unraveling principal component analysis

Peter Bloem

July 2, 2023

Contents

Who is this book for? It wasn't born from a desire to plug a gap in the market, and I didn't have a clear idea of the kind of book I wanted to create when I started. For a long time, it wasn't even meant to be a book.

It started when I began teaching machine learning in 2018. One of the topics was principal component analysis, and I wanted to do a good job, since it was a topic that had entranced and mystified me in equal measure as a student. I wanted to do it justice.

It turned out that I couldn't do that in the time that was available to prepare a single lecture. On the morning before the lecture, going over my slides I caught several mistakes that I couldn't quite fix, and many questions that I didn't know how to answer.

I brushed it off, survived the lecture despite feeling like an imposter, and resolved to do better next year. The next year, the same thing happened again. Different mistakes, different questions, but the same feeling of cheating myself and the students.

The thing is, as a student, I never delved deeply into the mathematics of anything. I never *really* got to grips with linear algebra and at best, I could only ever claim to have a working grasp of what I needed to apply it in simple settings, and to look up what I didn't know.

This, I expect, is how it is for many students and researchers alike: we are taught the fundamentals from the ground up, but we only start tuning in when things become concrete. The foundations, we either never learn, or quickly forget.

After three years of this, I decided it was enough, and I began to write a blog post on PCA. This would finally force me to properly come to grips with the subject, however deep the rabbit hole went. Once I was finished, I could use the blog post as part of the teaching material. Proof, if any were need, that I really did know what I was talking about.

The more I wrote, the more questions I generated for myself, and the more the blog post span out of control. I decided to split the thing in two: the first part a simple self-contained story for those, like my students, who just wanted to understand enough of PCA to apply it effectively, and a second part that delved into the fundamentals.

The pattern continued steadily: to really get down to the foun-

dations, I needed to prove the spectral theorem. This required so many dependencies it became a third blog post, so it didn't clutter up the second. Then, I felt, I needed to end on a clear algorithm for PCA. You can't say you really understand something if you don't know how to implement it. This is always done through the singular value decomposition (SVD), which I then felt I should also understand properly. In the end, the explanation of the SVD and the algorithm to compute it became too big to put into a single blogpost and I split those up too, bringing the tally to five. Five blog posts, with each little too big to function well as a blog post.

In short, I realized I had accidentally written a book. Like I said. I don't really know who it's for. It's about PCA, but it covers much more than that. Almost everything a linear algebra textbook does. But it's not a textbook by any stretch; I certainly wouldn't teach from it, and it kind of assumes you know linear algebra already when you start reading.

It's also not quite popular science. I occasionally go into narrative mode, but really, the aim is to go deep. To dig up the foundations.

I suppose it's a book for people like me. People who have learned linear algebra and then forgotten it. Who feel a measure of regret that they didn't pay full attention the first time around. If you've ever marveled at the magical results that PCA produces, and you'd like to really understand it, all the way down to the fundament, then this book will provide you with a guide. But perhaps it's best to think of this as a guided tour of the forests of linear algebra. I've been deep into the woods in search of treasure and I've made it back out. Let me show you what I've found.

Acknowledgements I am indebted to Emile van Krieken and to Charlie Lu for corrections and suggestions. My thanks go out to Nathan Young for the use of a figure from (Young et al., 2015) and to John Novembre for useful comments on the interpretation of the genomic PCA analysis in Chapter 1.

Licensing All figures in this book are released under a Creative Commons CC-BY-SA license. Source files may be found at `https://github.com/pbloem/pca-book`.

CHAPTER 1 · A FRIENDLY INTRODUCTION TO PRINCIPAL COMPONENT ANALYSIS

We will work from the outside in: we will view PCA first as a way of finding a smaller representation of a dataset. This is a typical machine learning problem: find a compressed representation of the data such that the reconstructions are as close to the original as possible. This is a simple view of PCA, an we'll be able to compute it with nothing more than gradient descent with a few extra tricks for satisfying constraints.

Most of the technical stuff only becomes necessary when we want to understand why PCA works so well: this is where the **spectral theorem** and the **eigenvalues and -vectors**, come in to the story, they give us a deeper understanding of what we're doing. We'll look at these subjects in Chapter 2.

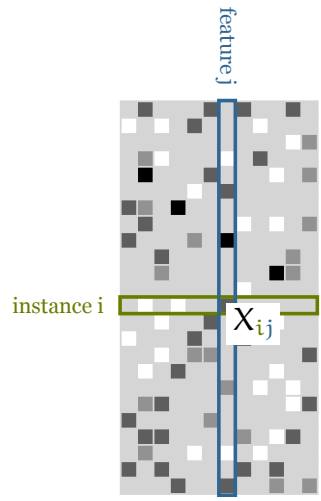
The spectral theorem is the heart of the method, so it pays to discuss it in some detail. We'll state it and explain what it means in Chapter 2, and leave the proof to Chapter 3.

I'll assume some basic linear algebra knowledge, but I'll try to explain the preliminaries where possible, even if they are fundamental to linear algebra. There is a small list of identities and properties in the appendix, which you may want to consult to refresh your memory.

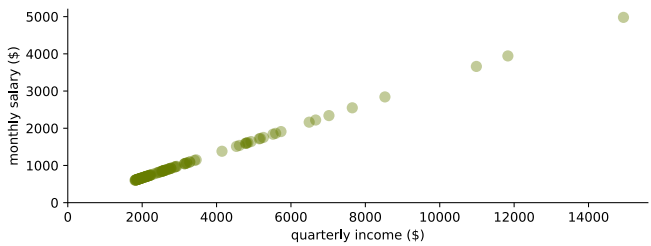
1.1 The basics

Let's begin by setting up some basic notation. We are faced with some high-dimensional dataset of instances (examples of whatever we're studying) described with real-valued features. That is, we have n *instances* $\mathbf{x}_i$ and each instance is described by a vector of m real values. We describe the dataset as a whole as an $n \times d$ matrix $\mathbf{X}$ that is, we arrange the examples as rows, and

the features as columns.



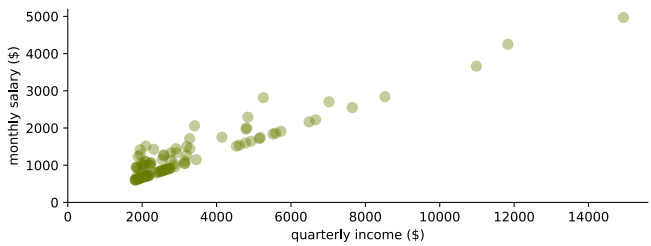
For a simple example, imagine that we have a dataset of 100 people with 2 features measured per person: their monthly salary and their income over the course of a quarter (i.e. three months). The second is just the first times 3, so this data is *redundant*. One value can be computed from the other, so we really only need to store one number per person. Here's what that looks like in a scatterplot.



Our intuition that we really only need one number to represent the data is reflected in the fact that *the data form a line*. So long as we know what that line is, we only need to know how far along the line each instance is, so we can store the whole dataset in one number per instance.

> *There are also more complex relations between two features that have this property, like a parabola or an exponential curve. In PCA we simplify things by only exploiting linear relations.*

Of course, data in the wild is never this clean. Let's introduce some small variations between the monthly salary and the income after three months. Some people may have changed jobs, some people may get bonuses or vacation allowances, some people may have extra sources of income. Here's a more realistic version of the data.



The data is no longer perfectly linear, but it still seems *pretty* linear. If we imagine the same line we had in the last plot, and represent each person as a dot along that line, we lose some information, but we still get a decent *reconstruction* of the data.

If you know how to do linear regression, you can probably work out how to draw such a line through the data, predicting the income from the salary or the other way around (and PCA is very similar to linear regression in many ways). However we'll need something that translates to higher dimensions, where we don't have a single target feature to predict.

To do so, we'll develop a method purely from these first principles:

1. We want to represent the data using a small set of numbers per instance.

2. We will limit ourselves to linear transformations of the data to derive this small set.

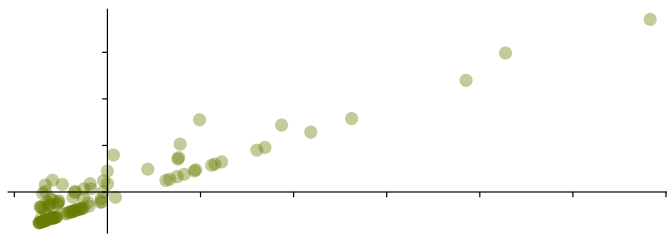
3. We want to minimize the error in our reconstruction of the data. That is, when we map the data back to the original representation, as best we can, we want it to be as close as possible to the original.

1.2 One-dimensional PCA

We'll develop a one-dimensional version of PCA first. That is, we will represent each instance $\mathbf{x}_i$ (row i in our data matrix $\mathbf{X}$), by a single number z_i as best we can. We will call z_i the *latent representation* of $\mathbf{x}_i$.

The phrase "latent" comes from the Latin for being hidden. *This will make more sense when we see some of the other perspectives on PCA.*

To start with, we will assume that the data are **mean-centered**. That is, we have subtracted the mean of the data so that the mean of the new dataset is 0 for all features.



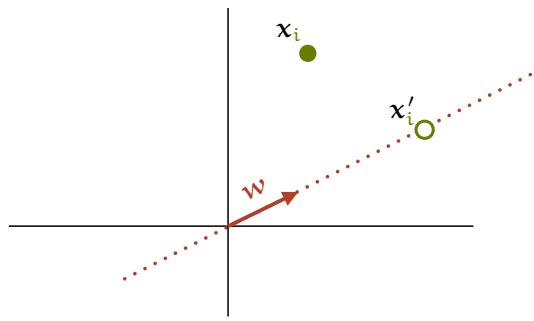
For now, think of this as a bit of necessary data pre-processing. We will see where this step comes from in the next chapter.

Our task is to find a linear transformation from $\mathbf{x}_i$ to z_i, and another linear transformation back again. A linear transformation from a vector $\mathbf{x}_i$ to a single number is just the dot product with a vector of weights. We'll call this vector $\boldsymbol{v}$. A linear transformation from a single number z_i back to a vector is just the multiplication of z_i by another vector of weights. We'll call this vector $\boldsymbol{w}$. This gives us

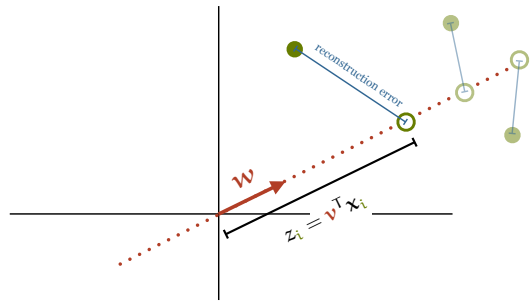
$$z_i = \boldsymbol{v}^T \mathbf{x}_i$$
$$\mathbf{x}'_i = z_i \boldsymbol{w}$$

where $\mathbf{x}'_i$ is the reconstruction for instance i.

Look closely at that second line. It expresses exactly the intuition we stated earlier: we will choose one line, represented by the vector $\boldsymbol{w}$, and then we just represent each data point by how far along the line it falls, or in other words, we represent $\mathbf{x}'_i$ as a multiple of $\boldsymbol{w}$.



All allowed values of $\mathbf{x}'_i$ are on the dotted red line, which is defined by our choice of $\boldsymbol{w}$. Where each individual ends up is defined by the multiplier z_i, wich is determined by the weights $\boldsymbol{v}$. Our objective is to choose $\boldsymbol{v}$ and $\boldsymbol{w}$ so that the reconstruction error, the distance between $\mathbf{x}_i$ and $\mathbf{x}'_i$ is minimized (over all i).



We can simplify this picture in two ways.

First, note that many different vectors w define the same dotted line in the image above. So long as the vector points in the same direction, any length of vector defines the same line, and if we rescale z_i properly, the reconstructed points x_i' will also be the same. To make our solution unique, we will *constrain* w to be a unit vector. That is, a vector with length one: $w^T w = 1$.

> *To be more precise, this doesn't leave a unique solution, but two solutions, assuming that a single direction is optimal, since the unit vector can point top right, or bottom left. In other words, if w is a solution, then so is $-w$.*

The second simplification is that we can get rid of the second vector v. Imagine that we have some fixed w (that is, the dotted line is fixed). Can we work out which choice of x_i on the line will minimize the reconstruction error? We will go into a bit of detail here, since it helps to set up some intuitions that will be important in the later chapter of this book.

If you remember your linear algebra, you'll know that this happens when the line of the reconstruction error is *orthogonal* to the dotted line. In more fancy language, the optimal x_i is the *orthogonal projection* of x_i onto the dotted line. If you look at the image, it's not too difficult to convince yourself that this it true. You can imagine the reconstruction error as a kind of rubber band pulling on x_i, and the point where it's orthogonal

is where it comes to rest.

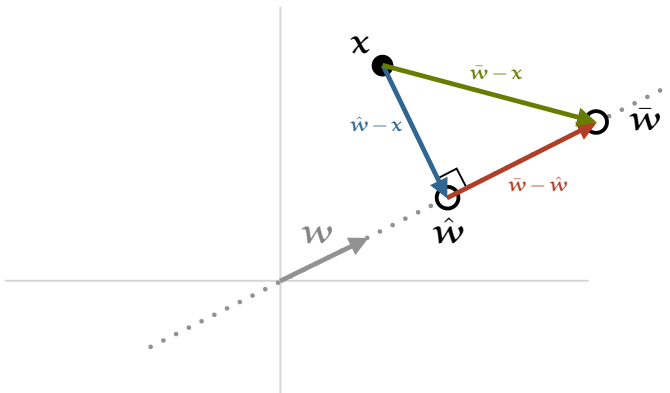
In higher dimensions, however, such physical intuitions will not always save us. Since the relation between orthogonality and least squares is key to understanding PCA, we will take some time to prove this properly.

Best approximation theorem (1D) Let $w, \mathbf{x} \in \mathbb{R}^n$, let W be the line of all multiples of w, and let $\hat{w}$ be the orthogonal projection of $\mathbf{x}$ onto W. Then, for any other $\bar{w}$ in W, we have

$$\text{dist}(\mathbf{x}, \hat{w}) < \text{dist}(\mathbf{x}, \bar{w})$$

where $\text{dist}(\mathbf{a}, \mathbf{b})$ denotes the Euclidean distance $\|\mathbf{a} - \mathbf{b}\|$.

Proof. (*Adapted from Thm. 9 Ch. 7 in Lay (1994)*) Note that $\mathbf{a} - \mathbf{b}$ is the vector that points from the tip of $\mathbf{a}$ to the bottom of $\mathbf{b}$. We can draw three vectors $\bar{w} - \mathbf{x}$, $\hat{w} - \mathbf{x}$ and $\bar{w} - \hat{w}$ as follows:



By basic vector addition, we know that

$$\bar{w} - \mathbf{x} = \hat{w} - \mathbf{x} + \bar{w} - \hat{w},$$

so the three vectors form a triangle (when we arrange them as shown in the picture).

We also know, by construction, that $\hat{w} - x$ is orthogonal to $\bar{w} - \hat{w}$, so **the triangle is right-angled**.

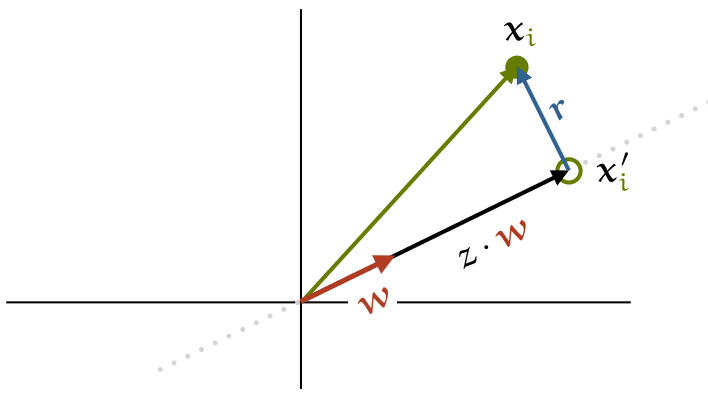
Since we have a right angled triangle, the Pythagorean theorem tells us that the lengths of the sides of the triangles are related by

$$\text{dist}(x, \bar{w})^2 = \text{dist}(\hat{w}, x)^2 + \text{dist}(\bar{w}, \hat{w})^2 .$$

Since $\text{dist}(\bar{w}, \hat{w}) > 0$ (because $\bar{w}$ and $\hat{w}$ are not the same point), we know that $\text{dist}(x, \bar{w})$ must be strictly larger than $\text{dist}(\hat{w}, x)$. □

This result generalizes easily to any linear subspace W spanned by a given set of vectors. We will show a more general proof in Chapter 4. This principle, the orthogonal projection as the best approximation is at the heart of a lot of optimization problems.

So, the best reconstruction x_i' of the point x_i on the line defined by $z \cdot w$ (however we choose w) is the orthogonal projection of x_i onto w. So how do we compute an orthogonal projection? Let's look at what we have:



We've projected x_i down onto w and we've given the vector from the projection to the original the name r. By vector addition we know that $zw + r = x_i$, so $r = x_i - zw$.

Two vectors are orthogonal if their dot product is zero, so we're looking for a z such that $zw^T\mathbf{r} = 0$, or equivalently, $w^T\mathbf{r} = 0$. We rewrite

$$0 = w^T\mathbf{r} = w^T(\mathbf{x}_i - zw) = w^T\mathbf{x}_i - zw^Tw.$$

This gives us $z = w^T\mathbf{x}_i / w^Tw$. And, since we'd already defined w to be a unit vector (so $w^Tw = 1$), we get $z = w^T\mathbf{x}_i$.

Let's retrace our steps. We had two weight vectors: v to encode $\mathbf{x}_i$ into the single number z_i, and w to decode $\mathbf{x}_i$ as $z_i w$. We've now seen that for any given w, the best choice of z_i is $w^T\mathbf{x}_i$. In other words, **we can set v equal to w and use it to encode and to decode**.

> *Note that an important requirement for this result (and its generalizations coming up) is that w is a unit vector.*

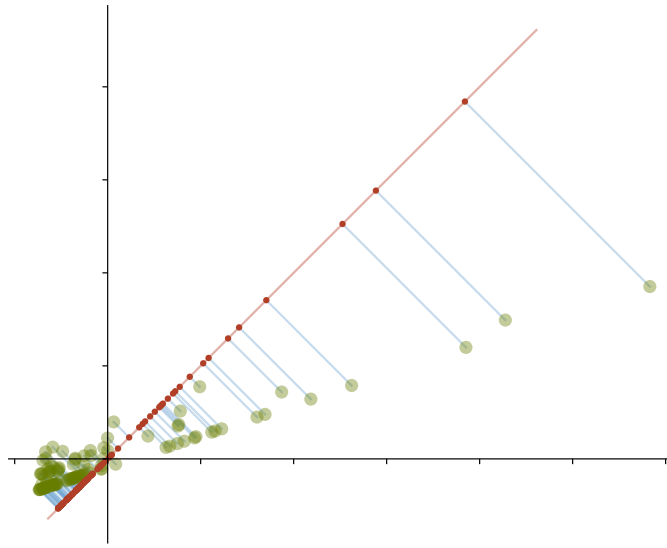
So, after all that, we can finally state precisely what we're looking for. Given w our reconstruction is $\mathbf{x}_i' = z_i \cdot w = w^T\mathbf{x}_i \cdot w$. This means we can state our goal as the following constrained optimization problem:

$$\operatorname*{argmin}_w \sum_i \|w^T\mathbf{x}_i \cdot w - \mathbf{x}_i\|$$
$$\text{such that } w^Tw = 1.$$

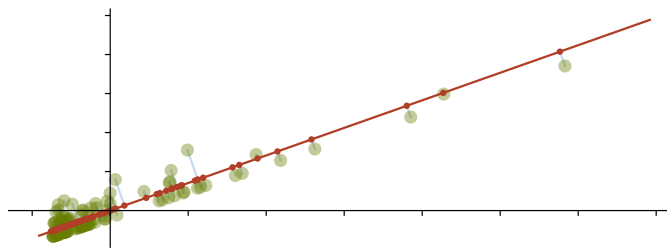
How do we solve this? This is a simple problem and there are fast ways to solve it exactly. But we've done a lot of math already, and it's time to show you some results, so we'll just solve this by gradient descent for now. Basic gradient descent doesn't include constraints, but in simple cases like these, we can use *projected* gradient descent: after each gradient update, we project the parameters back to the subset of parameter space that satisfies the constraint (in this case simply by dividing w by its length).

> *If you don't know how gradient descent works, you can just imagine a procedure that starts with a random choice for w and takes small steps in the direction that the function above decreases the most.*

We start by initializing w to some arbitrary direction. Here's what the projections of the income data onto that w look like.



The (squared) sum of the lengths of the blue lines is what we want to minimize. Clearly, there are better options than this choice of w. After a few iterations of gradient descent, this is what we end up with.



You can think of the blue lines of the reconstruction error as pulling on the line of w and of w as pivoting on the origin.

For any dataset (of however many dimensions), there is a unique, optimal line $\boldsymbol{w}$. It's called the **first principal component**.

> *If you've read other descriptions of PCA, you may be wondering at this point why I'm not talking about maximizing the variance. This is an alternative way to define PCA. We'll discuss it in the next chapter.*

What can we say about the meaning of the elements of $\boldsymbol{w}$? Remember that it does two things: it encodes from $\mathbf{x}$ to z and it decodes from z to $\mathbf{x}'$. The encoding is a dot product: a weighted sum over the elements of $\mathbf{x}$.

In the first example of the income data, before we added the noise, the second feature was always exactly three times the first feature. In that case, we could just remember the first feature, and forget the second. That would be equivalent to encoding with the vector $(1, 0)$. The compressed representation z would be equivalent to the first feature and we could decode with $z \times (1, 3)$. Or, we could encode with $(0, 1)$ and decode with $(\frac{1}{3}, 1)$.

Why are the encoding and decoding vectors different in these cases? Because when we proved that they were the same, we assumed that they were unit vectors. Our encoding vector is a unit vector, but the corresponding decoding vector isn't. PCA provides solution for which the encoder and the decoder are the same. It takes a mixture of both features, in different proportions (in our case $1/\sqrt{10}$ and $3/\sqrt{10}$) . There are a a lot of perspectives on exactly what this mixture means. We'll illustrate the first by looking at a higher dimensional dataset.

We'll use a dataset of grayscale images of faces produced by AT&T Laboratories Cambridge called the *Olivetti dataset*.[1]
We will describe each pixel as a feature with a value between 0 (black) and 1 (white). The images are 64×64 pixels, so each image can be described as a single vector of 4096 real values.

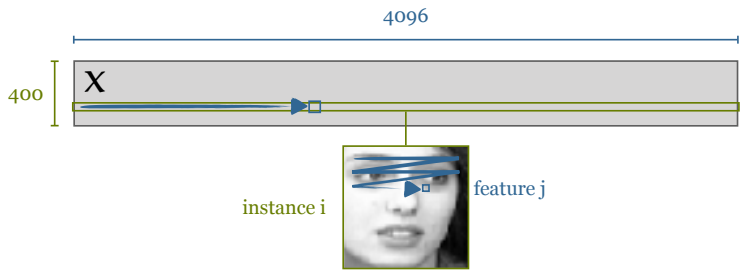
> *Note that by flattening the images into vectors we are entirely ignoring the grid structure of the features: we are not telling our*

[1] https://scikit-learn.org/stable/datasets/real_world.html#the-olivetti-faces-dataset

A small sample of images from the Olivetti data.

algorithm whether two pixels are right next to each other, or at opposite ends of the image.

The Olivetti data contains 400 images, so we end up with a data matrix $\mathbf{X}$ of 400×4096.



This is a data scientist's worst nightmare: data with many more features than instances. With so many features, the space of possible instances is vast, and we only have a tiny number to learn from. Our saving grace is that, like the income/salary example, the features in this dataset are highly *dependent*: knowing the value of one pixel allows us to say a lot about the value of other pixels.

For instance, pixels that are close together more often than not have similar values. The images are often roughly symmetric. All faces will have mostly uniform patches of skin in roughly

the same place, and so on.

In short, while our dataset is expressed in 4096 dimensions, we can probably express the same information in many fewer numbers, especially if we are willing to trade off a little accuracy for better compression.

The procedure we will use to find the first principal component for this data is exactly the same as before—search for a unit vector that minimizes the reconstruction loss—except now the instances have 4096 features, so w has 4096 dimensions. First, let's look at the reconstructions.

Reconstruction from a single principal component. Originals on the left, reconstructions on the right.

These are not very impressive yet, but to be fair, we've compressed each image into a single number, we shouldn't be surprised that there isn't much left after we reconstruct it. But that doesn't mean that 1D PCA doesn't offer us anything useful.

What we can do is look at the first principal component in data space: w is a vector with one element per pixel, so we can re-arrange it into an image and see what each element in the vector tells us about the original pixels of the data. We'll color the positive elements of w red and the negative values blue.

It looks like this:

If we think of this as the *encoding* vector, we can see a heatmap of which parts of the image the encoding looks at: the darker the red, the more the value of that pixel is added to z. The darker the blue, the more it is subtracted.

If we think of this as the *decoding* vector, we can see that the larger z is, the more of the red areas gets added to the decoded image, but the more of the blue areas get *subtracted*. That is, two red pixels are positively correlated, and a red and a blue pixel are negatively correlated. A bright red pixel and a light red pixel have the same relation as our monthly salary and quarterly income: one is (approximately) a multiple of the other.

Another interpretation of the principal component is that it places the images in the dataset on a single line, which mean it orders the images along a single direction.

To see if there's any interpretable meaning to this ordering, we can try moving along the line and decoding the points we find. We can start at the origin (the vector $\mathbf{0}$). If we decode that, we get the mean of our dataset (the so called mean face). By adding or subtracting a little bit of the principal component, we can see what happens to the face.

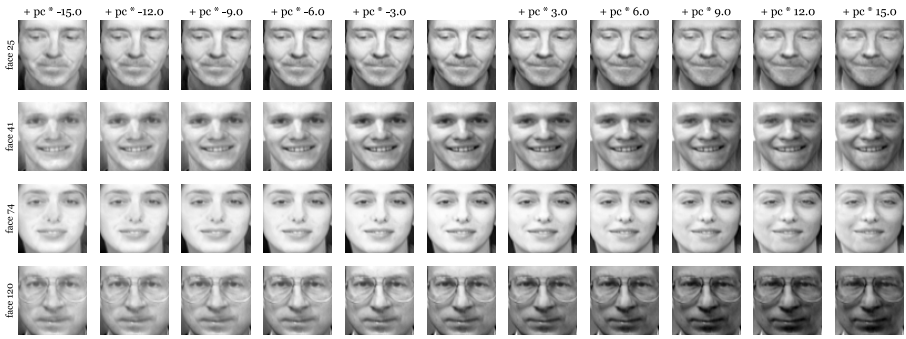
A few things are happening at once: the skin is becoming less clear, the lines in the face become more pronounced, the glasses become more pronounced and the mouth curves upward. Most of this is consistent with moving from a young subject to an old one.

We can test this on the faces in our dataset as well; our principal component is a direction in the data space, so we can start with an image from our data, and take small steps in the direction of the principal component, or in the opposite direction.

You can think of this as manipulating the single-number latent representation z by adding some small amount ϵ. If we decode such a point, we get $(z+\epsilon)\boldsymbol{w} = z\boldsymbol{w} + \epsilon\boldsymbol{w} = \mathbf{x}' + \epsilon\boldsymbol{w}$. We then just replace the reconstruction $\mathbf{x}'$ by the actual point $\mathbf{x}$. Note that

we depend on the linearity of our transformation: for nonlinear variants of PCA, this trick won't work anymore.

Here's what we get.



Note the manipulation of the mouth, in particular in face 41. As the corners of the mouth go up, the bottom lip goes from curving outward, with a shadow under the lip to curving inwards, folding under the teeth, with the shadow turning into a highlight.
Here we see the real power of PCA. While the reconstructions may not yet be much to write home about, the principal component itself allows us, using nothing but a linear transformation consisting of a single vector, to perform realistic manipulation of facial data, based on a dataset of just 400 examples.

1.3 n-Dimensional PCA

Enough playing around in a one-dimensional latent space. What if we want to improve our latent representations by giving them a little more capacity? How do we do that in a way that gives us better reconstructions, but keeps the meaningful directions in the latent space?

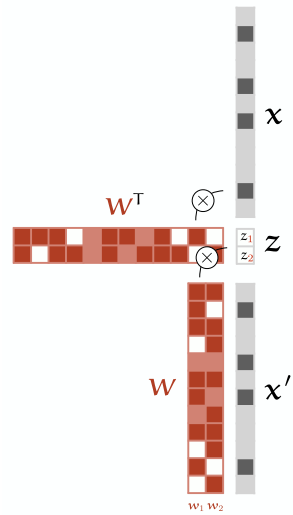
Let's start by updating our notation. $\mathbf{x}_i$ is still the same vector, row i in our data matrix $\mathbf{X}$, containing m elements, as many as we have features. $\boldsymbol{z}_i$ is now also a vector (note the boldface). $\boldsymbol{z}_i$ has k elements, where k is the number of latent features, a parameter we set. In the example above, $k = 1$. We'll drop the i subscript to clarify the notation.

Let's say we set $k = 2$. If we stick to the rules we've followed so far—linear transformations by unit vectors—we end up with the following task: find two unit vectors w_1 and w_2 and define $z = (z_1, z_2)$ by $z_1 = x^T w_1$ and $z_2 = x^T w_2$. Each latent vector gives us a reconstruction of x. We sum these together to get our complete reconstruction.

We can combine the two vectors w_1 and w_2 in a single matrix W (as its columns) and write

$$z = W^T x$$
$$x' = Wz$$

Or, in diagram form:



This would already work fine as a dimensionality reduction method. You can think of this as an autoencoder, if you're familiar with those. However, we can add one more rule to improve our reduced representation. **We will require that w_2 is orthogonal to w_1.**

This decision is important, and has many useful consequences. We'll save those for later. For now, we'll just take it at face value.

In general, each component w_r we add should be orthogonal to all components before it: for $k = 3$ we add another unit vector w_3, which should be orthogonal to both w_1 and w_3.

We can summarize these constraints neatly in one matrix equation: the matrix W, whose columns are our w vectors, should satisfy:

$$W^T W = I$$

where I is the $k \times k$ identity matrix. This equation combines both of our constraints: unit vectors, and mutually orthogonal vectors. On the diagonal of $W^T W$, we get the dot product of every column of W with itself (which should be 1 so that it is a unit vector) and off the diagonal we get the dot product of every column of W with every other column (which should be 0, so that they are orthogonal).

How do we find our W? The objective function remains the same: the sum of squared distances between the instances x and their reconstructions x'. To satisfy the constraints, we can proceed in two different ways. We'll call these the *combined* problem and the *iterative* problem.

The **combined** problem is simply to add the matrix constraint above and stick it into our optimization function. This gives us

$$\begin{aligned} &\underset{W}{\operatorname{argmin}} \sum_x \|WW^T x - x\|^2 \\ &\text{such that } W^T W = I \end{aligned}$$

The **iterative** problem defines optima for the vectors w_r in sequence. We use the same one-dimensional approach as before, and we find the principal components one after the other. Each step we add the constraint that the next principal component should be orthogonal to all the ones we've already found.

To put it more formally, we choose each $w_1, \ldots, w_k$ in sequence by optimizing

$$w_r = \begin{cases} \underset{w}{\operatorname{argmin}} \sum_x \|x^T w \times w - x\|^2 \\ \quad \text{such that } w^T w = 1, \\ \quad \text{and } w^T w_i = 0 \text{ for } i \in [1 \ldots r-1] \end{cases}$$

These approaches are very similar. In fact, they're sometimes confused as equivalent in the literature. Let's look how they relate in detail.

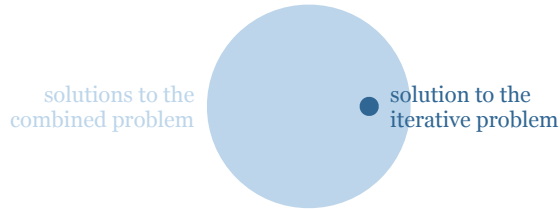
The vector w_1 defined in the iterative problem, is the same vector we found in the one-dimensional setting above: the first principal component. If the two problems are equivalent (i.e. they have the same set of solutions), this vector should always be one of the columns of W in the combined problem.

how that this isn't guaranteed, we can look at the case where $k = m$. That is, we use as many latent features as we have features in our data. In this case, the first vector w returned by the iterative approach is still the first principal component, as we've defined it above. However, for the combined approach, we can set $W = I$ for a perfect solution: clearly the columns of I are orthogonal unit vectors, and $WW^T x - x = II^T x - x = x - x = 0$, so the solution is optimal.

In short, a solution to the combined problem may not be a solution to the iterated problem. What about the other way around, does solving the iterated problem always give us a solution to the combined problem? Certainly the vectors returned are always mutually orthogonal unit vectors, so the constraint is satisfied. Do we also reach a minimum? It turns out that we do.

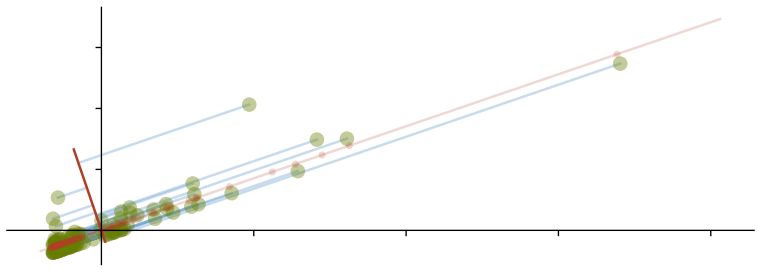
Optimality of the iterative approach A solution to the iterative problem is also a solution to the combined problem.

We will prove this in the second chapter. For now, you'll have to take my word for it. The combined problem has a large set of solutions, and the iterative approach provides a kind of unique point of reference within that space.



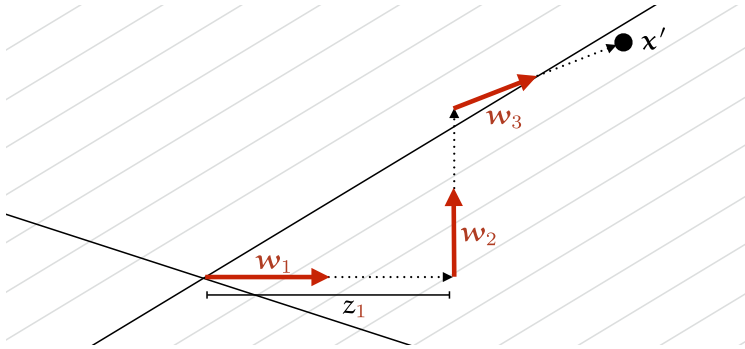
We can say more about this later, but for now we will equate the iterative solution with PCA: the W which not only minimizes the reconstruction error as a whole, but also each column of W minimizes the reconstruction error in isolation, constrained to the subspace orthogonal to the preceding columns. The combined problem does not give us the principal components.

Using the iterative approach, and solving it by projective gradient descent, we can have a look at what the other principal components look like. Let's start with our income data, and derive the second principal component.



This is a bit of an open door: in two dimensions, there is only one direction orthogonal to the first principal component. Still, plotting both components like this gives some indication of what the second component is doing. The first component captures the main difference between the people in our dataset: roughly their monthly salary. The second component captures whatever is left; all the noise we introduced like end of year bonuses and alternative sources of income.

Note how PCA reconstructs the original data points. Given the components $w_1, \ldots, w_k$, we represent each $\mathbf{x}$ as a weighted sum



Reconstruction by principal components in the case where $k = 3$. Each latent variable z_r tells us how much of the r-th principal component to add to our reconstruction.

over these vectors where the latent features $z_1, \ldots, z_k$ are the weights:

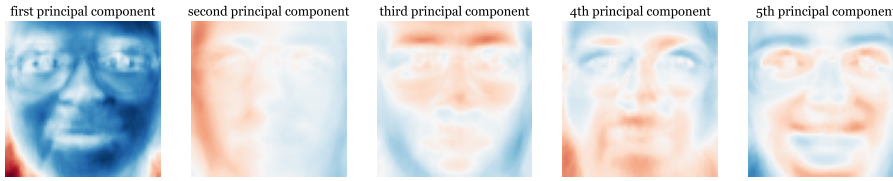
$$x' = z_1 w_1 + z_2 w_2 + \ldots + z_k w_k$$

That's it for the income dataset. We've reached $k = m$, so we can go no further.

Let's turn to the dataset of faces, where there are many more principal components to explore.

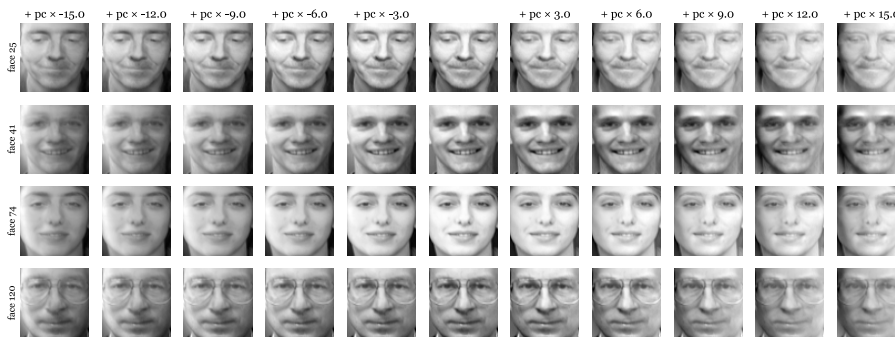
If we compute the first 30 principal components, we get the following reconstructions.

You can still tell the originals from the reconstructions, but many of the salient features now survive the compression process: the direction of the gaze, the main proportions of the face, the basic lighting, and so on. By looking at the first five principal components, we can see how this is done.

