rather than somehow recording the operations and replaying them later. A common method for doing these operations simultaneously is by constructing an augmented matrix like the following.

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 1 & 0 \\ 2 & 1 & 5 & 0 & 0 & 1 \end{array}\right]$$

If we perform the set of elementary row operations shown above to this 3×6 matrix, the matrix will be transformed into the following.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{11}{2} & -\frac{7}{2} & -4 \\ 0 & 1 & 0 & \frac{3}{2} & -\frac{1}{2} & -1 \\ 0 & 0 & 1 & -\frac{5}{2} & \frac{3}{2} & 2 \end{array}\right]$$

If the matrix is singular, *i.e.*, a square matrix with no inverse, then this method will not yield an inverse. In fact the normalization step, as shown in (A.2), (A.3), and (A.4), will be impossible because one or more rows will contain only zeros.

The following example shows how we can use the GJE matrix inversion technique to compute the inverse of a 2×2 matrix.

Example A.4.3 (*Inverse of* 2×2 *using GJE*)

First let's assume $a \neq 0$, otherwise we must swap the two rows.

$$\left[\begin{array}{cc|cc} a & b & 1 & 0 \\ c & d & 0 & 1 \end{array}\right]$$

$$-cR_1 + aR_2 \to R_2 \Rightarrow \left[\begin{array}{cc|cc} a & b & 1 & 0 \\ 0 & ad-cb & -c & a \end{array}\right]$$

$$-bR_2 + (ad-bc)R_1 \to R_1 \Rightarrow \left[\begin{array}{cc|cc} a(ad-bc) & 0 & ad & -ba \\ 0 & ad-bc & -c & a \end{array}\right]$$

$$\underbrace{\tfrac{1}{a}R_1 \to R_1}_{a \neq 0} \Rightarrow \left[\begin{array}{cc|cc} ad-bc & 0 & d & -b \\ 0 & ad-bc & -c & a \end{array}\right]$$

$$\begin{array}{c} \frac{1}{ad-bc}R_1 \to R_1 \\ \frac{1}{ad-bc}R_2 \to R_2 \end{array} \Rightarrow \left[\begin{array}{cc|cc} 1 & 0 & \frac{d}{ad-bc} & \frac{-b}{ad-bc} \\ 0 & 1 & \frac{-c}{ad-bc} & \frac{a}{ad-bc} \end{array}\right]$$

Thus

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \begin{bmatrix} \frac{d}{ad-bc} & \frac{-b}{ad-bc} \\ \frac{-c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Now let's assume $a = 0$, and $c \neq 0$; thus $ad - bc = -bc$. Since we are assuming the matrix in invertible, we can assume $bc \neq 0$, which means $b \neq 0$.

$$
\begin{array}{rl}
 & \left[\begin{array}{cc|cc} 0 & b & 1 & 0 \\ c & d & 0 & 1 \end{array}\right] \\
R_1 \leftrightarrow R_2 \Rightarrow & \left[\begin{array}{cc|cc} c & d & 0 & 1 \\ 0 & b & 1 & 0 \end{array}\right] \\
-dR_2 + bR_1 \to R_1 \Rightarrow & \left[\begin{array}{cc|cc} bc & 0 & -d & b \\ 0 & b & 1 & 0 \end{array}\right] \\
\underbrace{\begin{array}{c} \frac{1}{bc}R_1 \to R_1 \\ \frac{1}{b}R_2 \to R_2 \end{array}}_{b\neq 0 \ \wedge \ c\neq 0} \Rightarrow & \left[\begin{array}{cc|cc} 1 & 0 & \frac{-d}{bc} & \frac{1}{c} \\ 0 & 1 & \frac{1}{b} & 0 \end{array}\right]
\end{array}
$$

This means

$$
\begin{aligned}
\begin{bmatrix} 0 & b \\ c & d \end{bmatrix}^{-1} &= \begin{bmatrix} \frac{-d}{bc} & \frac{1}{c} \\ \frac{1}{b} & 0 \end{bmatrix} \\
&= \frac{-1}{bc}\begin{bmatrix} d & -b \\ -c & 0 \end{bmatrix} \\
&= \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}
\end{aligned}
$$

A.5 Gauss Jordan Inversion with Pivot

Here we look at an example of the Gauss Jordan Elimination procedure (with pivoting) to compute the inverse.

