Contents

1 Linear Algebra Introduction

Most of the math that you have done so far has likely involved rules for working with numbers like x and functions like $f(x)$. These rules are covered in courses like algebra and calculus.

In this course we will learn a new set of rules called Linear Algebra (an "algebra" is just a set of rules). These rules will allow us to work not only with numbers, but with higher-dimensional objects called *vectors* and *matrices*. The applications for vectors and matrices are practically limitless because they are the objects that nearly all programming languages use for storing and manipulating data. And data, as you know, is having a moment right now.

These notes are based heavily on Gilbert Strang's textbook, *Introduction to Linear Algebra*. I took from Strang what I deemed to be the absolute minimum amount of linear algebra required for a data science education, and then reworded, reorganized, and augmented the content to better fit the needs of this course. If you seek more rigorous proofs, more example problems, alternate explanations of concepts, or just more detail in general, I encourage you to consult the source material in Strang.

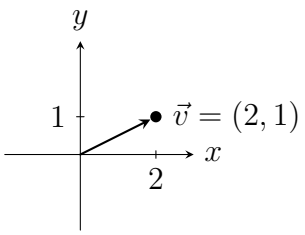
2 Vectors, Matrices, Linear Systems

a Vectors

a.1 Basics

> **(def 2.1) Vector**
>
> A vector is a mathematical object capable of representing something more than a single number can. The numbers that the vector contains are called its *components*.

That definition is very general, but that is because vectors can represent many things. The most common example is a line in the xy plane:



(2.1)

Here, the vector $\vec{v} = (2, 1)$ is the line pointing from the origin $(0, 0)$ to the point $(x = 2, y = 1)$. It has a length and a direction (we'll quantify these later), but for now it is clear that a single number cannot represent both of those concepts. In this course we will denote vectors with an arrow over them, like $\vec{v}$, to differentiate from a scalar value v.

Another example of a vector is a time series of data, with components being the observations from time $t = 1$ to $t = T$:

$$\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_T \end{bmatrix}$$

Even when we write a vector like $\vec{v} = (v_1, v_2, \ldots, v_T)$, we will assume that it is a column rather than a row. The distinction means nothing now but it will be important later.

(ex 2.1) Defining a vector in Python

In Python, we will use the `numpy` package to define vectors. In the code block below, any line beginning with `>>>` indicates a statement that will print something on the line below it.

```
# Initialize from a list
v = np.array([1, 2, 3])
>>> v.shape
(2,)
# Access components with []
>>> v[0]
1
```

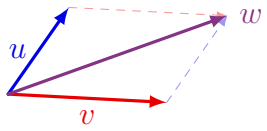
```
>>> v[1:]
array([2, 3])
```

a.2 Vector Operations

One of the most common vector operations is addition. We can add two vectors if they are the same size:

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \end{bmatrix} \tag{2.2}$$

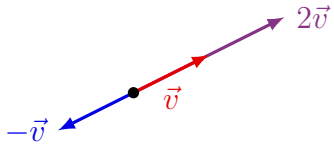
Graphically, you can visualize the sum of two vectors as the vector you get by appending the two vectors end to end. See how $\vec{u} + \vec{v} = \vec{w}$ in the diagram below:



The other thing we often do is scale vectors by a constant factor:

$$c \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} cu_1 \\ cu_2 \end{bmatrix} \tag{2.3}$$

Graphically, this can correspond to stretching, shrinking, or reflecting a vector:



The general term for doing one or both of these at the same time is a *linear combination*:

(def 2.2) Linear combination

Addition and/or scaling of vector(s), e.g., $c\vec{v} + d\vec{w}$, where c and d are arbitrary scalars and $\vec{v}$ and $\vec{w}$ are vectors of the same size.

(ex 2.2) Adding and scaling `numpy` arrays

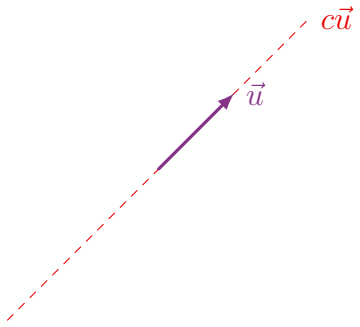
Numpy arrays support elementwise addition and scalar multiplication.

```
u = np.array([10, 20, 30])
v = np.array([0, 1, 2])
>>> u + v
array([10, 21, 32])
>>> 10 * v
array([ 0, 10, 20])
>>> 2 * u + 3 * v
array([20, 43, 66])
```

a.3 Span and Independence

When we talk about linear combinations of vectors, we are most commonly interested in the set of *all possible* linear combinations of a vector or set of vectors.

For example, consider the single vector $\vec{u} = (1, 1)$. What are all of its possible linear combinations? It is only a single vector, so all we can do is scale it up or down by a scalar value $c \in [-\infty, \infty]$ (the symbol $\in$ means "is a member of", so that expression says "c is a member of the numbers in the range $-\infty$ to ∞."). No matter what value we choose for c, we are restricted to producing a resulting vector along the infinite line $c\vec{u}$:

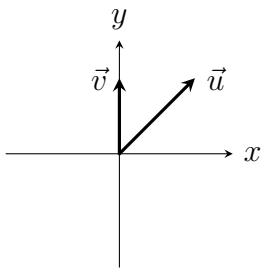


That infinite line is called the *span* of the vector $\vec{u}$.

(def 2.3) Span

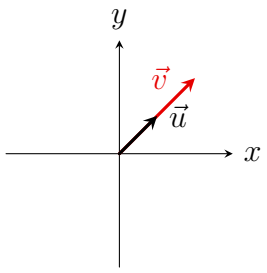
The set of all possible linear combinations of a set of vectors.

What if instead we had two vectors, $\vec{u} = (2, 2)$ and $\vec{v} = (0, 2)$?



In this case, the set of all linear combinations $c\vec{u} + d\vec{v}$, i.e., the span of $\{\vec{u}, \vec{v}\}$, could fill the entire xy plane. In other words, *any* vector of the form (x, y) could be generated by adding together scaled versions of $\vec{u}$ and $\vec{v}$. See if you can find a counterexample (you cannot).

An important question: can we fill the entire xy plane with any two two-component vectors $\vec{u}$ and $\vec{v}$? Let's see another example with $\vec{u} = (1, 1)$ and $\vec{v} = (2, 2)$:



In this case, we *cannot* fill the xy plane with linear combinations of $\vec{u}$ and $\vec{v}$, because $\vec{v}$ is already a linear combination of $\vec{u}$: $\vec{v} = 2\vec{u}$. In other words, $\vec{v}$ is in the span of $\vec{u}$. This means that they are not *independent*. It turns out we need two independent vectors, one for each dimension, in order to fill a two-dimensional space like the xy plane.

(def 2.4) Independence

A set of vectors are linearly independent if no vector in the set can be constructed from a linear combination of the other vectors.

Let's summarize, and ask a question: if the span of one vector $\vec{u}$ is a line, and the span of two independent vectors $\vec{u}$ and $\vec{v}$ is 2d space, what might the span of three independent vectors $\vec{u}, \vec{v}, \vec{w}$ be? If you guessed all of 3d space, you are correct. These independent vectors must have 3 components in order to represent an (x, y, z) coordinate, but otherwise they can be linearly combined the same as a 2d vector. The easiest way to come up with 3 independent vectors is the following, with a 1 in each component and 0 in the rest:

$$\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}$$

But why stop at 3? We can extend this logic all the way up to n dimensions: the span of any n independent vectors is n-dimensional space. This gets harder to visualize, but the beauty of linear algebra is that it allows us to describe these spaces mathematically.

a.4 Vector length and direction

Just as numbers have a magnitude and sign, vectors can be described with a length and direction. In order to define those concepts, we first introduce a new operation called the *dot product*.

(def 2.5) Dot product

For two n-component vectors $\vec{u}$ and $\vec{v}$, the dot product is defined as

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{n} u_i v_i$$

Example: For $\vec{u} = (0, 1)$, $\vec{v} = (3, 5)$...

$$\vec{u} \cdot \vec{v} = 0(3) + 1(5) = 5$$

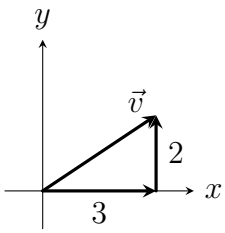


Figure 1: The definition of vector length follows from Pythagorean Theorem.

This allows us to define the *length* of a vector:

(def 2.6) Vector length

The length of a vector $\vec{v}$ is given by $\sqrt{\vec{v} \cdot \vec{v}}$. You will often see this denoted as $||\vec{v}||$.

This definition may seem arbitrary, but it can be understood from the Pythagorean theorem (remember that?). Consider a vector $\vec{v} = (3, 2)$, as shown in Figure 1. $\vec{v}$ is the hypotenuse of a right triangle with side lengths given by its two components, $x = 3$ and $y = 2$. Therefore, the length $||\vec{v}|| = \sqrt{3^2 + 2^2} = \sqrt{3(3) + 2(2)} = \sqrt{\vec{v} \cdot \vec{v}}$.

Let's use this new length definition to define another important concept in linear algebra: *unit vectors*.

(def 2.7) Unit vector

A unit vector is any vector $\vec{u}$ with length $||\vec{u}|| = 1$. An arbitrary vector $\vec{u}$ can become a unit vector $\hat{u}$ if you divide the vector by its length:

$$\hat{u} = \frac{\vec{u}}{||\vec{u}||}$$

So we have vector length, but what about direction? Let's start with the simple example in Figure 2.

The angle between $\vec{u}$ and $\vec{v}$ is 90 degrees. What is their dot product? Let's check:

$$\vec{u} \cdot \vec{v} = 1(0) + 0(1) = 0$$

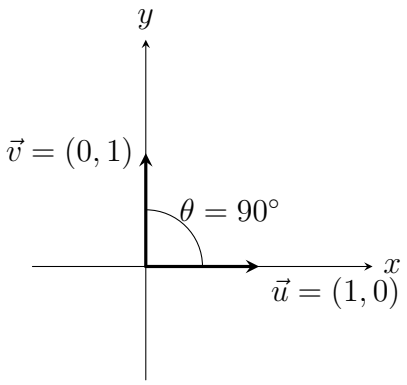


Figure 2: The angle θ between two vectors $\vec{u}$ and $\vec{v}$

It turns out this holds in general: if $\vec{u} \cdot \vec{v} = 0$, then $\vec{u}$ and $\vec{v}$ are perpendicular, and the angle between them is 90 degrees (or $\pi/2$ if you prefer radians). We can generalize this rule and show (see Strang for a proof) that the angle θ between any two vectors is given by the following definition.