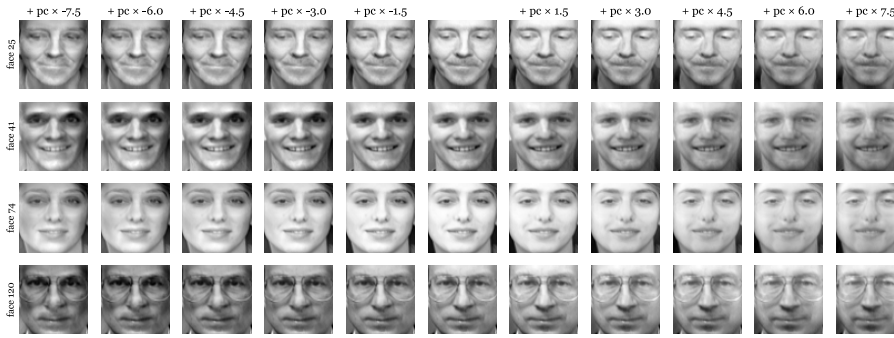


The first one we've seen already. It's flipped around, this time, with the blues and reds reversed, but it defines the same line in space. Note that the magnitude of the higher PCs is much lower: the first principal component does most of the work of putting the image in the right region of space, and the more PCs we add, the more they fine-tune the details.

The second PC captures mostly lighting information. Adding it to a picture adds to the left side of the image, and subtracts from the right side. We can see this by applying it to some faces from the data.



The third PC does the same thing, but for top-to-bottom lighting changes. The fourth is a bit more subtle. It's quite challenging to tell from the plot above what the effect is. Here's what we see when we apply it to some faces.



The PC seems to capture a lot of the facial features we associate with gender. The faces on the left look decidedly more ”female”, and the faces on the right more male. It's a far cry from the face manipulation methods that are currently popular, but considering that we have only 400 examples, we are only allowed a linear transformation, and that the method originated in 1901 (Pearson, 1901), it's not bad.

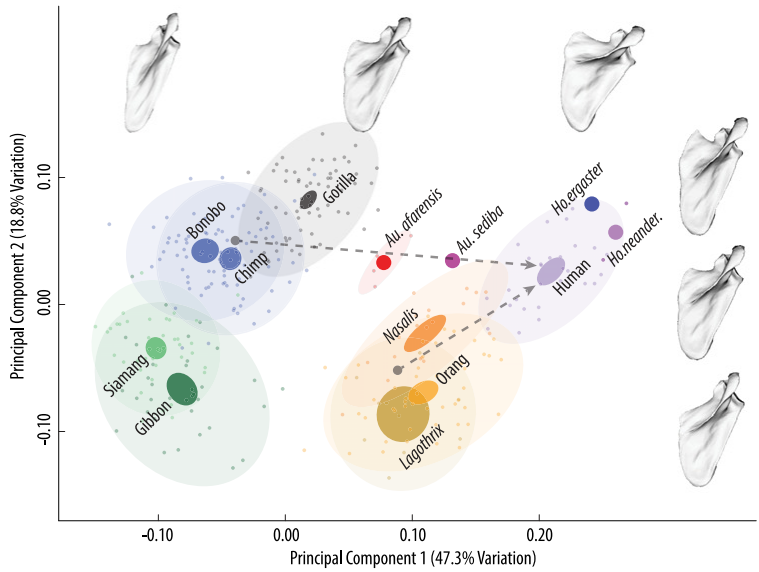
1.4 Applications of PCA

Before we finish up, let's look at two examples of how PCA is used in research.

Let's start with a problem which crops up often in the study of fossils and other bones. A trained anatomist can look at, say, a shoulder bone fossil, and tell instantly whether it belongs to a chimpanzee (which is not very rare) or an early ancestor of humans (which is extremely rare). Unfortunately such judgements are usually based on a kind of unconscious instinct, shaped by years of experience, which makes it hard to back it up scientifically. ”This is is Hominid fossil, because it looks like one to me,” isn't a very rigorous argument.

PCA is often used to turn such a snap judgement into a more rigorous analysis. We take a bunch of bones that are entirely indistinguishable to the layperson, and we measure a bunch of features, like the distances between various parts on the bone. We then apply PCA and plot the first two principal components.

Here is what such a scatterplot looks like for a collection of scapulae (shoulder bones) of various great apes and hominids.

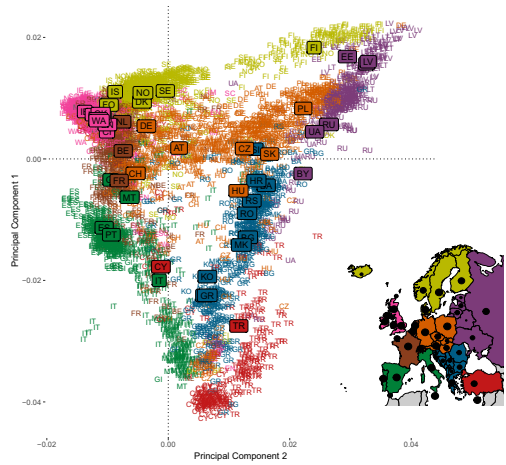


This particular figure is from Young et al. (2015) (reproduced with permission) which is available online, but the literature is full of images like these. Here, the authors took scans of about 350 scapulae. We can clearly see different species forming separate clusters. If we find a new scapula, we can simply measure it, project it to the first two principal components, and show that it ends up among the Homo Ergaster to prove that our find is special. What's more, not only can we tell the Hominin fossils apart, we see that they seem to lie on a straight line from chimpanzees to modern humans, giving a clue as to how human evolution progressed.

> *This analysis is based on full 3D scans of the bones, but it also works if you measure a number of features by hand.*

Let's add one final example, to really hammer home the magic of PCA. When large-scale genome databases began to be gathered, one of the first things researchers did was to perform principal component analyses. To do so, all you need to do is to extract some features from the DNA sequence. This is easy enough to

do: for instance, you can identify a few hundred thousand important genetic markers, and use the presence or absence of each as a binary feature.



A PCA analysis of a European population, similar to the one by Novembre et al. (2008), colored by ancestry. Adapted from Gilbert et al. (2022).

This was first done by Novembre et al. (2008) for a dataset of 1387 Europeans, measuring about 200 000 genetic markers. They applied PCA, and plotted the first two principal components. They then colored the instances by the person's ancestral country of origin (the country of origin of the grandparents if available, otherwise the person's own country of origin). What they saw was something like a blurry picture of Europe.

The first principal component corresponds roughly to how far north the person lives (or their ancestors did) and the second principal component to how far east they live. This means that a scatterplot shows up as a fuzzy map of Europe. If we sent a thousand DNA samples off to aliens on the other side of the galaxy, they could work out a rough image of the geography of Earth.

Note that while this result is impressive, it's easy to misinterpret what this means. These to principal components together explain no more than a few percent of the variance. That is, while

> *we can make a crude prediction of a person's origin from their DNA, we can predict almost nothing about their DNA from their origin. We saw something similar in the example of the Olivetti data: we could predict people's gender and age from the first few principal components, but when we reconstructed the photograph from just this information, the result removed almost all relevant details about the subject of the photograph.*

Wrapping up, what have we learned so far? We have defined PCA as an iterative optimization problem designed to compress high dimensional data into fewer dimensions, and to minimize the resulting reconstruction loss. We've shown one simple way to find this solution, a laborious and inaccurate way, but enough to get the basic idea across. We then looked at various practical uses of PCA: analyzing human faces, human fossils, and human DNA. We showed that in many cases, PCA magically teases out high-level semantic features hidden in the data: the species of the fossil, the location of the subject in Europe, or the subject's age.

What we haven't discussed fully, is where this magical property comes from. We've shown that it isn't *just* the compression objective, since optimizing just that in one single optimization doesn't lead to the PCA solution. Among the set of all solutions that minimize the reconstruction error, the PCA solution takes a special place. Why that's the case, and why it this should emerge form optimizing the principal components one by one, greedily if you will, instead of all together, we will discuss in the next chapter. To do so, we'll need to dig into the subject of *eigenvectors*, the underlying force behind almost everything that is magical about linear algebra.

CHAPTER 2 · EIGENVECTORS AND EIGENVALUES

In the first chapter, we took what I think is the most intuitive route to defining PCA: framing the method in terms of *reconstruction error*. The solution method we used wasn't very efficient or stable, and some parts of the "why" question were left unresolved, but we answered the "how" question well enough to show the method in action and hopefully convince you that it's worth digging a little deeper.

We'll start by tying up one of the loose ends from the last chapter. There, we defined PCA in terms of reconstruction error, but most other explanations instead define it in terms of **variance maximization**.

I started with the reconstruction error, since it requires fewer assumptions, and the required assumptions feel more intuitive. However, to explain the details of what happens under the hood, the variance perspective is more helpful, so we'll start by adding that to our toolbelt.

2.1 Minimal reconstruction error is maximal variance

In the previous chapter, we defined PCA as a solution to the following problem: for a mean-centered set of instances $\mathbf{x}_i$ find a sequence of k unit vectors $\mathbf{w}_1, \ldots, \mathbf{w}_k$ where each unit vector is defined by

$$\mathbf{w}_r = \begin{cases} \underset{\mathbf{w}}{\operatorname{argmax}} \sum_i \|\mathbf{x}_i{}^\top \mathbf{w}\mathbf{w} - \mathbf{x}_i\|^2 & \\ \quad \text{such that } \mathbf{w}^\top \mathbf{w} = 1 & \text{(a)} \\ \quad \text{and } \mathbf{w}^\top \mathbf{w}_j = 0 \text{ for } j \in [1 \ldots r-1] & \text{(b)} \end{cases}$$

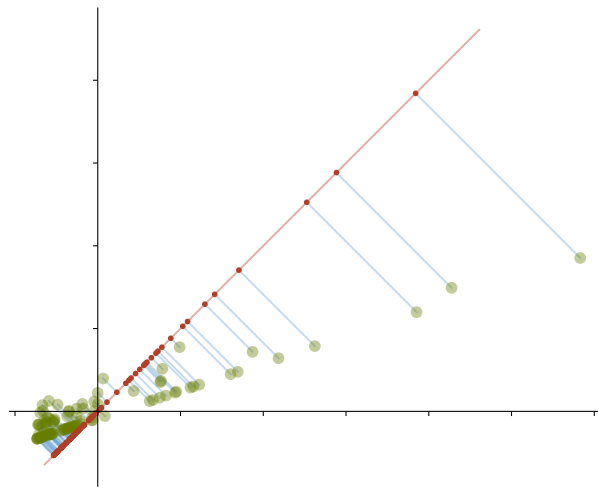
That is, each unit vector defines an approximation of $\mathbf{x}_i$, and the objective is to keep the distance between the reconstruction and

the original instance as small as possible, while ensuring that (a) the vector is a unit vector, and (b) the vector is orthogonal to all preceding vectors.

The variance maximization objective takes the same form, but instead of minimizing the reconstruction error, it chooses w to *maximize* the variance of the data projected onto w.

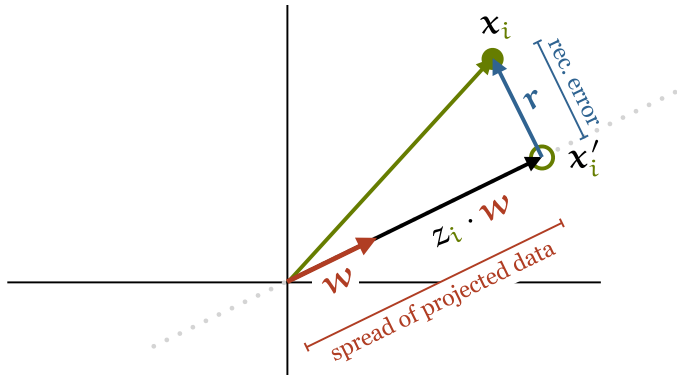
As in the previous chapter, we'll start with the one-dimensional case. We will choose a single unit vector w and project the data onto it orthogonally. That means that for every instance $\mathbf{x}_i$, we get a single number z_i which is defined by $\mathbf{x}_i{}^T w$.

This time, our objective is to choose w so that the variance among the set of z_i is maximized. That means that in this image



we want to choose w so that the red points are spread out as widely as possible.

Why is this the same as choosing the w that minimizes the reconstruction error? It's easy to see this if we draw a a diagram for a single instance.



Note that w is a unit vector, so for the length of the bottom edge of the triangle we have $\|x'_i\| = \|wz_i\| = z_i$.

By the Pythagoras, $\|x_i\|^2 = \|r\|^2 + {z_i}^2$. The vector x_i remains constant, since that is given by the data. The only thing we change is the direction of the vector w. If we change that to decrease the reconstruction error $\|r\|$, the distance z_i must decrease. The sum of the squares of all z_i's is the variance of the data.

Thus, the first principal component is the vector w for which the data has maximum variance when projected onto w. For the sake of completeness, let's work this into a proper proof. There's some technical stuff coming up, so we had better get into a more formal mindset.

Equivalence of error and variance optimization. The vector w that minimizes the reconstruction error among the reconstructed instances x'_i, maximizes the variance among the projections z_i.

Proof. The maximal variance objective can be stated as

$$\operatorname*{argmax}_w \frac{1}{n} \sum_i (\bar{z} - z_i)^2 .$$

The objective inside the argmax is simply the definition of variance. We can drop the constant multiplier $1/n$, since it doesn't

affect where the maximum is. We can show that $\bar{z}$, the mean of the projections is 0:

$$\bar{z} = \frac{1}{n}\sum_i w^T x_i = w^T\left(\frac{1}{n}\sum_i x_i\right) = w^T \mathbf{0}\,.$$

The last step follows from the assumption that the data is mean-centered.

Thus our objective simplifies to

$$\operatorname*{argmax}_w \sum_i {z_i}^2\,.$$

From our diagram above, we know that $\|x_i\|^2 = \|r\|^2 + {z_i}^2$, or, equivalently

$$\begin{aligned}\|x_i\|^2 &= \|x_i - x_i'\|^2 + {z_i}^2\\ {z_i}^2 &= \|x_i\|^2 - \|x_i - x_i'\|^2\,.\end{aligned}$$

Filling this in to the optimization objective, we get

$$\operatorname*{argmax}_w \sum_i \|x_i\|^2 - \|x_i - {x_i}'\|^2\,.$$

where $\|x_i\|^2$ is a constant we can remove without affecting the maximum and removing the minus turns the maximum into a minimum. Thus, we end up with

$$\operatorname*{argmin}_w \sum_i \|x_i - x_i'\|^2$$

which is the objective for minimizing the reconstruction loss. □

The rest of the procedure is the same as before. Once we've chosen w_1 to maximize the variance, we choose w_2 to maximize the

variance and to be orthogonal to w_1, we choose w_3 to maximize the variance and to be orthogonal to w_1 and to w_2 and so on.

In the previous chapter, we also defined a **combined problem**, which combined all the vectors together in one optimization objective. We can work out an equivalent for the variance perspective (the proof is in Section B.2 of the appendix).

Equivalence of combined optimization. The combined problem for reconstruction error minimization.

$$\begin{aligned} &\underset{W}{\operatorname{argmin}} \sum_i \|x_i - x_i'\| \\ &\text{such that } W^T W = I \end{aligned}$$

is equivalent to the following variance maximization problem

$$\begin{aligned} &\underset{W}{\operatorname{argmax}} \sum_{i,r} {z_{ir}}^2 \quad \text{with } z_{ir} = {w_r}^T x_i \\ &\text{such that } W^T W = I\,. \end{aligned}$$

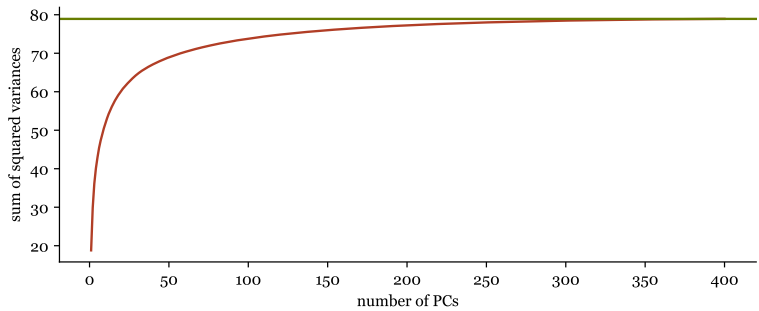
If we want to optimize a matrix W with mutually orthogonal unit vectors for columns in one go, then maximizing the sum of the variances in all latent directions is equivalent to minimizing the reconstruction loss defined in the combined approach

One consequence is that since $W = I$ was a solution of the error minimization problem (with $k = m$) it must be a solution for the variance maximization problem as well.

This tells us something interesting. If we set $W = I$, then $\sum_{i,r} {z_{ir}}^2$ is just the sum of all the variances of all the features in our data. For all solutions at $k = m$, this must be the total variance among the latent features. For solutions to the problem at some $k < m$ the variance is less than or equal to this value. Each principal component adds a little variance to the total sum of variance, and when we have enough principal components to reconstruct the data perfectly, the sum of the variances along the

principal components equals the sum total variance in the data.

Here's what that looks like for the Olivetti faces data.



We plot the sum total of the variances of the data as a green vertical line and the sum total of the PCA solution for increasing k as a red line.

> *In this case, we have data that is wider than it is tall, so we reach the ceiling before* $k = m$*, when* $k = n$*. This is to do with the* rank *of the matrix* $\mathbf{X}$*. We'll shed some light on this in Chapter 4.*

This shows that we can think of the total variance in all directions in the data space as an additive quantity in the data, which we can call its *information content*. The data contains a certain amount of information and the more latent dimensions we allow, the more of that information we retain.

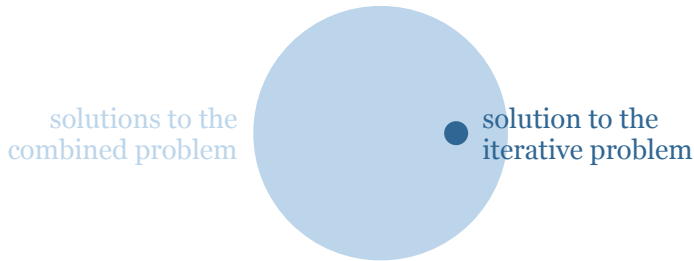
> *Don't read too much into the word* information *here. It's just a convenient phrase to use. We could relate it to formal notions of information content, like Shannon's, but not without some serious extra work.*

If we keep all the dimensions, we retain all the information. If we start with the first principal component and add components one by one, we add to the total of squared variances in a sum that eventually sums up to the total of squared variances in the data (much like the reconstruction loss eventually goes to zero). Both perspectives—retaining variance and minimizing reconstruction

loss—are formalizations of the same principle; that *we want to minimize the amount of information lost in the projection*.

2.2 Eigenvectors

Let's return to this image.



These solutions are the same for both perspectives: variance maximization and reconstruction error minimization. We have two unresolved questions about this image.

First, how is it that the solution to the iterative problem (the PCA solution) reaches the same optimum as the solutions to the combined approach? Take a second to consider how strange this is. Solving the iterative problem is a greedy search: once we have chosen w_1 we can't ever go back. The process for the combined approach solves all the vectors in sync. How is it that this ability to tune the vectors in concert doesn't give us any advantage in the optimum we find?

The second question, and the question we will answer first, is what is the meaning of the PCA solution among the combined solutions? How can we *characterize* this solution?

The answer to the second question can be summarized in one phrase:

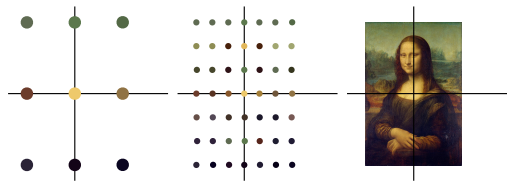
> The principal components are *eigenvectors*.

Depending on your background, this will raise one of two questions. *The eigenvectors of what?* or, more simply *What are eigenvectors?* Let's start with the second question, and work our way back to the first.

2.2.1 What are eigenvectors?

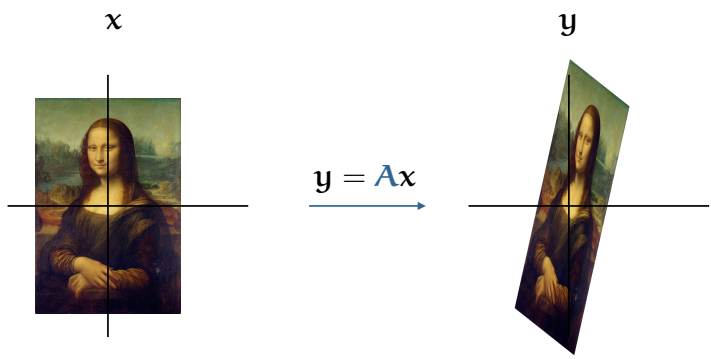
The most common, and probably the most intuitive way to think about matrices is *as transformations of points in space*. If we have some vector $\mathbf{x}$ and we multiply it by a matrix $\mathbf{A}$, we get a new point $\mathbf{y} = \mathbf{A}\mathbf{x}$. If $\mathbf{A}$ is square, then $\mathbf{x}$ and $\mathbf{y}$ are in the same space.

A good way to visualize this is by *domain coloring*. We take a large number of points, arranged in a grid, and we color them by some image. This could be a simple color gradient, but we can also choose a photograph or some other image. Following Wikipedia's example, we'll use a picture of the Mona Lisa.



An increasingly fine-grained domain coloring using the Mona Lisa.

If we apply the transformation $\mathbf{A}$ to each of these points, we can tell what effect the matrix has on this space.



All the points are mapped to a new position by $\mathbf{A}$ and poor Lisa

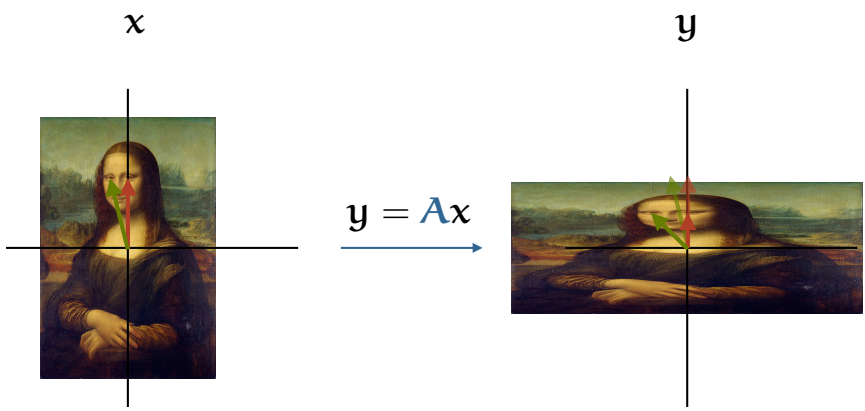
ends up squished and stretched in various directions. Transformations expressible in a matrix are linear transformations. These are the transformations for which a line in the original image is still a line in the transformed image. This means that we can rotate, stretch, squish and flip the image in any direction we like, but we can't warp, bend or tear it.

In this language of transformation, we can very naturally define what eigenvectors are. The **eigenvectors of a square matrix** $\mathbf{A}$ are defined as those vectors (i.e. points in the image) for which the *direction* doesn't change under transformation by $\mathbf{A}$.

It's simplest to see what this looks like for a *diagonal matrix*. For instance in the transformation

$$\mathbf{y} = \begin{pmatrix} 2 & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \mathbf{x}$$

the matrix acts independently on the first and second dimensions, squishing one, and stretching the other.



In this image we've also drawn two vectors: one to the middle of Mona Lisa's left eye, and one to middle of the right. Since Leonardo put the right eye dead center in the painting (not by accident, I imagine), the red vector shrinks, but doesn't change direction. The green vector is affected by both the squishing

and the stretching, so its direction and magnitude both change. Hence, the red vector is an eigenvector, and the green vector isn't.

In a diagonal matrix, the eigenvectors are always the vectors that point in the same directions as the axes, so they're easy to identify. In general square matrices, finding the eigenvectors is more tricky.

> *See if you can find an eigenvector for the first transformation, shown above. One trick is to start with a random vector, and if it changes direction, transplant it back to the untransformed image and iterate until the vector doesn't change anymore.*

Formally, a vector $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$ if the following holds for some scalar λ:

$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v}\,.$$

This is just a symbolic version of what we said in words above: if $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$, then transforming it changes its magnitude but not its direction, which is the same as saying we can multiply it by a scalar instead of by a matrix. The scalar corresponding to the eigenvector, in this case λ is called an **eigenvalue** of the matrix.

> *It's not at all clear from the definition why these vectors should be meaningful or special. For now, just trust me that eigenvectors are worth knowing about.*

To build your intuition, consider for a second the question of whether a pure rotation matrix has eigenvectors (the zero vector doesn't count). It shouldn't take long to convince yourself that a rotation in two dimensions doesn't. At least, not usually. There are two exceptions: rotating by 360 degrees (which is just $\mathbf{I}$) and rotating by 180 degrees. In both cases, every vector is an eigenvector. In the first with eigenvalue 1, and in the second with eigenvalue -1.

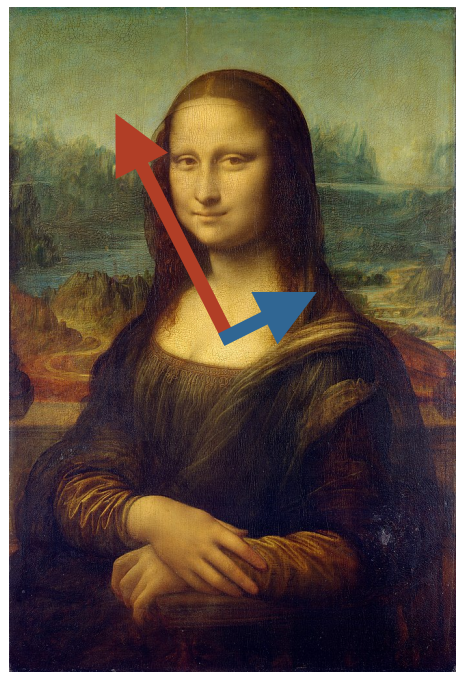
In three dimensions, rotation is a different story: try pointing straight up in the air and spinning around. The direction of your arm doesn't change as you rotate, so your arm is an

eigenvector. Your nose does change direction, so your nose is not an eigenvector.

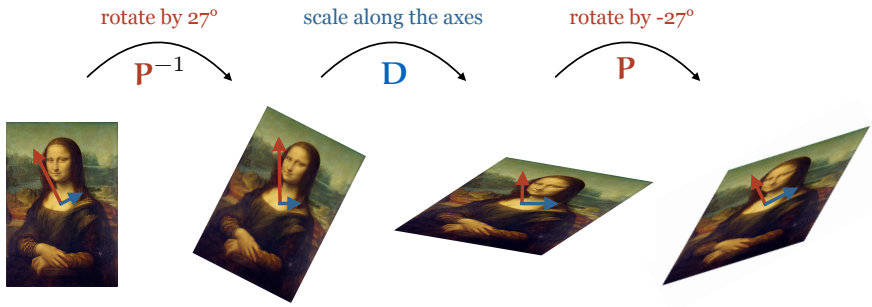
We saw that when a matrix is diagonal, its eigenvectors are *aligned with the axes*, so they're easy to find. For other matrices, we need to do some more work. One trick is to simply transform the matrix so that the eigenvectors are aligned with the axes, and then to transform it back again.

This is easiest to understand if we work backwards: given some eigenvectors, find the transformation for them.

Here are some vectors drawn on top of the Mona Lisa. What is the transformation for which these vectors do not change direction?



We have made things easy for ourselves by making the eigenvectors orthogonal. This means we can *rotate* the image to put the eigenvectors on the axes. We can then do any stretching and squishing we like along the axes, and rotate the image back.



Note how, comparing the first and the last image, the red and blue vector change their shape, but not their direction.

Any change we make to the direction of the vectors in the first step is reversed exactly in the last step: the only permanent change to any *directions* is made in the middle step. Therefore, those vectors which don't change direction in the middle step, never change direction at all, and must therefore be eigenvectors. These are the vectors that align with the axes in the middle figure, since the middle transformation is along the axes. Therefore, the vectors which are mapped to align with the axes by the first step are the eigenvectors of the complete transformation *consisting of all three steps*.

All three of these steps can be expressed as matrix transformations. We will call the matrix for the last step $\mathbf{P}$. The first step is then $\mathbf{P}^{-1}$, since it needs to undo $\mathbf{P}$ exactly. This means that we must require that $\mathbf{P}$ is invertible.

> *We could also call the first step* $\mathbf{P}$ *and the last step* $\mathbf{P}^{-1}$*, of course, but it works out more neatly if we call the last step* $\mathbf{P}$.

In the example, $\mathbf{P}$ is a rotation matrix, but the principle generalizes to all invertable matrices.

We have seen the approach for the middle step already: it is expressed by a diagonal matrix, which we'll call $\mathbf{D}$. The composition of these three steps is our transformation matrix $\mathbf{A}$.

We compose transformations by multiplying their matrices, so we have:

$$\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}$$

Note that for a transformation $\mathbf{A}\boldsymbol{x} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}\boldsymbol{x}$, the rightmost matrix is applied first, so the direction of the matrices is reversed from the steps in image above.

Now we can work backwards: For any given matrix $\mathbf{A}$, *if* we can decompose it in this way, as the product of some invertible matrix, some diagonal matrix and the inverse of the invertable matrix, then we know three things:

- The eigenvectors of $\mathbf{A}$ are the vectors that are mapped by $\mathbf{P}^{-1}$ to align to the axes. Any change of direction introduced by $\mathbf{P}^{-1}$ is undone by $\mathbf{P}$, so the only vectors whose direction is unchanged are those mapped to the eigenvectors of $\mathbf{D}$ (i.e. to the axes).

- The eigenvectors are also those vectors which to which axis-aligned vectors are transformed by $\mathbf{P}$.

- The eigen*values* of $\mathbf{A}$ are the elements along the diagonal of $\mathbf{D}$. Any change of magnitude introduced by $\mathbf{P}^{-1}$ is undone by $\mathbf{P}$, so only the changes made by $\mathbf{D}$ remain. An eigenvector mapped to axis i by $\mathbf{P}$ is scaled by D_{ii}, which is therefore the corresponding eigenvalue.

For a given matrix $\mathbf{A}$, finding a diagonal matrix $\mathbf{D}$ and an invertible matrix $\mathbf{P}$ so that $\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}$ is called a **diagonalization** or an **eigendecomposition** of $\mathbf{A}$.

So, given a diagonalization of $\mathbf{A}$, which are the eigenvectors? We use the second bullet point above. If we take an axis-aligned vector and transform it by $\mathbf{P}$, the result is an eigenvector.

We have one eigenvalue per axis, so we'll look for one eigenvector for each. For the vectors to feed to the transformation, we can just take axis-aligned unit vectors (also known as one-hot vectors). Each will transform to an eigenvector. We can do the transformation for all vectors in one go by concatenating the vectors as the columns of one matrix. For the unit vectors this

simply results in the identity matrix $\mathbf{I}$, and for the eigenvectors, this results in a matrix we will call E. So we are looking the matrix E for which

$$\mathbf{P}\mathbf{I} = \mathsf{E}\,.$$

Removing $\mathbf{I}$ tells us that the eigenvectors we are looking for are simply the columns of $\mathbf{P}$.

2.2.2 The spectral theorem

Note that we were careful above to say *if* $\mathbf{A}$ can be diagonalized. Not all square matrices can be diagonalized. A theorem showing if and when a particular class of transformations can be diagonalized is called a *spectral theorem*.

> *The set of eigenvalues of a matrix is sometimes called its spectrum, so methods and results using these principles often use the adjective* spectral. *For instance, we have spectral clustering, spectral graph theory and spectral ODE solvers.*

There are many spectral theorems, but we'll only need the simplest. The spectral theorem for *symmetric matrices*. A symmetric matrix is a square matrix $\mathbf{A}$ which is symmetric across the diagonal. That is, it has $A_{ij} = A_{ji}$, or equivalently $\mathbf{A}^T = \mathbf{A}$. We'll call this *the* spectral theorem in the context of this book.

To state the theorem, we first need to define **orthogonal matrices**. These are square matrices in which all columns are mutually orthogonal unit vectors.

This should sound familiar, it's the constraint we placed on our principal components in the previous chapter. In the combined problem, the constraint $\mathbf{W}^T\mathbf{W} = \mathbf{I}$ is simply a requirement that the matrix $\mathbf{W}$ be orthogonal.

Why? Remember that matrix multiplication is just a collection of all dot products of rows on the left with columns on the right, so in this case all columns of $\mathbf{W}$ with all other columns of $\mathbf{W}$. On the diagonal of $\mathbf{W}^T\mathbf{W}$, we find all dot products of columns of $\mathbf{W}$ with themselves, which are all 1, because they are all unit vectors. Off the diagonal we find all dot products of all columns with other columns. These are all zero, because they

are all mutually orthogonal.

> *Geometrically, orthogonal matrices represent those transformations that do not change the magnitude of any vector: that is, rotations and reflections.*

A very useful property of orthogonal matrices is that their inverse is equal to their transpose: $W^{-1} = W^T$. This follows directly from the fact that $W^T W = \mathbf{I}$, because the inverse of W is defined as a matrix W^{-1} such that $W^{-1}W = \mathbf{I}$.

This property makes orthogonal matrices especially nice to work with, since we can take the inverse—usually a costly and numerically unstable operation—by flipping the indices around, which we can do in constant time, with no numerical instability.

We can now state the spectral theorem.

The spectral theorem. Call a matrix A *orthogonally diagonalizable* if it is diagonalizable with the additional constraint that P is orthogonal

$$A = PDP^T.$$

A matrix A is orthogonally diagonalizable if and only if A is symmetric.

Proving this now would require us to discuss too many extra concepts that aren't relevant for this part of the story. On the other hand, this theorem is very much the heart of PCA: everything it is and can do follows from this result. We'll take it at face value for now, and answer the rest of our questions. The next chapter will be entirely dedicated to proving the spectral theorem.

For now, just remember that if we have a square, symmetric matrix, we can diagonalize it with an orthogonal matrix P and a diagonal matrix D. The diagonal elements of D will be the eigenvalues and the columns of P will be the corresponding eigenvectors.

Note that the spectral theorem implies that there are n eigenvalues (since D has n diagonal values). Some of them might be zero, but we need not worry about that at the moment. In up-

coming chapters, we'll develop some concepts that help us characterize what it means for eigenvalues to be zero.

Finally, notice that for any such diagonalization, we can shuffle the eigenvalues around and get another diagonalization (we just have to shuffle the columns of $\mathbf{P}$ in the same way). Since the ordering of the eigenvalues in $\mathbf{D}$ is arbitrary, we usually sort the from largest to smallest, calling the largest the **first eigenvector** and the smallest the **last eigenvector**. As you may expect, we'll see later that these match the ordering of the principal components. We'll call the decomposition with the eigenvectors ordered like this, the *canonical* orthogonal diagonalization.

2.2.3 The eigenvectors of which matrix?

Let's get back to the PCA setting. Where do we find eigenvectors in this picture? We have a matrix, the data matrix $\mathbf{X}$, but it isn't square, and it's never used as a transformation.

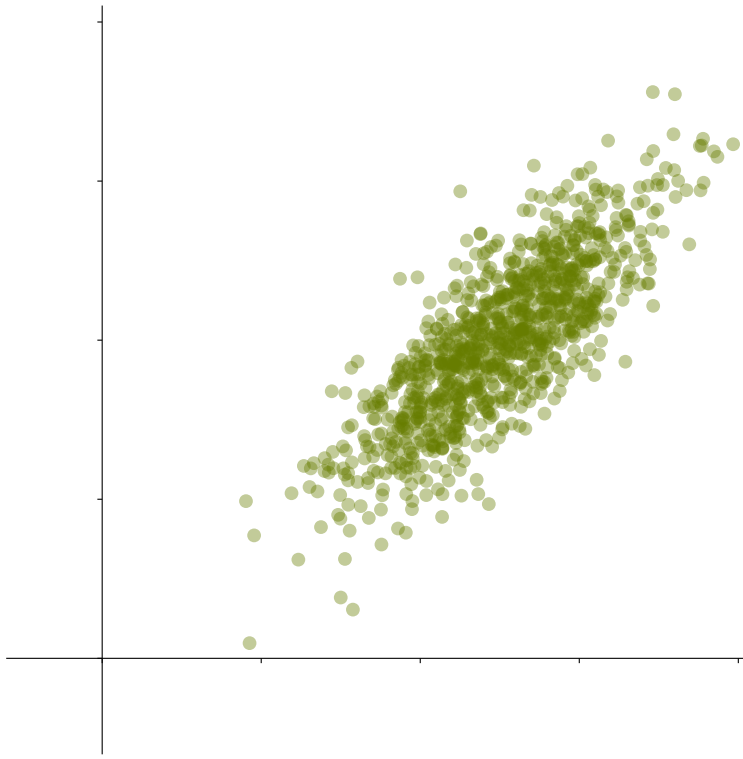
In fact, the eigenvectors that end up as the principal components are the eigenvectors of the *covariance matrix* of our data $\mathbf{X}$.

> The principal components are the eigenvectors of the covariance matrix.

Let's start by reviewing what a covariance matrix is. When we talk about one-dimensional data, we often discuss the variance: a measure for how spread out the numbers are. We can think of this as a measure of *predictability*. The more spread out the points are, the more difficult it is to predict where a randomly sampled point might be. If the variance is very small, we know any point is very likely to be near the mean of the data. If it's very large, we are less sure.

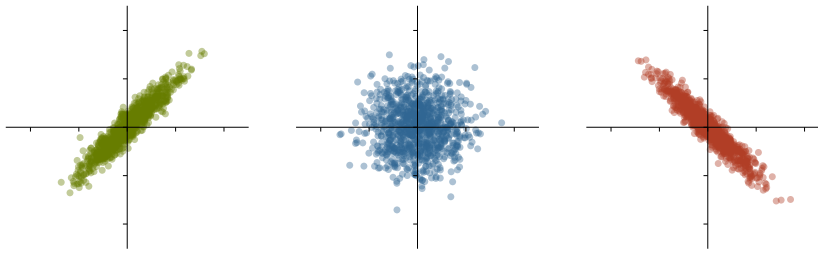
The covariance matrix is the analogue to this for m-dimensional data, like our dataset $\mathbf{X}$. It tells us not just how spread out the points are along the axes (the variance of each feature) but also how spread out the points of one feature are, *given the value of another feature*.

Consider the following 2D dataset:



The variance for both features is 1, so the data is pretty spread out. It has high variance, and is therefore relatively unpredictable. However, if I know the value of feature 1, suddenly the likely values of feature 2 become much narrower.