Example A.5.1 (*GJ* 4×4 *Inversion with Pivot*)

We compute the inverse of

$$M = \begin{bmatrix} 2 & 0 & 5 & 2 \\ -1 & 0 & 1 & -1 \\ 1 & -1 & -1 & 3 \\ -5 & 4 & -4 & -1 \end{bmatrix}$$

First we construct an augmented matrix by *adjoining* M on the left with the 4×4 identity on the right.

$$\left[\begin{array}{rrrr|cccc} 2 & 0 & 5 & 2 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 \\ 1 & -1 & -1 & 3 & 0 & 0 & 1 & 0 \\ -5 & 4 & -4 & -1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Pivot with R_4.

$$R_4 \leftrightarrow R_1 \Rightarrow \left[\begin{array}{rrrr|cccc} -5 & 4 & -4 & -1 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 \\ 1 & -1 & -1 & 3 & 0 & 0 & 1 & 0 \\ 2 & 0 & 5 & 2 & 1 & 0 & 0 & 0 \end{array}\right]$$

Scale and row operations.

$$\begin{array}{c} \frac{1}{5}R_1 \to R_1 \\ R_1 + R_2 \to R_2 \\ -R_1 + R_3 \to R_3 \\ -2R_1 + R_4 \to R_4 \end{array} \Rightarrow \left[\begin{array}{rrrr|cccc} 1 & -0.8 & 0.8 & 0.2 & 0 & 0 & 0 & -0.2 \\ 0 & -0.8 & 1.8 & -0.8 & 0 & 1 & 0 & -0.2 \\ 0 & -3.2 & -1.8 & 2.8 & 0 & 0 & 1 & 0.2 \\ 0 & 1.6 & 3.4 & 1.6 & 1 & 0 & 0 & 0.4 \end{array}\right]$$

Pivot with R_3.

$$R_3 \leftrightarrow R_2 \Rightarrow \left[\begin{array}{cccc|cccc} 1 & -0.8 & 0.8 & 0.2 & 0 & 0 & 0 & -0.2 \\ 0 & -3.2 & -1.8 & 2.8 & 0 & 0 & 1 & 0.2 \\ 0 & -0.8 & 1.8 & -0.8 & 0 & 1 & 0 & -0.2 \\ 0 & 1.6 & 3.4 & 1.6 & 1 & 0 & 0 & 0.4 \end{array}\right]$$

Scale and row operations.

$$\begin{gathered} \tfrac{-1}{3.2} R_2 \to R_2 \\ 0.8R_2 + R_1 \to R_1 \\ 0.8R_2 + R_3 \to R_3 \\ -1.6R_2 + R_4 \to R_4 \end{gathered}$$

$$\Rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 1.25 & -0.5 & 0 & 0 & -0.25 & -0.25 \\ 0 & 1 & 0.5625 & -0.875 & 0 & 0 & -0.3125 & -0.0625 \\ 0 & 0 & 2.25 & -1.5 & 0 & 1 & -0.25 & -0.25 \\ 0 & 0 & 2.5 & 3 & 1 & 0 & 0.5 & 0.5 \end{array}\right]$$

Pivot with R_4.

$$R_3 \leftrightarrow R_4$$

$$\Rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 1.25 & -0.5 & 0 & 0 & -0.25 & -0.25 \\ 0 & 1 & 0.5625 & -0.875 & 0 & 0 & -0.3125 & -0.0625 \\ 0 & 0 & 2.5 & 3 & 1 & 0 & 0.5 & 0.5 \\ 0 & 0 & 2.25 & -1.5 & 0 & 1 & -0.25 & -0.25 \end{array}\right]$$

Scale and row operations.