(def 2.8) Angle between vectors

The angle θ between any two vectors $\vec{u}$ and $\vec{v}$ is given by

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

Example: Let $\vec{u} = (3, 1)$ and $\vec{v} = (-1, 2)$. What is the angle θ between $\vec{u}$ and $\vec{v}$?

Calculating each of the terms we need...

$$\vec{u} \cdot \vec{v} = 3(-1) + 1(2) = -1$$

$$||\vec{u}|| = \sqrt{3^2 + 1^2} = \sqrt{10}$$
$$||\vec{v}|| = \sqrt{-1^2 + 2^2} = \sqrt{5}$$

Plugging into our equation above results in:

$$\cos\theta = \frac{-1}{\sqrt{50}}$$

And solving for θ:

$$\theta = \cos^{-1}\left(\frac{-1}{\sqrt{50}}\right) = 98.13^\circ$$

And in Python:

```
u = np.array([3, 1])
v = np.array([-1, 2])
# Length is given by the 2-norm:
>>> np.linalg.norm(u)
3.1622776601683795
>>> np.linalg.norm(v)
2.23606797749979
# And dot product between u and v:
>>> np.dot(u, v)
-1
# And the angle:
theta = np.arccos(-1 / np.sqrt(50))
>>> theta * 180 / np.pi
98.130102354156
```

b Matrices

b.1 Basics

Just as numbers can be collected into the higher dimensional vector, we can collect vectors into an even higher dimensional object called a *matrix*.

(def 2.9) Matrix

A collection of numbers with m rows and n columns. For example, an arbitrary matrix given by

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

has $m = 2$ rows and $n = 2$ columns. When $m = n$ the matrix is *square*.

Rather than thinking of a matrix as a big block of numbers, though, it is helpful to instead think of it as a collection of n column vectors, each with m components (we'll sometimes use the shorthand m-vector to describe such a vector). So from the example above, we can think of A as:

$$A = \begin{bmatrix} \vec{a_1} & \vec{a_2} \end{bmatrix}$$

with $\vec{a_1} = (1, 0)$ and $\vec{a_2} = (0, 1)$.

b.2 Matrix size and transpose

This is a good time to introduce notation for specifying the size of a vector or matrix. If a matrix has m rows and n columns, we would call that an $m \times n$ matrix. A column vector could similarly be described as being size $n \times 1$ if it has n components. But a row vector with the exact same components would be size $1 \times n$. The difference matters, as we will see in a bit, because operations between matrices and vectors are only valid when their shapes are compatible.

To enable operations that would otherwise be undefined, there is a common operation available called the *matrix transpose*.

(def 2.10) Matrix Transpose

A matrix A of size $m \times n$ has a *transpose* denoted $A^\top$ of size $n \times m$, which has the row and column indices switched. Mathematically, the elements of $A^\top$ are given by

$$[A^\top]_{ji} = [A]_{ij},$$

where $i = 1 \dots m$ denotes a row index of A and $j = 1 \dots n$ denotes a column index of A. Another way to think about the transpose is that it flips the matrix A over its diagonal elements. A matrix with $A = A^\top$ is called *symmetric*.

There are a few important properties to remember for transposes:

- $(A^\top)^\top = A$
- $(A + B)^\top = A^\top + B^\top$
- $(AB)^\top = B^\top A^\top$

Example 1: Let

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

Then

$$A^\top = \begin{bmatrix} 1 & 3 \\ -2 & 2 \end{bmatrix}$$

Example 2: Let

$$A = \begin{bmatrix} 0 & 5 \\ 5 & 1 \end{bmatrix}$$

In this case, $A = A^\top$, so A is symmetric.

Example 3: Let

$$\vec{v} = \begin{bmatrix} 5 \\ 15 \\ 20 \end{bmatrix}$$

be a 3×1 column vector. Then

$$\vec{v}^\top = \begin{bmatrix} 5 & 15 & 20 \end{bmatrix}$$

is a 1×3 row vector.

Another way you'll see matrices and vectors described is by their size *and* the types of numbers they contain. Most often, they contain real numbers. We would denote an $m \times n$ matrix of real numbers like $A \in \mathbb{R}^{m \times n}$. If it has complex numbers it would be $A \in \mathbb{C}^{m \times n}$, but in this class we will assume that all matrices contain real numbers. By most conventions, an n-vector is usually just denoted $\vec{a} \in \mathbb{R}^n$, with the assumption that it's a column vector and not a row vector. A row vector will always be denoted as a transposed column vector, e.g., $\vec{a}^\top$.

(ex 2.3) Initializing and transposing matrices with `numpy`

Numpy arrays can have an arbitrary number of dimensions. We'll ini-

tialize a 2D array (i.e., a matrix) here and demonstrate some common operations like transpose and slicing.

```
# Can initialize a matrix as a list of lists:
A = np.array([[1, 2, 3],[4, 5, 6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])

>>> A.shape
(2, 3)

# Can transpose with the .T operator:
>>> A.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> A.T.shape
(3, 2)

# Extract rows and columns with standard slicing:
>>> A[0, :]
array([1, 2, 3])
>>> A[:,0]
array([1, 4])
```

b.3 Matrix-matrix addition

If you have two matrices A and B and they are the same size, you can add them together element-wise to obtain a new matrix $C = A + B$. A specific example:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix} \tag{2.4}$$

(ex 2.4) Matrix addition and scalar multiplication in `numpy`

Just like 1D arrays, matrices are added together and scaled elementwise:

```
>>> A = np.array([[1, 2], [3, 4]])
>>> A + A
array([[2, 4],
       [6, 8]])

>>> 2 * A
array([[2, 4],
       [6, 8]])
```

b.4 Matrix-vector multiplication

Addition was easy, but the most common operation we perform with matrices is multiplication. Matrices can be multiplied by scalars, and they can multiply other matrices. But most often a matrix will be found multiplying a vector. It works like this:

Define $A \in \mathbb{R}^{m \times n}$ with columns $\vec{a_i} \in \mathbb{R}^m$:

$$A = \left[\begin{array}{cccc} \vec{a_1} & \vec{a_2} & \cdots & \vec{a_n} \end{array} \right],$$

and let $\vec{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ be an $n \times 1$ column vector (the vector $\vec{x}$ *must* have n elements – otherwise the multiplication is undefined). Then the product $A\vec{x}$ is given by:

$$A\vec{x} = \vec{y} = x_1\vec{a_1} + x_2\vec{a_2} + \cdots + x_n\vec{a_n} \tag{2.5}$$

In other words, the product $A\vec{x}$ is a linear combination of the columns of A, where the scaling factors for the n columns are the n elements of $\vec{x}$. The result of this linear combination is a vector $\vec{y} \in \mathbb{R}^m$, because each of the vectors $\vec{a_i}$ that was combined to produce $\vec{y}$ was an m-vector.

There is another way to view matrix multiplication that is more focused on the rows of A rather than the columns. Let us instead think of A as a

matrix composed of m row vectors $\vec{r}_1, \ldots, \vec{r}_m$, each of size $1 \times n$:

$$A = \begin{bmatrix} \vec{r_1} \\ \vec{r_2} \\ \vdots \\ \vec{r_m} \end{bmatrix},$$

Then the matrix-vector product $A\vec{x}$ is given by the dot product of each row $\vec{r_i}$ with $\vec{x}$:

$$A\vec{x} = \vec{y} = \begin{bmatrix} \vec{r_1} \cdot \vec{x} \\ \vec{r_2} \cdot \vec{x} \\ \vdots \\ \vec{r_m} \cdot \vec{x} \end{bmatrix},$$

You get the same answer either way. We will demonstrate that with a numerical example.

(ex 2.5) Matrix-vector multiplication example

Calculate $A\vec{x}$ for

$$A = \begin{bmatrix} 1 & 3 \\ 5 & 6 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} 1 \\ 7 \end{bmatrix}$$

The column way:

$$A\vec{x} = 1 \begin{bmatrix} 1 \\ 5 \end{bmatrix} + 7 \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 22 \\ 47 \end{bmatrix}$$

The row way:

$$A\vec{x} = \begin{bmatrix} 1(1) + 3(7) \\ 5(1) + 7(6) \end{bmatrix} = \begin{bmatrix} 22 \\ 47 \end{bmatrix}$$

With `numpy`:

```
A = np.array([[1, 3], [5, 6]])
x = np.array([1, 7])
>>> Ax = np.matmul(A, x)
>>> Ax
array([22, 47])
```

b.5 Matrix-matrix multiplication

Matrices multiplying other matrices is common too. Suppose we have two matrices, $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times m}$. Their product $C = AB \in \mathbb{R}^{m \times m}$. You can think of the inner dimensions n as "cancelling," leaving the product with the outer dimensions m from each matrix. This isn't just an outcome though, it's a requirement: if a matrix A multiplies a matrix B, then the number of columns of A must match the number of rows of B.

To see the result of the multiplication more clearly: say B has columns $b_1, \ldots, b_m$, then $C = AB$ is given by

$$C = AB = \begin{bmatrix} Ab_1 Ab_2 & \cdots & Ab_m \end{bmatrix}$$

In other words, columns of the product are given by matrix-vector multiplication between A and the columns of B.