This is because the data has high *co*variance between the two features: knowing the value of one, tells us a lot about the value of another. Another way of saying this is that the two features are highly *correlated*. Here are the different ways data can be *linearly* correlated in 2D.



positive correlation

$$\begin{pmatrix} 1 & 0.95 \\ 0.95 & 1 \end{pmatrix}$$

no correlation

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

negative correlation

$$\begin{pmatrix} 1 & -0.95 \\ -0.95 & 1 \end{pmatrix}$$

Three ways data can be correlated. The bottom row shows the covariance matrices of each dataset (explained below).

Pay particular attention to the middle example: perfectly *decorrelated* data. In such data, the features are independent: knowing the value of one tells us nothing about the value of the other. This is an important property of good *latent features*. For instance, in the Olivetti data from the last chapter, many of the observed features (the pixel values) were highly correlated, but the latent features we extracted by PCA (the gender, the age, the lighting) were largely decorrelated. If the data is not biased, knowing the age of a subject shouldn't tell you anything about the way they were lit or how feminine they appear.

The formula for the variance of feature j, as we've seen before, is

$$\text{Var}_X(j) = \frac{1}{n} \sum_i (\bar{x}_j - X_{ij})^2$$

Where $\bar{x}_j$ is the mean of feature j, which is 0 if the data is mean-centered. The *co*variance between features j and k is defined as

$$\text{Cov}_X(j, k) = \frac{1}{n} \sum_i (\bar{x}_j - X_{ij})(\bar{x}_k - X_{ik})$$

These are both estimates. *The distribution from which the data was sampled has some invisible (co)variance, which we estimate from the data by these formulas. For a maximum likelihood estimate, we divide by* n*, for an unbiased estimate by* $n-1$*. For large data, the difference is negligible, so I'll use the first to keep the formulas simple.*

For mean-centered data, these simplify to

$$\begin{aligned}\mathrm{Var}_{\mathbf{X}}(j) &= \frac{1}{n}\sum_i X_{ij}X_{ij} \\ \mathrm{Cov}_{\mathbf{X}}(j,k) &= \frac{1}{n}\sum_i X_{ij}X_{ik}\,.\end{aligned}$$

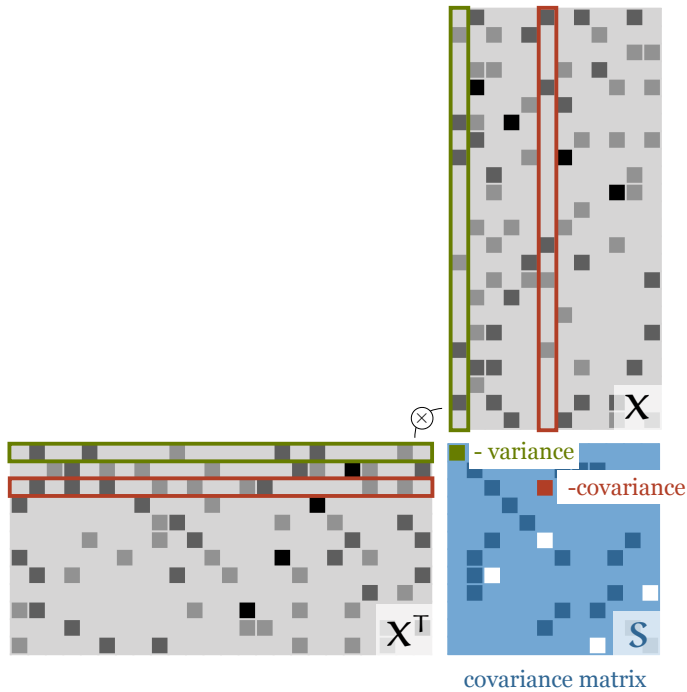
Note two things about these equations:

1. The variance is just the covariance of a feature *with itself*: $\mathrm{Var}_{\mathbf{X}}(j) = \mathrm{Cov}_{\mathbf{X}}(j,j)$.

2. If we ignore the multiplier $\frac{1}{n}$, the covariance is the dot product of one column of $\mathbf{X}$ with another.

This means that if we make a big matrix with all covariances between features j and k, we can compute that matrix by a simple matrix multiplication:

$$\mathrm{Cov}(\mathbf{X}) = \frac{1}{n}\mathbf{X}^T\mathbf{X}$$



This matrix is symmetric, since $\text{Cov}(j, k) = \text{Cov}(k, j)$, and it has the variances of the various features along the diagonal.

> *For any matrix, X of any size, $M = X^T X$ is square and symmetric: $M_{ij} = M_{ij}$ because both values are the dot product of columns i and j in the data.*

We'll denote the covariance matrix of our dataset X by S . This is the matrix that we're interested in: the eigenvectors of S coincide with the principal components of X.

I expect that that doesn't immediately make a lot of intuitive sense. We've developed eigenvectors in terms of matrices that transform points in space. We don't usually think of S as transforming space. It's not common to see a vector multiplied by S. Yet, we can easily diagonalize S. In fact, since it's symmetric,

the spectral theorem tells us that we can diagonalize it with an orthogonal matrix, and we can be sure that it has n eigenvalues.

To develop some intuition for what these eigenvalues *mean* we can have a look at the common practice of **data normalization**.

2.3 Data normalization and basis transformations

Data normalization is a very common data pre-processing step in many data science processes. For many applications we don't much care about the natural scale of the data, and we instead want the data to lie in a predictable range. For one-dimensional data, one way to do this is to rescale the data so that its mean equals 0, and its variance equals 1. We achieve this easily by first shifting the data so that the mean is at 0, and then scaling uniformly until the variance is 1.

To work out how to make this transformation, we can imagine that our data *originally* had mean 0, and variance 1, and was then transformed by scaling and then adding some constant value. That is, every point x we observed was derived from an unseen point z by two parameters s and t as

$$x = sz + t$$

with the z's having mean 0 and variance 1. We will call z the **hidden** or **latent** variable behind the **observed variable** x.

After scaling by s, the mean of the "original" data is still 0, so we should set $t = \bar{x}$ to end up with the correct mean for our observed data. To work out s, we move this term to the other side:

$$x - \bar{x} = sz\,.$$

The left-hand side is the mean-centered data, and the right hand side is a scaled version of the latent data. Since variance isn't affected by the additive term, we get

$$\mathrm{Var}(\{x\}) = \frac{1}{n}\sum_z (sz)^2 = s^2\frac{1}{n}\sum_z z^2 = s^2 \times \mathrm{Var}(\{z\}) = s^2$$

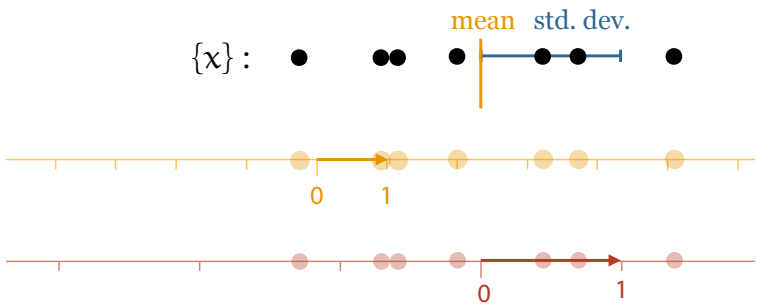
So, to end up with the correct variance for x, we should set the *square* of s equal to the data variance, or equivalently, we should

set s equal to the data standard deviation σ (the square root of the variance). So, the correct normalization is:

$$x' = \frac{x - \bar{x}}{\sigma}.$$

This may seem like an overly elaborate way to derive a pretty intuitive normalization, but we will generalize this approach to higher dimensions later, so it pays to understand the steps.

Instead of thinking of this operation as moving the data around, we can also think about it as keeping the data where it is, but just expressing it in different *axes*. We move the origin to coincide with the data mean, and then scale *the unit* (the length of the arrow from 0 to 1) so that its tip lies at the point $\bar{x} + \sigma$. On this new axis, the data is normalized.



We can see the operation of normalizing our one-dimensional data as simply expressing the same points on a different axis. We change the location of the origin and the length of the unit, and our data is normalized.

In higher dimensions, the *units* we use to express points in space are often called a *basis*. We take a bunch of vectors $\mathbf{b}_1, \ldots, \mathbf{b}_k$, called the *basis vectors* and express points as a sum of the basis vectors, each multiplied by a particular scalar, where the scalars are unique to the point we are expressing.

Strictly speaking, for the set $\mathbf{b}_1, \ldots, \mathbf{b}_k$ *to be a basis, the vectors should also be* linearly independent. *For our current purposes, we don't need to define this so precisely. We'll discuss linear independence in more detail in a later chapter.*

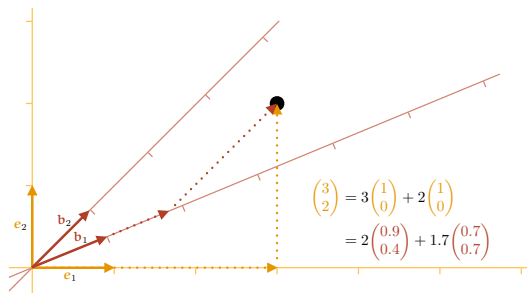
This is how our standard coordinate system works as well: in three dimensions, the basis vectors are $\boldsymbol{e}_1 = (1,0,0)$, $\boldsymbol{e}_2 = (0,1,0)$ and $\boldsymbol{e}_3 = (0,0,1)$. When we refer to a point $\mathbf{p}$ with the coordinates (7, 3, 5), we are implicitly saying that

$$\mathbf{p} = 7 \times \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 3 \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + 5 \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

This is called the **standard basis**. It's a little elaborate for something so familiar, but it shows a principle we can apply for other sets of basis vectors. With any set of vectors $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_k\}$, we can describe a point by writing down a vector $\mathbf{p}^B$, and computing

$$\mathbf{p} = p_1^B \mathbf{b}_1 + \ldots + p_k^B \mathbf{b}_k$$

Here, $\mathbf{p}$ are the coordinates of the point in the standard basis, and $\mathbf{p}^B$ are the coordinates in the basis B.



The point $(3, 2)$ as expressed in our standard basis becomes the point $(2, 1.7)$ when expressed in the basis defined by vectors $\mathbf{b}_1 = (0.9, 0.4)$ and $\mathbf{b}_2 = (0.7, 0.7)$.

If we concatenate the basis vectors into the columns of a matrix

$\mathbf{B}$, we can express this transformation as a simple matrix multiplication:

$$\mathbf{p} = \mathbf{B}\mathbf{p}^{\mathrm{B}}$$

This also suggest how to transform in the other direction: from a point in the standard basis to a point in the basis B: we require that $\mathbf{B}$ is invertible and use

$$\mathbf{p}^{\mathrm{B}} = \mathbf{B}^{-1}\mathbf{p}\,.$$

The set of points you can express in a particular basis is called its **span**. In the image above, the span is the same as that of the standard basis, but if you define two basis vectors in a three-dimensional standard basis, their span would usually be some plane crossing the origin.

If you want to actually compute the transformation into the basis, the computation of a matrix inverse is finicky and very likely to be numerically unstable. It's nice if you can ensure that $\mathbf{B}^{-1} = \mathbf{B}^{\mathrm{T}}$. We've seen matrices with this property already: they're called **orthogonal matrices**. To refresh your memory, orthogonal matrices are square matrices with columns that are all unit vectors and all mutually orthogonal.

> *A basis expressed by an orthogonal matrix is called an orthonormal basis. It's a rotated or flipped version of the standard basis, but the basis vectors are still all orthogonal and they are still all unit vectors.*

We can now say that our data normalization was nothing more than a simple basis transformation in $\mathbb{R}^1$. We mean-center the data, and replace the standard basis vector by one that matches the variance of our data. This is not an orthonormal basis, but we'll see a fix for that later.

More importantly, we can translate the idea of data normalization to higher dimensions.

In $\mathbb{R}^1$, we were after a basis in which the variance was 1. In $\mathbb{R}^n$ we will look for a basis in which the *covariance* is $\mathbf{I}$.

This requirement has two consequences:

- In our new coordinates, the variance is 1 along all axes.
- In our new coordinates all covariances are 0. That is, the data is perfectly *decorrelated*.

This kind of normalization is called whitening *(because standard normally distributed noise is sometimes called white noise). It's not usually necessary in data science, but it can be a very powerful preprocessing method if you can spare the required resources. We're primarily discussing it as a way of making intuitive what is happening under the hood of PCA.*

We'll proceed the same way we did before: we will imagine that our data was originally of this form, and has been transformed by an affine transformation. We'll call the matrix for this imagined "original" data $\mathbf{Z}$. This means that we assume that $\mathbf{X}$ was produced by sampling $\mathbf{Z}$ from a standard normal distribution and transforming with as:

$$\mathbf{X}^\mathsf{T} = \mathbf{A}\mathbf{Z}^\mathsf{T} + \mathbf{t}$$

with some $\mathbf{A}$ and $\mathbf{t}$.

As before, we will first figure out which $\mathbf{A}$ and which $\mathbf{t}$ will take us from the latent data $\mathbf{Z}$ to our observed data $\mathbf{X}$, and then we will invert this transformation to find the transformation that normalizes our data.

The logic for $\mathbf{t}$ is the same as it was before: since $\mathbf{Z}$ has zero mean, it still has zero mean after being transformed by $\mathbf{A}$. If we set $\mathbf{t} = \bar{\mathbf{x}}$, we transport this to the mean of the observed data.

We move this term to the left-hand side

$$\mathbf{X}^\mathsf{T} - \bar{\mathbf{x}} = \mathbf{A}\mathbf{Z}^\mathsf{T}$$

and observe that the mean-centered data on the left is equal to our $\mathbf{A}$-transformed latent data.

Now, we need to set $\mathbf{A}$ to achieve the right covariance. The covariance is unaffected by the additive term $-\bar{\mathbf{x}}$, so we can ignore that. The covariance of the transformed data is:

$$\text{Cov}(\mathbf{X}) = \frac{1}{n}\mathbf{A}\mathbf{Z}^\mathsf{T}(\mathbf{A}\mathbf{Z}^\mathsf{T})^\mathsf{T} = \frac{1}{n}\mathbf{A}\mathbf{Z}^\mathsf{T}\mathbf{Z}\mathbf{A}^\mathsf{T} = \mathbf{A}\text{Cov}(\mathbf{Z})\mathbf{A}^\mathsf{T} = \mathbf{A}\mathbf{A}^\mathsf{T}.$$

Where previously we needed to choose our scalar s so that its square was equal to the data variance σ^2, we now need to choose our transformation matrix $\mathbf{A}$ so that its "square" $\mathbf{A}\mathbf{A}^\mathsf{T}$ is equal to the data *co*variance $\mathbf{S}$.

If we find such an $\mathbf{A}$, we know that its *transformation* is what maps the decorrelated data to the data we've observed. So even though we never transform any points by the covariance matrix, we see that internally, it does contain a very natural transformation matrix.

There are a few ways to find $\mathbf{A}$ for a given $\mathbf{S}$. The Cholesky decomposition is the most natural analogue to the square root we used in the 1D case. This road leads to a technique known as *Cholesky whitening*.

But this is not a book about whitening, it's a book about PCA. We're trying build some intuition for what PCA is doing. So instead, we'll solve $\mathbf{S} = \mathbf{A}\mathbf{A}^\mathsf{T}$ using the orthogonal diagonalization we developed earlier, which will lead us to a method called *PCA whitening* (a kind of byproduct of the PCA analysis).

We know that $\mathbf{S}$ is square and symmetric, so we know it can be orthogonally diagonalized:

$$\mathbf{S} = \mathbf{P}\mathbf{D}\mathbf{P}^\mathsf{T}.$$

To turn this into a solution to $\mathbf{S} = \mathbf{A}\mathbf{A}^\mathsf{T}$ we need two factors, with the second the transpose of the first. We can do this easily by noting two things about the diagonal matrix in the middle. First, the square root $\mathbf{D}^{\frac{1}{2}}$ of a diagonal matrix $\mathbf{D}$ is just another diagonal matrix with the square roots of the original elements along the diagonal. This gives us $\mathbf{D} = \mathbf{D}^{\frac{1}{2}}\mathbf{D}^{\frac{1}{2}}$. Second, the transpose of a diagonal matrix is the same matrix, so that $\mathbf{D} = \mathbf{D}^{\frac{1}{2}}{\mathbf{D}^{\frac{1}{2}}}^\mathsf{T}$. Thus

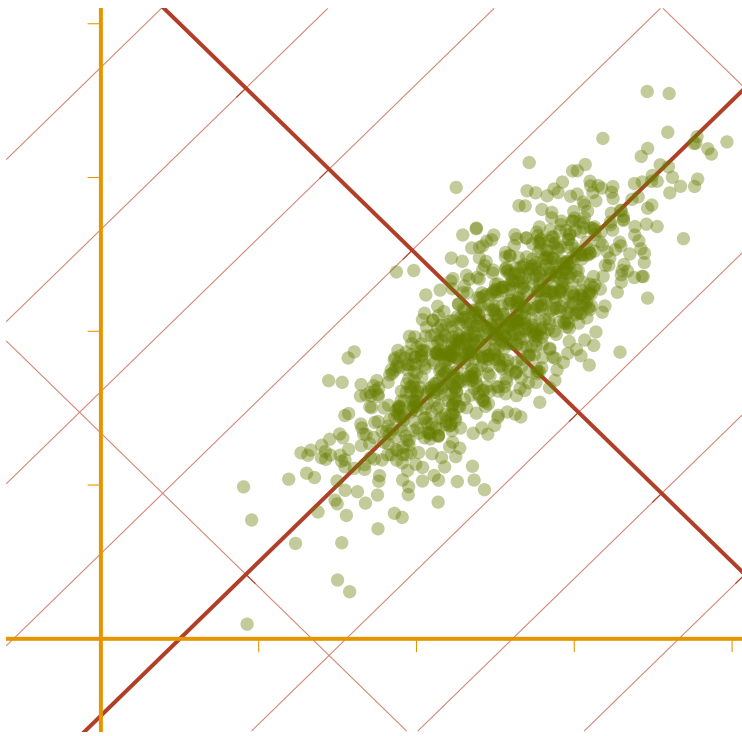
$$\begin{aligned}\mathbf{S} &= \mathbf{P}\mathbf{D}^{\frac{1}{2}}{\mathbf{D}^{\frac{1}{2}}}^\mathsf{T}\mathbf{P}^\mathsf{T} \\ &= \mathbf{A}\mathbf{A}^\mathsf{T} \qquad \text{with } \mathbf{A} = \mathbf{P}\mathbf{D}^{\frac{1}{2}}\end{aligned}$$

Finally, to whiten our data, we reverse the transformation from $\mathbf{Z}$ to $\mathbf{X}$ and get

$$\mathbf{Z} = \mathbf{A}^{-1}(\mathbf{X} - \mathbf{t}) = \mathbf{D}^{-\frac{1}{2}}\mathbf{P}^{\mathsf{T}}(\mathbf{X} - \mathbf{t}) .$$

So, to map our data to a zero-mean, unit-variance, decorrelated form, we map to the basis formed by the eigenvectors of $\mathbf{S}$ and then divide along each axis by the square root of the eigenvalues. We can see here that the eigenvalues of the covariance matrix are the variances of the data, *along the eigenvectors* (remember that we divided by the square of the variance before).

Taking the alternative perspective we took above, we can also keep the data where it is and *change our basis*. We scale the standard basis along the axes by $\mathbf{D}^{\frac{1}{2}}$ rotate by $\mathbf{P}$ and translate by $\bar{\mathbf{x}}$. In the resulting axes, our data has mean $\mathbf{0}$, and covariance $\mathbf{I}$.



Note how we're having our cake and eating it too. We are scaling our axes to control the variance, so we can't have an orthonormal basis, but the eigendecomposition breaks the basis transformation in two steps: first an orthonormal basis transformation, which allows us to use $\mathbf{P}^T$ *instead of* $\mathbf{P}^{-1}$*, and then a scaling along the new axes by the eigenvalues.*

2.4 Quadratic forms

Have we fully married our first intuition about eigenvectors in transformation matrices with the role eigenvectors play in PCA, as the eigenvectors of $\mathbf{S}$? Not quite. We've shown that $\mathbf{S}$ is in some sense composed of a very important transformation $\mathbf{A}$, which transforms decorrelated data with unit variance to have covariance $\mathbf{S}$, but the eigenvectors we're using are not the eigenvectors of $\mathbf{A}$. Rather, $\mathbf{A}$ is *made up* of our eigenvectors and may itself have different eigenvectors, or no (real) eigenvectors at all.

We will see in Chapter 4 that the eigenvectors of $\mathbf{S}$ *are called the* singular *vectors of* $\mathbf{A}$*.*

To develop an intuition for how $\mathbf{S}$ operates on space, it's more helpful not to look at the linear form

$$\mathbf{S}\boldsymbol{x}$$

but at the *quadratic* form

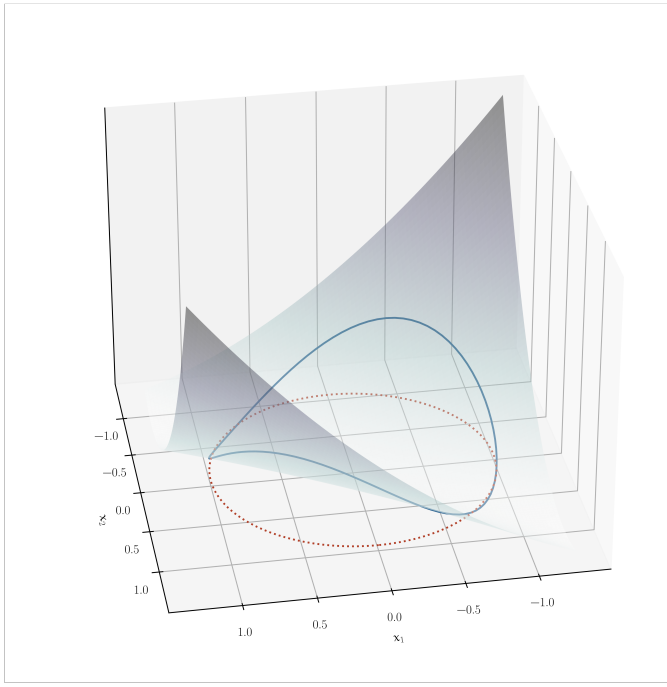
$$\boldsymbol{x}^T\mathbf{S}\boldsymbol{x}\,.$$

This may look mysterious, but it's just a concise way of writing second-order polynomials in n variables (just like $\mathbf{M}\boldsymbol{x}$ is a concise way of writing a linear function from n to m variables). For instance,

$$\boldsymbol{x}^T\begin{pmatrix}2 & 3\\ 4 & 5\end{pmatrix}\boldsymbol{x} = 2x_1{}^2 + 3x_2x_1 + 4x_1x_2 + 5x_2{}^2$$

The simplest quadratic is $\boldsymbol{x}^T\mathbf{I}\boldsymbol{x}$, or just $\boldsymbol{x}^T\boldsymbol{x}$. If we set this equal to 1, the points that satisfy the resulting equation are the unit

vectors. In 2 dimensions, these form a circle called the bi-unit circle. In higher dimensions, the resulting set is called the bi-unit (hyper)sphere.
There are two ways to use quadratic forms to study the eigenvectors of $\mathbf{S}$. The first is to look at $\mathbf{x}^\mathsf{T}\mathbf{S}\mathbf{x}$ and to study what this function looks like when constrained to the bi-unit sphere. Looking only at the points for which $\mathbf{x}^\mathsf{T}\mathbf{x} = 1$ what happens to the parabola $\mathbf{x}^\mathsf{T}\mathbf{S}\mathbf{x}$?



The bi-unit circle is deformed by the parabola $\mathbf{x}^\mathsf{T}\begin{pmatrix}2 & 3\\ 4 & 5\end{pmatrix}\mathbf{x}$

If we diagonalize $\mathbf{S} = \mathbf{P}\mathbf{D}\mathbf{P}^\mathsf{T}$, the quadratic becomes $\mathbf{x}^\mathsf{T}\mathbf{P}\mathbf{D}\mathbf{P}^\mathsf{T}\mathbf{x}$. The first and last two factors are just a change of basis so we can also write $\mathbf{z}^\mathsf{T}\mathbf{D}\mathbf{z}$ with $\mathbf{z} = \mathbf{P}^\mathsf{T}\mathbf{x}$. Since $\mathbf{P}$ is orthogonal, the

change of basis doesn't change the length of the vectors and the constraint that $\mathbf{x}$ should be unit vectors is equivalent to requiring that $\mathbf{z}$ be unit vectors.
The quadratic form $\mathbf{z}\mathbf{D}\mathbf{z}$ is particularly simple, because $\mathbf{D}$ is diagonal. We simply get

$$\mathbf{z}\mathbf{D}\mathbf{z} = z_1 z_1 \mathrm{D}_{11} + z_2 z_2 \mathrm{D}_{22} + \ldots + z_m z_m \mathrm{D}_{mm} .$$

This sum is very important. Note that all the factors $z_r z_r$ are not only positive (because they're squares), but they also sum to one, since $\|\mathbf{z}\|^2 = {z_1}^2 + \ldots + {z_m}^2$ is the squared length of vector $\mathbf{z}$, which we constrained to 1.

We'll call this a **weighted sum**: a sum over some set of numbers where each term is multiplied by a positive weight, so that the weights sum to 1.

In the next section, we will use this sum to prove just about every open question we have left. For now, just notice what happens when $\mathbf{x}$ is an eigenvector. In that case, $\mathbf{z}$ is a one-hot vector, because $\mathbf{P}\mathbf{z} = \mathbf{x}$, and only one of the terms in the sum is non-zero.

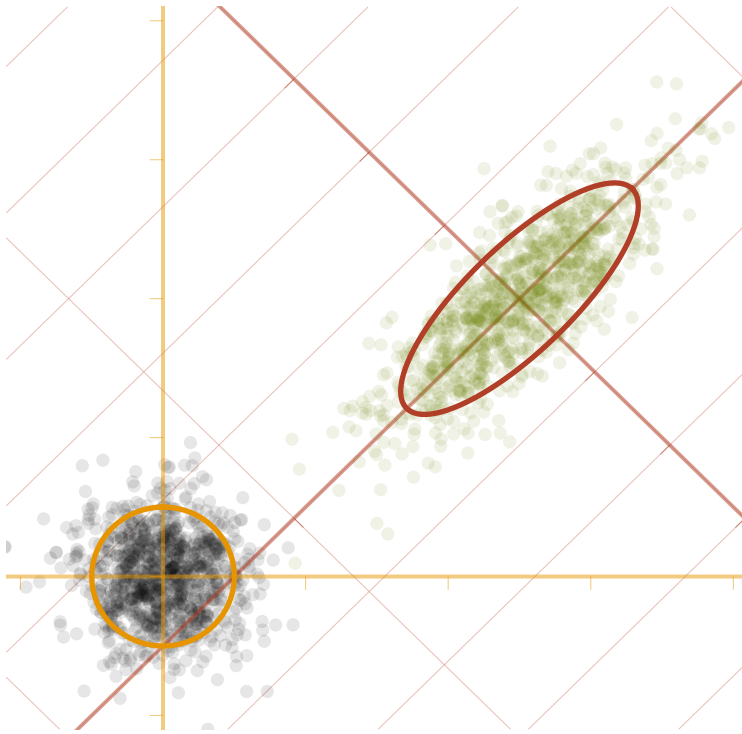
This is one way to think of the quadratic form of $\mathbf{S}$: it defines m orthogonal directions in space (the eigenvectors), at which the quadratic takes some arbitary value (the eigenvalues). For all other directions $\mathbf{x}$, the quadratic is a weighted mixture between the eigenvalues, with the weights determined by how much of $\mathbf{x}$ projects onto the corresponding eigenvectors.

> *Looking at the sum above, you may be able to figure out what the minimum and maximum values are of the quadratic form $\mathbf{x}^T\mathbf{S}\mathbf{x}$ under the constraint that $\mathbf{x}^T\mathbf{x} = 1$. Note the similarity of this optimization problem to the PCA problem. If you don't see it yet, don't worry. We'll dig deeper into this in the next section.*

For another geometric interpretation of the eigenvectors of $\mathbf{S}$, think back to the one-dimensional example of normalizing data.

In the normalized version of the data, the variance is equal to 1. This means that, for most distributions, we can be sure that the majority of the data lies in the interval $(-1, 1)$. This is called the bi-unit interval, since it is made up of two units around the origin. If our data is normally distributed, this interval captures

about 68% of it after normalization. The transformation by t and s maps this interval to an interval that captures the same proportion of the unnormalized data.
In higher dimensions, the analogue of the bi-unit interval is the bi-unit sphere, the set of all points that are at most 1 away from the origin. To follow the analogy, we can transform the bi-unit sphere, which captures the majority of $\mathbf{Z}$, by some $\mathbf{A}^{-1}$ so that we capture the majority of the observed data $\mathbf{X}$.



The bi-unit circle captures the majority of the normalized data. The ellipse we create by transforming the circle by $\mathbf{A}^{-1}$ (and translating to the mean) captures the same majority of the unnormalized data.

In two dimensions, the transformation by $\mathbf{A}$ and $\mathbf{t}$ that we derived above is the transformation that maps the bi-unit circle to an ellipse which captures the majority of the data. In more than two dimensions, we're mapping a hypersphere to an *ellipsoid*. Note that the standard basis vectors are mapped to the eigenvectors of $\mathbf{S}$. We call this new basis, in which the data is normalized, the **eigenbasis** of $\mathbf{S}$.

To work out the shape of the ellipsoid in quadratic form, we just start with the set of all unit vectors $C = \{\mathbf{u} : \mathbf{u}^T\mathbf{u} = 1\}$ and transform each by $\mathbf{A}$ individually (much like we transformed the Mona Lisa earlier).

$$\begin{aligned}
\mathbf{A}C &= \{\mathbf{A}\mathbf{u} : \mathbf{u}^T\mathbf{I}\mathbf{u} = 1\} \\
&= \{\mathbf{y} : \mathbf{u}^T\mathbf{u} = 1 \text{ and } \mathbf{y} = \mathbf{A}\mathbf{u}\} \\
&= \left\{\mathbf{y} : \left(\mathbf{A}^{-1}\mathbf{y}\right)^T \mathbf{A}^{-1}\mathbf{y} = 1\right\} \\
&= \{\mathbf{y} : \mathbf{y}^T\mathbf{S}^{-1}\mathbf{y} = 1\}
\end{aligned}$$

That is, to transform the bi-unit circle to an ellipsoid that covers the same proportion of $\mathbf{X}$ as the circle did of $\mathbf{Z}$, we turn the equation $\mathbf{u}^T\mathbf{u} = 1$ into the equation $\mathbf{y}\mathbf{S}^{-1}\mathbf{y} = 1$. This gives us a quadratic form that desribes the ellipsoid that covers the majority of our data.

Why do we get an inversion from $\mathbf{S}$ to $\mathbf{S}^{-1}$? Follow the transformation along the first eigenvector of $\mathbf{S}$. In this direction, we are multiplying the input unit vector by the square of the first eigenvalue (remember $\mathbf{A} = \mathbf{P}\mathbf{D}^{\frac{1}{2}}$). To keep the vector a unit vector, we should therefore *divide* by the square of the first eigenvalue. In other words, if we want to define a quadratic form for which the arguments transformed by $\mathbf{A}$ stay unit vectors, then the more the value of $\mathbf{x}^T\mathbf{S}\mathbf{x}$ grows, the more our quadratic form should shrink and vice versa.

Note that the inverse of a symmetric matrix has a particularly simple expression in terms of its orthogonal diagonalization: $\mathbf{S}^{-1} = \left(\mathbf{P}\mathbf{D}\mathbf{P}^T\right)^{-1} = \mathbf{P}\mathbf{D}^{-1}\mathbf{P}^T$. *That is, the eigenvectors stay the same, and we just take the reciprocals of the eigenvalues.*

2.5 Why is PCA optimal?

With the quadratic form added to our toolbox, we can finally start answering some of the deeper questions. Both visually, using transformations of ellipses, and formally, using the language of eigenvectors.

Let's start simply: why does the first principal component coincide with an eigenvector of S? And, while we're at it, which eigenvector does it coincide with?

Visually, this is easy to show. In the image above, we plotted the bi-unit circle, and its transformation into an ellipse that covers the same part of the observed data. The standard basis vectors are mapped to the eigenvectors. Note that these become the axes of the ellipse. One of the standard basis vectors is mapped to the ellipse's *major axis*, the direction in which it bulges the most.The direction in which the data bulges the most is also the direction of greatest variance, and therefore the first principal component.

The proof of this fact is not very complex.

First eigenvector. The first principal component is the eigenvector of the covariance matrix S with the largest eigenvalue.

Proof. The first principal component w_1 is defined as

$$\underset{w}{\operatorname{argmax}} \sum_i \left(w^T x_i\right)^2 \quad \text{such that } w^T w = 1$$

that is, the direction in which the variance of the projected data is maximized.

Rewriting the objective function, we get

$$\begin{aligned}
\sum_i \left(w^T x_i\right)^2 &= \sum_i w^T x_i {x_i}^T w \\
&= w^T \left(\sum_i x_i {x_i}^T \right) w \\
&= w^T X^T X w \\
&= N w^T S w .
\end{aligned}$$

This means that the direction in which the sum of variances is maximized, is the direction, represented by a unit vector $\boldsymbol{w}$, for which the quadratic form $\boldsymbol{w}^T \mathbf{S} \boldsymbol{w}$ is maximal.
If we orthogonally diagonalize $\mathbf{S}$, with the eigenvalues canonically arranged, we get

$$\begin{aligned} \boldsymbol{w}^T \mathbf{S} \boldsymbol{w} &= \boldsymbol{w}^T \mathbf{P} \mathbf{D} \mathbf{P}^T \boldsymbol{w} \\ &= \boldsymbol{z} \mathbf{D} \boldsymbol{z} \quad \text{with } \boldsymbol{z} = \mathbf{P}^T \boldsymbol{w} . \end{aligned}$$

In the last step, we've simplified to a diagonal quadratic form in the eigenbasis of $\mathbf{S}$. This quadratic form simplifies to

$$\boldsymbol{z}^T \mathbf{D} \boldsymbol{z} = {z_1}^2 \mathrm{D}_{11} + \ldots + {z_m}^2 \mathrm{D}_{mm}$$

where the constraint that $\boldsymbol{z}$ is a unit vector means that ${z_1}^2 + \ldots + {z_m}^2 = 1$. In other words, this is a weighted sum over the diagonal elements of $\mathbf{D}$. To maximize a weighted sum, we assign weight 1 to the largest element and weight 0 to the rest. Since we took D_{11} to be the largest eigenvalue, the vector $\hat{\boldsymbol{z}} = (1, 0, \ldots, 0)$ maximizes the quadratic form.

Mapping back to the standard basis, we get $\boldsymbol{w}_1 = \mathbf{P} \hat{\boldsymbol{z}}$. That is, the first column of $\mathbf{P}$, which is the first eigenvector of $\mathbf{S}$. □

We can extend this proof to show that all the other PCs are eigenvectors as well.

PCs as eigenvectors. The k-th principal component of $\mathbf{X}$ is the k-th eigenvector of the covariance matrix $\mathbf{S}$.

Proof. For the first principal component $\boldsymbol{w}_1$, the previous theorem provides a proof. For $\boldsymbol{w}_2$, follow the previous proof until the weighted sum

$$\boldsymbol{z}^T \mathbf{D} \boldsymbol{z} = {z_1}^2 \mathrm{D}_{11} + \ldots + {z_m}^2 \mathrm{D}_{mm} .$$

First, note that any vector $\boldsymbol{w}'$ that is orthogonal to $\boldsymbol{w}_1$ must also be orthogonal after transformation by $\mathbf{P}\mathbf{P}^T$:

$$0 = \boldsymbol{w}_1^{\mathsf{T}} \boldsymbol{w}' = \boldsymbol{w}_1{}^{\mathsf{T}} \mathbf{P}\mathbf{P}^{\mathsf{T}} \boldsymbol{w}' = \boldsymbol{z}_1{}^{\mathsf{T}} \boldsymbol{z}' .$$

Thus, the second eigenvector is orthogonal to the first (as required) if and only if their projections by $\mathbf{P}^{\mathsf{T}}$ are orthogonal as well (and similarly for higher eigenvectors).

Recall that the $\boldsymbol{z}$-vector of the first principal component is $(1, 0, \ldots, 0)$, so to be orthogonal, the $\boldsymbol{z}$ vector corresponding to the second principal component must have 0 at its first element. Since the D_{jj} are arranged in decreasing order, we maximize the sum under this constraint with the vector $\hat{\boldsymbol{z}} = (0, 1, 0, \ldots, 0)$. $\mathbf{P}^{\mathsf{T}} \hat{\boldsymbol{z}}$ selects the second column in $\mathbf{P}$, so the second principal component coincides with the second eigenvector.

The same logic holds for the other principal components. For each component r, we must set all weights ${z_i}^2$ with $i < r$ to zero in order for the $\boldsymbol{z}$ vector to be orthogonal to all principal components already chosen. In the remainder of the sum, we maximize the weight with the one-hot vector for r, which selects the r-th eigenvector. □

We have finally shown that the eigenvectors are the vectors that maximize the variance.

There is one question left to answer: Why is the set of the first k principal components a solution to the combined problem at k?

Optimality of PCA. For any k, a solution to the iterative problem is a solution to the combined problem.