$$
\begin{aligned}
\tfrac{1}{2.5}R_3 &\to R_3 \\
-1.25R_3 + R_1 &\to R_1 \\
-0.5625R_3 + R_2 &\to R_2 \\
-2.25R_3 + R_4 &\to R_4
\end{aligned}
$$

$$
\Rightarrow \left[\begin{array}{cccc|cccc}
1 & 0 & 0 & -2.0 & -0.5 & 0 & -0.5 & -0.5 \\
0 & 1 & 0 & -1.55 & -0.225 & 0 & -0.425 & -0.175 \\
0 & 0 & 1 & 1.2000000002 & 0.4 & 0 & 0.2 & 0.2 \\
0 & 0 & 0 & -4.2 & -0.9 & 1 & -0.7 & -0.7
\end{array}\right]
$$

For the bottom row, there is no pivot, so scale and row operations.

$$
\begin{aligned}
\tfrac{-1}{4.2}R_4 &\to R_4 \\
2R_4 + R_1 &\to R_1 \\
1.55R_4 + R_2 &\to R_2 \\
-1.2000000002R_4 + R_3 &\to R_3
\end{aligned}
$$

$$
\Rightarrow \left[\begin{array}{cccc|cccc}
1 & 0 & 0 & 0 & -0.071 & -0.476 & -0.166 & -0.166 \\
0 & 1 & 0 & 0 & 0.107 & -0.369 & -0.166 & 0.083 \\
0 & 0 & 1 & 0 & 0.142 & 0.285 & 0 & 0 \\
0 & 0 & 0 & 1 & 0.214 & -0.238 & 0.1666 & 0.1666
\end{array}\right]
$$

Finally, we simply extract the right-hand side of the augmented matrix to find the computed value of M^{-1}.

$$
M^{-1} = \begin{bmatrix}
-0.071 & -0.476 & -0.166 & -0.166 \\
0.107 & -0.369 & -0.166 & 0.083 \\
0.142 & 0.285 & 0 & 0 \\
0.214 & -0.238 & 0.1666 & 0.1666
\end{bmatrix}
$$

A.6 GJE Inversion in Python

The method gauss_jordan_inverse computes the inverse (if possible) and the determinant of the given square matrix. In the current implementation it uses make_diagonal_bareiss internally (see Section ??). The classical GJE implementation below used make_unit_diagonal; it is shown here for reference.

```
class SqMatrix(Matrix):

    # Classical GJE implementation (for reference; now uses Bareiss)
    def exact_inverse(a):
        augmented = a.adjoin_cols(SqMatrix.identity(a.dim))
        diag, det = augmented.make_unit_diagonal()
        if diag is None:
            return None, 0

        return diag.extract_cols(range(a.dim, 2 * a.dim)), det
```

A.7 Determinant

As mentioned earlier GJE can be used to compute the determinant. The idea is detailed by Theorem A.7.1.

Theorem A.7.1 (*Determinant from Row Operations*)

If a sequence of elementary row operations transforms a square matrix, A, to the identity, then the determinant of A is the product of factors, one factor for each row operations performed.

Row operation	Condition	Factor
$R_i \leftrightarrow R_j$	$i \neq j$	-1
$\alpha R_i \to R_i$	$\alpha \neq 0$	$\frac{1}{\alpha}$
$\alpha R_i + \beta R_j \to R_i$	$i \neq j$ and $\alpha \neq 0$	$\frac{1}{\alpha}$

We again take the 3×3 matrix which we used in the previous sections, $A = \begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix}$. We can use the Laplacian expansion to compute that $\det(A) = 2$.

Example A.7.2 (*Determinant by Row Operation*)

In the example in Section A.1, we did not see any cases of swapping row. However, we saw several examples of the other two operations.

Row	Operation	Factor
$R_1 + R_2$	$\to R_2$	1
$2R_1 - R_3$	$\to R_3$	-1
$-2R_2 + 4R_1$	$\to R_1$	$\frac{1}{4}$
$-3R_2 + 4R_3$	$\to R_3$	$\frac{1}{4}$
$8R_3 + 2R_1$	$\to R_1$	$\frac{1}{2}$
$R_3 + R_2$	$\to R_2$	1
$\frac{1}{8}R_1$	$\to R_1$	8
$\frac{1}{4}R_2$	$\to R_2$	4
$-\frac{1}{2}R_3$	$\to R_3$	-2