For a more explicit example we can define two 2×2 matrices,

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Then $AB = C \in \mathbb{R}^{2 \times 2}$ is given by

$$C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

(ex 2.6) Matrix multiplication in `numpy`

Numpy makes matrix multiplication easy with `np.matmul`. It will also tell you when you try to multiply two non-compatible matrices: see the example below where we try to multiply a 2×3 matrix by a 2×2 matrix.

```
A = np.array([[1, 2],[3, 4], [5, 6]])
B = np.array([[2, 0], [0, 1]])
>>> A
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
>>> B
array([[2, 0],
       [0, 1]])

>>> np.matmul(A, B)
array([[ 2,  2],
       [ 6,  4],
       [10,  6]])

# Create 2 x 3 matrix C:
C = A.T
np.matmul(C, B)
>>> np.matmul(C, B)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: matmul: Input operand 1 has a
mismatch in its core dimension 0, with
gufunc signature (n?,k),(k,m?)->(n?,m?)
(size 2 is different from 3)
```

b.6 Dot product as matrix multiplication

In section a.4 we defined the dot product between two vectors $\vec{u}$ and $\vec{v}$ as the sum of the product of corresponding elements of the two vectors, resulting in a single scalar value. That is still true, but we can write it another way using our new rules of matrix multiplication. Specifically:

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{n} u_i v_i = \vec{u}^\top \vec{v} \tag{2.6}$$

Check the dimensions to make sure this works: $\vec{u}^\top$ is size $1 \times n$, and $\vec{v}$ is size $n \times 1$. So the vector multiplication results in the n dimensions cancelling, and we are left with a 1×1 vector (which is just a scalar) equal to the dot product. Nothing new here conceptually, just a different way to write a common operation.

b.7 The Identity Matrix

One of the most important matrices in all of linear algebra is called the *Identity Matrix*. It is a square matrix denoted I and it looks like

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & & 1 \end{bmatrix} \tag{2.7}$$

All diagonal elements are 1, and all off-diagonal elements are 0. I is like the number 1 but for linear algebra: let $A \in \mathbb{R}^{m \times n}$, let I_m denote an $m \times m$ identity matrix, and let I_n denote an $n \times n$ identity matrix. We find that:

$$I_m A = A I_n = A \tag{2.8}$$

Thus, multiplication by I returns the original matrix, just like how multiplying a number by 1 returns the original number. If you need to quickly create an identity matrix of size $n \times n$ in Python, you can use the shortcut `I = np.eye(n)`.

b.8 Rules for matrix operations

Matrix operations have some important properties to remember. Specifically:

$$\begin{aligned} A + B &= B + A \\ c(A + B) &= cA + cB \\ AB &\neq BA \\ C(A + B) &= CA + CB \\ (A + B)C &= AC + BC \\ A(BC) &= (AB)C \end{aligned}$$

The most important of these is certainly $AB \neq BA$, because it behaves the most differently from standard (scalar) multiplication where $ab = ba$. There are situations where you'll need to solve a matrix equation by multiplying a matrix from the left, but not from the right, or vice versa.

c Linear Systems and the Matrix Inverse

So far we have just learned a bunch of rules for operating on these new objects, vectors and matrices. Now we will begin to see why they are useful in practice.

c.1 Linear Systems

You have probably been asked to solve systems of linear equations before. Something like,

$$\begin{aligned} x - 2y &= 1 \\ 3x + 2y &= 11 \end{aligned}$$

This is easy enough to solve by hand because there are only two equations with two unknowns. But if you had a million equations with a million unknowns you would probably appreciate a more automated solution. That is exactly what linear algebra provides.

Let's use our newfound knowledge of vectors and matrices to rewrite the equations above. Defining

$$A = \begin{bmatrix} 1 & -2 \\ 3 & 2 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 1 \\ 11 \end{bmatrix},$$

and using what we know about matrix-vector multiplication, the system of linear equations above is simply given by

$$A\vec{x} = \vec{b},$$

where we want to solve for the vector $\vec{x}$, with its components x and y. This works for two equations, it works for three equations, and it works for ten million equations.

So we have an equation, but now we need to solve it. It's tempting to write $\vec{x} = \vec{b}/A$ and call that a solution. In fact, we can *almost* do that, but we need to introduce one more concept first.

c.2 The Matrix Inverse

That concept is the matrix inverse. Rather than $\vec{x} = \vec{b}/A$, we write $\vec{x} = A^{-1}\vec{b}$, where A^{-1} is the *matrix inverse* of A. We pronounce it "A inverse."

(def 2.11) Matrix inverse

A matrix A is *invertible* if there exists a matrix A^{-1} such that

$$A^{-1}A = AA^{-1} = I$$

Some important notes about the matrix inverse, and its relationship to systems of linear equations:

- If A has an inverse, then there is only one solution to $A\vec{x} = \vec{b}$, and that solution is given by $\vec{x} = A^{-1}\vec{b}$.
- If there is a nonzero $\vec{x}$ such that $A\vec{x} = \vec{b} = \vec{0}$, then A cannot have an inverse. We call such a matrix *singular*.
  - Why does this condition mean that A has no inverse? Because the inverse would provide a solution $\vec{x} = A^{-1}\vec{b} = A^{-1}\vec{0}$, and there is no matrix A^{-1} that can produce a nonzero $\vec{x}$ by multiplying the zero vector $\vec{b} = \vec{0}$
  - Another way of saying this is that A must have n independent columns in order to have an inverse. We'll elaborate on this idea in the next section.

There is another way to determine whether or not a matrix A has an inverse, which is that the matrix's *determinant* must be nonzero.

(def 2.12) Determinant and Matrix Inverse

The determinant $\det(A)$ of a 2×2 matrix

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

is given by

$$\det(A) = ad - bc \tag{2.9}$$

For that same matrix A, the inverse A^{-1} is given by

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{2.10}$$

Therefore, if $\det(A) = ad - bc = 0$, then A^{-1} is undefined.

One last property related to matrix inverses and multiplication before we move on:

$$(AB)^{-1} = B^{-1}A^{-1} \tag{2.11}$$

Why? Because

$$AB(B^{-1}A^{-1}) = ABB^{-1}A^{-1} = AIA^{-1} = AA^{-1} = I$$

Let's get back to actually solving that example.

(ex 2.7) Solving a system of linear equations

We have

$$A = \begin{bmatrix} 1 & -2 \\ 3 & 2 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 1 \\ 11 \end{bmatrix},$$

We saw above how to find A^{-1} by hand for a 2×2 matrix, and in a traditional linear algebra class we would also see the 3×3 method, and you would practice that for approximately 2 weeks. But we don't have time for that, so we'll just see how to use Python to solve it. There are two primary ways:

1. Calculate A^{-1} directly, and then multiply $\vec{b}$ by A^{-1} to solve for $\vec{x}$. This is fine, but inefficient, because calculating the inverse for large matrices is computationally expensive.
2. Use `np.linalg.solve(A, b)` to solve the system $A\vec{x} = \vec{b}$ for $\vec{x}$. This uses fancy stuff under the hood to solve the system without directly calculating A^{-1}.

```
import numpy as np
A = np.array([[1, -2], [3, 2]])
b = np.array([1, 11])

# Method 1 (inefficient but clear)
A_inverse = np.linalg.inv(A)
print(A_inverse)
>> [[ 0.25   0.25 ],
    [-0.375  0.125]]

x = np.matmul(A_inverse, b) # matrix mult.
print(x)
>> array([3., 1.])

# Method 2 (more efficient, less clear)
x = np.linalg.solve(A, b)
print(x)
>> array([3., 1.])
```

Therefore, the solution to the system of equations was $\vec{x} = (3, 1)$, or $x = 3, y = 1$. You can verify this manually if you like.

3 Vector Spaces, Dimensionality, and Bases

This section will cover a few more abstract concepts that we need before getting to all the applications.

a Vector Spaces

You can think of a vector space as an infinite collection of vectors. This differs from a matrix, which is a collection of a finite number of vectors. The best way to get a handle on this is with some examples of common vector spaces:

- The vector space $\mathbb{R}$ is the set of all real numbers
- $\mathbb{R}^2$ is the xy-plane

- $\mathbb{R}^n$ is the space of all real n-vectors (i.e., column vectors with n real components)
- $\mathbb{C}^n$ is the space of all complex n-vectors

Rather than give a rigorous mathematical definition of a vector space, we'll stick with these examples and discuss the most important property of a vector space:

(def 3.1) The most important property of a vector space

If an arbitrary vector v is a member of the vector space $\mathbb{S}$, then all linear combinations of v remain in $\mathbb{S}$.

Corollary: A vector space must contain the zero vector because that is one of the possible linear combinations of v.

This leads us to another definition:

(def 3.2) Subspace

A subspace is a vector space within another vector space

Example: A line through the point $(0, 0)$ is a subspace of $\mathbb{R}^2$

In the following sections, we will learn about the most important subspaces in linear algebra, often called the Four Fundamental Subspaces.

a.1 The Column Space

Starting with a matrix $A \in \mathbb{R}^{m \times n}$, we can define a subspace called the *column space*.

(def 3.3) Column Space

Also called the *range* or *image* of a matrix, the column space $\mathcal{C}(A)$ is the set of all possible linear combinations of the columns of A. Equivalently:

- $\mathcal{C}(A)$ is *spanned* by the columns of A, or more colloquially that $\mathcal{C}(A)$ is the span of A.
- $\mathcal{C}(A)$ is the set of all vectors $A\vec{x}$

Example: The column space of the 2×2 identity matrix is all of $\mathbb{R}^2$,

the xy-plane.

Crucially, the column space of A will only fill an entire plane (e.g., the xy-plane in the case of a 2×2 matrix) if the columns of A are independent (refer to Section a.3). Certainly the identity matrix satisfies this requirement; in general though, any matrix that satisfies

$$A\vec{x} = \vec{0} \iff \vec{x} = \vec{0} \tag{3.1}$$

will have a column space $\mathcal{C}(A)$ that fills an entire plane (the symbol $\iff$ means "if and only if"). If $A\vec{x} = \vec{0}$ for some nonzero $\vec{x}$, then the columns of A are not linearly independent (recall discussion in Section c.2).

The column space is an important concept when it comes to solving systems of linear equations, $A\vec{x} = \vec{b}$. Specifically, we can say that $A\vec{x} = \vec{b}$ is solvable if and only if $\vec{b} \in \mathcal{C}(A)$. Put another way: if you cannot create $\vec{b}$ by linearly-combining columns of A, then you cannot find a vector $\vec{x}$ that solves $A\vec{x} = \vec{b}$.

Question: what vector space is $\mathcal{C}(A)$ a subspace of? Well, to start it's a subspace of itself (all vector spaces are), but let's think about it more generally in terms of dimensions: $A \in \mathbb{R}^{m \times n}$, and $\vec{x} \in \mathbb{R}^n$, so the product $A\vec{x} \in \mathbb{R}^m$. Therefore, the column space $\mathcal{C}(A)$, the set of all possible $A\vec{x}$, is a subspace of $\mathbb{R}^m$. Maybe it can fill all of $\mathbb{R}^m$ – it will if the columns of A are independent. But maybe it will only fill part of $\mathbb{R}^m$ – that will happen if the columns of A are not independent.

a.2 The Null Space

Still working with $A \in \mathbb{R}^{m \times n}$, we move to our second subspace, called the *null space*.

(def 3.4) Null Space

Also called the *kernel*, the null space $\mathcal{N}(A)$ is the set of all $\vec{x}$ such that $A\vec{x} = \vec{0}$.