Proof. Let $\boldsymbol{W}$ be a solution to the combined problem at k. Let $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k$ be the columns of $\mathbf{P}^{\mathsf{T}}\boldsymbol{W}$, that is, the solution vectors expressed in the eigenbasis of $\mathbf{S}$. The total variance captured by all these $\boldsymbol{z}$ is

$$\sum_r \boldsymbol{z}_r{}^{\mathsf{T}} \mathbf{S} \boldsymbol{z}_r = \sum_{r,j} {z_{rj}}^2 D_{jj} .$$

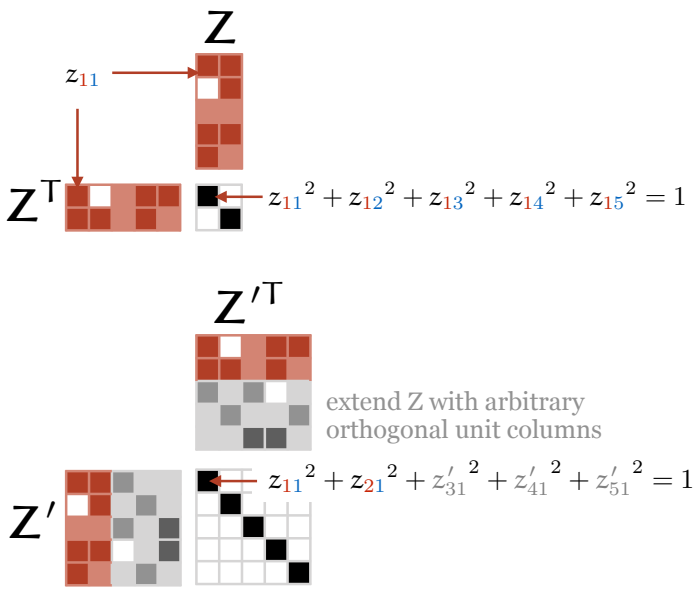
Where z_{rj} is the j-th element of vector $\boldsymbol{z}_r$. These are r weighted sums, summed together. We can group the weights that each $\boldsymbol{z}_r$ contributes to each eigenvalue D_{jj} in to a single sum:

$$\sum_{r,j} z_r{}^2 D_{jj} = \sum_j D_{jj} \sum_r z_{rj}{}^2 .$$

Now, since the different z_r's are orthogonal, and each of unit length, their sum contribution to each D_{jj} must be no more than 1, and the sum of each of their weights is 1.

Why? Imagine the matrix $\mathbf{Z}$ with the z_r for its columns. What we're saying is that its squared elements of $\mathbf{Z}$ should sum to 1 over the columns and not exceed 1 when summed over the rows. If we take $\mathbf{Z}^T\mathbf{Z}$, the squared and summed columns end up along the diagonal. We know $\mathbf{Z}^T\mathbf{Z} = \mathbf{I}$, so these are all 1.

If we take $\mathbf{Z}\mathbf{Z}^T$, the squared and summed *rows* end up along the diagonal. Extend $\mathbf{Z}$ with orthogonal unit vectors until it is square and orthogonal. Then $\mathbf{Z}\mathbf{Z}^T = \mathbf{I}$. Each 1 on the diagonal of $\mathbf{I}$ is the result of a sum of squared values. Some come from the original $\mathbf{Z}$, some from the columns we added, but all are squares, so it's a sum of nonnegative terms. Therefore, the terms contributed by the original vectors cannot sum to more than 1.



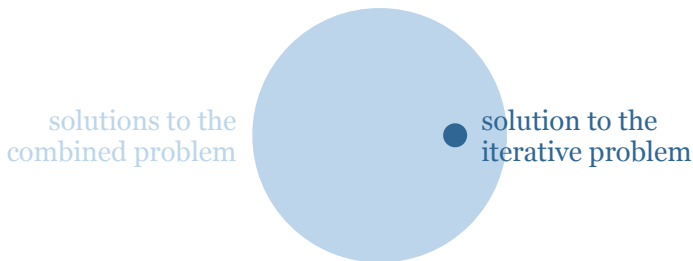
So, we have a weighted sum where the total weight allowed is k, and the maximum weight per element is 1. The optimal solution is to give a maximum weight of 1 to each of the largest k elements—that is, the first k eigenvalues—and zero weight to everything else.

One way to achieve this is by setting $\{z_r\}$ to be the first k one-hot vectors, which yield the first k eigenvectors when we transform back to the standard basis. □

Note that when we say "one way" in the last paragraph, we do not mean the *only* way. For instance, if we set $k = 2$, we get an optimum with $z_1 = (1, 0, 0, \ldots), z_2 = (0, 1, 0, \ldots)$ (the PCA solution), but rotating the vectors by 45 degrees in their shared plane gives us $z_1 = (\sqrt{1/2}, \sqrt{1/2}, 0, \ldots), z_2 = (\sqrt{1/2}, -\sqrt{1/2}, 0, \ldots)$. Filling these values into the sum, we see that they also result in a weight of 1 for D_{11} and a weight of 1 for D_{22}, which means that this is also a solution to the combined problem at $k = 2$.

More broadly, given the PCA solution, any other W whose columns span the same space as the span of the PCA solution is also a solution to the combined problem.

We have finally proved our Venn diagram correct, and we have illustrated what the rest of the light blue circle is made of.



2.5.1 Characterizing the PCA solution

Since the eigenvectors are the solution to the PCA problem, you may be forgiven for thinking of the eigenvectors themselves in terms error minimization or variance maximization. In that case, we should guard against a misconception.

Look back at the ellipse we drew above. The first eigenvector was the major axis of the ellipse, the direction in which the data bulged out the most. However, the other eigenvector is its *minor* axis. The direction in which the data bulges *the least*, this makes it the direction in which the variance is *minimized*.

To study this a bit more formally, we take the first two proofs of the previous section and turn them around. If we start with the *last* eigenvector and work backward, we are choosing the directions that minimize the variance (and hence maximize the reconstruction error).

Last eigenvector. The direction in which the variance is minimized, is the eigenvector of A with the smallest eigenvalue.

Proof. Follow the proof of the **first eigenvector** theorem until the sum

$$z^T D z = {z_1}^2 d_{11} + \ldots + {z_m}^2 d_{mm} .$$

A weighted sum is minimized when all the weight is given to the smallest term. Following the same logic as before, this leads to a one-hot vector $\hat{z} = (0, \ldots, 0, 1)$ that selects the last column of P, which is the last eigenvector. □

Did we make a mistake somewhere? We defined the principal components as directions for which variance is maximized. Then we showed that all principal components are eigenvectors. Now we learn that at least one eigenvector actually *minimizes* the variance. What gives?

The solution lies in the fact that the sum of all variances is fixed to the sum of the variances of the data, z_{total}. Imagine solving the combined problem for $k = m - 1$. The resulting variances along the columns of the solution W should be as high as possible. However since all these columns are orthogonal, there is only one direction v left which is orthogonal to all of them. The variance along this direction, z_v, is whatever variance we haven't captured in our solution:

$$z_1 + \ldots + z_{m-1} + z_v = z_{\text{total}}$$

Since z_{total} is fixed, maximizing the first $m-1$ terms is equivalent to minimizing the last.

We can define a kind of *reverse iterative problem* where we define the last principal component as the direction that minimizes the variance, the last-but-one principal component as the direction orthogonal to the last principal component, the last-but-two principal component as the direction orthogonal to the last two that minimizes the variance and so on.

We can show that optimizing for this problem gives us exactly the same vectors as optimizing for the original iterative problem which maximized the variance.

Reverse iteration. Under the reverse iterative problem, the last-but-r principal component chosen coincides with the k-th eigenvector of S, with $k = m - r$, and therefore with the k-th principal component.

The proof is the same as that of the **PCs as eigenvectors** theorem, except starting with the smallest eigenvector instead of the largest and choosing $\hat{z}$ to *minimize* at every step.

This shows us that it's not quite right to think of the eigenvectors as maximizing or minimizing some quantity like variance or reconstruction error (even though we've defined the principal components that way). The eigenvectors of S simply form a very natural orthonormal basis for the data, from which we can derive natural solutions to optimization objectives in both directions.

There is one question that we haven't answered yet. How do we refine the combined problem so that it coincides with the iterative problem? The one property that we use in our derivations above that is not stated in the combined problem, is that in the new basis, the data are *decorrelated*. If we add this requirement to the optimization objective, we get:

$$\begin{aligned} &\underset{W}{\text{argmax}} \sum_i \left(\mathbf{x}_i{}^T W\right)^2 \\ &\text{such that } W^T W = \mathbf{I} \\ &\quad \text{and } \frac{1}{N} W^T \mathbf{X}^T \mathbf{X} W \text{ is diagonal.} \end{aligned}$$

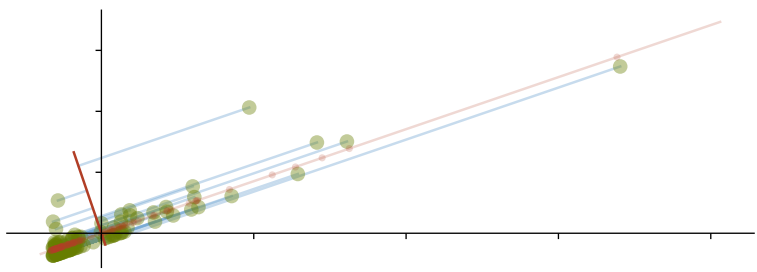
For this problem, there is only one solution (up to negations of the principal components): the PCA solution. With this, we can

finally let go of our iterative view of PCA, and embrace methods that compute all principal components in a single operation.

We have come home from a long walk. Let's settle by the fireplace and talk about all the things we've seen.

We started with a broad idea of PCA as a method that iteratively minimizes the reconstruction error, while projecting into a lower dimensional space. For some reason, we saw last time, this works amazingly well and exposes many meaningful latent dimensions in our data. In this chapter, we showed first that minimizing reconstruction error is equivalent to maximizing variance.

We then looked at eigenvectors, and we show that the eigenvectors of the data covariance $\mathbf{S}$ arise naturally when we imagine that our data was originally decorrelated with unit variance in all directions. To me, this provides some intuition for why PCA works so well when it does. We can imagine that our data was constructed by sampling independent latent variables z and then mixing them up linearly. In our income dataset, in the first chapter, there was one important latent variable: each person's salary. From this, we derived the monthly salary, and the majority of their quarterly income. The other latent variable captured random noise: whether people had some extra income, bonuses, etc.



We can imagine the same approach with the Olivetti faces. We get 4096 features, but under water, most of the work is done by a few *latent* dimensions which are largely independent of each other: the subject's age, the direction of the light source, their apparent gender and so on. All of these can be chosen independently from each other, and are likely mostly decorrelated in the data. That is, if we don't light all women from the left, or only chose old men and young women.

If these assumptions are violated, it may point to undesirable biases in our data. A very relevant topic at the moment. This shows that bias can be defined in terms of the assumed latent variables. Unfortunately, once the data is biased, it reduces our ability to extract the latent features, which makes it more difficult to counteract the bias.

Since the assumptions behind our transformation from decorrelated data to the observed data are mostly correct, finding this transformation, and inverting it retrieves the latent dimensions. The greater the variance along a latent dimension, the more variance that particular "choice" added to the the data. The choice of the subject's age adds more variance than the lighting, and the lighting adds more variance than the gender.

The heart of the method is the spectral theorem. Without the decomposition $\mathbf{S} = \mathbf{P}\mathbf{D}\mathbf{P}^\mathsf{T}$, none of this would work. Proving that such a decomposition always exists for a symmetric matrix, and that every matrix for which the decomposition exists is symmetric, is not very difficult, but it takes a lot more background than we had room for here: this includes matrix determinants, the characteristic polynomial and complex numbers. In the next chapter, we will go over all these subjects carefully, building our intuition for them, and finish by thoroughly proving the spectral theorem.

Finally, you may wonder if any of these new insights help us in computing the principal component analysis. The answer is yes, the eigendecomposition $\mathbf{S} = \mathbf{P}\mathbf{D}\mathbf{P}^\mathsf{T}$ can be computed efficiently, and any linear algebra package allows you to do so. This gives you the principal components $\mathbf{P}$, and the rest is just matrix multiplication.

The eigendecomposition is certainly faster and more reliable than the projected gradient descent we've used so far, but it can still be a little numerically unstable. In practice, PCA is almost always computed by **singular value decomposition** (SVD). The SVD is such a massively useful method that it's worth looking at in more detail. It's inspired very much by everything we've set out above, but its practical applications reach far beyond just the computation of principal components. We'll develop the SVD in

Chapter 4 and provide some algorithms for computing both the eigendecomposition and the SVD in Chapter 5. But before all that, we'll return to the spectral theorem, and see exactly what is required to prove it.

CHAPTER 3 · PROVING THE SPECTRAL THEOREM

When I started writing this, it was not meant to be a book. I was going for a short explanation of Principal Component Analysis that was simple, but that also didn't skip any steps. I was frustrated with other explanations that leave things out, or require the reader to take things at face value.

This chapter illustrates why that so often happens. In this chapter we will prove the **spectral theorem**, which we introduced in the previous chapter. This is very much the dark heart of PCA: the one result from which everything else follows, so it pays to understand it properly. The drawback is that the proof of the spectral theorem adds a boatload of preliminaries to the story.

Suddenly, just to understand this one statement, we need to understand **determinants**, the **characteristic polynomial**, **complex numbers, vectors and matrices** and **the fundamental theorem of algebra**. All interesting, of course, and worth knowing about, but it's a lot of baggage if you just want to know how PCA works. So I decided to move it all into one self-contained chapter.

3.1 Restating the spectral theorem

In the last chapter, we learned the following.

An *orthogonal matrix* is a square matrix whose columns are mutually orthogonal unit vectors. Equivalently, an orthogonal matrix is a matrix $\mathbf{P}$ for which $\mathbf{P}^{-1} = \mathbf{P}^{\mathsf{T}}$.

Any square matrix $\mathbf{A}$ is *orthogonally diagonalizable* if there exists an orthogonal matrix $\mathbf{P}$ and a diagonal matrix $\mathbf{D}$ such that $\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{\mathsf{T}}$. A matrix $\mathbf{A}$ is symmetric if $\mathbf{A} = \mathbf{A}^{\mathsf{T}}$.

> **The spectral theorem** A matrix is orthogonally diagonalizable if and only if it is symmetric.

We call this "the" spectral theorem in the context of this book. In general, there are many spectral theorems about which operators can be diagonalized under which conditions.

Previously, we saw how much follows from this one simple theorem. If we take this to be true, we get eigenvectors, whitening and principal components.

In the rest of this chapter we'll build a toolkit step by step, with which to analyze this problem. At the end, we'll return to the theorem and apply our tools to prove it.

The first is a very useful function of a matrix: the determinant.

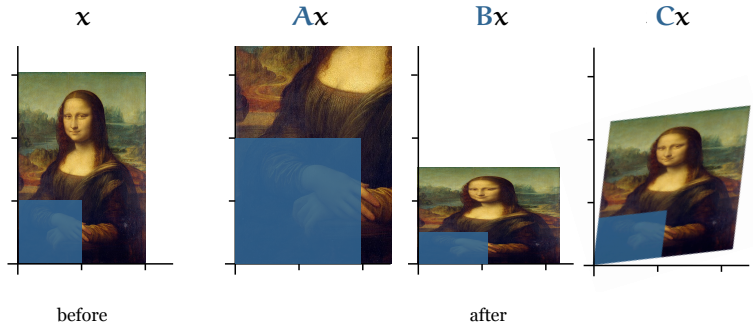
3.2 Determinants

The determinant started life long before linear algebra. As early as the 3rd century BCE, the function was used as a property of a set of linear equations that would allow you to determine whether the equations had a solution.

Later, determinants were studied as functions in their own right. In this context, they were seen as very opaque and abstract: something that was useful in higher mathematics, but hard to explain to the lay person. It wasn't until *matrices* become popular, and in particular the view of a matrix as representing a geometric transformation, that determinants finally acquired an intuitive and simple explanation.

That explanation—apart from some subtleties which we'll discuss later—is that for a square matrix $\mathbf{A}$, thre determinant expresses how much $\mathbf{A}$ *inflates* the space it transforms.

For example, here are three different ways that a matrix might transform space to squish and stretch in different directions.



Three linear transformations, showing the effect on the Mona Lisa, and the unit square.

In the first, everything is stretched equally in all directions by a factor of 2. That means that a square with area 1 in the original (a *unit* square) has area 4 after the transformation by $\mathbf{A}$ (since both its sides are doubled). This is what we mean by inflating space: the determinant of $\mathbf{A}$ is 4 because transforming something by $\mathbf{A}$ increases its area by a factor of 4. In the second example, We stretch by 1.1 in one direction, and shrink to 0.5 in the other. The result is that that a unit square in the original ends up smaller after the transformation: the determinant of $\mathbf{B}$ is $0.5 \times 1.1 = 0.55$.

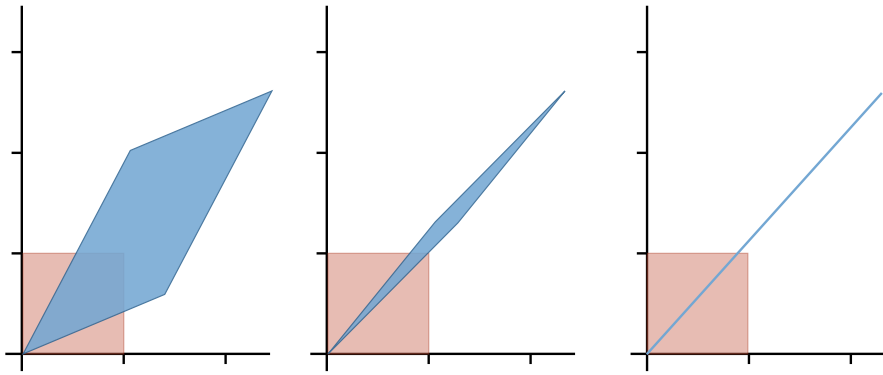
To see that objects other than squares are inflated by the same amount, just subdivide it into small squares. Each of the squares is inflated by the same amount, so the total is as well.

The third example is a little trickier. The Mona Lisa is again squished and stretched in different directions, but these are not aligned with the axes. The area of the unit square seems to be getting a little smaller, but how can we tell by how much exactly?

Before we dig into the technical details, let's first look at why it is worth doing so. Why is it so important to know by how much a matrix inflates space? There are many answers, but in the context of this series, the most important reason to care about the determinant is that it gives us a very convenient characterization of *invertible matrices*.

An invertible matrix is simply a matrix whose transformation is invertible. That is, after we apply the transformation $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ we can always transform $\mathbf{y}$ back to $\mathbf{x}$, and end up where we started.

When is a matrix not invertible? When multiple inputs $\mathbf{x}$ are mapped to a single output $\mathbf{y}$. In linear transformations, this happens when the input is squished so much in one direction, that the resulting space has a lower dimensionality than the original.



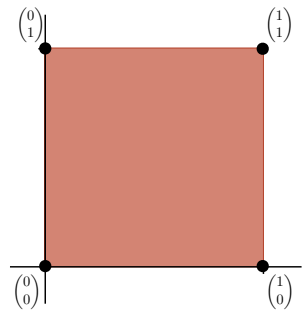
Three transformations with increasingly small determinant. In the third, the unit square is squeezed into a line. Note that the two edges on the bottom left of the square are mapped to the same part of the line, so the transformation is not invertible.

We don't need to know how to compute the determinant to know what its value is in this case. The unit square is mapped to a line segment, so its area goes from 1 to 0. This is how the determinant helps us to characterize invertible matrices: if the determinant is non-zero, the matrix is invertible, if the determinant is zero, the matrix is not invertible, or *singular*.

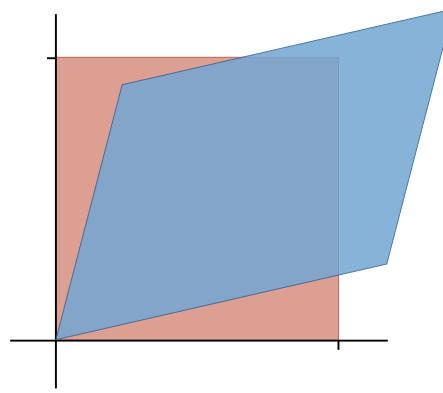
3.2.1 Computing the 2×2 determinant

Using this definition, it's pretty straightforward to work out what the formula is for the determinant of a matrix $\mathbf{A}$ that transforms a 2D space. We'll start by drawing a unit square, and labeling

the four corners:



The corner points $(0,0)$, $(0,1)$ $(1,0)$ and $(1,1)$ can be transformed by multiplying them by $\mathbf{A}$. we know that under a linear operation like this, line segments stay line segments, so the four edges of the square are transformed to line segments, and the resulting figure between the four points must be a quadrilateral. We also know that *parallelism is preserved*: two line segments that were parallel before the transformation are parallel after. Lastly, we know that the origin stays where it is, unless we apply a translation, so corner $(0,0)$ is not affected by the transformation. All this means that the picture after the transformation will look something like this.



A parallelogram with one corner touching the origin. The determinant tells us the ratio between the area of the paralellogram and the original square. Since the original square has area 1, the area of the parallelogram is the determinant of $\mathbf{A}$.

Working this out requires only a small amount of basic geometry. Here's the simplest way to do it.

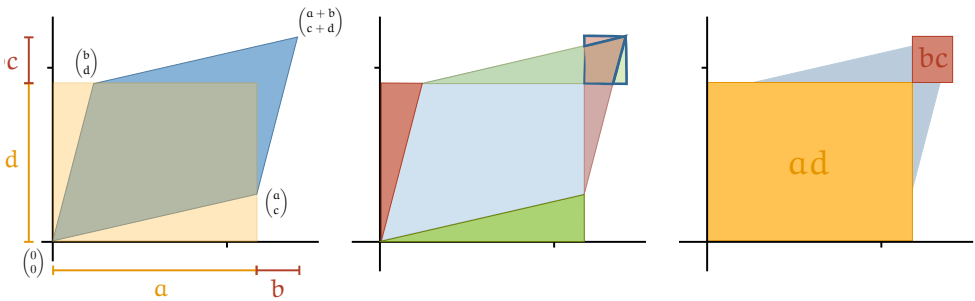
We first name the four elements of our matrix:

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

We can now name the four corners of our parallelogram in terms of these four scalars, by multiplying the corner points of the unit square by $\mathbf{A}$:

$$\begin{array}{cccccc} & \times & \begin{pmatrix}0\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}0\\1\end{pmatrix} & \begin{pmatrix}1\\1\end{pmatrix} \\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} & & \begin{pmatrix}0\\0\end{pmatrix} & \begin{pmatrix}a\\c\end{pmatrix} & \begin{pmatrix}b\\d\end{pmatrix} & \begin{pmatrix}a+b\\c+d\end{pmatrix} \end{array}$$

This gives us the following picture.



Here we can see the area of the paralellogram clearly: there is one large rectangle of area ad. To get from this area to the area of our parallelogram, we should subtract the area of the green triangle in the bottom, which is part of the rectangle but not the parallelogram.

But then, there's a green triangle at the top, with the same size, which is (mostly) part of the parallelogram but not of the

rectangle, so these cancel each other out. We follow the same logic for the red triangles.

Putting all this together, the rectangle with area ad has the same area as the parallellogram, except that we are overcounting three elements (outlined in blue): the two small triangles in the box at the top, which are not part of the paralellogram, and the overlap between the green and the red triangles, which we've counted twice. These three overcounted elements add up precisely to the box at the top-right, which has area bc.

So, the area of the paralellogram, and therefore the determinant of the matrix A is $ad - bc$. Or, in words: the determinant of a 2×2 matrix is the diagonal product minus the antidiagonal product.

We will write the determinant of a matrix A with two vertical bars around the matrix. When we write out the values of the matrix explicitly, we will remove the matrix parentheses for clarity:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc\,.$$

3.2.2 Negative determinants

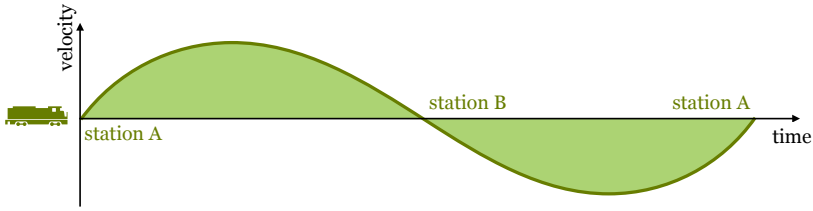
In the picture we drew to derive this, ad was bigger than bc, so the determinant was positive. But this is not guaranteed. Look at the two column vectors of our matrix. If we flip them around then the determinant becomes:

$$\begin{vmatrix} b & a \\ d & c \end{vmatrix} = bc - da\,.$$

Assuming the values in the matrix are the same, this is the same quantity as before, but negative. Areas can't be negative, so how do we interpret this?

The magnitude remains the same, so the simplest solution is just to adjust our definitions: *the absolute value* of $|A|$ is the amount by which A inflates space.

However, in many situations, the idea of a "negative area" actually makes a lot of sense. Consider, for instance, this graph of the velocity of a train along a straight track from one station to another and back again:



Here, we've used a negative velocity to represent the train traveling backwards. If you've done some physics, then you'll know that the area under the speed curve represents distance traveled. Here we have two options: we can look at the absolute value of the area, and see that the train has, in total, travelled twice the distance between the stations. We can also take areas below the horizontal axis to be *negative*. Then, their sum tells us that the total distance between the train's starting point and its final position is exactly zero.

All this is just to say, if you need positive areas, just take the magnitude of the determinant, but don't be too quick to throw away the sign. It may have some important meaning in your particular setting. For our purposes, we'll need these kinds of areas when we want to think about determinants for larger matrices.

We'll call this kind of positive or negative area a *signed area*, or signed *volume* in higher dimensions. You can think of the parallelogram as a piece of paper. If $\mathbf{A}$ stretches the paper, but doesn't flip it around, the singed area is positive. If the paper is flipped around, so that we see the reverse, the area is negative. If you flip the paper around twice, the sign becomes positive again.

3.2.3 Towards $n \times n$ determinants

Let's think about what we'll need to generalize this idea to 3×3 matrices and beyond, to general $n \times n$ matrices. The basic intuition generalizes: we can start with a unit (hyper)cube in n dimensions. A square matrix transforms this into an analogue of a parallelogram, called a parallellotope.

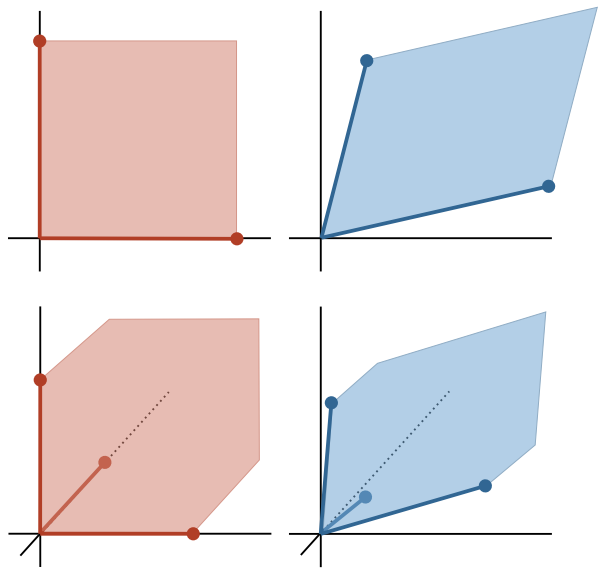
For a given dimensionality we can define a notion of n-volume. The 3-volume is simply the volume we already know. The n-volume of an n-dimensional "brick", the analogue of a rectangle, is the product of its extent in each direction: height times width

times length and so on in all directions. This means that the unit hypercube, which has sides of length 1 in all directions, always has n-volume 1.

We will assume, by analogy with the 2×2 case, that the determinant of an $n \times n$ matrix $\mathbf{A}$ is the n-volume of the parallellotope that results when we transform the unit hypercube by $\mathbf{A}$.

> *Note that there are* 3×3 *matrices that will flatten the unit cube into a parallelogram. In this case, we are* ***not*** *interested in the* area *of the parallelogram as we were before. The matrix is* 3×3*, so we care about the resulting* volume*, which in such cases would be* 0*.*

We can generalize a few useful properties from the 2×2 case. The columns of $\mathbf{A}$ are those vectors that describe the edges that touch the origin. We'll call these the **basis vectors** of the parallelogram/-tope.



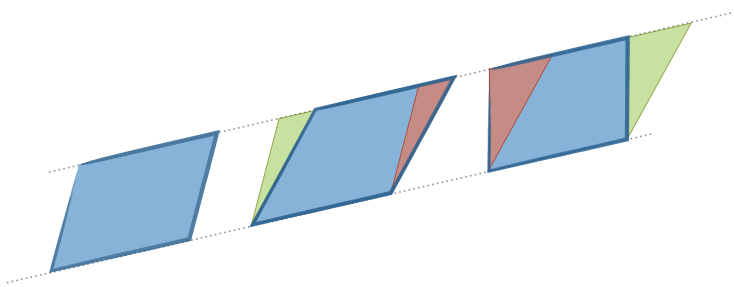
> The edges of the unit cube that touch the origin are the standard basis vectors. These are mapped to the edges of the parallelotope that touch the origin. These are the column vectors of $\mathbf{A}$. We call these the *basis vectors of the parallelotope*.

The proof we gave for the 2×2 determinant is very neat, but it isn't very easy to generalize to the $n \times n$ case in an intuitive way.

Instead, let's re-prove our result for the 2×2 case in a way that's easier to generalize. We'll need to convince ourselves of three properties of the area of a parallellogram.

These are not difficult to prove, but we'll focus here on the geometric intuition. If you want a more rigorous proof, it's easier to let this intuition go, and work purely symbolically.

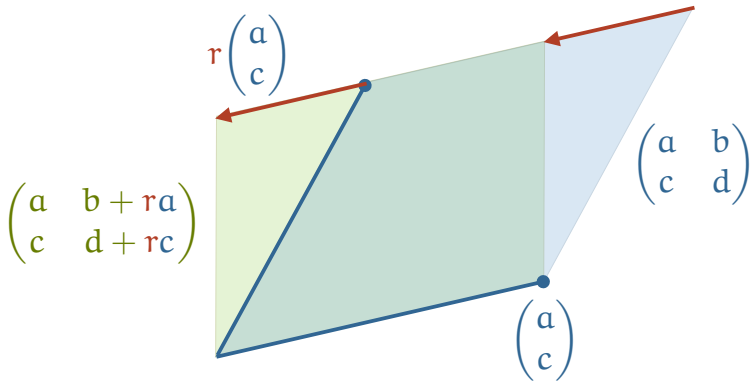
The **first property** we need is that if we move one of the sides of the parallelogram without changing its direction, the area of the parallellogram remains the same. This is easy to see visually.



A skew transformation preserves the area of a parallelogram. This means we can align one of the edges with the axes without changing the area.

Note that shifting one of the sides always adds a triangle to the parallellogram and takes away a triangle of the same size, so the total stays the same. The last example is particularly relevant: we can shift the parallelogram so that one of its edges is aligned with one of the axes. If we do this twice, we'll have a *rectangle* with an area equal to that of the original parallellogram.

What does this look like in the original matrix? Remember that the columns of the matrix are the two edges of the parallelogram that touch the origin. Shifting one of them in this way is equivalent to adding or subtracting a small multiple of the other to it.



To axis-align a parallellogram defined by a matrix, we take one of the columns, and add or subtract some multiple of the other column.

That is, if we take one of the columns of A, multiply it by any non-zero scalar, and add it to another column, the area of the resulting parallelogram is unchanged. If we name the column vectors v and w, and write $|v, w|$ for the determinant of the matrix with these column vectors, then we have

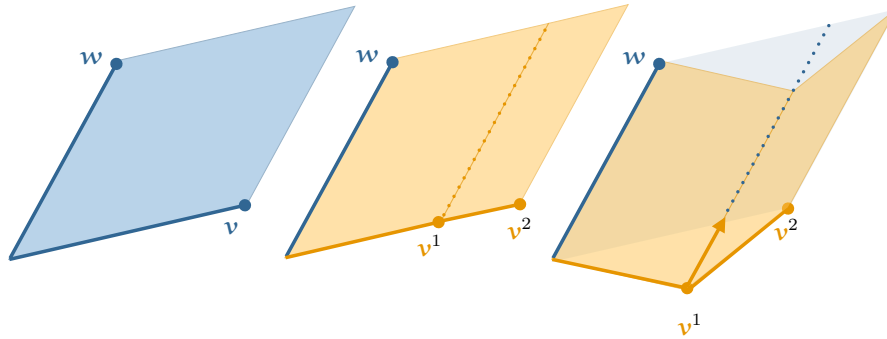
$$|v, w| = |v + rw, w| \text{ for any nonzero } r.$$

This kind of transformation is called a skew or a shear, so we'll call this property **skew invariance**: the area of a parallelogram is skew invariant.

The **second property** we need, is that if we take one of the column vectors of the matrix, say v, and write it as the sum of two other vectors $v = v^1 + v^2$, then the area of the paralellogram made by basis vectors v, w is the sum of the area of the two smaller parallelograms with basis vectors v^1, w and v^2, w respectively.

This is easy enough to see if v^1 and v^2 point in the same direction. Then the two smaller parallelograms together simply combine to form the larger one.

If they don't point in the same direction, we can skew them until they do. Since we've already shown that the area is skew invariant, none of this changes the area of the paralellogram.



If we break one of the basis vectors into the sum of two other vectors, the original area is the sum of the two parts. This is easy to see if the sub-vectors point in the same direction as the original. If they don't, we simply skew them until they do.

Symbolically, this means that if we break one of the vectors in a matrix into a sum of two other vectors, then the determinant distributes over that sum:

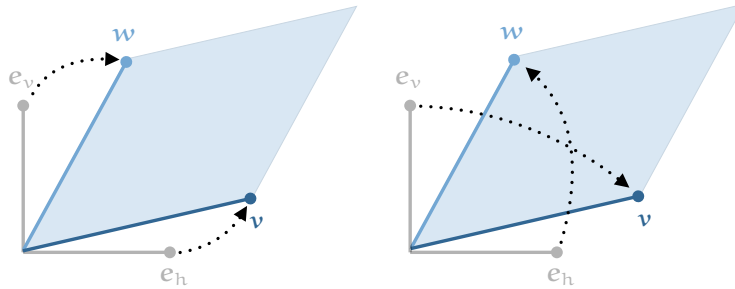
$$|\, v^1 + v^2, w \,| = |\, v^1, w \,| + |\, v^2, w \,|.$$

We won't need it here, but we can also show that multiplying one of the vectors by some scalar scales the determinant by the same value. These two properties together are called **multilinearity**: the area of a paralellogram is a multilinear function of the basis vectors. It's a linear function of one of its arguments if we keep the others fixed.

We need **one more property**: if we start with a parallelogram with basis vectors v, w, and we flip around the vectors, w, v, what happens to the area? If we look at the picture of the parallellogram, at first, it's difficult to see that anything changes at all. The two basis vectors are still the same. To see what happens, we need to look at the operation of the matrix with these vectors as its columns.

The matrix with columns v, w maps the horizontal unit vector e_h to v, and the vertical unit vector e_h to w. For the matrix with the columns swapped, we reverse this mapping. We

get the same, parallelogram, but it's as if we've turned it over. Since the unit square has positive area, the flipped-over paralellogram has negative area.



Swapping the columns of a matrix swaps the basis vectors of the paralellogram, turning a positive area into a negative area.

Put simply, swapping basis vectors around maintains the magnitude of the area of a parallellogram, but changes the signs.

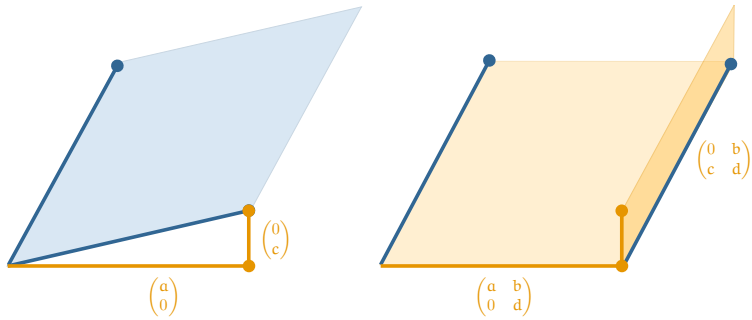
$$|\, v, w \,| = -|\, w, v \,|\,.$$

We call this property **alternativity**. As in, the area of a parallelogram is and alternating function of its basis vectors.

With these three properties: skew invariance, multilinearity and alternativity, we can work out our new proof of the determinant formula. First, using multilinearity, we write the first column of matrix $\mathbf{A}$ as the sum of two vectors:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} a & b \\ 0 & d \end{vmatrix} + \begin{vmatrix} 0 & b \\ c & d \end{vmatrix}\,.$$

We'll call this kind of vector, where only one element is non-zero, a **simple vector**. Here's a visualization of that step.

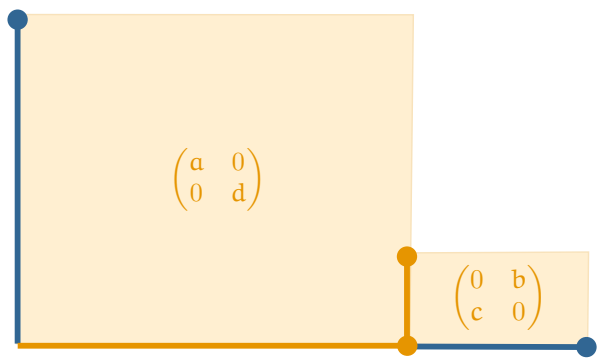


Breaking the determinant up in to two terms. Note that in the second image, the smaller parallelogram has negative area, because the two basis vectors are reversed.

Next, we use the property of skew invariance to subtract multiples of these new columns from the others. We use whatever multiple is required to make the rest of the row 0. The other rows are unaffected, since these all have 0's in the original column.

$$\begin{vmatrix} a & b \\ 0 & d \end{vmatrix} + \begin{vmatrix} 0 & b \\ c & d \end{vmatrix} = \begin{vmatrix} a & 0 \\ 0 & d \end{vmatrix} + \begin{vmatrix} 0 & b \\ c & 0 \end{vmatrix}.$$

Visually, that looks like this.



Axis-aligning the remaining basis vectors.

We have two parallelograms with one edge axis-aligned, and we simply skew them so that the other edge is axis-aligned as well. the area we're looking for is now the sum of two rectangles, one of which of facing away from us.

For the first term, we can work out the determinant easily. A diagonal matrix transforms the unit cube to a rectangle, so we just multiply the values along the diagonal: ac. For the second term, we have an anti-diagonal matrix. We can turn this into a diagonal matrix by swapping the two columns. By the property of alternativity, this changes the sign of the area, so the resulting signed area is $-bc$:

$$\begin{vmatrix} a & 0 \\ 0 & d \end{vmatrix} + \begin{vmatrix} 0 & b \\ c & 0 \end{vmatrix} = \begin{vmatrix} a & 0 \\ 0 & d \end{vmatrix} - \begin{vmatrix} b & 0 \\ 0 & c \end{vmatrix} = ac - bc\,.$$

This was certainly a more involved way of deriving the formula for the area of a polygon, but the benefit here is that this method generalizes to higher dimensions.