Multiplying together the factors from the row operations, we arrive at:

$$\begin{aligned} \det(A) &= 1 \times -1 \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{2} \times 1 \times 8 \times 4 \times -2 \\ &= \frac{8 \times 4 \times 2}{4 \times 4 \times 2} \\ &= 2 \end{aligned}$$

Example A.7.3 (*GJ* 3×3 *Inversion with Determinant*)

Recall again from Section 3.4.1 we saw two matrices, $A = \begin{bmatrix} 0 & -3 & -2 \\ 1 & -4 & -2 \\ -3 & 4 & 1 \end{bmatrix}$, and $B = \begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix}$, where $AB = I$. Let's try to show using GJE that $A^{-1} = B$. Let's also compute the determinant of A.

First we construct an augmented matrix by *adjoining* A on the left with the 3×3 identity on the right.

$$\left[\begin{array}{ccc|ccc} 0 & -3 & -2 & 1 & 0 & 0 \\ 1 & -4 & -2 & 0 & 1 & 0 \\ -3 & 4 & 1 & 0 & 0 & 1 \end{array}\right]$$

Pivot row 3 into row 1:

$$R_1 \leftrightarrow R_3 \Rightarrow \left[\begin{array}{ccc|ccc} -3 & 4 & 1 & 0 & 0 & 1 \\ 1 & -4 & -2 & 0 & 1 & 0 \\ 0 & -3 & -2 & 1 & 0 & 0 \end{array}\right]$$

Now, we zero out the off-diagonal elements of column 1.

$$R_1 + 3R_2 \to R_2 \Rightarrow \left[\begin{array}{ccc|ccc} -3 & 4 & 1 & 0 & 0 & 1 \\ 0 & -8 & -5 & 0 & 3 & 1 \\ 0 & -3 & -2 & 1 & 0 & 0 \end{array}\right]$$

Now, we zero out the off-diagonal elements of column 2.

$$\begin{array}{c} 4R_2 + 8R_1 \to R_1 \\ -3R_2 + 8R_3 \to R_3 \end{array} \Rightarrow \left[\begin{array}{ccc|ccc} -24 & 0 & -12 & 0 & 12 & 12 \\ 0 & -8 & -5 & 0 & 3 & 1 \\ 0 & 0 & -1 & 8 & -9 & -3 \end{array}\right]$$

Now, we zero out the off-diagonal elements of column 3.

$$\begin{matrix} -12R_3 + 1R_0 \to R_0 \\ -5R_3 + 1R_2 \to R_2 \end{matrix} \Rightarrow \left[\begin{array}{ccc|ccc} -24 & 0 & 0 & -96 & 120 & 48 \\ 0 & -8 & 0 & -40 & 48 & 16 \\ 0 & 0 & -1 & 8 & -9 & -3 \end{array}\right]$$

At this point in the left-hand part of the augmented matrix we have 0's off the diagonal (above and below) and non-zero elements along the diagonal. We now need to get 1's on the diagonal. So we can scale each row by the reciprocal of these diagonal elements.

$$\begin{matrix} \frac{-1}{24}R_0 \to R_0 \\ \frac{-1}{8}R_1 \to R_1 \\ -1R_2 \to R_2 \end{matrix} \Rightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & -5 & -2 \\ 0 & 1 & 0 & 5 & -6 & -2 \\ 0 & 0 & 1 & -8 & 9 & 3 \end{array}\right]$$

So, we verify that the right-hand half of the augmented matrix contains exactly the content of $B = \begin{bmatrix} 4 & -5 & -2 \\ 5 & -6 & -2 \\ -8 & 9 & 3 \end{bmatrix}$ as predicted.

Now we can compute the determinant of A by looking at which row operations we performed, and accumulating the associated multiplicative factors from Theorem A.7.1.