Example 1: The null space of the 2×2 identity matrix contains only the zero vector, because the columns of I_2 are linearly independent. Thus, $I_2\vec{x} = \vec{0} \iff \vec{x} = (0, 0)$.

Example 2: The null space of the matrix

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

is given by the span of the line $\vec{x} = (2, -1)$ (or equivalently, $\vec{x} = (1, -0.5)$, $\vec{x} = (-2, 1)$, etc.)

Asking the same question as we did for $\mathcal{C}(A)$, what space is $\mathcal{N}(A)$ a subspace of? If $\mathcal{N}(A)$ is made up of some $\vec{x} \in \mathbb{R}^n$ that satisfy $A\vec{x} = \vec{0}$, then it follows that $\mathcal{N}(A)$ is a subspace of $\mathbb{R}^n$. It's probably not all of $\mathbb{R}^n$ though, unless A is the zero matrix: in that case, any $\vec{x} \in \mathbb{R}^n$ will satisfy $A\vec{x} = \vec{0}$.

a.3 The Row Space

The third fundamental subspace is the *row space* of a matrix.

(def 3.5) Row space

The row space of $A \in \mathbb{R}^{m \times n}$ is the set of all linear combinations of the rows of A. Equivalently:

- The row space is simply $\mathcal{C}(A^\top)$, the set of all linear combinations of the columns of $A^\top$.
- The rows of A *span* $\mathcal{C}(A^\top)$

Example: The row space of the matrix

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

is given by the span of the line $\vec{x} = (1, 2)$.

Because we decided that the column space $\mathcal{C}(A)$ was a subspace of $\mathbb{R}^m$, it follows that the row space $\mathcal{C}(A^\top)$ is a subspace of $\mathbb{R}^n$.

a.4 The Left Nullspace

The fourth and final of our fundamental subspaces is called the *left nullspace*.

(def 3.6) Left Nullspace

The left nullspace is given by the set of vectors $\vec{x}$ that satisfy $A^\top \vec{x} = \vec{0}$. This is equivalent to $\mathcal{N}(A^\top)$.

Recalling that $A^\top \in \mathbb{R}^{n \times m}$, and looking at the equation $A^\top \vec{x} = \vec{0}$, it is clear that $\vec{x}$ must have m elements for the dimensions to work. Therefore, the left nullspace $\mathcal{N}(A^\top)$ must be a subspace of $\mathbb{R}^m$, because it is a collection of vectors $\vec{x}$ that satisfy $A^\top \vec{x} = \vec{0}$.

b Rank, Basis, and Dimensionality

In both casual conversation and in these notes, it is common to refer to the "dimension" of a space. For example, the xy-plane (also known as $\mathbb{R}^2$) is 2-dimensional. The world that we walk around in ($\mathbb{R}^3$) is 3-dimensional. These statements are hopefully obvious, so perhaps you have never thought about *why* 3D space has 3 dimensions, or what a dimension even means. In this section we will formalize a definition for the dimension of a space. This requires a couple new concepts first.

b.1 Rank

One of the most important concepts in all of linear algebra is the matrix *rank*.

(def 3.7) Rank

For $A \in \mathbb{R}^{m \times n}$, the matrix rank, denoted r, $\mathbf{rk}(A)$, or $\mathbf{rank}(A)$, is the number of independent columns of A. This is also equal to the number of independent rows of A.

Example 1: Consider the matrix

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

Row 2 is simply 2(Row 1), and Columns 1 and 3 are both scalar multiples of Column 2. So there is one independent row, and one independent column, therefore the rank $r = 1$.

Example 2: Consider the matrix

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 5 \end{bmatrix}$$

Columns 1 and 2 are independent. Rows 1 and 2 are independent too, but Row 3 can be constructed from Row 1 + 2(Row 2). So the rank of this matrix is $r = 2$.

Some facts and terminology related to rank, for $A \in \mathbb{R}^{m \times n}$:

- If $r = n$, we say that A has *full column rank*. This means that none of the columns are linear combinations of the other columns. See Example 2 above.
  - This is equivalent to saying that $A\vec{x} = \vec{0}$ can only be solved by $\vec{x} = \vec{0}$, i.e., the nullspace $\mathcal{N}(A)$ contains only $\vec{0}$.
- Any matrix with $r < n$ is *rank deficient*. See Example 1 above.
- Any matrix with $n > m$ (more columns than rows) will be rank deficient. See Example 1 above.
- A matrix with $n \leq m$ could be either full rank or not.

The rank of a matrix has important consequences for solving systems of linear equations, $A\vec{x} = \vec{b}$. There are four cases to consider:

1. $r = m = n$: The matrix A is square and full-rank. The system of linear equations has r equations, r unknowns, and 1 solution.
2. $r = m$, $r < n$: The matrix A has full row rank, but not full column rank. This means there are more unknowns than equations. We call this an *underdetermined* system, and it has infinitely many solutions.
3. $r < m$, $r = n$: The matrix A has full column rank, but not full row rank. There are more equations than unknowns. This is an *overdetermined* system, and it will have either no solution or one solution depending on whether $\vec{b} \in \mathcal{C}(A)$ or not.
4. $r < m$, $r < n$: The matrix A is rank deficient. $A\vec{x} = \vec{b}$ will have either no solution or infinitely many solutions.

b.2 Basis

(def 3.8) Basis

For a vector space $\mathcal{S}$, a basis for the space is a sequence of vectors that (1) are linearly independent, and (2) span the entire space.

Example: Consider 3D space, $\mathbb{R}^3$.

The vectors

$$\vec{u} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \vec{v} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \vec{w} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

form a basis because they are linearly independent and span all of $\mathbb{R}^3$: you can form any 3-component vector through linear combinations of $\vec{u}$, $\vec{v}$, $\vec{w}$.

On the other hand, the vectors

$$\vec{u} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \vec{v} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

do not form a basis. They are linearly independent, but they do not span $\mathbb{R}^3$ because you have no way to form a vector with a nonzero third component.

Finally, the vectors

$$\vec{u} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \vec{v} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \vec{w} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$$

do not form a basis either. They do span all of $\mathbb{R}^3$, but $\vec{x}$ is not independent from $\vec{w}$.

In other words, a basis has *just enough* independent vectors to span a space. Not too many, and not too few. Some other important notes on bases:

- Any vector v in a vector space is a *unique* combination of the basis vectors for that space.

  - This is probably the most important thing to remember about bases, so remember it.

- However, a vector space can have infinitely many valid bases. Depending on which basis you choose, the arbitrary vector v will be constructed from a slightly different linear combination of the basis vectors.

How does all this apply to our favorite matrix $A \in \mathbb{R}^{m \times n}$? Instead of thinking of the vectors in the example above as solitary vectors, think of them as columns of a matrix. In that case, the columns would always span $\mathcal{C}(A)$ – this follows from the definition of the column space. However, they might not be a *basis* for $\mathcal{C}(A)$, because there might be too many columns (i.e., one or more columns that are not linearly independent from the others).

b.3 Dimension

We can finally define the *dimension* of a space.

(def 3.9) Dimension

The dimension of a vector space, denoted d or dim, is the number of vectors in every basis for the space.

Example: The xy-plane, also known as $\mathbb{R}^2$, has dim = 2 because the two vectors

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

are a basis for the space.

The key phrase in our definition for dimension is *every basis*. The basis shown in the example above is known as the "standard basis" or "canonical basis," but any and every basis for $\mathbb{R}^2$ will have two vectors. In fact, we can say quite generally:

- Any n independent vectors in $\mathbb{R}^n$ must span $\mathbb{R}^n$, so they are a basis.

b.4 The Fundamental Theorem of Linear Algebra, Part 1

Believe it or not, we have learned enough by now to state something that sounds very important: Part 1 of the Fundamental Theorem of Linear Alge-

bra. To get there, we will need to apply our definition of dimension to the four fundamental subspaces.

Let's start with $A \in \mathbb{R}^{m \times n}$ and its column space $\mathcal{C}(A)$ and its row space $\mathcal{C}(A^\top)$. We know from the definition of matrix rank that the number of independent columns equals the number of independent rows, and that this number is equal to $r = \textbf{rank}(A)$. Therefore, any *basis* for either $\mathcal{C}(A)$ or $\mathcal{C}(A^\top)$ must have exactly r vectors in it. This allows us to define the dimension of the column space and row space,

$$\dim(\mathcal{C}(A)) = \dim(\mathcal{C}(A^\top)) = r. \tag{3.2}$$

What about the nullspace, $\mathcal{N}(A)$? It's easiest to think about the extreme cases first:

- If $A \in \mathbb{R}^{m \times n}$ has full column rank ($r = n$), then the only solution to $A\vec{x} = \vec{0}$ is the zero vector. By definition, the dimension of the space that is spanned by $\vec{0}$ is 0. So the dimension of the nullspace is given by $\dim(\mathcal{N}(A)) = 0$.
- If $A \in \mathbb{R}^{m \times n}$ is the zero matrix with rank $r = 0$, then the nullspace that satsifies $A\vec{x} = \vec{0}$ is all of $\mathbb{R}^n$, which has n vectors in its basis. Thus, the dimensionality of the nullspace is given by $\dim(\mathcal{N}(A)) = n$.

In each of these cases, we find that $r + \dim(\mathcal{N}(A)) = n$. It turns out that this holds in general, and is often named the Rank-Nullity Theorem.

(def 3.10) Rank-Nullity Theorem

For $A \in \mathbb{R}^{m \times n}$, rank-nullity theorem states that

$$\dim(\mathcal{C}(A)) + \dim(\mathcal{N}(A)) = r + \dim(\mathcal{N}(A)) = n$$

This is often written to explicitly give the dimension of the null space as

$$\dim(\mathcal{N}(A)) = n - r$$

In other words, the rank of a matrix and the dimension of its column space is equal to r, the number of independent columns. And the dimension of the nullspace is equal to the number of *dependent* columns, $n - r$.

We can use similar arguments to find the dimension of the left null space,

$$\dim(\mathcal{N}(A^\top)) = m - r.$$

In this case, the dimension is the number of dependent *rows*, rather than columns, which is simply equal to the difference between the total number of rows, m, and the number of independent rows, r

Putting all this together gives us Part 1 of the Fundamental Theorem:

(def 3.11) Fundamental Theorem of Linear Algebra, Part 1

- The column space $\mathcal{C}(A)$ and row space $\mathcal{C}(A^\top)$ both have dimension r
- The nullspace $\mathcal{N}(A)$ and left nullspace $\mathcal{N}(A^\top)$ have dimensions $n - r$ and $m - r$, respectively

In the next section, we will further explore the connections between these subspaces and use what we learn to do something useful: least squares.