3.2.4 Determinants for $n \times n$ matrices

We'll start with the three properties we used above, and see how they generalize to higher dimensions.

Skew invariance also holds in higher dimensions. If we have an $n \times n$ matrix A with n column vectors $a^1, \ldots, a^n$, adding a multiple of one to another does not change the volume of the resulting parallelotope. For instance:

$$|u, v, w| = |u + rv, v, w|$$

Multilinearity also carries over in the same way. We can break one of the column vectors up into a linear combination of two (or more) other vectors and the area of the resulting paralellotope breaks up in the same way. For instance:

$$|u^1 + u^2, v, w| = |u^1, v, w| + |u^2, v, w|$$

Finally, **alternativity**. This requires a little more care. You'd be forgiven for thinking that since we have a higher-dimensional space, we now have more ways for our paralellotope to orient as

well. We thought of our paralellogram as a piece of paper which could lie on the table in two ways. If we hold a parallellotope up in space, we can rotate it in all sorts of directions.

But the metaphor of a piece of paper is slightly misleading. When we flip a piece of paper upside-down, we rotate it, but that's not what really happens when we swap the basis vectors of the paralellogram. What really happens is that we turn the piece of paper inside-out: we *flip* it by pulling the right edge to the left and the left edge to the right.

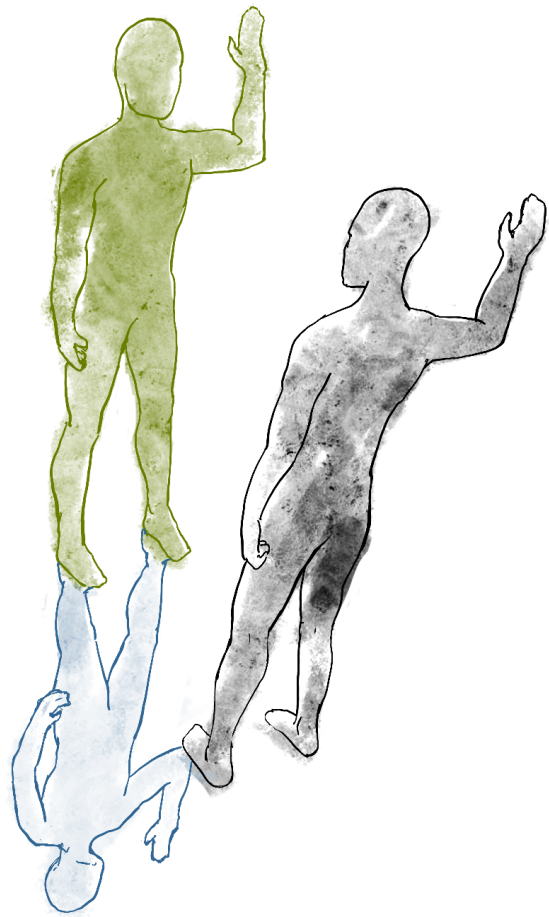
A better metaphor is a mirror: imagine standing in front of a mirror and holding up your right hand, palm forward.

It looks like your twin inside the mirror is holding up their left hand, with the thumb facing in the opposite direction. But if the mirror flips the image left-to-right, why doesn't it flip the image up-side-down as well? Why doesn't anything change if we turn the mirror ninety degrees?How does the mirror keep track of which direction the floor is?
The answer is that the mirror *doesn't* flip the image left-to-right. It flips it *back-to-front*. In a manner of speaking, it pulls the back of your hand forward and the front of your hand backwards until the whole hand is flipped.

Putting the mirror to your side will flip you left-to-right, also turning your right hand into a left hand. Putting the mirror on the floor and standing on it turns you upside-down, and again turns the right hand into a left hand. If you use two mirrors, one in front of you and one below, you are flipped back-to-front *and* upside-down, and if you look at this mirror-twin, you'll see that their hand has been flipped twice, to become a right hand again.

The result is that we still have only two orientations. Each time anything gets mirrored, it gets pulled inside out along some line, and the sign of the volume changes: your right hand turns into a left hand and vice versa. If it gets flipped an even number of times the sign stays the same, and if it gets flipped an odd number of times, the sign changes.

This means that alternativity in higher dimensions is defined as follows: if we swap around any two columns in a matrix, we flip the space (along a diagonal between the two corresponding axes). Therefore, the magnitude of the determinant stays the



A mirror in front of you seems to flip you left-to-right, but it actually flips you back-to-front. This *also* turns your right hand into a left hand. A second mirror on the floor flips you upside-down, turning your hand back into a right hand again.

same, but *the sign changes*. If we flip two more axes, the sign changes back. For instance:

$$|\,\mathbf{u},\mathbf{v},w\,| = -|\,\mathbf{v},\mathbf{u},w\,| = |\,w,\mathbf{u},\mathbf{v}\,|.$$

That's our three properties in place. Finally, before we start our derivation, note that in the 2×2 case, our ultimate aim was to work the matrix determinant into a sum of determinants of diagonal matrices. The idea was that the determinant of a diagonal matrix is easy to work out.

That's still true in higher dimensions: the columns of a diagonal matrix $\mathbf{A}$ each map one of the unit vectors to a a basis vector of length A_{ii} that lies along the i-th axis. Together these form the sides of an n-dimensional "brick" whose volume is just these lengths multiplied together. So the plan stays the same: use our three properties to rewrite the determinant into a sum of determinants of diagonal matrices.

We'll work this out for a 3×3 matrix explicitly as an example, but the principle holds for any number of dimensions.

We start by taking the first column of our matrix, and breaking it up into three simple vectors (using multilinearity).

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = \begin{vmatrix} a & b & c \\ 0 & e & f \\ 0 & h & i \end{vmatrix} + \begin{vmatrix} 0 & b & c \\ d & e & f \\ 0 & h & i \end{vmatrix} + \begin{vmatrix} 0 & b & c \\ 0 & e & f \\ g & h & i \end{vmatrix}$$

As before, we've broken our parallellotope into three parallellotopes, each with one of their edges axis-aligned. Next, we subract multiples of the first column from each of the other columns to turn the rows into vectors with only one non-zero element.

$$\begin{vmatrix} a & 0 & 0 \\ 0 & e & f \\ 0 & h & i \end{vmatrix} + \begin{vmatrix} 0 & b & c \\ d & 0 & 0 \\ 0 & h & i \end{vmatrix} + \begin{vmatrix} 0 & b & c \\ 0 & e & f \\ g & 0 & 0 \end{vmatrix}$$

This doesn't yet look quite as simple as it did in our 2×2 case, but we can go back to the first step and break the second column vector of each term into simple vectors as well. For the first term that looks like this:

$$\begin{vmatrix} a & 0 & 0 \\ 0 & e & f \\ 0 & h & i \end{vmatrix} = \begin{vmatrix} a & 0 & 0 \\ 0 & 0 & f \\ 0 & 0 & i \end{vmatrix} + \begin{vmatrix} a & 0 & 0 \\ 0 & e & f \\ 0 & 0 & i \end{vmatrix} + \begin{vmatrix} a & 0 & 0 \\ 0 & 0 & f \\ 0 & h & i \end{vmatrix}$$

In the first term, one of the column vectors is zero. This means the parallelotope becomes a parallelo*gram*, with 0 volume, so we can remove this term. For the other two, we apply the multilinear property to sweep the rows, and we get

$$\begin{vmatrix} a & 0 & 0 \\ 0 & e & f \\ 0 & h & i \end{vmatrix} = \begin{vmatrix} a & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & i \end{vmatrix} + \begin{vmatrix} a & 0 & 0 \\ 0 & 0 & f \\ 0 & h & 0 \end{vmatrix}$$

The logic is the same for the green and blue terms. If we ignore the zeros that we added, we end up with a 2×2 submatrix, to which we can apply the same trick again to turn each term into two more terms. If we do this we get six terms in total.

$$\begin{vmatrix} a & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & i \end{vmatrix} + \begin{vmatrix} a & 0 & 0 \\ 0 & 0 & f \\ 0 & h & 0 \end{vmatrix} + \begin{vmatrix} 0 & b & 0 \\ d & 0 & 0 \\ 0 & 0 & i \end{vmatrix} + \begin{vmatrix} 0 & 0 & c \\ d & 0 & 0 \\ 0 & h & 0 \end{vmatrix} + \begin{vmatrix} 0 & b & 0 \\ 0 & 0 & f \\ g & 0 & 0 \end{vmatrix} + \begin{vmatrix} 0 & 0 & c \\ 0 & e & 0 \\ g & 0 & 0 \end{vmatrix}$$

The result is that we have separated the determinant into several terms, so that for each, the matrix in that term has only one non-zero element in each row and each column. The structure of the non-zero values is that of a *permutation matrix*, with the exception that permutation matrices contain only 1s and 0s. These are like the original matrix $\mathbf{A}$ with a permutation matrix used to mask out certain values.

In fact, what we have done is to enumerate *all possible permutations*. We started by creating one term for each possible choice for the first column, and then for each term we separated this in to the remaining two choices for the second column (after which the choice for the third column was fixed as well).

This idea naturally generalizes to higher dimensions. Moving from left to right, we pick one element of each column and zero out the rest of its column and row. At each subsequent step we limit ourselves to whatever non-zero elements remain. We sum the determinants of all possible ways of doing this.

We can now turn each of these matrices into a diagonal matrix, by swapping around a number of columns. By the property of alternativity, this doesn't change the magnitude of the determinant, only the sign: for an even number of swaps, it stays the same and for an odd number it flips around. This gives us

$$\begin{vmatrix} a & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & i \end{vmatrix} - \begin{vmatrix} a & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & h \end{vmatrix} - \begin{vmatrix} b & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & i \end{vmatrix} + \begin{vmatrix} c & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & h \end{vmatrix} + \begin{vmatrix} b & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & g \end{vmatrix} - \begin{vmatrix} c & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & g \end{vmatrix}.$$

And since the determinant of a diagonal matrix is simply the diagonal product, as we worked out earlier, we get

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei - afh - bdi + cdh + bfg + ceg.$$

To generalize this to n dimensions, we represent a permutation of the first n natural numbers with the symbol σ. For instance, if $n = 6$ we may have $\sigma = \langle 1, 5, 4, 3, 2 \rangle$. We'll call the *sign* of a permutation, $\operatorname{sign}(\sigma)$, -1 if the permutation can be placed in the correct order with an odd number of swaps and 1 if this can be done with an even number of swaps.

Then, the process we described above leads to the following formula for the determinant:

$$|\mathbf{A}| = \sum_{\sigma} \operatorname{sign}(\sigma) \prod_{i} A_{\sigma(i),i}$$

where the sum is over all permutations of the first n natural numbers, and the product runs from 1 to n.

Note that for each term in our sum, corresponding to the permutation σ, the value $A_{\sigma(i),i}$ marks out one of the elements that we haven't zeroed out. The value $\prod_i A_{\sigma(i),i}$ is the product of all these values.

This is called the **Leibniz formulation** of the determinant.

We got here by defining the determinant as the volume of a parallelotope and then deriving the Leibniz formulation from that.

This is useful for building a visual intuition, but if you want to be rigorous it's not the most efficient approach.

For this reason, the determinant is usually first defined as any *multilinear, alternative function, which yields 1 for the identity matrix. You can then show that there is only one such function, and that it's the Leibniz function above. From there, you can then prove all the other interpretations of the determinant, including the geometrical one.*

The determinant is a powerful tool, with many uses. Here, we only care about one of them: *it lets us characterize whether a matrix is invertible or not*. The determinant is precisely zero if and only if the matrix is not invertible. The reason we care about this, is that it will lead to a very useful way of characterizing the eigenvalues: the **characteristic polynomial** of a matrix.

3.3 The characteristic polynomial

Let's start with how we originally characterized eigenvalues in Chapter 2. There, we said that an eigenvector of a matrix $\mathbf{A}$ is any vector $\boldsymbol{v}$ for which the direction doesn't change, under operation of the matrix. The magnitude *can* change, and the increase of that magnitude we call the eigen*value* λ corresponding to the eigenvector.

To summarize, for any eigenvector $\boldsymbol{v}$ and its eigenvalue λ of $\mathbf{A}$, we have

$$\mathbf{A}\boldsymbol{v} = \lambda\boldsymbol{v}\,.$$

Moving the right-hand-side over to the left, we get

$$\mathbf{A}\boldsymbol{v} - \lambda\boldsymbol{v} = \mathbf{0},$$

where both sides are vectors, with a vector of zeros on the right. To allow us to manipulate the left hand side further we rewrite $\lambda\boldsymbol{v}$ as $(\lambda\mathbf{I})\boldsymbol{v}$. This gives us:

$$\begin{aligned}\mathbf{A}\boldsymbol{v} - \lambda\mathbf{I}\boldsymbol{v} &= \mathbf{0}\\ (\mathbf{A} - \lambda\mathbf{I})\boldsymbol{v} &= \mathbf{0}\,.\end{aligned}$$

This last line is basically a linear problem of the form $\mathbf{M}\boldsymbol{v} = \mathbf{0}$, with $\mathbf{M} = (\mathbf{A} - \lambda\mathbf{I})$. The solutions to this problem, the set of all vectors $\boldsymbol{v}$ that we can multiply by $\mathbf{M}$ to get the null vector $\mathbf{0}$ are called the *null space* of $\mathbf{M}$. What we have just shown is that **any eigenvector of $\mathbf{A}$ with eigenvalue λ must be in the null space of the matrix $\mathbf{A} - \lambda\mathbf{I}$.**

This is where invertibility and the determinant come in: if $\mathbf{M}$ is invertible, it can only map one point to any other point. Every matrix maps $\mathbf{0}$ to $\mathbf{0}$, so for any invertible matrix the null space consists only of the point $\mathbf{0}$. The only matrices with more interesting null spaces are non-invertible matrices. Or, matrices with determinant 0.

> *Note that we're not talking about the invertibility of $\mathbf{A}$ itself, only of the derived matrix $\mathbf{M} = \mathbf{A} - \lambda\mathbf{I}$*

So, now we can tie it all together. Choose some scalar value λ. If we have

$$|\,\mathbf{A} - \lambda\mathbf{I}\,| = 0$$

then the matrix $\mathbf{A} - \lambda\mathbf{I}$ has a non-trivial null-space, and λ is an eigenvalue. We want to study the left-hand-side of this equation as a function of λ, taking the values in $\mathbf{A}$ as constants.

As we've seen, expanding the determinant into an explicit form can get a little hairy for dimensions larger than 3, but we don't need to make it explicit, so long as we can tell what *kind* of function it is. To illustrate, say we have a 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

In that case, the value $|\,\mathbf{A} - \lambda\mathbf{I}\,|$ works out as

$$(a - \lambda)(d - \lambda) - bc\,.$$

We can multiply out these brackets, and we would see that this is a polynomial with λ as its variable. This polynomial has λ^2 as the highest power. The values for which this polynomial equal zero, its **roots**, are the eigenvalues of the original matrix $\mathbf{A}$.

For an $n \times n$ matrix, as we saw, the Leibniz form of the determinant gives us one term for each possible permutations of length n, each of which consists of n elements of the matrix multiplied together. If any of these elements come from the diagonal, they contain λ. That means each term contains at most n λs, giving us an n-th order polynomial.

As you may have guessed, this function is what we call the **characteristic polynomial** of $\mathbf{A}$. The points where this function is 0, the roots of the polynomial, are the eigenvalues of $\mathbf{A}$.

And this means that we can apply a whole new set of tools from the analysis of polynomials, to the study of eigenvectors. We never have to work out the characteristic polynomial explicitly, we can just use the knowledge that the determinant *is a polynomial* and use what we know about polynomials to help us further along towards the spectral theorem.

And one of the richest and most versatile tools to come out of the analysis of polynomials, is the idea of *complex numbers*. This is a counterintuitive idea, so we'll take some time to set it up carefully, before we dig into the mathematics.

3.4 Complex numbers

Complex numbers spring from the idea that there exists a number i for which i^2 is -1. We don't know of any such number, but we simply assume that it exists, and investigate the consequences. For many people this is the point where mathematics becomes too abstract and they tune out. The idea that squares can be negative clashes too much with our intuition for what squares are. The idea that we just pretend that they can be negative and keep going, seems almost perverse.

And yet, this approach is one that humanity has followed again and again in the study of numbers. If you step back a bit, you start to see that it is actually one of the most logical and uncontroversial things to do.

The study of numbers started somewhere before recorded history, in or before the late stone age, when early humans began counting things in earnest, and they learned to *add*. I have five apples, I steal three apples from you, now I have eight apples. That sort of thing.

At some point, these early humans will have solidified their concept of "numbers." It is a set of concepts (whose meaning we understand intuitively) which starts $1, 2, 3, \ldots$ and continues. If you add one number to another, you always get another number. If the number was big, they may not have had a name for it, but a patient paleolithic human with enough time could certainly have carved the required number of tally marks into an animal bone.

The operation of addition can also be reversed. If $5 + 3$ gives 8, then taking 5 away from 8 gives 3. If I steal 5 apples from your collection of 8, you still have 3 left. Thus, subtraction was born. But subtraction, the *inverse* of addition, required some care. Where adding two numbers always yields a new number, subtracting two numbers doesn't always yield a new number. You can't have $5 - 8$ apples, because if you have 5 apples I can't steal more than 5 of them.

As societies grew more complicated, financial systems developed and *debt* became an integral part of daily life. At some point, the following thought experiment was considered. What if $5 - 8$ is a number after all? Maybe it's just a number we don't have a name for yet.

So, we'll just give it a name and see if we can make some sense of how it behaves. No doubt many people were outraged by such a suggestion, protesting that it was unnatural, and an insult to whatever God they believed had designed the numbers. But simple investigation soon showed that if these numbers were assumed to exist, they followed simple rules and, it made sense to think of them as a kind of mirror image of the natural numbers, extending to infinity in the opposite direction. $5 - 8$ was the mirror image of 3, so it made sense to call it "-3".

The skeptics might argue that this made no sense, because there is no such thing as having -3 apples, but the mathematicians will have countered that in other areas, such as finance, there were concepts that could be expressed very beautifully by the negative numbers. If I owe you 3 apples, because my earlier theft was found out, but you also stole 8 apples from me, I now owe you -5 apples, or rather, you owe me 5.

The same principle can be applied to multiplication. If your tribe has 8 families, and every family is entitled to 5 apples, you

need to find 8×5 apples. Again, an operator, and any two numbers you care to multiply will give you a new number (even if you believe in negative numbers).

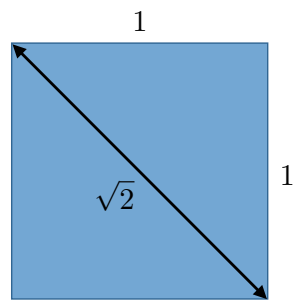
And again, you can do the reverse the operation: if the harvest has yielded 48 apples, you can work out that every family in your tribe gets 6 of them. But again, you have to be careful about which numbers you apply the inverse to. Sometimes you get a known number, and sometimes you don't. If you have 50 apples, suddenly there is no known number that is the result of $50/8$.

But what if there was? What if we just gave $50/8$ a name and started investigating? We'd find out pretty quickly that it would make sense to think of these numbers as lying *in between* the integers. We call these the *rational* numbers. Whoever it was that invented the rationals must have run into less resistance than the inventor of the negative numbers; it's much easier to imagine half an apple than to imagine -3 of them.

The pattern is hopefully becoming clear. Let's have one more example, to really drive the point home, and also to bring us far enough into recorded history so we can actually see how people dealt with these revelations. Adding is repeated counting, and multiplication is repeated adding so *raising to a power*, repeated multiplication, is the next operator in the hierarchy.

The story should be familiar at this point. Any two natural numbers a and b can be "exponentiated" together as a^b and the result is another natural number.

The inverse operation is a b-th root, but we can stick with square roots to illustrate our point. In fact the square root of 2, the length of the diagonal of a unit square, is all we need. In this case, there is nothing abstract or perverse about the quantity $\sqrt{2}$: in a square room with sides of 1 meter it's the distance from one corner to the corner opposite.



And yet, when people investigated, it caused great upset.

The man who gave his name to the theorem we would use to work out the above picture, Pythagoras, was the head of a cult. A cult dedicated to mathematics. They lived ascetically, much like monks would, centuries later, and dedicated themselves to the study of nature in terms of mathematics. When asked what the purpose of man was, Pythagoras answered "to observe the heavens." One fervent belief of the Pythagoreans was that number and geometry were inseparable: all geometric quantities could be expressed by (known) numbers.

The story of the Pythagoreans is a mathematical tragedy. It was one of their own, commonly identified as Hippasus of Metapontum, who showed that no rational number corresponded exactly to $\sqrt{2}$. Some aspects of geometry were outside the reach of the known numbers. According to legend, he was out at sea when he discovered this, and was promptly thrown overboard by the other Pythagoreans.

Of course, with the benefit of hindsight, we know how to manage such upsetting discoveries. We simply give the new number a name, "$\sqrt{2}$," and see if there's some place among the numbers where it makes sense to put it. In this case, somewhere between $141/100$ and $142/100$, in a space we can make infinitely small by choosing better and better rational approximations.

With this historical pattern clearly highlighted, the discovery of the complex numbers should be almost obvious. In fact, we don't even need a new operation to invert, we are still looking at square roots, but instead of applying the square root a positive integer, we apply it to a *negative integer*. To take the simplest example, we'll look at $\sqrt{-1}$. No number we know gives -1 when we multiply it by itself, so our first instinct is to dismiss the operation. The square root is only allowed for a subset of the real-valued numbers. Just like subtraction was only allowed for a subset of the natural numbers, and division was only allowed for a subset of the integers.

But, what if the number $\sqrt{-1}$ did exist? What would the consequences be?

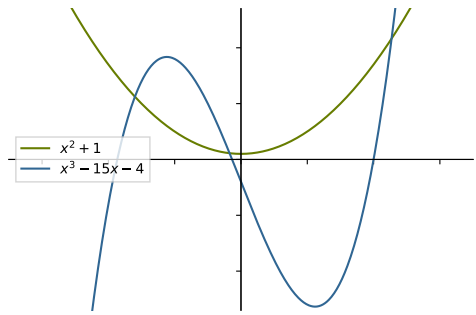
As the previous paragraphs should illustrate, this kind of investigation is usually born out of necessity. Like a fussy child

given a new food, people are consistently reluctant to accept new types of numbers. In this case, what pushed us over the edge was the study of polynomials; functions of the form:

$$f(x) = ax^3 + bx^2 + cx + d$$

where the highest exponent in the sum indicates the *order* of the polynomial.

The problem of finding the *roots* of a polynomial, the values of x for which $f(x)$ is equal to 0 crops up in all sorts of practical problems. In some cases, this leads to squares of negative numbers, as we see when we try to solve $x^2 + 1 = 0$. This didn't worry anybody, of course, since this function lies entirely above the horizontal axis—it has no roots—so it's only natural that solving for the roots leads to a contradiction.



The function $f(x) = x^2 + 1$ has no roots: it doesn't cross the horizontal axis. Therefore, it make sense that $x^2 = 0$ or $\sqrt{-1}$ has no solutions.

However, when people started to work out general methods for finding the roots of *third*-order polynomials, like $x^3 - 15x - 4$, which *does* have roots, it was found that the methods worked if one temporarily accepted $\sqrt{-1}$ as an intermediate value, which later canceled out. This is where the phrase *imaginary* number

originates. People (Descartes, to be precise) were not ready to accept these as numbers, but no one could deny their utility.

Eventually, people followed the pattern that they had followed for the integers, the rationals and all their successors. We give the new number a name, $i = \sqrt{-1}$, and we see if there's any way to relate it, geometrically, to the numbers we know.

Let's start with addition. What happens if we add i to some number, say 3? The simple answer is that nothing much happens. The most we can say about the new number is that it is $3 + i$.

Multiplication then. Again $2 \times i$ doesn't simplify in any meaningful way, so we'll just call the new number $2i$. What if we combine the two? With a few subtleties, we can rely on the basic rules of algebra to let us multiply out brackets and add things together. So, if we start with i, add 3 and multiply by 2, we get:

$$2(i + 3) = 2 \cdot 3 + 2 \cdot i = 6 + 2i$$

This is a very common result: we've applied a bunch of operations, involving the imaginary number i, and the result can be written as the combination of a real value r, another real value c and i as:

$$r + ci\,.$$

We will call any number that can be written in this way a *complex number*. The set of all complex numbers is written as

$$\mathbb{C}\,.$$

At this point you may be worried. What if we come up with another operation that is not defined for all complex numbers? Are we going to have to make another jump? Are we going to find ever bigger families of numbers to deal with? It turns out that in many ways, $\mathbb{C}$ is the end of the line. So long as we stick to algebraic operations, we can do whatever we like to complex numbers, and the result will always be well defined as another complex number.

To illustrate, let's show this for a few simple examples. Lets say we have two complex numbers $a + bi$ and $c + di$. If we add them, we get

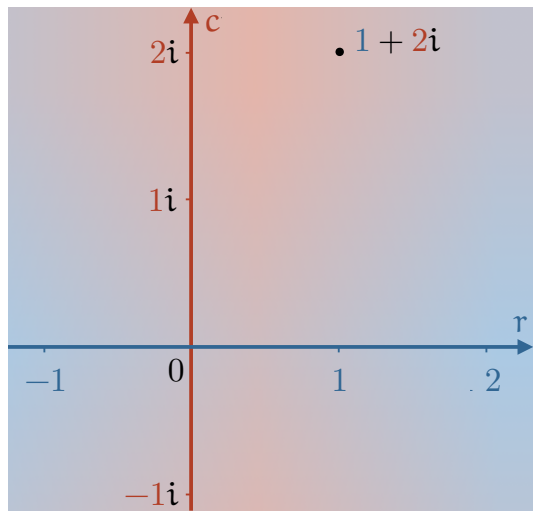
$$(a + bi) + (c + di) = a + c + bi + di = (a + c) + (b + d)i$$

If we multiply them, we get

$$\begin{aligned}(a + bi)(c + di) &= ac + adi + bic + bidi \\ &= (ad + bdi^2) + (ad + bc)i \\ &= ad - bd + (ad + bc)i\,.\end{aligned}$$

That is, one real-valued number, added to i times another real-valued number. Note that in the second line of the derivation for the multiplication, we can use $i^2 = -1$, since we know that $i = \sqrt{-1}$. In short, multiplying or adding together any two complex numbers gives us another complex number.

Since every complex number can be written as the combination of two real-valued numbers, it makes sense to visualize them in a plane. We plot the value of the real term along the horizontal axis and the value of the imaginary term along the vertical.

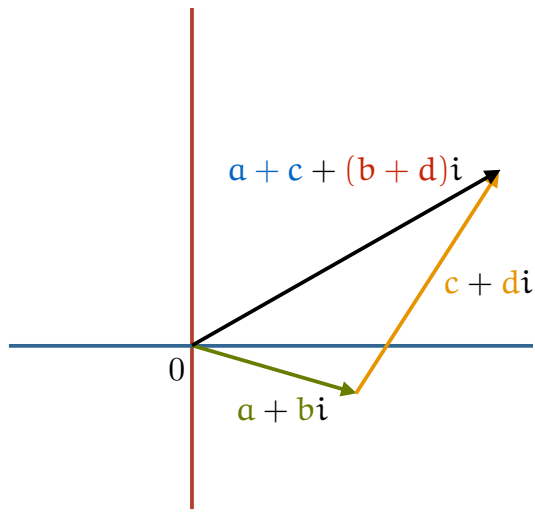


The real-valued numbers that we already knew are a subset of the complex numbers: those complex number for which the imaginary part is zero. In this picture, the real-valued numbers are on the horizontal axis.

Note that this is just a visualization. There is nothing *inherently* two-dimensional about the complex numbers, except that there is a very natural mapping from $\mathbb{C}$ to $\mathbb{R}^2$. At heart, it's just a set of numbers with a bunch of operations defined for them.

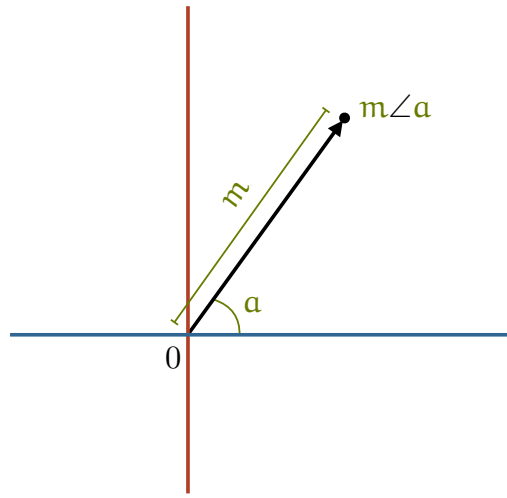
The nice thing about the mapping to the plane, however, is that we can take operations like multiplication, addition and so on, and see what they look like in this picture. This way, we can build a very helpful visual intuition for how the complex numbers behave.

Let's look at the most important concepts we'll need going forward. For addition, we can build on our existing intuitions. Adding two complex numbers works the same as adding two vectors in the plane: we place the tail of one on the head of the other.



The same logic shows that subtraction of complex numbers behaves as you'd expect. To compute $x-y$, we subtract the real part of x from the real part of y and likewise for the imaginary part. Geometrically, this corresponds to vector subtraction in the plane.

To see what multiplication looks like, we can switch to a different way of representing complex numbers. Instead of giving the Cartesian coordinates (r, c) that lead to the number $z = r + ci$, we use the *polar* coordinates. We give an angle a from the horizontal axis and a distance m from the origin. The angle is also called the *phase* and the distance is called the *magnitude* or the *modulus*. When we write a number like this, we'll use the notation $z = m\angle a$. To refer to the magnitude of a complex number z, which we'll be doing a lot, we use the notation $|z|$.



We call this representation of a complex number *polar notation*, and the earlier representation *Cartesian notation*.

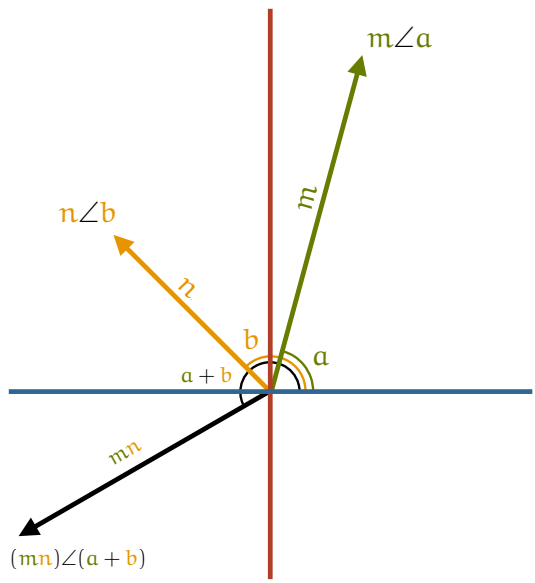
The reason polar notation is so useful, is that multiplication looks very natural in it. To see the relation, assume that we have a number $z = m\angle a$. Then basic trigonometry tells us that in Cartesian notation, this number is written as $z = m\cos(a) + m\sin(a)i$.

Let's see what happens if we take two numbers, in polar notation, and multiply them:

$$
\begin{aligned}
&(m\angle a)(n\angle b) \\
&= (m\cos(a) + m\sin(a)i)\ (n\cos(b) + n\sin(b)i) \\
&= m\cos a\ n\cos b + m\sin a\ n\sin b + (m\cos a\ n\sin b + n\cos b\ m\sin a)i \\
&= mn(\cos a\cos b - \sin a\sin b) + mn(\cos a\sin b + \cos b\sin a)i \\
&= mn\cos(a+b) + mn\sin(a+b)i \\
&= (mn)\angle(a+b)
\end{aligned}
$$

In the third line, we apply the multiplication in Cartesian notation that we already worked out earlier. Then, in the fifth line, we apply some basic trigonometric sum/difference identities.

What this tells us, is that when we view complex numbers in polar coordinates, multiplication has a very natural interpretation: the angle of the result is the *sum* of the two original angles, while the magnitude of the result is the *product* of the two original magnitudes.



The easiest way to define *division* is as the operation that cancels out multipliciation. For each z, there should be a z^{-1} so that multiplying by z and then by z^{-1} brings you back to where you were. Put simply $zz^{-1} = 1$. Dividing by z can then be defined as multiplying by z^{-1}. Using the polar notation, we can see that the following definition of z^{-1} does the trick:

$$z^{-1} = (m\angle a)^{-1} = \frac{1}{m}\angle -a\,.$$

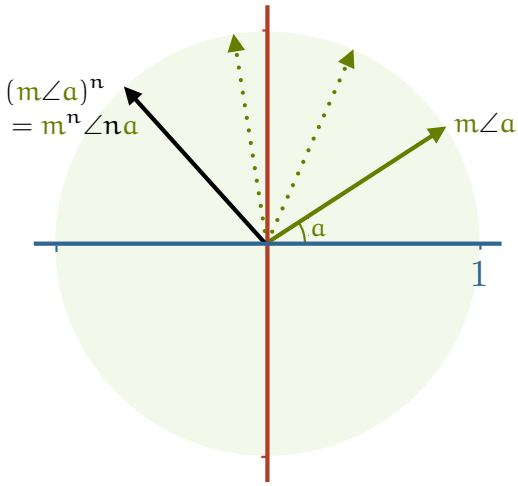
Note how this view of multiplication agrees with special cases that we already know. For real numbers, the angle is always 0, and the magnitude is equal to the real value. Therefore, multiplying real numbers together reduces to the multiplication we already knew.

The number i is written as $1\angle 90\,\text{deg}$ in polar coordinates. That means that multiplying a number z by i keeps the magnitude of z the same, but rotates it by 90 degrees counter-clockwise. A real number multiplied by i is rotated from the horizontal to the vertical axis. If we multiply by i twice, we rotate 180 degrees, which for real numbers means negating them. This makes sense too, because $z \cdot i \cdot i = zi^2 = z \cdot -1$.

Which brings us to exponentiation. Raising complex numbers to arbitrary values, including to complex ones, is an important topic, but one which we can sidestep here. All we will need is the ability to raise a complex number to a natural number. That follows very naturally from multiplication:

$$(m\angle a)^n = (m\angle a)(m\angle a)\ldots(m\angle a) = m^n\angle na\,.$$

Again, let's look at some special cases. If the angle is 0, we stay on the real number line, and the operation reduces to ordinary exponentiation. If the magnitude is 1 but the angle is nonzero, then we rotate about the origin over the unit circle in n steps of angle a.



The main thing we need, however, is not integer exponentiation, but its inverse: the n-th root. Given some complex number $z = m\angle a$, which other number do we raise to the power n so that we end up at z? The answer follows directly from our polar view of the complex plane: the magnitude should be $\sqrt[n]{m}$, which is just the real-valued n-th root, and the angle should be a/n.

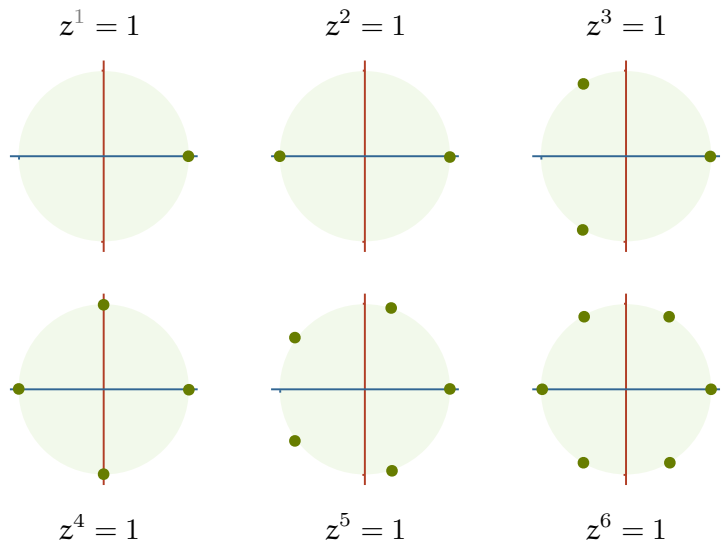
Let's check for $\sqrt{-1}$, which started all this business. Which number should we raise to the power 2, so that we get -1? The magnitude of -1 is 1, so our number has magnitude $\sqrt{1} = 1$. Now we need a number with magnitude one, so that twice its angle equals 180°. This is a 90° angle, so our number is $1\angle 90°$, which is exactly where we find i.

Notice how this solves the problem we had when we were constrained to the real line. Then we had negative numbers to deal with, and the real n-th root does not exist for negative numbers. Now, we are only ever applying the n-th root to *magnitudes*, which are positive. The rest is dealt with by rotating away from the real numbers. This means that when it comes to complex numbers, we can always find some number that, when raised to n, gives us z. We call this the complex n-th root $\sqrt[n]{z}$.

Note however, that this is not always a *unique* number. Let's say we raise $1\angle 10°$ to the power of 4. This gives us $1\angle 40°$, so

$1\angle 10°$ is a fourth root of $1\angle 40°$. However, if we raise $1\angle 92.5°$ to the power of 4, we get $1\angle 370°$, which is equal to $1\angle 10°$ as well. Any angle a' for which $a'\frac{1}{n} \bmod 360 = a$ will give us an n-th root of $m\angle a$.

How many solutions does this give us for any given number? It's easiest to visualize this if we plot the n-th roots of 1.

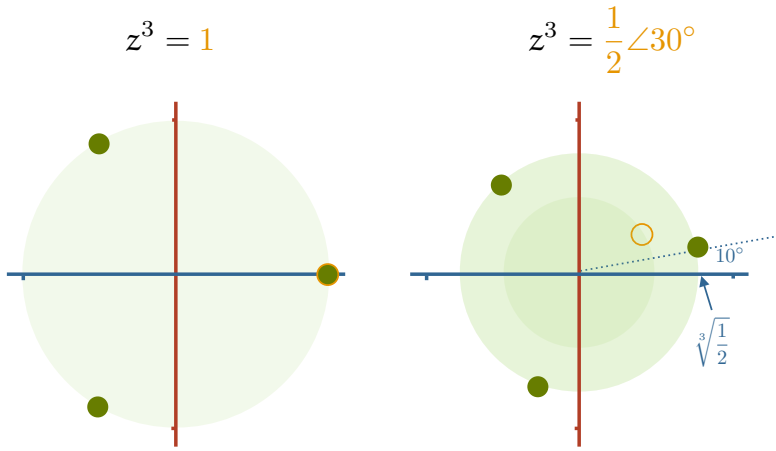


For each, of course, the real value 1 is a solution, but for the higher powers, there are additional solutions on the unit circle. For $\sqrt[2]{1}$, for instance, multiplying -1 by itself rotates it by 180 degrees to coincide with 1. For $\sqrt[3]{1}$, we get three roots, two of which are non-real. The solution with angle 120 deg, when raised to the power of 3 gives us an angle of 360 deg $=$ 0 deg. The solution with angle 240 deg puts the angle after cubing at 720 deg $=$ 0 deg.