Example A.7.4 (*GJE* 3×3 *Inversion with Determinant*)

Let's combine the constants associated with the elementary row operations from Example A.7.3

Row Operation	multiplicative factor
$R_1 \leftrightarrow R_3$	-1
$R_1 + 3R_2 \rightarrow R_2$	$\frac{1}{3}$
$4R_2 + 8R_1 \rightarrow R_1$	$\frac{1}{8}$
$-3R_2 + 8R_3 \rightarrow R_3$	$\frac{1}{8}$
$-12R_3 + 1R_0 \rightarrow R_0$	1
$-5R_3 + 1R_2 \rightarrow R_2$	1
$\frac{-1}{24}R_0 \rightarrow R_0$	-24
$\frac{-1}{8}R_1 \rightarrow R_1$	-8
$-1R_2 \rightarrow R_2$	-1

So $\det(A) = (-1)(\frac{1}{3})(\frac{1}{8})(\frac{1}{8})(1)(1)(-24)(-8)(-1) = 1$

We can enhance the code shown in the implementation of `make_unit_diagonal` from Section A.3 to also compute the determinant. To implement this enhancement, we start with an initial value of 1, `det = 1`. We update `det` in three different cases in accordance with Theorem A.7.1.

1. When `swap_rows(i,j)` is called and $i \neq j$, then `det *= -1`. Attention, if a row is swapped with itself, do not execute `det *= -1`.

2. When `scale_row(s,j)`, multiply the determinant by $\frac{1}{s}$: `det *= 1/s`.

3. When `row_operation(t,i,s,j)`, multiply the determinant by $\frac{1}{s}$: `det *= 1/s`.

A.8 Complexity of Inversion by GJE

In general computing the inverse of an $n \times n$ matrix requires $n-1$ row operations of the form $\alpha R_i + \beta R_j \to R_j$ for each column. *I.e.*, $n(n-1)$. Furthermore each such row operation consists of $2n$ scalar multiplications, and n scalar additions. The final normalization step can be optimized, because at this point we are sure that all non-diagonal entries are 0, Thus the scaling only requires n scalar multiplications (if optimized).

Thus there are

$$T(n) = 2n \times n \times (n-1) + n = 2n^3 - 2n^2 + n \tag{A.5}$$

scalar multiplications (worst case), both to compute a matrix inverse, or equivalently to compute the determinant using Gauss-Jordan elimination. Compare this with $T(n) = (e-1)n!$ operations necessary to compute the determinant using the Laplacian expansion algorithm from Section 4.5.

A.9 Solving Linear Equations by Matrix Inversion

In Section 5.3 we discussed a method for solving $Ax = b$ by matrix inversion. However, at that time, we did not have a way to compute the inverse of an arbitrary invertible matrix. Now that we can compute the inverse, provided it exists, we have yet another way to solve a system of linear equations.

Recall once again Equation (5.7).

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & 2 & -1 \\ 2 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 11 \end{bmatrix}$$

If we consider this to be in the form $Ax = b$ where A is an $n \times n$ matrix, and b is a $n-$vector, (equivalently an $n \times 1$ matrix), then we can solve this equation in the case A^{-1} exists.

$$\begin{aligned} Ax &= b \\ A^{-1}b = A^{-1} \times (Ax) &= (A^{-1} \times A)x = Ix = x \end{aligned}$$

Example A.9.1 (*Solution by Matrix Inversion*)

To solve the system of equations, Equation (5.7), we need only multiply b on the left by A^{-1}.

$$x = \begin{bmatrix} \frac{11}{2} & -\frac{7}{2} & -4 \\ \frac{3}{2} & -\frac{1}{2} & -1 \\ -\frac{5}{2} & \frac{3}{2} & 2 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \\ 11 \end{bmatrix} = \begin{bmatrix} -\frac{33}{2} \\ -\frac{7}{2} \\ \frac{19}{2} \end{bmatrix}$$

A.10 Programming Exercises

A.10.1 Homework

- Continue with `Matrix.py`
  - `make_diagonal_bareiss`
- Provided in `SqMatrix.py`

  The methods `gauss_jordan_inverse` and `gauss_jordan_elimination` are provided; they use `make_diagonal_bareiss` internally. Study these methods to understand how to implement the methods they call.
- Test with:

  - gauss_jordan_test.py
  - bareiss_test.py

- Enrichment: implement make_unit_diagonal (classical GJE, Section A.3) and verify with gauss_jordan_test.py.