4 Orthogonality, Projections, and Least Squares

a Orthogonality

a.1 Vector spaces

In Section a.4 we discussed the condition for two vectors $\vec{v}$ and $\vec{w}$ to be orthogonal to each other:

$$\vec{v} \cdot \vec{w} = v^\top w = 0$$

It turns out, we can extend this definition from individual vectors up to vector spaces.

(def 4.1) Orthogonality of Vector Spaces

Two vector spaces V and W will be orthogonal to each other if:

$$v^\top w = 0 \; \forall v \in V, \; \forall w \in W. \tag{4.1}$$

> Here, the symbol $\forall$ means "for all." In plain language, we would say that all vectors v in V are orthogonal to all vectors w in W.

Picture a room like the one sketched in Figure 3. The vector space V is the floor, w_1 and w_2 are individual walls, and W is the corner between the two walls. The vector spaces V and W are orthogonal to each other because any vector along W will be orthogonal to any vector in V. However, w_1 and w_2 are not orthogonal to each other, because they share the corner W where their respective vectors would be parallel.

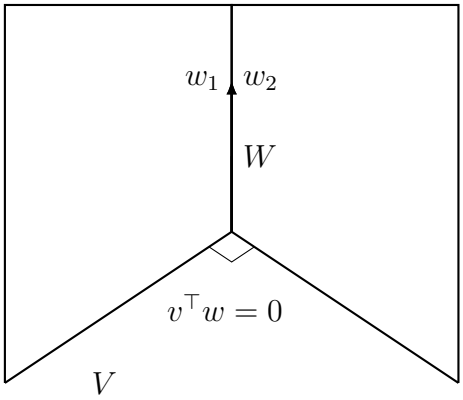


Figure 3: Orthogonality (and lack thereof) for vector spaces.

Here is another example. Consider two vector spaces we learned about in the last chapter: $\mathcal{N}(A)$ and $\mathcal{C}(A^\top)$, the null space and row space of $A \in \mathbb{R}^{m \times n}$. The null space contains all $\vec{x}$ for which $A\vec{x} = \vec{0}$. But another way to think about the matrix-vector multiplication is that it is the dot product of every row of A with the vector $\vec{x}$. So if each row r_i in A has a dot product of zero with every $\vec{x}$ in the null space, then by definition $\mathcal{N}(A) \perp \mathcal{C}(A^\top)$, where $\perp$ denotes orthogonality. By similar arguments, we can find that the left null space and the column space are orthogonal: $\mathcal{N}(A^\top) \perp \mathcal{C}(A)$.

a.2 Matrices

Orthogonal matrices are an important concept too, especially when we get to the Singular Value Decomposition (Section b).

(def 4.2) Orthogonal Matrices

A matrix $Q \in \mathbb{R}^{m \times n}$ is called orthogonal if its columns are orthonormal (i.e., unit vectors of length 1 that are orthogonal to each other), and its rows are all orthonormal vectors. This implies that the matrix will satisfy the relation

$$Q^\top Q = QQ^\top = I \tag{4.2}$$

Crucially, this also implies the following relationship between the matrix inverse and matrix transpose:

$$Q^{-1} = Q^\top \tag{4.3}$$

b Projections

b.1 Graphical Interpretation

The easiest way to conceptualize projections is with an example. In Figure 4, we have a vector $\vec{b} \in \mathbb{R}^3$, pointing from the origin to the 3D spatial coordinate (x, y, z). The *projection* of $\vec{b}$ onto the z-axis is simply given by $\vec{p_1} = (0, 0, z)$, which only contains the z component of the original vector. Similarly, the projection onto the xy-plane is given by $\vec{p_1} = (x, y, 0)$, where now we have squashed the z component to zero, and are left with a line in the xy plane.

All that is very qualitative though. How might you obtain the projected vectors $\vec{p_1}$ and $\vec{p_2}$ from $\vec{b}$ mathematically? One way is with matrix multiplication. Consider the "projection matrix"

$$P_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

You can check for yourself that $\vec{p_1} = P_1\vec{b}$. The matrix multiplication zeros out every component of $\vec{b}$ except for the z-component. We can get $\vec{p_2}$ in a similar way after multiplying by a different projection matrix P_2:

$$\vec{p_2} = P_2\vec{b} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \tag{4.5}$$

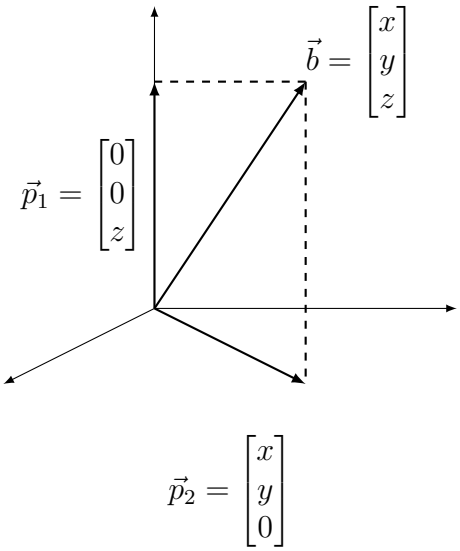


Figure 4: Projection of $\vec{b}$ onto the z axis and xy-plane

b.2 Vector projection

The examples above help to visualize projections, but the matrices that produce projections can be defined mathematically too.

(def 4.3) Projection Matrix

An $n \times n$ square matrix P is a projection matrix if and only if it satisfies $P^2 = P$ and $P = P^\top$

That definition helps us identify projection matrices, but it doesn't really help us create projection matrices. The more general problem we encounter is how to project one arbitrary vector onto another arbitrary vector.

Consider the following setup, illustrated in Figure 5: we have $\vec{a} = (a_1, \ldots, a_m)$ and $\vec{b} = (b_1, \ldots, b_m)$. What is the projection of $\vec{b}$ onto $\vec{a}$?

The projection $\vec{p}$ is the vector that goes from the origin up to the point on $\vec{a}$ that is *closest* to $\vec{b}$. The point closest to $\vec{b}$ will be the point that makes an orthogonal line $\vec{e}$ between $\vec{b}$ and $\vec{a}$. Because the projection will be in the same direction as $\vec{a}$, but scaled to a different magnitude, we can write it as

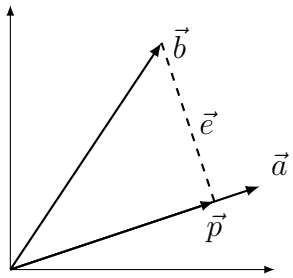


Figure 5: Projection of $\vec{b}$ onto $\vec{a}$. Both vectors are visualized as 2-dimensional, but Equation 4.6 applies to vectors with an arbitrary number of components.

$\vec{p} = \hat{x}\vec{a}$. All we need to do now is solve for $\hat{x}$:

$$\begin{aligned}\vec{e} &= \vec{b} - \vec{p} \\ &= \vec{b} - \hat{x}\vec{a}\end{aligned}$$

Noting that $\vec{a} \perp \vec{b} - \hat{x}\vec{a}$,

$$\begin{aligned}\vec{a} \cdot \left(\vec{b} - \hat{x}\vec{a}\right) &= 0 \\ \hat{x} &= \frac{a^\top b}{a^\top a}\end{aligned}$$

Putting all this together and noting that we dropped the $\vec{()}$ in the fraction for clarity we find:

$$\vec{p} = \hat{x}\vec{a} = \frac{a^\top b}{a^\top a}\vec{a} \tag{4.6}$$

We can write this in terms of a projection matrix too,

$$\vec{p} = P\vec{b} \tag{4.7}$$

where P is defined as

$$P = \frac{aa^\top}{a^\top a} \tag{4.8}$$

(def 4.4) Vector projection

The projection of an arbitrary vector $\vec{b}$ onto another vector $\vec{a}$ is given by

$$\vec{p} = \frac{\vec{a}^\top \vec{b}}{\vec{a}^\top \vec{a}} \vec{a}$$

b.3 Higher-dimensional projection

In the last section we figured out how to project one vector onto another vector. Let's extend that to a higher-dimensional case, where we want to project a vector onto a *vector space* spanned by a whole set of vectors.

Consider

$$\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n \in \mathbb{R}^m$$

as the columns of a matrix $A \in \mathbb{R}^{m \times n}$. Our goal is to project a vector $\vec{b} \in \mathbb{R}^m$ onto the vector space $\mathcal{C}(A)$, i.e., the space spanned by the columns of A. Mathematically, we seek a projection $\vec{p} \in \mathcal{C}(A)$, i.e., a linear combination of the columns of A of the form

$$\vec{p} = \hat{x}_1 \vec{a}_1 + \cdots + \hat{x}_n \vec{a}_n \tag{4.9}$$

that is closest to $\vec{b}$, where $\hat{x}_i$ are the components of a vector $\hat{x}$ that scale each of the columns of A in linear combination. Therefore, another way to phrase Equation 4.9 is that we seek $\hat{x}$ such that

$$\vec{p} = A\hat{x} \tag{4.10}$$

is as close as possible to $\vec{b}$. This is illustrated in Figure 6.

Analogous to the vector projection case (Figure 5), we can define a vector $\vec{e}$ that is orthogonal to $\mathcal{C}(A)$, the space we are projecting onto. Instead of being orthogonal to a single vector $\vec{a}$, it will be orthogonal to every column of A, i.e., $\vec{a}_1, \ldots, \vec{a}_n$. And instead of defining $\vec{e} = \vec{b} - \hat{x}\vec{a}$, we have $\vec{e} = \vec{b} - A\hat{x}$, where $A\hat{x}$ is the particular combination of the columns of A (rather than the particular scaling of the vector $\vec{a}$) that is closest to $\vec{b}$. Putting all this together, we find n equations (one for each column of A) to solve for the n

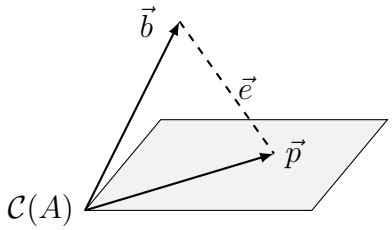


Figure 6: Projection of $\vec{b}$ onto $\mathcal{C}(A)$. Here, $\mathcal{C}(A)$ is visualized as a 2-dimensional plane, but the vector space can have an arbitrarily high dimension.

components of $\hat{x}$:

$$\begin{aligned} \vec{a}_1^\top(\vec{b} - A\hat{x}) &= 0 \\ \vec{a}_2^\top(\vec{b} - A\hat{x}) &= 0 \\ &\vdots \\ \vec{a}_n^\top(\vec{b} - A\hat{x}) &= 0 \end{aligned} \tag{4.11}$$

But we can simplify Equation 4.11 further with matrix multiplication. What is a matrix that has every column of A as one of its rows? That would be $A^\top$, so Equation 4.11 is equivalent to

$$A^\top(\vec{b} - A\hat{x}) = 0. \tag{4.12}$$

Manipulating this a bit, we find

$$A^\top\vec{b} = A^\top A\hat{x}. \tag{4.13}$$

And if $A^\top A$ is invertible, then we can solve for $\hat{x}$ as

$$\hat{x} = \left(A^\top A\right)^{-1} A^\top\vec{b} \tag{4.14}$$

When is $A^\top A$ invertible? It is invertible when A is full rank. In other words, all columns of A must be linearly independent.