In short, every multiple of 360: 0, 360, 720, 1080, ..., can be divided by n to give us a solution. Once we get to $360n$, dividing by n gets us back to a solution we've already seen, so we get n unique solutions in total.

To translate this to roots of any complex number $m\angle a$, we

simply scale the circle so that its radius is $\sqrt[n]{m}$ and then rotate it so that the first solution points in the direction of a/n.



3.5 The fundamental theorem of algebra

The reason we are bringing in complex numbers, is that we are interested in talking, in general terms, about the roots of the chracteristic polynomial. When all we have access to are real-valued numbers, this becomes a messy and unpredictable business. A polynomial of order n can have anywhere between 0 and n roots. When we add complex numbers to our toolbelt, the whole picture becomes a lot simpler. And that is down to a result called *the fundamental theorem of algebra*.

The theorem has many equivalent statements, but this is the one most directly relevant to our purposes.

The fundamental theorem of algebra Any non-constant polynomial of order n has exactly n complex roots, counting multiplicities.

For now, don't worry about what is meant by multiplicities. We'll dig into that later. A constant polynomial is a function like $f(x) = 3$, which will not have any roots.

To prove this theorem, the first thing we need is to show that each such polynomial has *one root*. After that, the rest is straightforward. So straightforward (to some) that this is often seen as an alternative statement of the fundamental theorem:

The fundamental theorem of algebra (variant) Any non-constant polynomial of order n has at least one complex root.

Let

$$p(z) = c_n z^n + \ldots + c_1 z + c_0$$

be our polynomial. For our purposes, we can think of the coefficients as real-valued, but the theorem holds for complex coefficients as well. The argument z and the result $p(z)$ can always be complex.

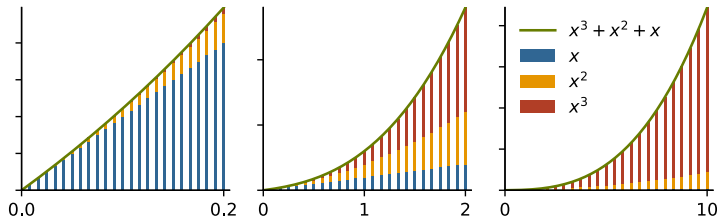
To find a root of p, we will consider the function $|p(z)|$. That is, the magnitude of the complex number that we get out of p. This provides the following benefits:

- The magnitude is always non-negative. That means that the lowest possible value that $|p(z)|$ can take on is 0, at which point we must have $p(z) = 0$. In short, for a root of $p(z)$, $|p(z)|$ is both 0 and at a minimum.
- Since the magnitude of a complex number is a single real value, $|p(x)|$ is a function from two dimensions (the complex plane) to one dimension (the reals) and we can easily visualize it in three dimensions. This is not so easy for $p(x)$ itself, since we have a two-dimensional input *and* a two-dimensional output.

Our big shortcut in this proof will be to look at what the magnitude does in extreme cases: for very large inputs, and for inputs very close to the minimum. We will see that in both cases, the function can be approximated well by the magnitude of a simple polynomial.

To see this, let's start with a simple real-valued example. The polynomial $p(x) = x^3 + x^2 + x$ in the positive range. In this

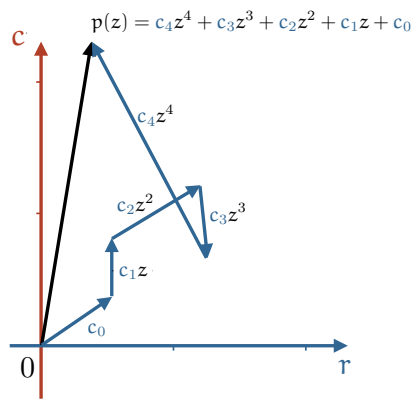
area, $p(x)$ is equal to its magnitude, so we don't need to worry about the distinction yet.



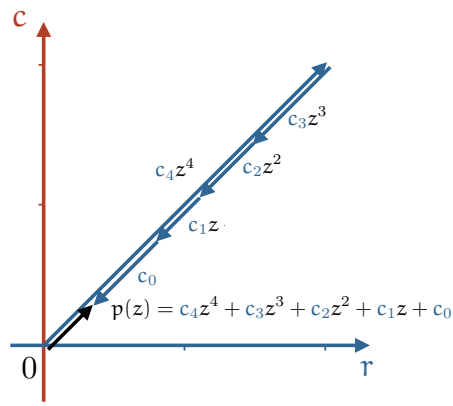
What we see here, is that as x gets bigger, the term x^3 dominates. Almost all the contribution to the magnitude comes from this term, and pretty soon, the simpler polynomial x^3 becomes a pretty good approximation of the polynomial $x^3 + x^2 + x$. This is not surprising, since the cube grows much faster than the square which grows much faster than the identity.

Toward $x = 0$, where p has a minimum, the opposite happens. As x^3 grows the fastest when $x > 1$, so it *shrinks* the fastest when $0 < x < 1$. In this regime the term that shrinks the slowest, x begins to dominate, and x becomes a good approximation of the function p.

Of course, this is just one polynomial. If we move to complex polynomials, and we allow for any order and all possible coefficients, does this pattern still hold? Let's imagine a generic complex polynomial. In this case, all terms in the polynomial are complex numbers, and the value of the polynomial is their sum.



The magnitude $|p(z)|$ of the polynomial at z is the distance of the end result to the origin. Each term contributes to this magnitude in a different direction. If we want to show that a particular term dominates, we can look at the worst case: that term points in one direction, and all other terms point in the exact opposite direction.



In this case, we can ignore the angles of the terms and focus only on their magnitudes. If we assume the highest-order term points in the opposite direction of the rest, the total magnitude is

$$\begin{aligned} |p(z)| &= |c_n z^n| - \left|c_{n-1} z^{n-1}\right| - \ldots - |c_1 z| - |c_0| \\ &= |c_n||z|^n - |c_{n-1}||z|^{n-1} - \ldots - |c_1||z| - |c_0| \end{aligned}$$

Note that the terms we are subtracting are all magnitudes, so they are all positive.

We will first use this to show that $|p(z)|$ has some definite minimum. One alternative situation would be if $p(z)$ is a function that is positive everywhere and monotonically increasing in some direction, like e^x is on the real number line. We'll need to exclude such possibilities first.

Assume that $|z| > 1$ for some z. If so, we make the total

value of the sum *smaller* if we replace all lower-order powers by z^{n-1}. This means that

$$\begin{aligned} |p(z)| &> |c_n||z|^n - |c_0| - \sum_{i \in 1..n-1} |c_i||z|^{n-1} \\ &= |c_n||z|^n - |c_0| - |z|^{n-1} \sum_{i \in 1..n-1} |c_i| \\ &= |z|^{n-1} \left(|c_n||z| - \sum_{i \in 1..n-1} |c_i| \right) - |c_0| \end{aligned}$$

If we choose z so that its magnitude is larger than $\frac{1}{|c_n|} \sum_i |c_i|$, the factor in brackets becomes positive. Beyond that, we know that there is some value of $|z|$ large enough that the first term is bigger than the second. In short, for a suffciently large B, we can always choose a value of z such that $|p(z)|$ is larger than B.

This means we can draw some large circle with radius R, find the smallest value of $|p(z)|$ inside the circle, and then draw a second circle with radius B so that all values outside of $p(z)$ outside the second circle are larger than this minimum inside the first circle. This means $|p(x)|$ has a definite minimum inside the second circle.

Now, all we need to do is show that this minimum can be expressed by a complex number. To do that, we'll follow the same sort of argument, but with the magnitude going to 0, so that the *lower-order* terms dominate.

First, let z_0 be the minimum we've just shown must exist. Translate p so that this minimum coincides with the origin, and call the result $q(z)$. Specifically, $q(z) = p(z - z_0)$.

This is another n-th order polynomial. We'll call its coefficients d_i. Note that $q(0)$ has the same value as $p(z_0)$ by construction. What we want to show is that $p(z_0) = 0$.

In many polynomials the lowest-order term is the first-order term $c_1 z$. However, we need to account for cases where this term is always zero. To be general, we write q as

$$q(z) = d_0 + d_k z^k + d_{k+1} z^{k+1} + \ \ldots \ + d_n z^n$$

where k is the order of the lowest-order, non-constant term.

We'll show that the proportion of the sum contributed by the higher order terms vanishes as we get near zero, so we can take the simpler function $q'(z) = c_0 + c_k z^k$ as a good approximation, that becomes perfect at the origin.

Note that $q(0) = q'(0)$.

More formally, lets look at the ratio between the higher-order terms and the kth-order term:

$$r = \frac{|d_{k+1}||z|^{k+1} + \ldots + |d_n||z|^n}{|d_k||z|^k}.$$

Assuming that $|z| < 1$, the numerator is made bigger by reducing all exponents to z^{k+1}, so

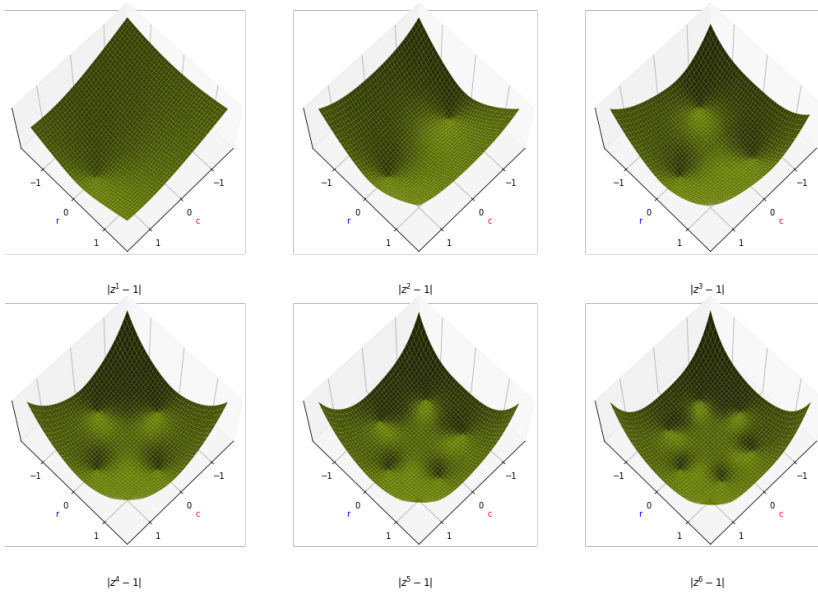
$$r < \frac{|z|^{k+1} \sum_{i \in k+1..n} |d_i|}{|d_k||z|^k} = |z| \frac{\sum_i |d_i|}{|d_k|}.$$

The second factor is a constant, so if we want to make the contribution of the higher order terms less than some given ϵ, we just need to choose a z with small enough magnitude.

Specifically $|z| < \epsilon \frac{|d_k|}{\sum_i |d_i|}$. *Fill this into the above, and note that the two fractions cancel out, leaving only* ϵ.

This allows us to continue our analysis with q' instead of q. Next, we can show that because $|q'(z)|$ has a minimum at 0, that minimum must be equal to 0, so that $d_0 = 0$.

At first, this may not be obvious. Why should a function $q'(z) = c_k z^k + c_0$ necessarily have $c_0 = 0$ if its magnitude has a minimum at 0? It becomes clearer if we take the image of the roots of $z^n - 1$ that we showed earlier, and create a 3d plot of the corresponding magnitude functions $|z^n - 1|$.



This kind of picture applies to the more general function $|c_k z^k + c_0|$ as well: the two constants rotate the image and change how close the roots are to the origin, but unless $c_0 = 0$, we get a ring of roots some distance from the origin and no root at the origin. This contradicts what we know: that $|q(z)|$ has a minimum at 0, so we know that c_0 must be 0.

To prove this formally, let $z = \sqrt[k]{-d_0/d_k}$, one of the roots of q', and let ϵ be some real value near 0. Then we have:

$$\begin{aligned} |q'(\epsilon z)| &= |d_0 + d_k \cdot -\epsilon^k \frac{d_0}{d_k}| \\ &= |d_0 - \epsilon^k d_0| \end{aligned}$$

Note that both terms in this last line point in the same direction, so if $d_0 > 0$, the resulting magnitude is smaller than $|d_0|$, which contradicts what we already know: that $|d_0|$ is the minimum of q'. Therefore $d_0 = 0$, $q(0)$ is a root of q and $p(z_0)$ is a root of p.

It's instructive to look over this proof, and try to figure out why the same argument wouldn't work for real-valued polynomials.

The answer is in the step where we chose $z = \sqrt[k]{-\epsilon d_0/d_k}$. *This allowed us to approach the origin from one of the roots of* $c_k z^k + c_0$, *and to observe that the magnitude increases. In the real-valued world, we cannot always make this choice, because the root may be of a negative number.*

3.5.1 From one root to n roots

Now that we know that each polynomial has at least one root, how do we get to multiple roots? In high school, we learned that when we were faced with a (real valued) second-degree polynomial to solve, sometimes, if we were lucky, we could find its factors. For instance, the function:

$$f(x) = x^2 - 3x + 2$$

can be rewritten as

$$f(x) = (x-1)(x-2)\,.$$

Now, the function is expressed as a multiplication of two factors, and we can deduce that if x is equal to 1 or to 2, then one of the factors is 0, so the whole multiplication is zero. Put simply, if we can factorize our polynomial into factors of the form $x - r$, called **linear factors**, then we know that the r's are its roots.

This is how we'll show that any $p(z)$ of degree n has n complex roots: we'll factorize it into n factors of the form $z - r$, where we will allow r to be complex.

To allow us to factor any polynomial into linear factors, we'll use a technique called *Euclidian division*, which allows us to break up polynomials into factors. The general method works for any polynomial factor, but we can keep things simple by sticking to one specific setting.

Euclidian division (simplified) Given a polynomial $p(z)$ of degree n and a linear factor $z - r$, there is a polynomial $q(z)$ of degree $n-1$ and a constant d, called the *remainder* such that

$$p(z) = (z-r)q(z) + d\,.$$

The proof is short, but a bit dense and it doesn't add much to the intuition we need. It's in the appendix if you're curious.

Now, let r be a root of $p(z)$—we know there must be one—and apply the Euclidian division so that we get

$$p(z) = (z - r)q(z) + d\,.$$

Since r is a root, $p(r)$ must be zero. The first term is zero because of the factor $(z - r)$, so d must be zero as well. In short, if we apply Euclidean division with a root r, we get

$$p(z) = (z - r)q(z)\,.$$

And with that, we can just keep applying Euclidean division. First to $q(z)$, then to the $n - 1$ polynomial resulting from that and so on. Each time we do this, we get one more factor, and the degree of $q(x)$ is reduced by one.

This tells us what we were looking for: every polynomial $p(z)$ of degree n can be decomposed into a product of n linear terms

$$p(z) = (x - r_1)(x - r_2)\dots(x - r_n)$$

so it must have n roots.

What we haven't proved yet, is whether all of these roots are distinct. And indeed, it turns out they need not be. We can factor any $p(z)$ into n linear factors, but it may be the case that some of them are the same. For instance,

$$p(z) = z^2 - 6z + 9 = (z - 3)(z - 3)$$

We call these **multiplicities**. If we count every root by the number of factors it occurs in, then the total comes to n.

We also haven't shown yet that an n-th degree polynomial can't have more *than n roots. This follows from the fact that any roots r' of p we don't use, must be roots of q, since $p(r') = 0 = (r' - r)q(r')$. If we start with $n + 1$ distinct roots, we therefore end up with a 0-order polynomial, a constant function, with a root, giving us a contradiction. If we start with $n + 1$ roots with*

multiplicities, we can add some small noise to them make the roots distinct, deriving a contradiction that way.

We'll call a root *real* if it is a real number and *complex* otherwise. Given the n roots of a particular polynomial, what can we say about how many of them are real and how many of them are complex?

We can have a polynomial with all roots complex and one with all roots real, or something in between, but there *is* a constraint: if our polynomial has only real-valued coefficients, then **complex roots always come in pairs**. This is because if a complex number $r + ci$ is a root of $p(z)$, then that same number with the complex part subtracted instead of added, $r - ci$, is *also* a root of $p(z)$.

The second number is called the *conjugate* of the first. We denote this with a vertical bar: $\overline{z}$ is the conjugate of z. Visually, the conjugate is just the reflection image in the real number line.

Why is it the case that if z is a root that $\overline{z}$ is a root too? Well, it turns out that taking the conjugate distributes over many operations. For our purposes, we can easily show that it distributes over addition and multiplication and that it commutes over integer powers.

$$\begin{aligned}\overline{z+w} &= \overline{z} + \overline{w} \quad \text{since} \\ \overline{a+bi+c+di} &= a+c-(b+d)i \\ &= a-bi+c-di = \overline{a+bi} + \overline{c+di}\end{aligned}$$

$$\begin{aligned}\overline{zw} &= \overline{z}\,\overline{w} \quad \text{since} \\ \overline{m\angle a \cdot n\angle b} &= \overline{mn\angle(a+b)} = mn\angle(-a-b) \\ &= m\angle -a \cdot n\angle -b = \overline{m\angle a} \cdot \overline{n\angle b}\end{aligned}$$

$$\begin{aligned}\overline{z^n} &= \overline{z}^n \quad \text{since} \\ \overline{(m\angle a)^n} &= \overline{m^n \angle na} = m^n\angle -na = (m\angle -a)^n = \overline{m\angle a}^n\end{aligned}$$

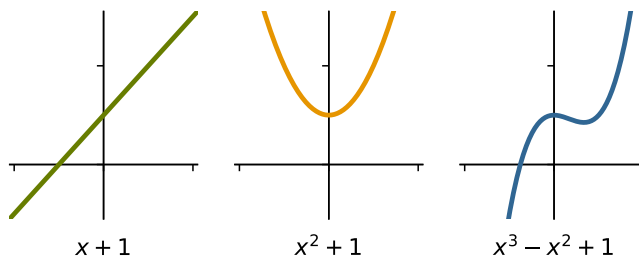
If $p(z) = 0$, then $\overline{p(z)} = 0$ too, since 0 is a real value, so it's equal to its own conjugate. For the roots of a polynomial p, this gives us:

$$\begin{aligned} 0 &= \overline{p(z)} \\ &= \overline{c_n z^n + \ldots + c_1 z + c_0} \\ &= c_n \bar{z}^n + \ldots + c_1 \bar{z} + c_0 \\ &= p(\bar{z}) \,. \end{aligned}$$

In short, if a complex number is a root of $p(z)$, then its conjugate is too. This means that if we have a 2nd degree polynomial (with real coefficients), we can have two real roots, or one pair of complex roots. What we *can't have* is only one real root, since then the other would be complex by itself, and complex roots have to come in pairs.

Similarly, if we have a 3rd degree polynomial, we must have at least one real root, since all complex roots together must make an even number.

This should make sense if you think about the way real valued polynomials behave at their extremes. A 2nd degree polynomial either moves off to positive or to negative infinity in both directions. That means it potentially never crosses the x axis, resulting in two complex roots, or it does cross the x axis, resulting in two real roots. If it touches rather than crosses the x axis, we get a multiplicity: a single real-valued root that occurs twice.



A 3rd degree polynomial always moves off to negative infinity in one direction, and positive infinity in the other. Somewhere in

between, it has to cross the x axis, so we get one real-valued root at least. The rest of the curve can take on a single bowl shape, so the remaining two roots can be both real, when this bowl crosses the horizontal axis, or both complex when it doesn't.

Remember, this only holds if the coefficients are real-valued. If the coefficients of the polynomial are complex, then we get

$$\overline{c_n z^n + \ldots + c_1 z + c_0} = \overline{c_n}\,\bar{z}^n + \ldots + \overline{c_1}\,\bar{z} + \overline{c_0}$$

where the conjugation of the coefficients cannot be removed.

3.5.2 Back to eigenvalues

This was a long detour, so let's restate what brought us here.

We were interested in learning more about the eigenvalues of some square matrix $\mathbf{A}$. These eigenvalues, as we saw, could be expressed as the solutions λ to the following equation

$$|\,\mathbf{A} - \lambda\mathbf{I}\,| = 0\,.$$

We found that the determinant on the left is a polynomial in λ, so we can use what we've learned about polynomials on this problem: if we allow for complex roots, then we know that the characteristic polynomial of $\mathbf{A}$ has exactly n complex roots, counting multiplicities. The entries of $\mathbf{A}$ are real values, so the polynomial has real coefficients, and the complex roots must come in pairs.

In the last part, we said that a given square matrix had between 0 and n eigenvalues. Now, we can refine that by allowing *complex eigenvalues*. An $n \times n$ matrix always has n eigenvalues, counting multiplicities, some of which may be complex, in which case, they come in pairs. Let's see what all these concepts mean in the domain of matrices and vectors.

First, lets look at a simple rotation matrix:

$$\mathbf{R} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\,.$$

This matrix rotates points by 90 degrees counter-clockwise around the origin. All non-zero vectors change direction under this

transformation, so previously, we would have said that $\mathbf{R}$ has no eigenvalues. Now, let's look at the roots of its characteristic polynomial:

$$\begin{aligned} |\,\mathbf{R} - \lambda\mathbf{I}\,| &= \begin{vmatrix} -\lambda & -1 \\ 1 & -\lambda \end{vmatrix} \\ &= (-\lambda)(-\lambda) - (-1)(1) \\ &= \lambda^2 + 1 = 0 \end{aligned}$$

As expected, a polynomial with complex roots. In fact, a classic. Its roots are i, and its conjugate $-\mathrm{i}$.

What does this mean for the eigenvectors? Remember that an eigenvector is the vector that doesn't change direction when multiplied by the matrix. There are no such vectors containing real values, but if we allow vectors filled with complex numbers, there are.

This can get a little confusing: a vector in $\mathbb{R}^2$ is a list of two real numbers. A vector in $\mathbb{C}^2$ is a list of two complex numbers. For instance:

$$\boldsymbol{x} = \begin{pmatrix} 2 & +3\mathrm{i} \\ 1 & -2\mathrm{i} \end{pmatrix}.$$

The confusion usually stems from the fact that we've been imagining complex numbers as 2-vectors, so now we are in danger of confusing the two. Just remember, a complex number is a *single* value. It just so happens there are ways to *represent* it by two real values, which can help with our intuition. When we start thinking about complex matrices and vectors, however, it may *hurt* our intuition, so it's best to think of complex numbers as just that: single numbers. Complex matrices and vectors are just the same thing we know already but with their elements taken from $\mathbb{C}$ instead of $\mathbb{R}$.

Linear algebra with complex matrices and vectors is a very useful field with many applications, but here, we will only need the basics. Addition and multiplication are well-defined for complex numbers, and all basic operations of linear algebra are simply repeated multiplication and/or addition. If we write things

down symbolically, they usually look exactly the same as in the real-valued case.

For instance, if $\boldsymbol{x}$ and $\boldsymbol{y}$ are two complex vectors, then

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

where $x_i + y_i$ represents complex addition as we defined it before. Similarly, for a complex number z and a complex vector $\boldsymbol{x}$:

$$z\boldsymbol{x} = \begin{pmatrix} zx_1 \\ zx_2 \\ \vdots \\ zx_n \end{pmatrix}.$$

Matrix multiplication also works the same as it does in the real-valued case: the result of multiplying a complex matrix $\boldsymbol{A}$ with complex matrix $\boldsymbol{B}$ is the matrix $\boldsymbol{C}$ where C_{ij} is the sum-product of the elements of row i in $\boldsymbol{A}$ and column j in $\boldsymbol{B}$: $\sum_k A_{ik}B_{kj}$.

In the real-valued case we would describe such a sum-and-product operation as the dot product or inner product of two vectors. But this is where we have to be careful in the complex world. The definition of the inner product takes a little bit of care.

The problem is that if we define the inner product of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ as $\boldsymbol{x}^T\boldsymbol{y}$, as we do in the real-valued case, it doesn't behave quite as we want to. Specifically, when we take the inner product of a vector *with itself*, it doesn't give us a well-behaved *norm*. A norm is (roughly) an indication of the length of the vector, and one important property is that there is only one vector that should have norm 0, which is the zero vector.

However, a complex vector like:

$$\boldsymbol{x} = \begin{pmatrix} i \\ 1 \end{pmatrix}$$

will also lead to $\boldsymbol{x}^T\boldsymbol{x} = 0$. The problem is in the transpose. When we move from the real-valued to the complex-valued world, it turns out that simply transposing a matrix doesn't always behave

analogously to how it did before. For things to keep behaving as we expect them to, we need to replace the transpose with the *conjugate transpose*.

The conjugate transpose is a very simple operation: to take the conjugate transpose of a complex matrix, we simply replace all its elements by their conjugates, and then transpose it. If we write the conjugation of a matrix with an overline as we do in the scalar case, and the conjugate transpose with a $*$, then we can define:

$$\mathbf{A}^* = \overline{\mathbf{A}}^{\mathsf{T}}$$

This may seem like a fairly arbitrary thing to do. Why should this particular operation be so fundamental in the complex world? To get some motivation for this, we can look at one more representation of complex numbers. We've seen the cartesian representation, and the polar representation, and here is one more: we can also represent a single complex number as a 2×2 matrix.

Let $r + ci$ be a complex number. We can then arrange the two components in a 2×2 matrix as follows:

$$\begin{pmatrix} r & -c \\ c & r \end{pmatrix}$$

This is a single complex number represented as a real-valued matrix. The benefit of this representation is that if we matrix-multiply the complex numbers x and y in their matrix representations, it is equivalent to multiplying the two complex numbers together: the result is another 2×2 matrix representing the result of the multiplication xy *as a matrix*.

$$\begin{pmatrix} a & -b \\ b & a \end{pmatrix} \quad \begin{pmatrix} c & -d \\ d & c \end{pmatrix} \quad \begin{pmatrix} ac - bd & -ad - bc \\ ad + bc & ac - bd \end{pmatrix}$$

Another way to see this is to write the Cartesian coordinates in terms of the angle and magnitude: $m\cos(a) + m\sin(a)i$. *If you arrange these values into a matrix, you see that the result is a rotation matrix for angle* a*, multiplied by a scalar* m*. Rotation, and uniform scaling is exactly the operation of complex multiplication.*

With this perspective in hand, we can also rewrite complex matrix multiplication. Start with a normal multiplication of a complex matrix $\mathbf{A}$ by a complex matrix $\mathbf{B}$. Now replace each element A_{ij} in both with a 2×2 matrix of real values, representing the complex number A_{ij} as described above. Then, concatenate these back into a matrix $\mathbf{A}^R$, which is twice as tall and twice as wide as $\mathbf{A}$, and filled with only real values. Do the same for $\mathbf{B}$.

Multiplying $\mathbf{A}^R$ and $\mathbf{B}^R$ together performs exactly the same operation as multiplying $\mathbf{A}$ and $\mathbf{B}$ together, except that the result is also in this 2×2 representation. This way, we can transform a complex matrix multiplication into a real-valued matrix multiplication.

This shows us the motivation for the complex conjugate. Compare the number $r+ci$ in a 2×2 representation to its conjugate $r-ci$:

$$\begin{pmatrix} r & -c \\ c & r \end{pmatrix} \quad \begin{pmatrix} r & c \\ -c & r \end{pmatrix}.$$

They are transposes of each other. That means that if we take a complex matrix like $\mathbf{A}$, transform it to 2×2 representation $\mathbf{A}^R$ and then transpose it, the result $\mathbf{A}^{R\,T}$ interpreted as a 2×2 representation of a complex matrix, is not the transpose of $\mathbf{A}$, but the *conjugate* transpose.

The conjugate transpose will be important for what's coming up, so let's look at a few of its properties.

First, note that if $\mathbf{A}$ contains only real values, the conjugate transpose reduces to the regular transpose: real values are unchanged by conjugation, so the conjugation step doesn't change $\mathbf{A}$ and only the transpose remains.

Second, note that the conjugate transpose distributes over multiplication the same way the transpose does: $(\mathbf{AB})^* = \mathbf{B}^*\mathbf{A}^*$. This is because the conjugation distributes over the sums and multiplications inside the matrix multiplication so that we get:

$$\overline{\mathbf{AB}} = \overline{\mathbf{A}}\,\overline{\mathbf{B}}.$$

With the conjugate transpose, we can also define a dot product that will give us a proper norm. By analogy with the real-valued

dot product written as $\mathbf{x}^T\mathbf{y}$, we define the dot product of complex vectors $\mathbf{x}$ and $\mathbf{y}$ as

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{y}^*\mathbf{x} = \sum_i x_i \overline{y_i}\,.$$

Note that this is not a symmetric function anymore: it matters for which of the two vectors we take the conjugate transpose. By convention, it's the second argument of the dot product.

This suggests a natural norm for complex vectors. In the real-valued case, the norm is the square of the vector's dot product with itself: $|\mathbf{x}| = \sqrt{\mathbf{x} \cdot \mathbf{x}}$. The same holds here.

This is a little more abstract and hard to visualize than the dot product in the real-valued case. We'll just have to accept for now that the math works out. We'll need to carry over the following properties of norms and dot products from the real-valued case:

1. A vector with norm 1 is called a **unit vector**.

2. Two vectors whose dot product is 0 are called **orthogonal**. In this case it doesn't matter in which order we take the dot product: if it's zero one way around, it's also zero the other way around. This is easiest to see in the 2×2 representation of the dot product.

3. A matrix U whose column vectors are all unit vectors, and all mutually orthogonal is called **unitary**. This is the complex analogue of the orthogonal matrix we introducted in Chapter 2. Just like we had $U^TU = \mathbf{I}$ and $U^{-1} = U^T$ for orthonormal matrices, we have $U^*U = \mathbf{I}$ and $U^{-1} = U^*$ for unitary matrices.

4. The standard basis for $\mathbb{R}^n$, the columns of $\mathbf{I}$, serves as a basis for $\mathbb{C}^n$ as well. For $\mathbf{I}$ to be a basis, we should be able to construct any complex vector $\mathbf{z}$ as a linear combination of the basis vectors. Here we can simply say $\mathbf{z} = z_1\mathbf{e}_1 + \ldots + z_n\mathbf{e}_n$, where $\mathbf{e}_i$ are the columns of $\mathbf{I}$.

With these properties in place, we can return to the question of eigenvalues and eigenvectors.

Let's go back to our example. Here is the operation of our rotation matrix:

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \begin{pmatrix} -z_2 \\ z_1 \end{pmatrix}.$$

An eigenvector of this matrix is one for which this operation is the same as multiplying by the eigenvalue i (or $-i$):

$$\begin{aligned} z_1 i &= -z_2 \\ z_2 i &= z_1 . \end{aligned}$$

Remember that z_1 and z_2 are both complex numbers. We know already that on *individual* complex numbers, multiplying by i has the effect of rotating in the complex plane by 90 degrees counterclockwise. That means that we're looking for a pair of complex numbers, such that rotating them this way turns the first into the negative of the of the second and the second into the first. This is true for any complex numbers of equal magnitude with a 90 degree angle between its two elements. For instance

$$\begin{pmatrix} 1 \\ i \end{pmatrix}$$

is an eigenvector.

In the real-valued case, a given eigenvector could be multiplied by a scalar and it would still be an eigenvector. The same is true here as well. If we multiply the eigenvector above by a complex scalar $s = m\angle a$, this multiplication rotates both complex numbers in the vector above by the same angle a, so the angle between them stays 90 degrees.

This allows us to scale the eigenvector so that its norm becomes 1, giving us a *unit eigenvector*.

3.6 The spectral theorem

We are finally ready to begin our attack on the spectral theorem. The structure of the proof is as follows. We will first de-

fine a slightly different decomposition of a matrix, called the *Schur decomposition*.

We first show that any square matrix, complex or real, can be Schur-decomposed. Then, we show that the Schur decomposition of a symmetric real-valued matrix coincides with the eigendecomposition.

3.6.1 The Schur decomposition

Let A be any complex-valued, $n \times n$ matrix. The Schur decomposition rewrites A as the following product: $A = U^* T U$, where U is a unitary matrix, and T is an upper triangular matrix (i.e. a matrix with non-zero values only on or above the diagonal). Compare this to the eigendecomposition $A = P^T D P$, where P is orthogonal, and D is diagonal.

Unlike the eigendecomposition, however, we can show that the Schur decomposition exists for *any* square matrix.

> *This is a proof by induction. If you've never seen that before, it may look a little confusing.*
>
> *The idea is that we state our problem first for some specific value n, for instance the size of the matrix ($n \times n$) we're dealing with. We prove the specific case $n = 1$ first, and then we prove that **if** the result holds for $n - 1$, we can prove that it does for n as well. Combining the two shows that the result must hold for all n. If you're struggling with this, try following the inductive step with $n = 2$ first, and then again with $n = 3$.*

Schur decomposition. Any $n \times n$ complex matrix A has a Schur decomposition $A = U^* T U$, where U is unitary, and T is upper triangular.

Proof. We will prove this by induction on n.

Base case. First, assume $n = 1$. That is, let A be a 1×1 matrix with value a. Then, the Schur decomposition reduces to simple scalar multiplication with $A = (u)^* (a) (u) = (uau)$, which is

true for $u = 1$ and $a = A_{11}$.

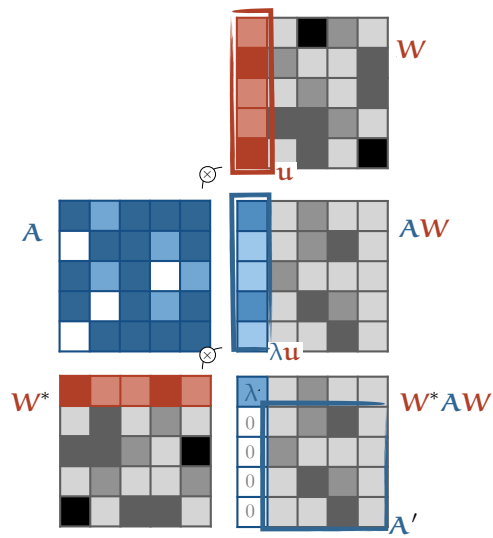
Induction step. Now we assume that the theorem holds for $n-1$, from which we will prove that it also holds for n.

We know that A has n eigenvalues, counting multiplicities and allowing complex values. Let λ be one of these, and let $\mathbf{u}$ be a corresponding unit eigenvector.

Let W be a matrix with $\mathbf{u}$ as its first column, and the remaining unit vectors orthogonal to $\mathbf{u}$ and to each other. This makes W a unitary matrix.

> *In $\mathbb{R}^n$ we know that sufficient orthogonal vectors are always available. In the appendix, we prove that this property carries over to $\mathbb{C}^n$.*

Now consider the matrix W^*AW. As illustrated below, the first column of AW is $A\mathbf{u}$, which is equal to $\lambda\mathbf{u}$ since $\mathbf{u}$ is an eigenvector. This means that $(W^*AW)_{11}$ is equal to $\mathbf{u}^*\lambda\mathbf{u} = \lambda$. The other elements in the first column are equal to the dot product of a scaled $\mathbf{u}$ and another column of W. Since the columns of W are mutually orthogonal, these are all 0.



Call this matrix $\mathbf{R}$ (note that it is not triangular yet). So far we have shown that $W^*AW = \mathbf{R}$, or if we multiply both sides by W^* and W, that $A = W\mathbf{R}W^*$.

Call $\mathbf{R}$'s bottom-right block A' as indicated in the image. A' is an $n-1 \times n-1$ matrix, so by the assumption made above, it can be factorized: $A' = VQV^*$, with V unitary and Q upper triangular.

Now note that if we extend the matrix V to an $n \times n$ matrix as follows

$$V' = \begin{pmatrix} 1 & 0 \dots 0 \\ 0 & \\ \vdots & V \\ 0 & \end{pmatrix}$$

we can move it out of the submatrix A', so that

$$A = WV' \begin{pmatrix} \lambda & * \dots * \\ 0 & \\ \vdots & Q \\ 0 & \end{pmatrix} V'^* W^*$$

with the $*$'s representing arbitrary values.

We call the matrix in the middle T. Note that this is now upper triangular. Note also that V' is unitary, since the column we added is a unit vector, and it's orthogonal to all other columns. Let $U = WV'$. Since W and V' are unitary, their product is as well, and with that we have

$$A = UTU^*$$

proving the theorem. □

The important thing about the Schur decomposition is that it *always works*. No matter what kind of square matrix we feed it, real or complex valued, with or without real eigenvalues, symmetric or asymmetric, singular or invertible, we always get a Schur decomposition out of it.

With this result in hand, the main difficulty of proving the

spectral theorem is solved. We simply need to look at how the Schur decomposition behaves if we feed it a real-valued symmetric matrix

3.6.2 Proof of the spectral theorem

The spectral theorem A matrix is orthogonally diagonalizable if and only if it is symmetric.

Proof. We will prove the two directions separately.

(1) If a real-valued matrix A is orth. diagonalizable, it must be symmetric. Note that in an orthogonal diagonalization we have $D = D^T$ because D is diagonal. Thus, if A is orthogonally diagonalizable, we know that

$$A = PDP^T = P^{T^T} D^T P^T = (PDP^T)^T = A^T$$

which implies that A is symmetric.

(2) If a real-valued matrix A is symmetric, it must be orth. diagonalizable. For this direction, we need the Schur decomposition. By assumption A is symmetric and real-valued, so that $A^* = A^T = A$. Let $A = UTU^*$ be its Schur decomposition.

> *Note that we have assumed that A is real-valued, but U and T could still contain complex values.*

From the symmetry of A, we have $UTU^* = (UTU^*)^* = UT^*U^*$, so $T = T^*$. This tells us two things. First that all values off the diagonal are zero (remember that T is upper triangular), so T is actually diagonal. Second, that the values on the diagonal are equal to their own conjugate so they must be real values.