(def 4.5) Projection onto a subspace

The projection of an arbitrary vector $\vec{b}$ onto the subspace $\mathcal{C}(A)$ is given by

$$\vec{p} = A\left(A^\top A\right)^{-1} A^\top \vec{b}$$

c Least Squares

Believe it or not, by arriving at Equation 4.14, we derived the least-squares solution to the regression problem, which may be something you have heard of. The setup was rather abstract though, so let's view it from a more familiar lens.

Say we have a vector $\vec{b} \in \mathbb{R}^m$ containing $m = 47$ home sale prices from Portland, OR. We have another vector $\vec{a}_1 \in \mathbb{R}^m$ containing the corresponding square footage for those houses, and $\vec{a}_2 \in \mathbb{R}^m$ with the number of bedrooms in each house. Our goal is to find a model that predicts the home price as a linear function of the square footage and number of bedrooms. Mathematically, we can pose this as

$$b_i \approx \hat{x}_1 a_{1,i} + \hat{x}_2 a_{2,i} \text{ for } i = 1, \ldots, m, \tag{4.15}$$

where $\hat{x}_1$ and $\hat{x}_2$ are fit coefficients for the values in $\vec{a}_1$ and $\vec{a}_2$, respectively, and $a_{1,i}$ denotes the i^{th} element of vector $\vec{a}_1$.

Rather than writing $m = 47$ separate equations, we can use matrix multiplication rules and define $A \in \mathbb{R}^{m \times n}$, with $n = 2$ as

$$A = \begin{bmatrix} | & | \\ \vec{a}_1 & \vec{a}_2 \\ | & | \end{bmatrix}$$

We can then define a vector $\hat{x} \in \mathbb{R}^{n \times 1}$ as

$$\hat{x} = \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix}$$

This simplifies Equation 4.15 to

$$A\hat{x} \approx \vec{b}, \tag{4.16}$$

where our goal is to find the vector $\hat{x}$ that maps the columns of A (square footage and number of bedrooms) as close as possible to the elements of $\vec{b}$

(home price). In machine learning terminology, the $n = 2$ columns of A are the *features*, i.e., independent variables, that are available to us. The vector $\vec{b}$ is the *response* or *target* (dependent variable) that we are trying to predict.

Let's be careful though, and remember what we learned in Section b.1. A linear system $A\vec{x} = \vec{b}$ where A has full column rank and $m > n$ will have either no solution, or one solution depending on whether $\vec{b} \in \mathcal{C}(A)$. This is called an overdetermined system, because we have more equations ($m = 47$) than unknowns ($n = 2$, for $\hat{x}_1$ and $\hat{x}_2$). It is pretty rare to find an exact solution to Equation 4.16 when there is that big of a mismatch between m and n. Finding an exact solution would mean that $\vec{b}$ is in the column space of A, which would imply that we can predict the home price *exactly* for those 47 homes based on the square footage and number of bedrooms, which doesn't seem realistic.

But this is where projections come in handy. We don't need an exact solution, we just want the closest mapping between the columns of A and the data in $\vec{b}$, and that mapping is found by projecting $\vec{b}$ onto the subspace spanned by the columns of A. In other words, the closest solution to Equation 4.16, which we call the "least-squares" solution, is found from Equation 4.14, which we'll write again here because it is so important:

$$\hat{x} = \left(A^\top A\right)^{-1} A^\top \vec{b} \tag{4.17}$$

Let's see this with actual numbers and plots to drive home the point.

c.1 Least Squares Example: Home Prices

Sticking with the example discussed above, we have 47 data points of home square footage, number of bedrooms, and sale price. We seek a model that maps the square footage and number of bedrooms to the price.

To start, we can plot individual features against the target to get a sense of correlation. This is shown in Figure 7, where we see reasonable positive correlation for each of the features.

When it comes to constructing our model, we will need a coefficient for each of the features plus a constant intercept term, so that in total we solve:

$$\hat{x}_0 + \hat{x}_1\vec{a}_1 + \hat{x}_2\vec{a}_2 = \vec{b},$$

With $\hat{x}_0$ being the intercept, $\hat{x}_1$ the coefficient (or slope) associated with the square footage vector $\vec{a}_1$, and $\hat{x}_2$ the coefficient associated with the number

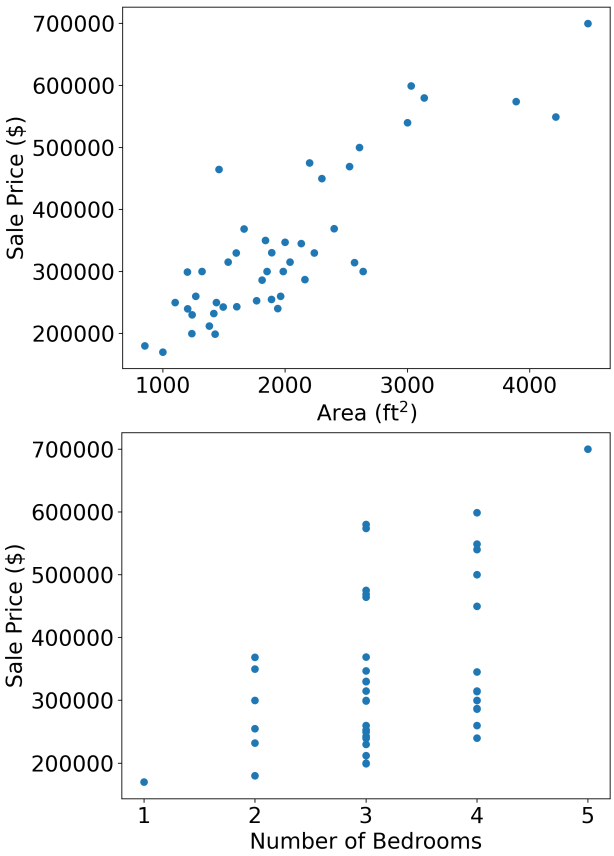


Figure 7: Home sale price (target) as a function of each of the features.

of bedrooms vector $\vec{a}_2$. As a matrix equation, this looks like $A\hat{x} = \vec{b}$, with:

$$A = \begin{bmatrix} | & | & | \\ 1 & \text{area}_i & \text{beds}_i \\ | & | & | \end{bmatrix}, \; \hat{x} = \begin{bmatrix} \hat{x}_0 \\ \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} \; \vec{b} = \begin{bmatrix} | \\ \text{price}_i \\ | \end{bmatrix},$$

for $i = 1 \ldots 47$. We can solve this in Python using matrix inverses and multiplication:

```
import numpy as np
A = np.hstack((np.ones_like(area), area, beds))
b = price
x = np.matmul(np.linalg.inv(np.matmul(A.T, A)), np.matmul(A.T, b))
print(x)
```

```
>>
[[87807.75019324]
 [  138.75587842]
 [-8186.38287595]]
```

This tells us that the optimal values for $\hat{x}$ are $\hat{x}_0 \approx 87807$, $\hat{x}_1 \approx 138$, $\hat{x}_2 \approx -8186$. We can confirm this by using a more standard approach, relying on `scikit-learn`:

```
from sklearn.linear_model import LinearRegression
X = np.hstack((area, beds))
y = price
model = LinearRegression().fit(X, y)
print(model.coef_)
>> array([[  138.75587842, -8186.38287595]])
print(model.intercept_)
>> array([87807.75019324])
```

Same answer! Overall, our prediction vs the actual price is shown in Figure 8.

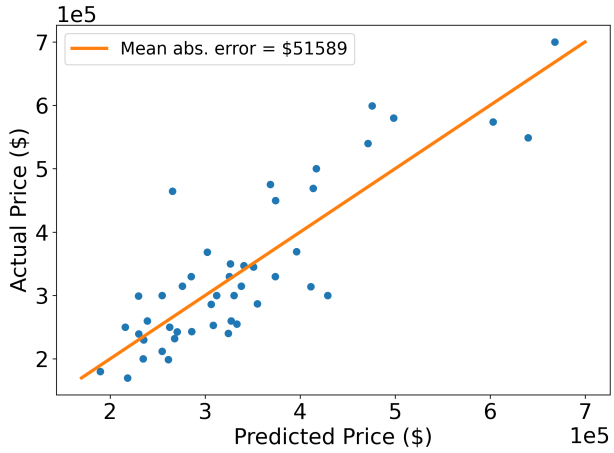


Figure 8: Predicted vs actual housing price predicted via linear regression using square footage and number of bedrooms as features.

5 Decompositions

This is the part where Linear Algebra gets really useful for engineering applications.

a Eigenvectors and Eigenvalues

a.1 Basic definitions

We begin with a matrix A, which for now must be square:

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

If we use A to multiply some vector $\vec{x}$, what happens? Usually, it produces some other vector $A\vec{x}$ with a different magnitude and a different direction. For example, if we set $\vec{x} = (1, -1)$, we obtain

$$A\vec{x} = \begin{bmatrix} 1 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix} \tag{5.1}$$

This is visualized in Figure 9.

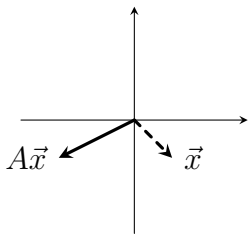


Figure 9: Change in direction and magnitude of $\vec{x}$ after multiplication by A

Sometimes, though, we encounter special vectors $\vec{x}$ that only change their magnitude by either stretching, shrinking, or reflecting by some factor λ when they are multiplied by A. Crucially though, the matrix multiplication does not change their direction. These special vectors are called the *eigenvectors* of A, and the factors λ that modify their magnitude are called the *eigenvalues* associated with those eigenvectors. An eigenvalue $|\lambda| > 1$ will stretch the vector, $|\lambda| < 1$ will shrink it, and $\lambda < 0$ will reflect it.

(def 5.1) Eigenvectors and Eigenvalues

The n eigenvalues $\lambda_1, \ldots, \lambda_n$ and corresponding eigenvectors $\vec{x}_1, \ldots, \vec{x}_n$ associated with a square matrix $A \in \mathbb{R}^{n \times n}$ satisfy the equation

$$A\vec{x}_i = \lambda_i \vec{x}_i \tag{5.2}$$

Example: For the matrix

$$A = \begin{bmatrix} 1 & 9 \\ 0 & 2 \end{bmatrix}$$

We have two pairs of eigenvalues and eigenvectors. These are $\lambda_1 = 1$ and $\vec{x}_1 = (1, 0)$, which satisfy:

$$A\vec{x}_1 = \begin{bmatrix} 1 & 9 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \lambda_1 \vec{x}_1$$

And $\lambda_2 = 2$, $\vec{x}_2 = (1, 1/9)$, which satisfy:

$$A\vec{x}_2 = \begin{bmatrix} 1 & 9 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1/9 \end{bmatrix} = \begin{bmatrix} 2 \\ 2/9 \end{bmatrix} = \lambda_2 \vec{x}_2$$

For a given matrix A, how can you find its eigenvalues and eigenvectors? The practical way (and the only way for large matrices) is with a programming language like Python:

```
import numpy as np
A = np.array([[1, 9], [0, 2]])
eigenvalues, eigenvectors = np.linalg.eig(A)
lambda_1 = eigenvalues[0]
lambda_2 = eigenvalues[1]
x1 = eigenvectors[:, 0]
x2 = eigenvectors[:, 1]
print(lambda_1)
>> 1.0
print(lambda_2)
>> 2.0
print(x1)
>> [1. 0.]
```

```
print(x2)
>> [0.99388373, 0.11043153]
```

Here we see a slight difference in $\vec{x}_2$ compared to the value in the example above. This is not due to floating point precision issues. It highlights an interesting property of eigenvectors, which is that if $\vec{x}$ is an eigenvector of A associated with eigenvalue λ, then any scalar multiple of it $c\vec{x}$ is also an eigenvector of A associated with eigenvalue λ.

In simple cases though (2×2 or 3×3 matrices), we can solve for eigenvalues and eigenvectors by hand. To do so, we can rearrange Equation 5.2 as:

$$(A - \lambda I)\,\vec{x} = 0 \tag{5.3}$$

In other words, $\vec{x}$ will be be an eigenvector of A if it is in the nullspace of $A-\lambda I$, where I is the identity matrix with the same size as A. Furthermore, if $\vec{x}$ is nonzero (this must be true in order for $\vec{x}$ to be an eigenvector), it follows that $A - \lambda I$ is *singular*, i.e., not invertible, meaning that

$$\det(A - \lambda I) = 0 \tag{5.4}$$

This is the expression we will use to find $\vec{x}$ and λ. Let's see an example.

(ex 5.1) Eigenvalues and Eigenvectors by hand

Find the eigenvalues and eigenvectors of the matrix

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

Solution: First we define $A - \lambda I$:

$$A - \lambda I = \begin{bmatrix} 1-\lambda & 2 \\ 2 & 4-\lambda \end{bmatrix}$$

Then we calculate the determinant and set it to zero:

$$\det(A - \lambda I) = (1-\lambda)(4-\lambda) - 2(2) = 0$$

Next, simplify and solve for λ, which will be the roots of the polynomial:

$$\lambda^2 - 5\lambda = 0$$
$$\lambda(\lambda - 5) = 0$$

Therefore, we have eigenvalues $\lambda_1 = 0$ and $\lambda_2 = 5$. To find the eigenvectors, we need to solve $(A - \lambda_i I)\vec{x}_i = \vec{0}$ for each of the two sets. Starting with $\lambda_1 = 0$:

$$(A - 0I)\vec{x}_1 = x_{11}\begin{bmatrix}1\\2\end{bmatrix} + x_{12}\begin{bmatrix}2\\4\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

This can be solved by $\vec{x}_1 = (x_{11}, x_{12}) = (2, -1)$.

For $\lambda_2 = 5$:

$$(A - 5I)\vec{x}_2 = \begin{bmatrix}1-5 & 2\\2 & 4-5\end{bmatrix}\vec{x}_2 = \begin{bmatrix}-4 & 2\\2 & -1\end{bmatrix}\vec{x}_2 = \begin{bmatrix}0\\0\end{bmatrix}$$

This can be solved by $\vec{x}_2 = (1, 2)$

A final thing to note about this example: the matrix A is singular (has no inverse), because the second column is a linear combination of the first. For a singular A, one of the eigenvalues will always be zero.

There are two quick "sanity checks" we can perform to validate the eigenvalues we computed:

1. The product of the n eigenvalues of A is equal to $\det(A)$.
2. The sum of the n eigenvalues of A is equal to the *trace* of A, $\text{Tr}(A)$, which is the sum of the diagonal elements of A.

a.2 Diagonalization

We will now see something special that happens when we have a matrix $A \in \mathbb{R}^{n\times n}$ with n independent eigenvectors $\vec{x}_1, \ldots, \vec{x}_n$. This will always be the case if you have n distinct eigenvalues (i.e., no repeated eigenvalues). Specifically, we can *diagonalize* A.

(def 5.2) Diagonalization

Suppose $A \in \mathbb{R}^{n\times n}$ has n independent eigenvectors $\vec{x}_1, \ldots, \vec{x}_n$. We can

construct a matrix S with the eigenvectors as columns, i.e.,

$$S = \begin{bmatrix} | & & | \\ \vec{x}_1 & \cdots & \vec{x}_n \\ | & & | \end{bmatrix}$$

and a diagonal matrix Λ with the eigenvalues along its diagonal, i.e.,

$$\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$$

and can then write our original matrix A as

$$A = S\Lambda S^{-1} \tag{5.5}$$

This is referred to as a diagonalization or the Eigenvalue Decomposition of A.

Why does Equation 5.5 work?

$$AS = \begin{bmatrix} A\vec{x}_1 & \cdots & A\vec{x}_n \end{bmatrix} = \begin{bmatrix} \lambda_1\vec{x}_1 & \cdots & \lambda_n\vec{x}_n \end{bmatrix} = \underbrace{\begin{bmatrix} \vec{x}_1 & \cdots & \vec{x}_n \end{bmatrix} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}}_{S\Lambda}$$

So if we have $AS = S\Lambda$, we can multiply both sides by S^{-1} to obtain $A = S\Lambda S^{-1}$.

Ok so we can write A in this new form, who cares? Lots of people, it turns out. One of the most useful properties of this decomposition is a simplification of the matrix power A^k. Usually, to compute $A^3 = AAA$ you would need to do 3 matrix multiplications. Not so bad, but to compute A^{100} you would need to do 100 matrix multiplications, which gets expensive for your computer. All of this is simplified with the eigenvalue decomposition through the property

$$A^k = S\Lambda^k S^{-1}. \tag{5.6}$$

Because Λ is diagonal, the matrix power Λ^k is just the power of the diagonal elements, so it is very cheap and easy to compute. Matrix powers come up

all the time in dynamical systems theory, controls problems, and mechanical and electrical engineering in general, so this diagonalization saves a lot of compute time in the machines that make the world go 'round.

A few other quick facts to summarize the past two sections and round out our knowledge of eigenvalues:

- A singular (non-invertible) matrix will have $\lambda = 0$ as at least one of its eigenvalues.
- Eigenvalues can be complex numbers, i.e., numbers with an imaginary component.
- The eigenvalue decomposition of a symmetric matrix $A = A^\top$ is special: it is guaranteed to have only real eigenvalues, and the eigenvectors can be chosen to be orthonormal such that $A = Q\Lambda Q^{-1} = Q\Lambda Q^\top$. This is called the *spectral theorem*, and it will come up again when we talk about the Singular Value Decomposition.
- A matrix with all positive eigenvalues is called *positive definite*, and will also satisfy the relationship $\vec{x}^\top A\vec{x} > 0$ for any $\vec{x}$. These matrices are important in many convex optimization problems.
- A matrix with all nonnegative eigenvalues is called *positive semidefinite*, and will have all $\lambda_i \geq 0$ for $i = 1, \ldots, n$.
- A triangular matrix (a matrix with all zeros above and/or below its main diagonal) will have its eigenvalues along the diagonal.
- An $n \times n$ matrix with n distinct eigenvalues will always be diagonalizable.

a.3 Application: Differential Equations

Consider the single variable differential equation

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \lambda u \tag{5.7}$$

with the initial condition $u(t = 0) = u_0$. This can be solved with

$$u(t) = u_0 e^{\lambda t}. \tag{5.8}$$

That's a nice simple example, but in real world systems, like airplanes or wind turbines or your cell phone, engineers often needs to solve hundreds of equations at the same time. This can be represented in the language of linear algebra with an equation like

$$\frac{\mathrm{d}\vec{u}}{\mathrm{d}t} = A\vec{u}, \tag{5.9}$$

with $\vec{u} \in \mathbb{R}^n$ containing the n different variables we want to solve for, and $A \in \mathbb{R}^{n\times n}$ containing the coefficients in front of each variable for each of the equations.

This sounds challenging, but it turns out that eigenvalues help us tremendously, allowing us to write a solution to Equation 5.9 of the form

$$\vec{u}(t) = \sum_{i=1}^{n} c_i e^{\lambda_i t}\vec{x}_i \tag{5.10}$$

where λ_i and $\vec{x}_i$ are eigenvalues and eigenvectors of A, and the constants c_i are determined from initial conditions.