This gives us a real-valued diagonalization $A = UTU^*$. But what about the matrix U? Could that still contain complex values? It could, but knowing that A is real-valued, and so are the diagonal values of T, we can perform the Schur decomposition specifically so that U ends up real valued as well.

Follow the construction of the Schur decomposition. In the **base case**, U is real-valued by definition. In the **inductive step**, assume that we can choose V real-valued for the case $n-1$. When we choose the eigenvector for λ, we choose a real eigenvector. These always exist for real eigenvalues (proof in the appendix). When we choose the other columns of W we choose real valued vectors as well.

This means that W and V' are real, so their product U is real as well.

That completes the proof: if we perform the Schur decomposition in such a way that we choose real vectors for W where possible, the decomposition of a symmetric matrix gives us a diagonal real-valued matrix T and an orthogonal matrix U. □

It's been a long road, but we have finally reached the end. It's worth looking back at all the preliminaries we discussed, and briefly seeing why exactly they were necessary to show this result. Let's retrace our steps in reverse order.

The last thing we discussed, before the proof of the spectral theorem was the **Schur decomposition**. Its usefulness was clear: the Schur decomposition *is* the eigendecomposition, if we're careful about its construction. The main benefit of the Schur decomposition is that it always works. With the real-valued eigendecomposition, we knew that it sometimes exists and sometimes doesn't. From that perspective it's very difficult to characterize the set of matrices for which it exists. The Schur decomposition allowed us to zoom out to the set of all matrices, so that we could ask what the decomposition looks like for real-valued, symmetric matrices.

The **complex numbers** make this possible. Filling matrices and vectors with complex numbers gives us a Schur decomposition that always works. The key to this is that the construction of the Schur decomposition requires us to pick one eigenvalue and corresponding eigenvector for various matrices. If we allow for complex eigenvalues, we ensure that this is always possible.

This result, that every $n \times n$ matrix has n eigenvalues if complex values are allowed, follows from two ideas. The first is the the **characteristic polynomial**. This is an n-th order polynomial, constructed from an $n \times n$ matrix A, that is zero, exactly when

the **determinant** of $\mathbf{A} - \lambda\mathbf{I}$ is zero. This means that the roots of the characteristic polynomial are the eigenvalues. The second idea is the **fundamental theorem of algebra** which tells us that every n-th order polynomial has exactly n roots in the complex plane, counting multiplicities.

That gives us the spectral theorem and, as we saw in the last part, the spectral theorem gives us principal component analysis.

Now that we know how PCA works, why it works, and we have a thorough understanding of the spectral theorem, there is only one question left: **how do we implement it?** There are a couple of ways to go about this, but the best option by far is to make use of the singular value decomposition (SVD). This is a very versatile operation, which has many uses beyond the implementation of PCA. We will dig into the SVD into the next chapter.

CHAPTER 4 · THE SINGULAR VALUE DECOMPOSITION

In the previous chapters, we learned that principal components are *eigen*vectors. Specifically, they are the eigenvectors of the covariance matrix $\mathbf{S}$ of our data $\mathbf{X}$.

In this chapter, we'll develop a slightly different perspective: that the principal components are **singular vectors**. Not of the covariance matrix $\mathbf{S}$, but of the data matrix $\mathbf{X}$ itself. Singular vectors, which we will define below, are closely related to eigenvectors, but unlike eigenvectors they are defined for all matrices, even non-square ones.

The two perspectives are complementary: they are simply different ways of looking at the same thing. The benefit we get from the singular vectors is that we can develop the **singular value decomposition** (SVD).

The SVD is firstly, the most popular and robust way of computing a principal component analysis. But it is also something of a Swiss army knife of linear algebra. It allows us to compute linear projections, linear regressions, and many other things.

Put simply, if you want to use linear algebra effectively in data science or machine learning, the singular value decomposition is the beginning and the end.

So, a topic well worth a chapter. We'll focus on the PCA use case first, since that's what brought us here, but like the eigendecomposition, the rabbit hole goes much deeper, and we'll finish by looking at some of the other uses of the SVD. We will look in particular detail at the concepts of **matrix rank**, the **pseudoinverse** and the **Eckart-Young-Mirsky theorem**.

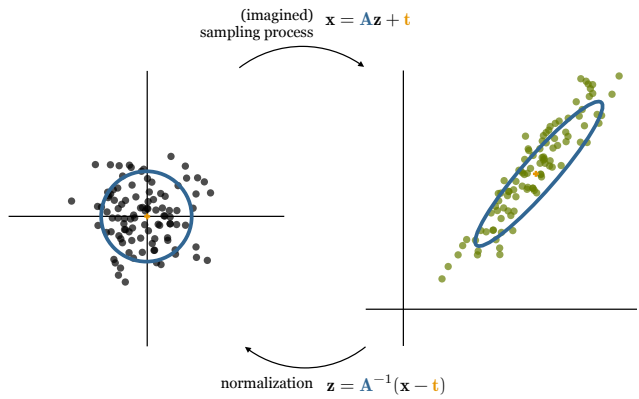
4.1 Eigenvalues and singular values

In the previous two chapters we explained eigenvalues and eigenvectors in detail. As we saw, these are well-defined and pre-

dictable for *square, symmetric matrices*. What happens if our matrix is not so well-behaved? What if it's not symmetric, not invertible, or not even square? Do some of the ideas that underlie eigenvalues and eigenvectors still carry over?

The best place to start is the intuition we built at the end of Chapter 2, when we discussed *normalization*. Let's review what we said there.

Let's say that we have a dataset consisting of a set of instances $\mathbf{x}$. We assume that there is some unobserved data $\boldsymbol{z}$, which is standard-normally distributed, and that a linear transformation $\mathbf{x} = \mathbf{A}\boldsymbol{z} + \mathbf{t}$ has transformed it into to the data we observed.



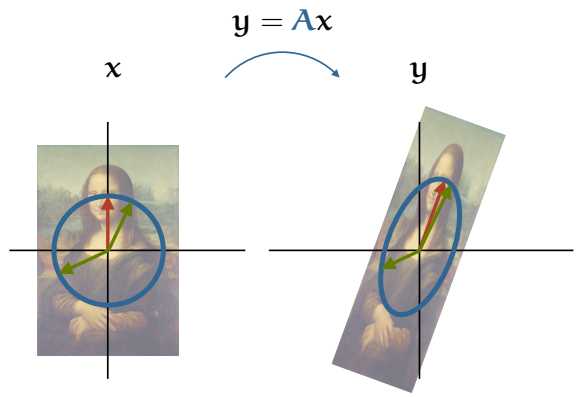
Summary of normalization. We imagine some linear process has transformed the data from some standard-normally distributed data, and we invert that transformation. The sphere of all unit vectors in the "original" space is transformed into an ellipse in the space of the data we observed.

What we saw in Chapter 2, is that if we can find $\mathbf{A}$ from the observed data, then the way in which $\mathbf{A}$ stretches space gives us the principal components of the data.

More specifically, imagine taking the (hyper)sphere of all unit vectors, and transforming them by $\mathbf{A}$. The result is an ellipsoid, and the major axis of this ellipsoid—the direction in which it bulges out the most—is the principal component of the data.

We'll start with that intuition: *the direction in which space is stretched the most by a transformation is important*, and see what it gets us when we apply it to general matrices.

Note that this is *not* an eigenvector of $\mathbf{A}$ (It's an eigenvector of the covariance $\mathbf{S} = \mathbf{A}^T\mathbf{A}$, but we'll get to that later). An eigenvector of $\mathbf{A}$ is a vector whose direction doesn't change when multiplied by $\mathbf{A}$. This is different, in general, from the direction in which the transformation has the greatest effect.



The vector that is stretched the most by the transformation is different from the eigenvectors, which are vectors that don't change direction under the transformation.

If you are wondering why the eigenvectors in the image above aren't orthogonal to each other it's because the transformation matrix isn't symmetric, so the spectral theorem doesn't apply. It has two real eigenvalues, but not with orthogonal eigenvectors.

We can now define the rest of the axes of our resulting ellipsoid in the same way we did with the principal components: for the second axis, we constrain it to be orthogonal to the first, and then see in which direction $\mathbf{A}$ causes the greatest stretch. For the third, we constrain it to be orthogonal to the first and second, and so on.

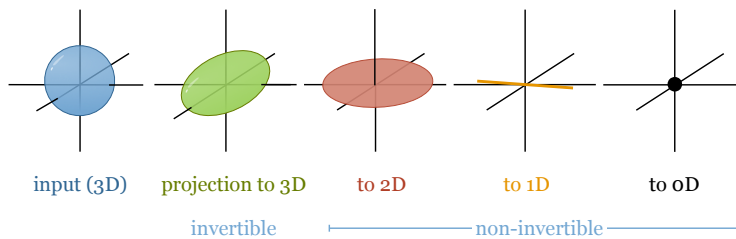
In this fashion, we get a series of vectors of "greatest stretch" that are all orthogonal to each other.

The reason we are revisiting these intuitions is that **they still hold if A isn't square**. If we allow any matrix, including non-square ones, we can still carry this particular intuition over from the principal components. Let M be any matrix, with dimensions $n \times m$. The multiplication

$$y = Mx$$

maps a vector $x \in \mathbb{R}^m$ to a vector $y \in \mathbb{R}^n$. If we constrain x to be a unit vector, then the possible inputs form a (hyper)sphere in $\mathbb{R}^m$. This sphere is mapped to an ellipsoid in $\mathbb{R}^n$. We don't need to know all the details about what ellipsoids are, just the basics: they are linear transformations of spheres, and the *axes* of the ellipsoid are the directions where it bulges the most (with the second axis the biggest bulge orthogonal to the first, and so on).

For some matrices, for instance singular square ones, the resulting ellipsoid is not fully n-dimensional, but of some lower dimensionality. For instance, if $m = n = 3$, there are matrices that produce a 2d ellipse, or even a 1d ellipsoid (a line segment). The matrix could even compress everything into a single point, which we'll call a 0d ellipsoid.



The same is true for matrices where n is bigger than m, for instance a 3×2 matrix. No linear transformation will turn a 2d sphere into a 3d ellipsoid, so the ellipsoid we get from a 3×2 matrix must be at most 2 dimensional.

We can put this more precisely when we've discussed matrix rank.

Nevertheless, the output is always an ellipsoid of some dimension, so our intuition carries over. We can always ask in which direction the resulting ellipsoid bulges out the most. It's just that after a few directions, the remainder may all be compressed to 0. For now, let's see what we can say about the directions that aren't.

So, the question is which unit vector is stretched the most by our matrix $\mathbf{M}$? we can state this as an optimization problem:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{argmax}} \;\; \|\mathbf{M}\mathbf{x}\| \\ &\text{such that} \;\; \|\mathbf{x}\| = 1\,. \end{aligned}$$

This problem simply asks for the input $\mathbf{x}$ for which the resulting vector $\mathbf{M}\mathbf{x}$ has maximal magnitude, subject to the constraint that $\mathbf{x}$ is a unit vector.

To find a solution to this problem, we can rewrite both norms as dot products. In the constraint, we know that setting the norm equal to 1 is the same as setting the dot product $\mathbf{x}^\mathsf{T}\mathbf{x}$ equal to 1. In the optimization objective, the norm and the dot product *aren't* the same value (one is the square root of the other), but they *are* maximized at the same $\mathbf{x}$. So, we get the equivalent problem:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{argmax}} \;\; \mathbf{x}^\mathsf{T}\mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{x} \\ &\text{such that} \;\; \mathbf{x}^\mathsf{T}\mathbf{x} = 1\,. \end{aligned}$$

Something quite exciting has happened here. By one simple rewriting step, the maximum of the linear function $\mathbf{M}\mathbf{x}$ has become the maximum of the *quadratic* function $\mathbf{x}^\mathsf{T}\mathbf{S}\mathbf{x}$, with $\mathbf{S} = \mathbf{M}^\mathsf{T}\mathbf{M}$.

This is similar to what we derived in Chapter 2: there, we started with a (square) transformation matrix $\mathbf{A}$, which took our imagined standard-normally distributed data into the form we observed. We showed that $\mathbf{A}$ could be derived from the covariance matrix $\frac{1}{n}\mathbf{X}^\mathsf{T}\mathbf{X}$ by the relation $\mathbf{A}\mathbf{A}^\mathsf{T} = \frac{1}{n}\mathbf{X}^\mathsf{T}\mathbf{X}$, and that optimizing for the direction of maximum stretch under $\mathbf{A}$ corresponds to the direction in which the quadratic form $\mathbf{x}^\mathsf{T}\left(\mathbf{A}\mathbf{A}^\mathsf{T}\right)\mathbf{x}$ is maximized.

Most importantly, we saw there that maximizing the value $\mathbf{x}^T \mathbf{S}\mathbf{x}$ gives us the first eigenvector of $\mathbf{S}$. What we are seeing here is that the same thing holds for any matrix $\mathbf{M}$. Optimizing for maximum stretch gives us a direction that is equal to the first eigenvector $\mathbf{v}$ of $\mathbf{M}^T\mathbf{M}$. Using this we can work out the relation between the amount that $\mathbf{v}$ is stretched in the multiplication $\mathbf{M}\mathbf{v}$, and the corresponding eigenvalue of $\mathbf{M}^T\mathbf{M}$:

$$\|\mathbf{M}\mathbf{v}\|^2 = \mathbf{v}^T\mathbf{M}^T\mathbf{M}\mathbf{v} = \mathbf{v}^T\lambda\mathbf{v} = \lambda\mathbf{v}^T\mathbf{v} = \lambda\,.$$

That is, if $\mathbf{v}$ is an eigenvector of $\mathbf{M}^T\mathbf{M}$ with eigenvalue λ, then multiplying $\mathbf{v}$ by $\mathbf{M}$ stretches $\mathbf{v}$ by $\sqrt{\lambda}$. Moreover, even though $\mathbf{M}$ may be non-square or singular, we know that $\mathbf{M}^T\mathbf{M}$ is *always* symmetric, which tells us that it has exactly m real eigenvalues, including multiplicities. The square roots of these eigenvalues indicate how much $\mathbf{M}$ stretches space along the various axes of the ellipsoid we get if we transform the unit vectors by it.

We call these values the **singular values** of $\mathbf{M}$. For each singular value σ, we have used two vectors in its definition. The unit vector $\mathbf{v} \in \mathbb{R}^m$ which we multiplied by $\mathbf{M}$ and the vector that resulted from the multiplication. The latter has length σ so we can represent it as $\sigma\mathbf{u}$, where $\mathbf{u} \in \mathbb{R}^n$ is also unit vector. With this, we can make our definition of the singular vectors similar to that of the eigenvectors: if σ is a singular value of $\mathbf{M}$, then for its two singular vectors $\mathbf{v}$ and $\mathbf{u}$

$$\mathbf{M}\mathbf{v} = \mathbf{u}\sigma$$

We call $\mathbf{v}$ a **right singular vector** of σ and $\mathbf{u}$ a **left singular vector**.

> *We currently have the right singular vectors on the left and vice versa, which I admit is confusing, but they will change place as we develop a more complete definition.*

You may ask what happens if an eigenvalue of $\mathbf{M}^T\mathbf{M}$ is negative. We know that all eigenvalues are real, but we don't know

that they are all positive. And we've defined the corresponding singular values as their square roots.

So, does the corresponding singular value of M, become undefined, or complex? The answer is that we can prove that this doesn't happen. Note that if we pass a unit eigenvector v of A into its quadratic $v^T A v$, the result is its eigenvalue $v^T \lambda v = \lambda v^\top v$. Now, let $y = Mx$. Then if x is an eigenvector of $M^T M$, the quadratic $x^T M^T M x = y^T y$ is the corresponding eigenvalue. This is simply the sum of the squared elements of y. Whether these are positive or negative, the result is always nonnegative, which shows an eigenvalue of $M^T M$ will always be nonnegative.

> *In technical terms, we have just shown that* $M^T M$ *is* positive semidefinite.

So, we can be sure that there are always m nonnegative singular values for M, even though some of them may be zero.

4.1.1 Singular vectors and principal components

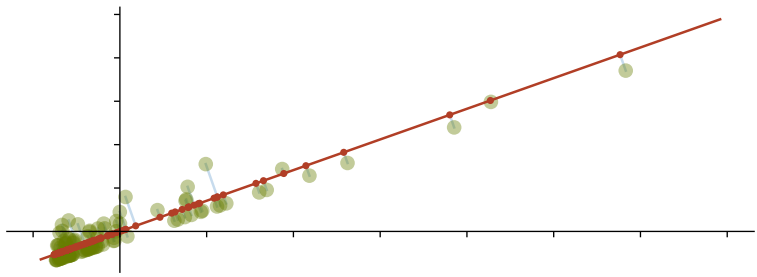
Before we develop the singular values and vectors further, let's see what their relevance is in the context of principal component analysis. In PCA, we start with a data matrix X, of size $n \times m$, which arranges the n instances in our data along the rows, and the m features of each along the columnns.

We can ask what the singular values of X are. You may be able to predict the answer, but first let's look at what how this squares with our intuition of singular values. We said that they represent the maximal amount multiplying a vector by X stretches that vector. What does it mean to multiply a vector by the data matrix?

Let's take a vector $p \in \mathbb{R}^m$ (where m is the number of features in our data), and let's assume p is a unit vector. That way, we are limited in how big we can make each entry. Now multiply $y = Xp$. We get a vector of length n: one value for each instance x in our data, which is the dot product of that instance and our vector p. This value is high if the features of the instance match the corresponding values of p in magnitude and sign: they should be big where p is big and negative where p is negative.

That is, the dot product expresses a kind of *similarity* between $\mathbf{p}$ and the instances in our data. The elements of $\mathbf{y}$ tell us which instances in the data "match" $\mathbf{p}$ the best. This means that the vector $\mathbf{p}$ that is stretched the most by $\mathbf{X}$, is the vector that is most similar to *all instances*. This is a tradeoff: making it more similar to one instance may make it less similar to other instances. Balancing this over all instances, we get the direction that maximizes $\|\mathbf{y}\|$: the first singular vector of $\mathbf{X}$.

If we assume that the data is mean-centered, we can imagine this direction pivoting around the origin, averaging the angles to all instances, proportional to how far from the origin each instance is. This should remind you of how we plotted the first principal component earlier.



From Chapter 1: the first PC for a simple dataset.

It's not hard to prove that this direction of maximal stretch is indeed the first principal component. As we've seen the maximal direction is an eigenvector of the matrix $\mathbf{X}^T\mathbf{X}$. What we also saw, in Chapter 2, is that the covariance matrix is estimated with $\frac{1}{n}\mathbf{X}^T\mathbf{X}$.

The constant factor $\frac{1}{n}$ doesn't really matter much for our purposes. If we diagonalize $\mathbf{X}^T\mathbf{X}$ as $\mathbf{P}^T\mathbf{D}\mathbf{P}$, then we can write $\mathbf{P}^T\frac{1}{n}\mathbf{D}\mathbf{P}$ for the properly normalized covariance. In other words $\mathbf{X}^T\mathbf{X}$ and $\frac{1}{n}\mathbf{X}^T\mathbf{X}$ have the same eigenvectors, and their eigenvalues only differ by a multiplicative constant $\frac{1}{n}$.

Thus, the directions corresponding to the singular values of $\mathbf{X}$ are the same as the eigenvectors of the covariance matrix. We've already established that the singular values of $\mathbf{X}$ are the square roots of the corresponding eigenvalues of $\mathbf{X}^T\mathbf{X}$, so we can say

that they are proportional to the eigenvalues of S.

We can now show that the singular values and eigenvalues are in some sense generalizations of standard deviation and variance. To illustrate, assume that our data X is one-dimensional. In this case, we could estimate the variance with the formula:

$$\text{var } X = \frac{1}{n} \sum_i {X_{i1}}^2$$

which we can also write as

$$\frac{1}{n} X^T X = X'^T X'$$

with $X' = \frac{1}{\sqrt{n}} X$.

In this case ${X'}^T X'$ has one eigenvalue, which is simply the scalar value ${X'}^T X'$. As shown above, this corresponds to the estimated variance of the data. The corresponding *singular* value of X' is its square root: the standard deviation.

In short, if we ignore a scaling factor of $\frac{1}{\sqrt{n}}$, the singular values of X are analogous to the standard deviation and the eigenvalues of the covariance matrix $X^T X$ are analogous to the variance.

4.1.2 The singular value decomposition

When we developed eigenvalues and eigenvectors, we saw that they allowed us to decompose square matrices as the product of three simpler matrices: $A = PDP^T$. We can do the same thing with singular values and vectors.

We've seen that any matrix has *singular values*, which correspond to two kinds of singular *vectors*, defined in a way that is similar to the way eigenvectors are defined. Here, we have the eigenvalue definition on the left and the singular value definition on the right

$$Aw = w\lambda \qquad\qquad Mv = u\sigma .$$

From the definition of the eigenvalue and vector, we managed to construct a decomposition of A.

A straightforward way to do this is the following recipe.

1. Assume that the $n \times n$ matrix A has n real eigenvalues λ_i, with unit eigenvectors w_i.
2. Arrange the eigenvectors as columns of a matrix so that we get $A[w_1 \dots w_n] = [w_1\lambda_1 \dots w_n\lambda_n]$
3. Let $P = [w_1 \dots w_n]$ and $D = \text{diag}(\lambda_1 \dots \lambda_n)$ so that $AP = PD$
4. Note that P is orthogonal, so that $A = PDP^T$

The fact that the eigenvectors are orthogonal comes from the iterative way in which we chose them: each was chosen to be orthogonal to all the previous choices. We did the same for the singular vectors. The assumption that we have n real eigenvalues comes from the spectral theorem.

One thing we haven't examined yet in any detail, is what happens if our eigenvectors are zero. This can happen if we have a square, symmetric matrix which is non-invertible, like

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}.$$

In such cases, we get one or more 0 eigenvalues. What are the corresponding eigenvectors? In many ways it doesn't matter: the defining relation $Aw = w0$ always holds, whatever w we choose. To make the above recipe work, all we need to do is to choose w so that it is a unit vector, and orthogonal to any w we already chose.

If we try to follow the same recipe with the singular vectors, we might get something like this.

1. Assume that the $n \times m$ matrix M has k singular values σ_i, with unit left and right singular vectors $u_i \in \mathbb{R}^n$ and $v_i \in \mathbb{R}^m$.
2. Arrange the singular vectors as columns of matrices so that we get $M[v_1 \dots v_k] = [u_1\sigma_1 \dots u_k\sigma_k]$.
3. Let $V = [v_1 \dots v_k]$, $U = [u_1 \dots u_k]$ and $\Sigma = \text{diag}(\sigma_1 \dots \sigma_k)$ so that $MV = U\Sigma$.

This is as far as we get. There aren't necessarily sufficient singular values to ensure that V is square. That means V is not invertable, so we can't take it to the other side of the equation.

We have $V^T V = I$, but to take V to the other side, we'd need $VV^T = I$, which we only get if V is square.

A simple solution is to extend V to a complete basis. We know that the k singular vectors making up V are mutually orthogonal, and we know that we can always choose $m - k$ additional unit vectors orthogonal to these and to each other. If we add them to V as columns, V becomes an $m \times m$ orthonormal matrix.

If we add one such vector v' to V, what do we need to change to keep our equation $M[v_1 \dots v_k] = [u_1\sigma_1 \dots u_k\sigma_k]$ intact? We need to extend the right with the vector Mv'. What can we say about this vector? First, we can always write any vector as a unit vector times a scalar length, so let's call it $u'\sigma'$.

$$M[v_1 \dots v_k\ v'] = [u_1\sigma_1 \dots u_k\sigma_k\ u'\sigma']$$

Note that we assumed that we have all k singular values of M already. This means that there is no vector v orthogonal to all v_i such that $\|Mv\| > 0$, or it would be a candidate for the $k+1$th singular vector. Therefore, $\|Mv'\| = 0$.

v' is in the null space *of M.*

This means that we can set $\sigma' = 0$ and choose u' however we like. We will choose some vector that is orthogonal to all u_i already chosen.

Note the similarity to the eigenvector case: once we've run out of nonzero singular values we extend the rest of the diagonal with zeros.

We could just keep adding vectors like this until V is square, and it would be orthogonal so we would be allowed to move it to the other side. But we'd be missing a trick if we stopped there. The matrix M represents a function from one space, $\mathbb{R}^m$

to another $\mathbb{R}^n$. If we make V square and orthogonal, we have constructed an orthonormal *basis* for $\mathbb{R}^m$, just like the analogous P is a basis in the eigendecomposition. It would be quite nice, if we could ensure that U is also an orthonormal basis for $\mathbb{R}^n$. That way, we would have two orthonormal bases U and V for the two spaces that M transforms between, and a diagonal matrix of singular Σ values in the middle.

To do this, we need to add different numbers of orthogonal vectors to V and U. V needs $k - m$ extra vectors to become $m \times m$, and U needs $k - n$ extra vectors to become $n \times n$. The trick to accomplish this is to make Σ *non-square*. We make Σ an $m \times n$ matrix, with all zeros except for the diagonal of the top $k \times k$ matrix in the top left, which contains the singular values. Here is an illustration.

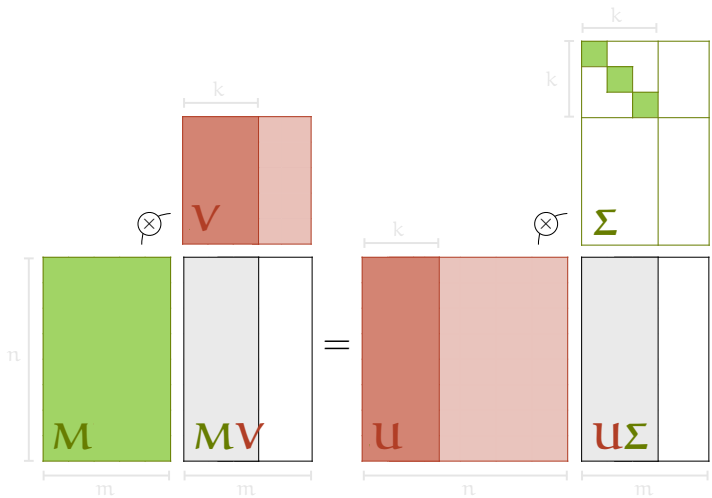


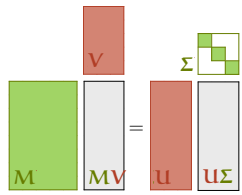
Diagram for the equation $MV = U\Sigma$. White cells represent 0s. The light columns of V and U are the vectors we add to make these matrices square.

We see that the effect of extending V with orthogonal vectors is to add zero columns to the product MV. This is not surprising, since we noted already that these must be in the null space of M.

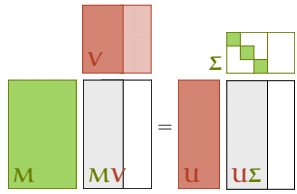
On the right, the product $U\Sigma$ needs to be equal to MV. The extra zero columns can easily be added by adding zero columns to Σ.

How about the extra basis vectors we would like to add to U? We need to add rows to Σ in order to make the matrix dimensions match. If the rows we add are zero rows, then we can be sure that when we multiply the two matrices, any entry in any column added to U will be multiplied by 0, so it won't affect the product $U\Sigma$.

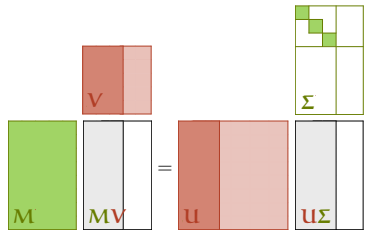
Here is the process in three steps:



Set up the singular values and vectors in a matrix multiplication.



Add orthogonal vectors to V to make it square. Add zero columns to Σ to make the sizes match.



Add orthogonal vectors to U to make it square. Add zero rows to Σ to make the sizes match.

Now, we can safely take V to the other side (that is, multiply both sides by V^{-1}).

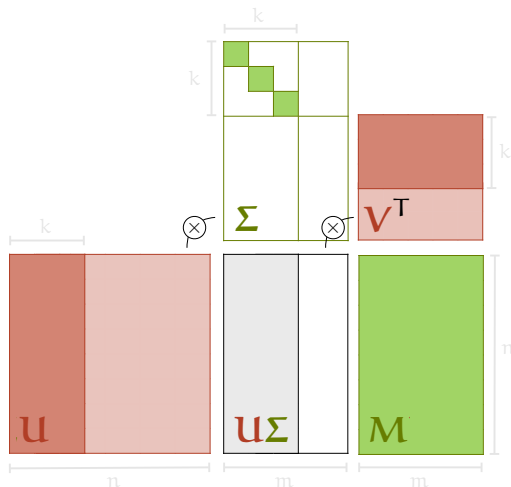
Since we've carefully constructed V to be orthonormal, we know that $V^{-1} = V^T$ and we get:

$$M = U\Sigma V^T$$

Where, to reiterate:

- M is *any* $n \times m$ matrix. We call the number of singular values it has k.
- V and U are orthogonal matrices of $m \times m$ and $n \times n$ respectively. The first k columns of each are the right and left singular vectors respectively.
- Σ is an $n \times m$ matrix with all entries 0 except the first k elements of the diagonal, which correspond to the singular values of M.

Here it is in a diagram.



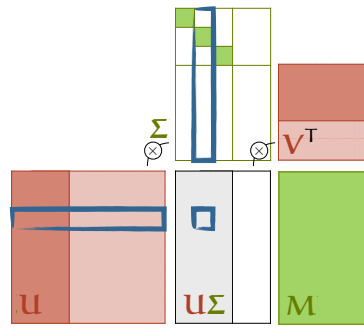
This is called a ***full* singular value decomposition**. It gives us all singular values, and two orthonormal bases for $\mathbb{R}^n$ and $\mathbb{R}^m$.

This is not a very economical way of representing M: all these matrices are very big. Let's see if we can trim the fat a little

bit. If we look at Σ, we see that we added a lot of zeros, in order to move V over to the right. It seems reasonable, that all these zeros, and their associated basis vectors, aren't actually necessary to represent M.

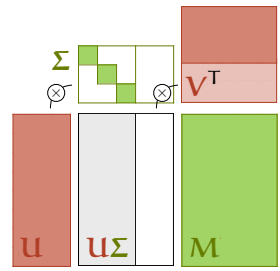
How much do we need? We know that M maps the axes of a sphere in the input space to the axes of an ellipsoid in the output space. Each singular value corresponds to one such mapping from axis to axis. We also know that to get from the equation with only the singular values and vectors to the full SVD, we only added zeros to Σ and null-space vectors to V. It's hard to escape the intuition that we had what we needed already before we extended V and U to be square, and the singular vectors and values by themselves already provided a complete description of M.

To prove this, we can walk back the exact steps we took to get to the full SVD, but now with V on the right hand side. First, here is the full SVD, for reference.



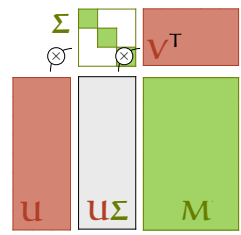
We've highlighted how the elements of $U\Sigma$ in the middle are computed: each is the dot product of a row of U and a column of Σ. The elements of each are matched up, multiplied and the result is summed together. What we see here, is that the elements of the columns we added to U, the ones that don't correspond to a singular value, are always matched up to a zero. Removing them won't change the dot product, and thus won't change the elements of the matrix $U\Sigma$. This means we can safely remove the columns we added to U, and each corresponding row of Σ.

Here's the result:



We can use the same logic on the multiplication of $U\Sigma$ by V^T. The added rows of V^T are always matched up to zero columns of $U\Sigma$. Removing these will not affect the value of each dot product. To remove the zero columns from $U\Sigma$, all we need to do is remove the rightmost block of zeros.

Removing all three, we arrive at a more economical representation of M:



It turns out that without extending V and U to full bases, we already had a complete decomposition of M, although we had to first make the extension to prove the fact.

We've done a lot of variable reassignments here: first using V to refer to just the singular vectors, then extending it to a square matrix, and then clipping it back again. To make our notation more precise, from now on we'll use U and V for the complete, square, basis matrices and U_r and V_r for the matrices made up of the first r columns of each.

If we assume that M has k singular values, we can now write this latest decomposition as:

$$M = U_k \Sigma_k V_k^T$$

This is called the **compact SVD**.

See the appendix for a more formal proof of the correctness of the compact SVD.

Finally, if we're not interested a perfect decomposition, we can extract only the top $r < k$ singular values and vectors. If we then construct the matrices V_r with the corresponding right singular vectors, U_r with the corresponding left singular vectors and Σ_r with these singular values along the diagonal, we can still perform the multiplication

$$M_r = U_r \Sigma_r V_r^T .$$

This is called the **truncated SVD**. It is not a proper decomposition of M, since M_r won't be *equal* to M, but it will be, in a very precise sense, the closest we can get under these constraints. We'll dig into the details of that later.

4.1.3 Principal Component Analysis by SVD

Before we move on, let's convince ourselves we can actually compute the SVD for a given M. The derivation of the SVD we've used above suggests a simple algorithm to decompose a matrix M. We can use gradient descent to find the singular vector v for which $\|Mv\|$ is maximal, subject to the constraint that v is a unit vector. Once gradient descent has converged, we find the next vector in the same way, but with the added constraint that it is orthogonal to all the vectors we've already chosen.

This is of course, similar to how we computed the PCA in the first chapter. Like in that chapter, we can enforce the constraints by projecting back to an orthogonal unit vector after the gradient update step.

If we stop collecting vectors at some arbitrary r, we get the truncated SVD. If we stop when we see the first v for which $Mv = 0$, we get the compact SVD, and finally, if we then also extend V and U to full bases, we get the full SVD.

This isn't the most precise or efficient SVD algorithm, and it isn't much better than the algorithm that we already developed for computing PCA in Chapter 1, but for now it should suffice to convince us that we *can* compute the SVD. We'll first see how exactly to implement PCA on the basis of a given algorithm for the SVD. We'll see some better algorithms in the next chapter.

Let X be the data matrix for our mean-centered data, and let $U\Sigma V$ be its (full) SVD.

It should be clear from the exposition above that once we have the SVD, there isn't a whole lot left to do.

We know that the principal components are the eigenvectors of $\frac{1}{n}X^TX$. The constant factor $\frac{1}{n}$ affects the eigenvalues, but not the eigenvectors, so we are equivalently looking for the eigenvectors of X^TX. These are the right singular vectors of X, which are the columns of V in our SVD.

So, the principal components follow directly from the singular value decomposition. Let's look at some of the things we may want to do with the principal components in practice, and see how we would achieve that.

Projecting the data to a lower dimension This is probably the main use case of principal component analysis. Reduce the dimensionality of the data so that we can feed it to some expensive algorithms or visualize it in 2D or 3D. To reduce the data to r dimensions, we need to project it orthogonally onto the first r principal components. We achieve this by computing the SVD truncated to r, and computing $Z = XV_r$.

As we can see in the decomposition, this is equal to $U_r\Sigma_r$. This multiplication is cheaper and more stable to compute, because Σ_r is a diagonal matrix.

Recovering the data As we saw in Chapter 1, we can partly reconstruct the data based on these lower-dimensional representations. Each element z_i of the latent representation tells us that we should multiply the i-th principal component by this value,

and and sum all these together.

In matrix terms $\mathbf{X}' = \mathbf{Z}\mathbf{V}_r^\mathsf{T} = \mathbf{U}_r\mathbf{\Sigma}_r\mathbf{V}_r^\mathsf{T}$.

Whitening In whitening, discussed in Chapter 2, we don't want to reduce the dimensionality of the data, we want to transform it so that it takes the shape of a standard normal distribution (or as close as we can get with a linear transformation). For this, we need a full basis $\mathbf{V}$, so we should compute at least the first m singular vectors, and extend $\mathbf{V}$ to be square if necessary. Then, we can compute a complete represention of the data $\mathbf{Z} = \mathbf{X}\mathbf{V}^\mathsf{T} = \mathbf{U}_m\mathbf{\Sigma}_m$.

This projects the data onto the eigenbasis, but we still need to scale the data along each axis, so that it has unit variance. In 1D, we to this by dividing by the standard deviation, and we characterized the singular values as the analogue of the standard deviation, so it would be intuitive if we should divide by the singular values.

To prove this, remember from Chapter 2 that the whitening transformation is the inverse of $\boldsymbol{x} = \mathbf{A}\boldsymbol{z} + \mathbf{t}$. That is

$$\boldsymbol{z} = \mathbf{A}^{-1}(\boldsymbol{x} - \mathbf{t}) = \mathbf{A}^{-1}\boldsymbol{x} - \mathbf{A}^{-1}\mathbf{t}\,.$$

We'll assume that our data is mean-centered, so $\mathbf{t} = \mathbf{0}$.

We also saw that $\mathbf{A}$ should be a square root of the covariance matrix:

$$\mathbf{A}\mathbf{A}^\mathsf{T} = \frac{1}{n}\mathbf{X}^\mathsf{T}\mathbf{X}$$

If we expand $\mathbf{X}$ into its SVD, we see that the SVD allows us to compute this square root:

$$\begin{aligned} AA^T &= \frac{1}{n}X^TX \\ &= \frac{1}{n}(U\Sigma V^T)^T U\Sigma V^T \\ &= \frac{1}{n}V\Sigma U^T U\Sigma V^T \\ &= \frac{1}{\sqrt{n}}V\Sigma\frac{1}{\sqrt{n}}\Sigma V^T \end{aligned}$$

This means that with $A = \frac{1}{\sqrt{n}}V\Sigma$ we have found our square root of the covariance. Not only that, we also have it in a form that is easy to invert:

$$\begin{aligned} A^{-1} &= \left(\frac{1}{\sqrt{n}}V\Sigma\right)^{-1} \\ &= \sqrt{n}\Sigma^{-1}V^T \end{aligned}$$

where Σ^{-1} consists simply of Σ, but with the diagonal elements inverted. We can now use A^{-1} to normalize our data.

Note that because we computed an SVD, the inversion, which is normally a very unstable operation, becomes very stable. It isolates the inversion of scalars to the diagonal elements of a matrix, and reduces the rest to transposing an orthogonal matrix.