Why does this work? Let's plug in $\vec{u} = e^{\lambda t}\vec{x}$ to Equation 5.9:

$$\frac{\mathrm{d}\vec{u}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\left(e^{\lambda t}\vec{x}\right) = \lambda e^{\lambda t}\vec{x} \tag{5.11}$$

And if $\vec{x}$ and λ are eigenvectors and eigenvalues, they satisfy

$$A\vec{x} = \lambda\vec{x} \tag{5.12}$$

So plugging Equation 5.12 into Equation 5.11:

$$\frac{\mathrm{d}\vec{u}}{\mathrm{d}t} = Ae^{\lambda t}\vec{x} = A\vec{u}, \tag{5.13}$$

and we are back with the original equation we wanted to solve. This is only possible if Equation 5.12 holds, which is why eigenvalues are so important to the solution of differential equations.

b Singular Value Decomposition

The Eigenvalue Decomposition in the preceding section had a severe restriction: it only works for a square matrix. Now we'll learn about a more general decomposition that works for any $m \times n$ matrix. Here's how it works.

b.1 Full SVD

Start with an $m \times n$ matrix A with rank r. We'll assume for now that it has real components, but this isn't required. Our goal is still to diagonalize A, but because A isn't square we need two sets of "eigenvectors" to handle the two different dimensions. We will call these more general "singular vectors" $\vec{v}$ and $\vec{u}$, and instead of eigenvalues we will have "singular values" σ.

We can put these vectors and values into matrices,

$$\underbrace{V = \begin{bmatrix} \vec{v}_1 \cdots \vec{v}_n \end{bmatrix}}_{n \times n}, \quad \underbrace{U = \begin{bmatrix} \vec{u}_1 \cdots \vec{u}_m \end{bmatrix}}_{m \times m}, \quad \underbrace{\Sigma = \begin{bmatrix} \sigma_1 & & & 0 \\ & \ddots & & \\ & & \sigma_r & 0 \\ 0 & & 0 & 0 \end{bmatrix}}_{m \times n}$$

and write the equation

$$AV = U\Sigma, \tag{5.14}$$

which is analogous to the eigenvalue decomposition $AS = S\Lambda$, but now we have two different sets of singular vectors instead of one set of eigenvectors.

Some details about the singular values and vectors:

- $\vec{v}_i$ for $i = 1, \ldots, n$ are the right singular vectors of A. They are sometimes called the "input" singular vectors of A. They are a basis for the row space of A and they are equivalent to the eigenvectors of the symmetric matrix $A^\top A$, which means that they are all orthogonal to each other.

- $\vec{u}_i$ for $i = 1, \ldots, m$ are the left singular vectors of A, also called the output vectors. They are a basis for the column space of A and are equivalent to the eigenvectors of $AA^\top$. Therefore, they are also orthogonal to each other.

- σ_i are the strictly positive singular values that satisfy $\sigma_1 > \cdots > \sigma_r > 0$. They live on the diagonal of Σ, but there are only r of them, so any remaining space on the diagonal is filled with zeros. The singular values can equivalently be defined in terms of the eigenvalues of $A^\top A$ and $AA^\top$, $\sigma_i = \sqrt{\lambda_i(A^\top A)} = \sqrt{\lambda_i(AA^\top)}$.

Recognizing that $V^\top = V^{-1}$ we can rearrange Equation 5.14 to give the explicit diagonalization of A,

$$A = U\Sigma V^\top \tag{5.15}$$

b.2 Reduced SVD

It gets better. Equation 5.15 is nice because we can diagonalize any matrix, but it can be wasteful. If r (the rank of A) is small, then Σ will have a lot of zeros in it, and none of those $(m-r)$ rows of zeros or $(n-r)$ columns of zeros will contribute to the matrix multiplication $U\Sigma V^\top$. If we instead define *reduced* matrices

$$\underbrace{V_r = \begin{bmatrix}\vec{v}_1 \cdots \vec{v}_r\end{bmatrix}}_{n\times r},\ \underbrace{U_r = \begin{bmatrix}\vec{u}_1 \cdots \vec{u}_r\end{bmatrix}}_{m\times r},\ \underbrace{\Sigma_r = \begin{bmatrix}\sigma_1 & & \\ & \ddots & \\ & & \sigma_r\end{bmatrix}}_{r\times r}$$

then we can decompose A with no loss of information, yet (potentially many) fewer matrix multiplications, as

$$A = U_r\Sigma_r V_r^\top. \tag{5.16}$$

This reduced form can save a lot of memory and compute time. Imagine if A is size $100 \times 100{,}000$. If we decompose A using Equation 5.15, then our matrices will be $V \in \mathbb{R}^{100{,}000\times 100{,}000}$, $U \in \mathbb{R}^{100\times 100}$, and $\Sigma \in \mathbb{R}^{100\times 100{,}000}$. But we know that rank$(A)$ <= 100, so it would be much nicer to only store in memory (and use in computations) $V \in \mathbb{R}^{100{,}000\times 100}$, $U \in \mathbb{R}^{100\times 100}$, and $\Sigma \in \mathbb{R}^{100\times 100}$.

b.3 Connection to PCA

You may have encountered Principal Components Analysis (PCA), a popular dimensionality reduction tool, in another class. PCA is almost the same thing as SVD, it just gives different names to the output.

Start with a data matrix $A \in \mathbb{R}^{m\times n}$ with m samples of n features. Each feature is normalized to have zero mean and unit variance. The SVD decomposes A as $A = U\Sigma V^\top$. In PCA lingo, we can define the following:

- The columns of V (or the rows of $V^\top$; same thing) are the *principal axes*, or directions of maximum variance for the data.
- The columns of $AV = U\Sigma$ are the original data projected onto the principal axes. These are often called *principal components* or *scores*.
- The dimensionality of the data can be reduced by keeping the first k singular values in Σ, setting the rest to 0, and constructing the reduced $A_k = U_k \Sigma_k V_k^\top$. If $k = \text{rank}(A)$ then you obtain the original A (Equation 5.16).
- The *proportion of variance explained* by principal component i is given by $\sigma_i^2 / \sum_j \sigma_j^2$. Therefore, the most important principal components are associated with the largest singular values.

b.4 Application 1: Image Compression via Low-Rank Approximation

The interpretation of SVD as a dimensionality reduction tool like PCA gives rise to one of its most useful applications: the *low-rank approximation*.

(def 5.3) Low-rank approximation

The rank-k approximation of the matrix $A \in \mathbb{R}^{m \times n}$, with $\text{rank}(A) = r > k$ is given by the reduced SVD

$$A_k = U_k \Sigma_k V_k^\top, \tag{5.17}$$

where only the first k singular values are retained but the rest are set to zero. This is often called a low-rank approximation of A. It is particularly useful when A is close to singular: A is technically full-rank, but most of the information it contains can be reconstructed from a small fraction of the singular values/vectors. Matrices such as these will have a high *condition number*, which is the ratio of the largest to smallest singular value.

Let's see why this is so useful. Say you are working on a computer vision problem involving very high resolution images of my dogs, Mandy and Ginny (Figure 10). These images are stored as matrices of grayscale pixel values, with size 3338×2625 pixels. This full-size image has rank 2625, but it can

be approximated as a matrix with rank $k < 2625$ using the SVD. The fewer singular values (i.e., the lower the rank), the more blurry and pixelated the image becomes. But even with the significant memory savings of $k = 100$ and $k = 50$, the image is still easy to see. You can play around with the demo here to explore this compression technique with other images.

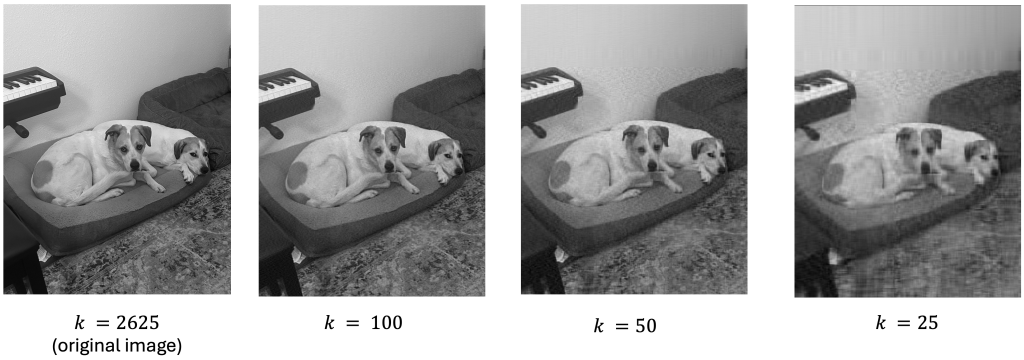


Figure 10: SVD-based compression of an image, with k indicating the number of retained singular values/vectors (i.e., the rank of the output image).

```
from PIL import Image
import numpy as np
from scipy.linalg import svd

full_image = Image.open(
    "images/mandy_ginny.jpg"
).convert("L")
full_image_array = np.array(full_image)
plt.imshow(full_image_array, cmap="gray")

# SVD
U, S, Vt = svd(
    full_image_array, full_matrices=False
)
S = np.diag(S)

# Reconstructing with k = 100
k = 100
```

```
Ak = np.matmul(U[:, :k] @ S[:k, :k], Vt[:k, :])
plt.imshow(Ak, cmap="gray")
```

b.5 Application 2: Pseudoinverse and Least Squares

As we noted in Section b.3, the least-squares solution

$$\hat{x} = (A^\top A)^{-1} A^\top \vec{x}$$

only works when $(A^\top A)^{-1}$ exists, which requires the columns of A to be independent. This isn't always the case: columns of your data matrix (the features) could be correlated to each other, and we don't want that to get in the way of solving a least squares problem. Luckily, SVD and the *pseudoinverse* saves the day.

(def 5.4) Pseudoinverse

Let $A \in \mathbb{R}^{m \times n}$ be a matrix with SVD $A = U\Sigma V^\top$. The pseudoinverse is given by

$$A^\dagger = (U\Sigma V^\top)^{-1} = V\Sigma^{-1}U^\top, \tag{5.18}$$

with

$$\Sigma^{-1} = \begin{bmatrix} \sigma_1^{-1} & & & \\ & \ddots & & \\ & & \sigma_r^{-1} & \\ & & & 0 \end{bmatrix}$$

The pseudoinverse $A^\dagger$ is guaranteed to exist, even if A^{-1} does not exist. And if A^{-1} does exist, then it is equal to $A^\dagger$.

Once we have the pseudoinverse, then the least squares solution to $A\vec{x} = \vec{b}$ is given by

$$\hat{x} = A^\dagger \vec{b} \tag{5.19}$$

b.6 Application 3: Data Matrix Structure

The SVD of a data matrix can highlight the matrix's underlying structure. In other words, if there are patterns in the data that help to simplify inter-

pretation, the SVD can automatically find those patterns. Strong patterns that explain a large proportion of the data will be assigned large singular values, and the original matrix can often be approximated from just a few of those strong patterns. To see an example of this, refer to Example 4.14 in the MML textbook, which looks at the SVD of a movie rating matrix.