> *This idea, that an SVD is a good way to invert a matrix, runs much deeper, as we shall see when we come to the pseudo-inverse.*

Wide data So far, we've assumed that our data is *tall*. That is, we have many more instances than features. If the reverse is true, as it was for the dataset of faces in Chapter 1, we may want to be careful how we compute the SVD. If we have precise control over how many singular vectors we compute, the distinction doesn't matter. Some implementations, however always

compute a number corresponding to the width of the input matrix. In this case, with wide data, it may be better to compute the SVD of the transpose, from which we can get the correctly reduced version of the original SVD:

$$X^T = \left(U_n \Sigma_n V_n^T\right)^T = V_n \Sigma_n U_n^T$$

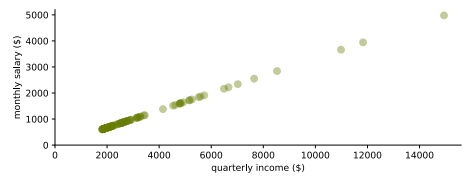
So that's the SVD: a decomposition that is so similar to PCA, you may be wondering why we separate the two at all? Isn't computing the SVD the same as computing the PCA? The answer is that the SVD is not *just* a way to compute a PCA. It can be used to compute many other things as well.

Next, we'll take a look at some of the most important things the SVD can do for us, beside computing a PCA. Along the way we'll try to build some additional intuition for how the SVD operates.

The first use case will provide an intuition for the number of singular values we can expect in a matrix.

4.2 Rank

Think back to the income dataset from the very first chapter, before we made it more realistic by adding some noise.



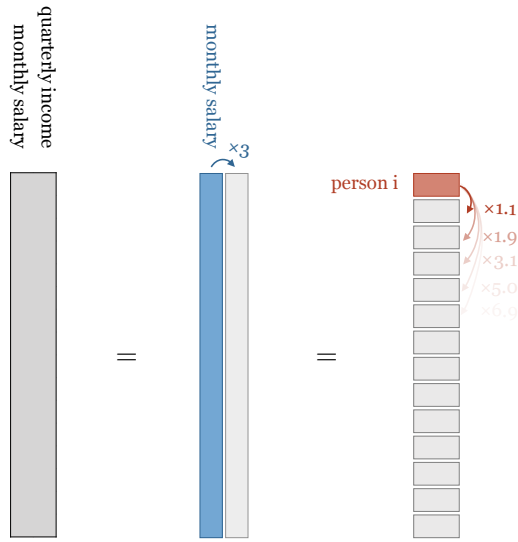
A plot of monthly salary vs. quarterly income for a small sample of people. Since there is a simple linear relation between these two quantities, the data lies on a straight line.

Even though this data is presented as two-dimensional—with two features—it is really inherently one-dimensional. This is easy to see in the picture, since the data lies on a line. What does this

look like in the data matrix $\mathbf{X}$? Note that each x value can be derived from the y value by multiplying it by 3. This means that the second column of our data matrix is a *multiple* of the first. For every row r, $\mathbf{X}_{r1} = 3\mathbf{X}_{r2}$.

Interestingly, the same holds true in the vertical direction. If we think of each person in the data as a vector, all these vectors point in the same direction. Only their lengths differ. This means that if we fix some arbitrary instance i, desribed by $\mathbf{x}_i$, the i-th row of $\mathbf{X}$, every other instance (every row in $\mathbf{X}$) can be described as some scalar m times $\mathbf{x}_i$.

In some sense, this tells us that the matrix $\mathbf{X}$ is *compressible*. We are given $300 \cdot 2 = 600$ numbers, but we really only need one vector in $\mathbb{R}^{300}$ and a single scalar, or one vector in $\mathbb{R}^2$ and 299 scalars.



We can represent the data by storing the monthly salary of all subjects and multiplying it by 3 to get the quarterly income, or we can store one reference person, together with a multiplier for every other person in the data.

How much a matrix can be compressed in this way is indicated by its **rank**. In this case the horizontal reduction tells us that the matrix has a *column rank* of one—we can represent it by just one of its columns and a bunch of multipliers.

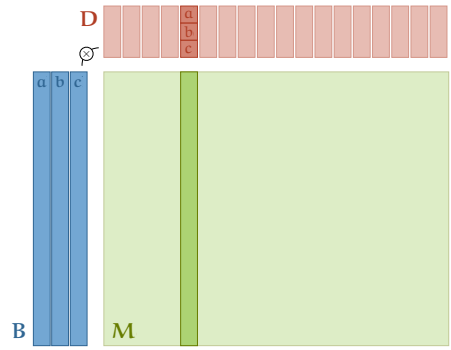
It has *row rank* of one as well—we can also represent it by one of its rows and a bunch of multipliers. We will see in a bit that it is no accident that the row and column rank are the same.

What does it mean to have a rank higher than one? Let's say that a matrix can be *compressed* to n columns if a subset of n of its columns is enough to express the rest as a linear combination of these n. For instance, if a matrix has column rank of 3, then there are three column vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, so that every other column can be expressed as $a\mathbf{a} + b\mathbf{b} + c\mathbf{c}$ with some numbers a, b, c unique to that column. The column rank of a matrix is the smallest number of columns that it can be compressed to, and and the row rank is the smallest number of rows that it can be compressed to.

It's helpful to draw such a compression in as a matrix multiplication. Let's assume we have a large rectangular matrix $\mathbf{M}$ with dimensions $n \times m$ and let's assume that it has column rank 3. This means that we can pick three of its columns $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ and represent all other columns by a set of just three numbers a, b and c.

Before we draw the diagram, note that $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ don't actually *have* to be columns of $\mathbf{M}$. We could, for instance, scale them to unit vectors, and we could still represent all columns of $\mathbf{M}$ as linear combinations. The key point is that we can express all column vectors of $\mathbf{M}$ in some *basis* of three vectors. They are all points in some 3D subspace of $\mathbb{R}^n$.

Now, to our diagram. Put the three basis vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ side by side in a matrix $\mathbf{B}$ of $n \times 3$. For a given column of $\mathbf{M}$, we can work out what a, b and c should be and put these together in a column vector $\mathbf{d}$. Multiplying $\mathbf{Bd}$ gives us the linear combination $a\mathbf{a}+b\mathbf{b}+c\mathbf{c}$, and thus reconstructs our column vector of $\mathbf{M}$. If we work out all the vectors $\mathbf{d}$ for all columns of $\mathbf{M}$ and concatenate them as the columns of a big $3 \times m$ matrix $\mathbf{D}$, then multiplying $\mathbf{BD}$ reconstructs all columns of $\mathbf{M}$. Or, put simply, $\mathbf{M} = \mathbf{BD}$.

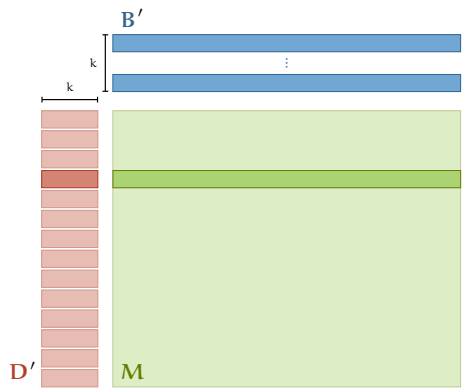


A matrix $\mathbf{M}$ with column rank 3 can be written as the multiplication of an $n \times 3$ matrix and a $3 \times m$ matrix.

This is called a **rank decomposition** of $\mathbf{M}$.

Looking at this diagram, we can almost directly deduce one of the more magical facts about ranks: **that the column rank is equal to the row rank**.

Imagine if we were to follow the same recipe for the row rank. We don't know the row rank, so let's call it k. If the row rank is k, there will be k rows that can serve as a basis for all rows. We concatenate them into a $k \times m$ matrix $\mathbf{B}'$ and work out the row vectors $\mathbf{d}'$ for each row of $\mathbf{M}$. Concatenating these into a matrix $\mathbf{D}'$ of $n \times k$, we get $\mathbf{D}'\mathbf{B}' = \mathbf{M}$. In other words, we get exactly the same diagram, but with the roles reversed.



This means that in the second diagram, we can also interpret $\mathbf{D}'$ as representing k column vectors and $\mathbf{B}'$ as representing the scalars required to reconstruct the columns of $\mathbf{M}$. And from this we can deduce directly what k should be. Since 3 is the column rank of $\mathbf{M}$, k can't be less than 3 or we would have found a representation of $\mathbf{M}$ in fewer columns (and the rank is the lowest number of columns possible). It also can't be more than 3, because then we could go the other way around: start with the second diagram, interpret it as a row rank diagram, and we'd find a representation with fewer rows than k, even though k is the row rank. So k must be 3.

This argument shows us that the column and the row rank of any matrix *must* be equal. We can drop the qualifier, and just refer to the matrix's **rank**. We can also conclude that the maximum possible value of the rank is $\min(n, m)$: the column rank cannot exceed the number of columns, and the row rank cannot exceed the number of rows, so the minimum of these two must be the matrix rank.

A matrix with rank equal to $\min(m, n)$ is called *full rank*. A matrix that isn't full rank is called *rank deficient*.

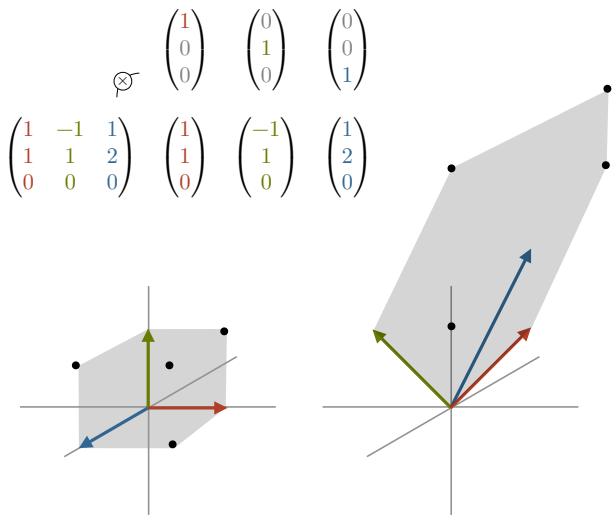
The rank is a very fundamental property of matrices with many interpretations in many contexts. Here are three examples, the last of which will bring us back to the singular value decomposition.

Dimension of the image of $\mathbf{M}$ In Chapter 3, to explain the determinant, we modeled the operation of a square matrix as a mapping from a unit cube to a polyhedron. We can extend this intuition to rectangular matrices as well. Take an $n \times m$ matrix $\mathbf{M}$. The input space of this matrix is spanned by m unit vectors pointing along the axes of $\mathbb{R}^m$. If we imagine these as m sides defining a unit cube, then we can ask how this cube is transformed by $\mathbf{M}$ into a parallelotope in $\mathbb{R}^n$.

The input vectors are one-hot, so multiplying one by $\mathbf{M}$ just picks out one of the columns of $\mathbf{M}$. Thus, we are creating the parallelotope spanned by the columns of $\mathbf{M}$. If all of these point in different directions, the parallelotope is of dimension m. If we only get $r < m$ unique directions, we get an r-dimensional shape.

We also get a shape of less than m dimensions if one or more of the vectors lies in a plane spanned by two of the others, or more generally in an subspace spanned by some of the others. In short, if one of the vectors is a linear combination of the rest.

The rank of $\mathbf{M}$ tells us the dimension of the *image*: the shape we get after operating on a shape of dimension m. Since column and row rank are equal, we now know that this is also the dimension of an image of $\mathbf{M}^T$. In the image below, we look at the image of the unit cube under a matrix. Since the matrix has rank two, the resulting figure is two-dimensional.



The matrix above maps the three standard basis vectors to three different points in space that correspond to the columns of the matrix. The matrix has low rank because we can express one column as the sum of the the other two. This means that the unit cube is flattened into a two-dimensional sheet containing all three vectors.

Invertibility and determinants A square matrix $\mathbf{A}$ with dimensions $n \times n$ can be invertible. That means that given $\mathbf{y}$, there is always only one $\mathbf{x}$ such that $\mathbf{y} = \mathbf{A}\mathbf{x}$. We've already seen that the

determinant tells us when a matrix is invertible: if the image of the matrix has volume 0, then the matrix is not invertible. Note that we are talking about n-volume here. If the matrix is 3×3 and the image of the unit cube is 2 dimensional, then it has a nonzero area but zero volume. In general if the image of an $n \times n$ matrix has nonzero n-volume, the determinant of $\mathbf{A}$ is zero.

Any shape of dimension less than n has zero n-volume: a square has zero volume, and a line has zero area. This means that if the rank tells us that the image of a $\mathbf{A}$ has dimension less than n, $\mathbf{A}$ must have determinant 0 and must therefore be non-invertible.

The reverse direction also works. If $\mathbf{A}$ is full-rank, the image, however small, is n-dimensional, and must have a nonzero n-volume. If we look at the image of the unit cube, we see that the images of the standard basis vectors *must* point in different directions if $\mathbf{A}$ is full rank, and the resulting parallellotope must have nonzero volume.

This provides one way to check for matrix singularity: if any of the columns is a linear combination of the rest, your matrix must be rank deficient, and therefore non-invertible, and have determinant 0. If there is no such column, the matrix is full rank, invertible and has nonzero determinant.

Number of singular values Here, we come back to the topic at hand. We've defined what singular values are, and how to enumerate them one by one. We've seen that this enumeration stops at some point (when $\|\mathbf{M}\boldsymbol{v}\| = 0$), but we haven't talked about *when* we can expect it to stop. How many singular values should we expect a given matrix to have?

You can probably guess the answer, so let's start there and work backwards: *the number of non-zero singular values a matrix has, is equal to its rank*.

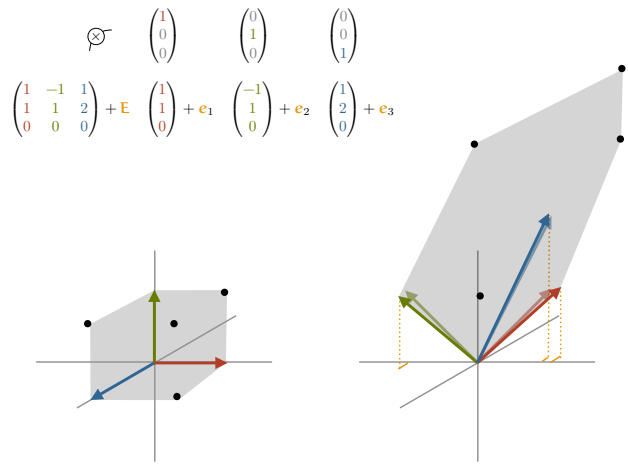
How do we show this? We discussed above that the rank is the dimension of the image of a matrix. That means that if we start with all unit vectors in $\mathbb{R}^m$, a sphere of dimension m, and look at the image of this set under $\mathbf{M}$, we will see a k-dimensional ellipsoid. This means that the process that enumerates the singular vectors can spit out k mutually orthogonal vectors. After

that, every vector $\boldsymbol{v}$ that is orthogonal to the ones already produced must have $\mathbf{M}\boldsymbol{v} = \mathbf{0}$.

4.2.1 Computing rank

So, that's rank. A fundamental, and very useful property of matrices. Now, given a matrix $\mathbf{M}$, how do we *compute* its rank? There are many algorithms: QR decomposition, row reductions, etc. The problem with all of these, is that when we encounter a matrix in the wild, there is often a little bit of noise. Either the matrix contains measurements, which are always subject to noise, or it's the result of some numeric computation, in which case floating point errors probably add a little bit of imprecision.

Any small amount of noise means that, with overwhelming probability, no column vectors will lie in *exactly* the subspace spanned by the others. The noise will always push it a tiny bit out of the subspace. This will technically make the matrix full rank, but what we are actually interested in, is what rank the matrix would have if we could remove the noise.

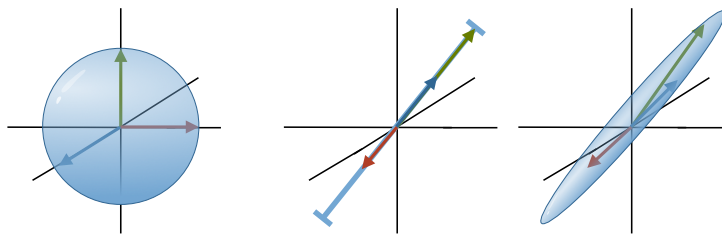


The low-rank matrix we used above, but now with a noise matrix $\mathbf{E}$ added to it, containing random values of small magnitude. The result is that each of the columns is nudged a little in a random direction, pushing it out of the xy-plane. The image of the unit cube is now no longer a 2D sheet, but a very flat 3D shape.

This is why the SVD is the preferred way of computing the rank of a matrix. If have a column vector that lies almost but not quite in the subspace spanned by the others, the result is *a very small singular value*.

To see why, imagine a rank 1 matrix with dimensions 3×3. This matrix would have all column vectors pointing in the same direction, and the image of the unit sphere would be a line segment. As a result, we get one singular value.

If we apply a tiny bit of noise to one of the columns, one of the vectors in the image won't quite point in the same direction as the others. As a result, we get an ellipse that looks a lot like a line segment: very thin in one direction. The image has 2 axes, so we get two singular vectors: one very similar to what we had before, for the main axis of the ellipse, and one very tiny one orthogonal to it, representing the extra dimension created by the noise.



For a rank-1 matrix, the image of the unit sphere is a line segment. If a little noise is added, the image becomes an elongated ellipsoid. Its major axis has large magnitude, corresponding to the original singular value and vector. The other two are very small: these correspond to the singular values introduced by the noise.

If the noise is small, then the singular values created by the noise are of an entirely different magnitude to the original singular values. This is why the SVD helps us to compute the rank in the presence of noise: we simply set a threshold to some low number

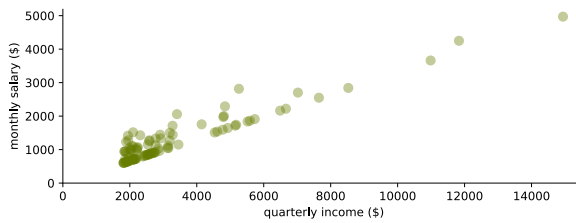
like 10^{-7}, and treat any singular value below the threshold as zero. The number of singular values remaining is the rank of our matrix *if we ignore the noise*.

What if we were to remove the corresponding singular *vectors* as well? If r is the rank we've established with the method described above, this means performing an r-truncated SVD. Multiplying the matrices $\mathbf{U}_r$, $\mathbf{\Sigma}_r$, $\mathbf{V}_r$ back together gives us a matrix $\mathbf{M}'$ which approximates $\mathbf{M}$, but removes the singular vectors corresponding to the noise. Can we treat this as a denoised version of our matrix? If so, what can we say about the remainder?

We can answer that question very precisely, but to do so, we'll first look at how the SVD can help us solve equations.

4.3 The pseudo-inverse

A very common use of the singular value decomposition is to solve systems of linear equations. To put that in a familiar context, let's return again to the example that we started with in Chapter 1: the salary dataset. This time, we'll look at the more realistic version.

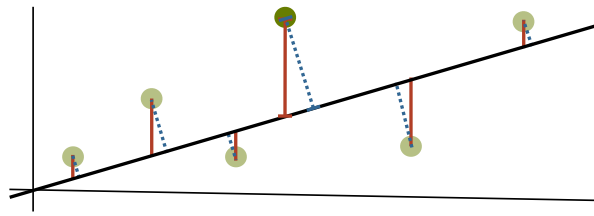


When I first showed this picture, I tried to steer you away from seeing this as a linear regression problem, where the task is to predict the variable on the vertical axis from the one on the horizontal. PCA, after all, is an *unsupervised* method: no feature in our data is marked as the target to be predicted. Instead, we want a compressed representation from which we can "predict" all of them as well as possible.

Let's forget about PCA for the moment, and let's return to the supervised setting. If we do want to predict one feature given the others, how does the SVD help us do this, and how does the solution to this problem relate to the solution to the PCA problem?

We'll stick with linear regression. This means we want to draw a line through these points that provides a prediction for the variable on the vertical axis. We want to choose this line so that the squares of the vertical distance to the data are minimized.

Here's a comparison to the definition of the first principal component. For the linear regression, we minimize the squares of the vertical distance, while for the principal component, we minimize the squares of the distances across both dimensions.



In PCA, the objective is to choose the black line to minimize the squared distances between the data and the line. In linear regression, we are minimizing only the squares of the the vertical distances.

For a solution with an arbitrary number of features, we will assume we have n instances with m input features each, collected into an $n \times m$ data matrix $\mathbf{X}$, and that for each instance $\mathbf{x}_i$, we have one target value y_i, collected into a vector $\mathbf{y} \in \mathbb{R}^n$.

Our goal is to find a vector $\boldsymbol{w}$ of m weights, and a scalar t so that the prediction $\mathbf{x}_i{}^T\boldsymbol{w} + t$ is as close to the target value y_i as possible. Ideally, we'd want to find a $\boldsymbol{w}$ so that $\mathbf{x}_i{}^T\boldsymbol{w}$ is exactly equal to y_i for all instances. Or in matrix notation

$$\mathbf{X}\boldsymbol{w} + t = \mathbf{y}$$

with $\mathbf{y}$ a long vector collecting all n target values.

In most cases, such an ideal solution is unlikely to exist. Even if there is a perfectly linear relation between the features and the target, there is probably a little noise in the data.

Instead, we'll measure how close our predictions are to the target by the square of the difference between them: $\left(\mathbf{x}_i{}^T \boldsymbol{w} + t - y_i\right)^2$.

Summing these squares over all instances, our objective becomes

$$\underset{\boldsymbol{w},t}{\operatorname{argmin}} \sum_i \left(\mathbf{x}_i{}^T \boldsymbol{w} + t - y_i\right)^2 .$$

We could solve this by gradient descent, or some other standard optimization algorithm, but when it comes to least-squares problems for linear models like these, we can find a more direct solution. Even better, we can get there mostly by geometric intuitions.

First, let's get rid of the term t. The simplest trick here is to add a feature to the data whose value is always one. The result is that $\boldsymbol{w}$ gains an extra value which is always added to the output, just like t is in the formulation above.

In neural networks, this trick is implemented by adding a so-called "bias node."

If we assume that this trick has been applied to $\mathbf{X}$ and $\boldsymbol{w}$, our problem reduces to

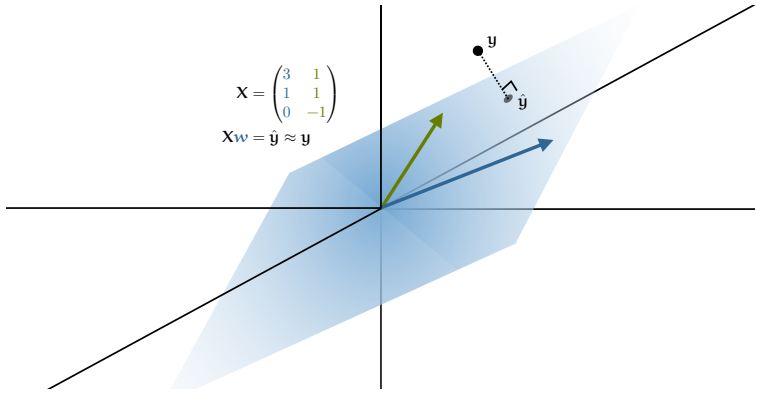
$$\underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_i \left(\mathbf{x}_i{}^T \boldsymbol{w} - y_i\right)^2$$

or in matrix notation

$$\underset{\boldsymbol{w}}{\operatorname{argmin}} \ \|\mathbf{X}\boldsymbol{w} - \mathbf{y}\|^2 .$$

Note that the term $\mathbf{X}\boldsymbol{w}$ is computing a linear combination of the columns of $\mathbf{X}$: each element of $\boldsymbol{w}$ is multiplied by one of the columns of $\mathbf{X}$ and the result is summed together. The space of all linear combinations of the columns of a matrix $\mathbf{X}$ is called its **column space**, denoted col $\mathbf{X}$. Every possible $\boldsymbol{w}$ results in a unique point $\mathbf{X}\boldsymbol{w}$ in the column space of $\mathbf{X}$. The set of all these points together coincides with the entire column space.

So, we can now say that we are looking for the point in the column space of $\mathbf{X}$ that is closest, by least squares, to the vector $\mathbf{y}$ (which may be outside the column space).



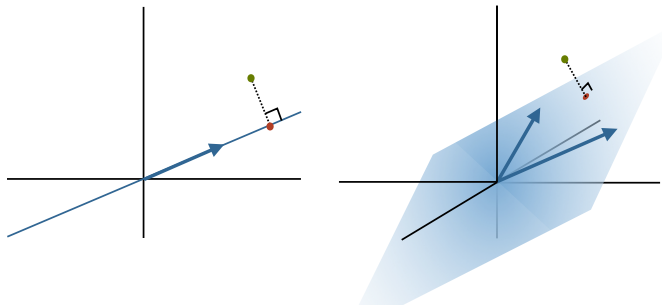
The values $\mathbf{X}\boldsymbol{w}$ are constrained to the column space of $\mathbf{X}$, represented by the blue plane. The best solution $\hat{\mathbf{y}}$ to our linear regression problem is the closest we can get to $\mathbf{y}$ while staying in the the plane.

This is a bit of a change of perspective. We are used to visualizing the space $\mathbb{R}^m$ where every instance in our dataset is a point in space and every feature is an axis. Now, we're visualizing the space $\mathbb{R}^n$, where every feature in our dataset is a point, and every instance is an axis.

In Chapter 1, we saw a simple result called the "best approximation theorem." It stated that if we have a point $\mathbf{q}$ and a line P, the closest to $\mathbf{q}$ we can get while staying on the line P is the orthogonal projection of q onto P.

We have a similar situation here. There is a point $\mathbf{y}$ that we want to approximate as best we can, but we have to stay in a specific part of space: not a line, but a *subspace*. Specifically the column space of $\mathbf{X}$.

A (linear) subspace is a subset of a larger space which contains the origin and for which the linear combination of any two vectors of the subspace is also in the subspace. Aany line, plane or hyperplane that crosses the origin is a subspace. A column space is always a subspace.



If our subspace is a line, the closest we can get to any point is the orthogonal projection of that point onto the line. If we define orthogonal projections correctly for n-dimensional subspaces, like a plane that crosses the origin, this principle applies in general.

Happily, the best approximation theorem holds also when the subspace we're restricted to is more than a line: the closest we can get to a point $\mathbf{q}$ while staying in some subspace S is the orthogonal projection of $\mathbf{q}$ onto S. In our case, that means that the best approximation to $\mathbf{y}$ within the column space of $\mathbf{X}$ is the orthogonal projection of $\mathbf{y}$ onto col $\mathbf{X}$.

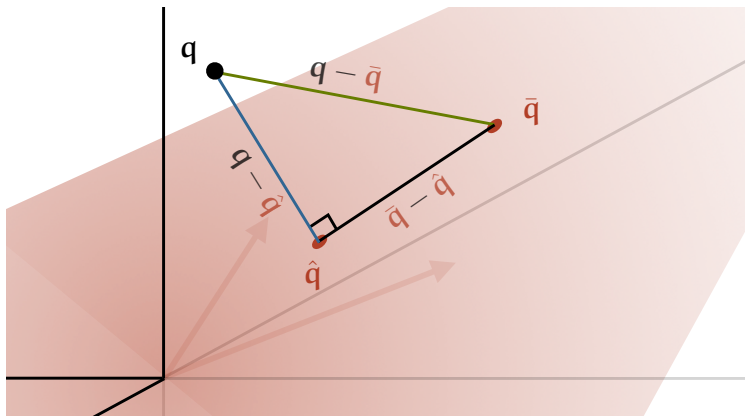
The first thing we should do is to define what an orthogonal projection *means* in this context. We'll call a vector orthogonal to a subspace S if it is orthogonal to every vector in S. Think of a plane intersecting the origin: a vector orthogonal to the plane points out of the plane at a right angle to every direction in the plane.

An orthogonal projection of $\mathbf{q}$ onto S is a point $\hat{\mathbf{q}}$ in S so that the difference vector $\mathbf{q} - \hat{\mathbf{q}}$ is orthogonal to S. Note how this

generalizes the one-dimensional case: the line segment pointing out of the line P at the orthogonal projection is orthogonal to every vector that points along P.

Now, we want to show that if we find an orthogonal projection of $\mathbf{q}$ onto S, that that is the closest we can get to $\mathbf{q}$ while staying in S. We'll follow the same logic as we did in the 1D case. Imagine that we have an orthogonal projection $\hat{\mathbf{q}}$ and any other point $\bar{\mathbf{q}}$ in S. No matter how many dimensions our space has, the three points $\mathbf{q}$, $\hat{\mathbf{q}}$ and $\bar{\mathbf{q}}$ form a triangle.

In this triangle, we know that the points $\hat{\mathbf{q}}$ and $\bar{\mathbf{q}}$ are in S so the vector $\hat{\mathbf{q}} - \bar{\mathbf{q}}$ must be as well. Because this vector is in S, we know that $\mathbf{q} - \hat{\mathbf{q}}$ is orthogonal to it so our triangle has a right angle at $\hat{\mathbf{q}}$.



From the image, you can already see that the vector $\mathbf{q} - \hat{\mathbf{q}}$ must be shorter than or equal to $\mathbf{q} - \bar{\mathbf{q}}$. The Pythagorean theorem lets us formalize this:

$$\|\mathbf{q} - \bar{\mathbf{q}}\|^2 = \|\hat{\mathbf{q}} - \bar{\mathbf{q}}\|^2 + \|\mathbf{q} - \hat{\mathbf{q}}\|^2 .$$

This tells us that the only way $\bar{\mathbf{q}}$ can be as good an approximation as $\hat{\mathbf{q}}$ is if $\|\hat{\mathbf{q}} - \bar{\mathbf{q}}\| = 0$, which implies that they are the same vector.

We haven't shown that $\hat{\mathbf{q}}$ exists or that it is unique, but if we can find a set of one or more orthogonal projections of $\mathbf{q}$, they must be the closest we can get to $\mathbf{q}$ while staying inside the subspace S.

As it happens, if S is a proper subspace, and $\mathbf{q}$ is outside it, then there is a unique orthogonal projection $\hat{\mathbf{q}}$. The proof is simple, but we don't need it here.

Let's return to the problem at hand. We've noted that the predictions our model w can produce for the data $\mathbf{X}$ are restricted to the column space of $\mathbf{X}$. By the argument above, the closest we can get to the target values $\mathbf{y}$ is the orthogonal projection of $\mathbf{y}$ onto this space. The vector $\hat{\mathbf{y}} = \mathbf{X}w$ provides an orthogonal projection if the vector e pointing from $\hat{\mathbf{y}}$ to $\mathbf{y}$ is orthogonal to the column space of $\mathbf{X}$.

This happens if it is orthogonal to all columns of $\mathbf{X}$. Any vector orthogonal to all columns is also orthogonal to any linear combination of all columns, so it must be orthogonal to col $\mathbf{X}$.

We can phrase this symbolically as

$$\mathbf{X}^T e = \mathbf{0}$$

with $e = \mathbf{y} - \hat{\mathbf{y}}$, and $\mathbf{0}$ a vector of zeros. With a bit of filling in and rewriting, we get

$$\begin{aligned} \mathbf{X}^T (\mathbf{y} - \hat{\mathbf{y}}) &= \mathbf{0} \\ \mathbf{X}^T \mathbf{y} &= \mathbf{X}^T \hat{\mathbf{y}} \\ \mathbf{X}^T \mathbf{y} &= \mathbf{X}^T \mathbf{X} w \end{aligned}$$

This is called the **normal equation** for our linear problem. Any w that satisfies this equation is a least squares solution. Note that $\mathbf{X}^T\mathbf{X}$ has popped up again.

In fact, we can see here that **if $\mathbf{X}^T\mathbf{X}$ is invertible**, we can multiply both sides by the inverse and get

$$\left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{y} = w$$

as a single, unique solution for w.

The term $\left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T$ is an instance of something called the **pseudo-inverse** of $\mathbf{X}$, written $\mathbf{X}^\dagger$. We'll provide a more robust

definition later, but it's interesting to see where the name comes from. Think back to the idealized version of our problem $\mathbf{X}\boldsymbol{w} = \mathbf{y}$. When we are lucky enough to have a square and full-rank $\mathbf{X}$, we can compute the exact solution by multiplying both sides by the inverse of $\mathbf{X}$: $\boldsymbol{w} = \mathbf{X}^{-1}\mathbf{y}$. We take the multiplier "to the other side" just like we would do with a scalar equation. General rectangular matrices don't have an inverse, but when we replace it by the pseudo-inverse, we get the least-squares solution, which will coincide with the ideal solution if one exists.

In fact, if $\mathbf{X}$ is invertible, $\mathbf{X}^\dagger$ coincides with the inverse, because, from basic properties of the inverse we get $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T = \mathbf{X}^{-1}{\mathbf{X}^T}^{-1}\mathbf{X}^T = \mathbf{X}^{-1}$.

Note that if $\mathbf{X}$ has dimensions $n \times m$, then $\mathbf{X}^\dagger$ has dimensions $m \times n$. This is consistent with the idea that if $\mathbf{X}$ represents a function from $\mathbb{R}^m$ to $\mathbb{R}^n$, then something analogous to its inverse should represent a function from $\mathbb{R}^n$ to $\mathbb{R}^m$

What happens if $\mathbf{X}^T\mathbf{X}$ is not invertible? It turns out this this happens precisely when the rank of $\mathbf{X}$ is strictly less than the number of columns. One such case is if $\mathbf{X}$ is wider than it is tall, like the dataset of portrait images we saw in Chapter 1. This makes sense if you think back to the ideal solution to our problem: $\mathbf{X}\boldsymbol{w} = \mathbf{y}$. If $\mathbf{X}$ is wider than it is tall (and full-rank) then this is a system of n equations with m variables: we will get many different solutions.

We can prove this fact simply from the SVD. This is a good illustration of how the SVD can be used for theoretical purposes. What we want to show is that if $\mathbf{X}^T\mathbf{X}$ is non-invertible, $\mathbf{X}$ must have a column rank less than m. Let $k < m$ be the rank of $\mathbf{X}$ and let

$$\mathbf{X} = \mathbf{U}_m\mathbf{\Sigma}_m{\mathbf{V}_m}^T$$

be the SVD of $\mathbf{X}$ truncated at m. Note that we only have k singular values so some of the diagonal of $\mathbf{\Sigma}_m$ is zero. Note also that the columns of $\mathbf{V}_m$ have length m, so $\mathbf{V}_m$ is square.

With this decomposition, we can write

$$\begin{aligned} \mathbf{X}^T\mathbf{X} &= \mathbf{V}_m\mathbf{\Sigma}_m{}^T\mathbf{U}_m{}^T\mathbf{U}_m\mathbf{\Sigma}_m\mathbf{V}_m{}^T \\ &= \mathbf{V}_m\mathbf{\Sigma}_m{}^2\mathbf{V}_m{}^T \end{aligned}$$

where the $\mathbf{U}_m$'s in the middle disappear, because $\mathbf{U}_m$'s columns are orthogonal to one another, so together they form an identity matrix. Note also that $\mathbf{\Sigma}_m$ is diagonal, so it's equal to its transpose and multiplying it by itself boils down to squaring the diagonal values.

The resulting decomposition is equal to the eigendecomposition of $\mathbf{X}^T\mathbf{X}$, which we already knew existed. However, what we've now shown as well, is that because $\mathbf{X}$ has lower rank than m, *some of its eigenvalues are zero*. We know that $\mathbf{V}_m$ and $\mathbf{V}_m{}^T$ are orthogonal, so they are invertible.

The composition of invertible matrices is invertible, so whether $\mathbf{X}^T\mathbf{X}$ is invertible boils down to whether $\mathbf{\Sigma}_m{}^2$ is invertible. A diagonal matrix with only *non*zero values on the diagonal is invertible: when we multiply a vector by it, we multiply each element of the vector by one of the diagonal elements, so we can get back the original elements by multiplying by their inverses. However, a diagonal matrix with a zero anywhere on the diagonal is singular. We lose information if we multiply by it, because those elements of the input vector that were multiplied by zero cannot be recovered from the output vector.

That means that if $\mathbf{X}$ has rank m, $\mathbf{\Sigma}^2$ must have m non-zero diagonal values, i.e. $\mathbf{X}^T\mathbf{X}$ must be invertible. If the rank of $\mathbf{X}$ is less than m, we get fewer nonzero values on the diagonal of $\mathbf{\Sigma}^2$, and $\mathbf{X}^T\mathbf{X}$ is singular.

What should we do when $\mathbf{X}^T\mathbf{X}$ is singular? This means we have multiple exact solutions to our problem $\mathbf{X}w = \mathbf{y}$, so we need an additional constraint to choose a single one. One common heuristic in machine learning is to pick models for which the parameter values are small. The simplest option is to pick the w that satisfies $\mathbf{X}w = \mathbf{y}$ for which $\|w\|$ is minimal.

One of the most beautiful properties of the pseudo-inverse is that if we define it carefully, it automatically delivers this solution if multiple solutions are available.

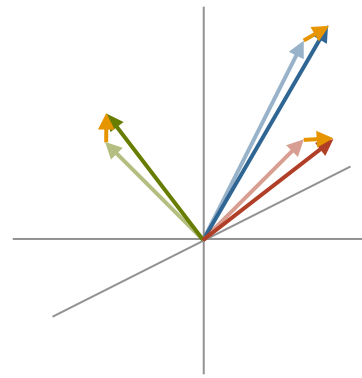
To show this, we will investigate what happens if we add a small invertible matrix to $\mathbf{X}^T\mathbf{X}$ to make it invertible. Specifically, we'll add a small multiple ϵ of the identity matrix. Doing this, the normal equation becomes

$$(\mathbf{X}^T\mathbf{X} + \epsilon\mathbf{I})\, w = \mathbf{X}^T\mathbf{y}\,.$$

The idea is that we can let ϵ go to zero to see which solution the pseudo-inverse gives us.

To see why $\mathbf{X}^T\mathbf{X} + \epsilon\mathbf{I}$ must be invertible, consider the vectors represented by its columns. If $\mathbf{X}^T\mathbf{X}$ is singular, and thus rank deficient, these must all lie in some subspace of lower dimension than m. By adding $\epsilon\mathbf{I}$, we are pulling each vector a little bit in the direction of a *different* axis. Imagine two vectors on the same line. If I pull one a little toward the horizontal, and the other a little toward the vertical, they must both leave the line they originally had in common.

$$\begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 2 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} \epsilon & 0 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon \end{pmatrix}$$



Adding a matrix $\epsilon\mathbf{I}$ to a low-rank matrix, nudges each column vector of the original by ϵ in the direction one of the axes, making the matrix full rank.

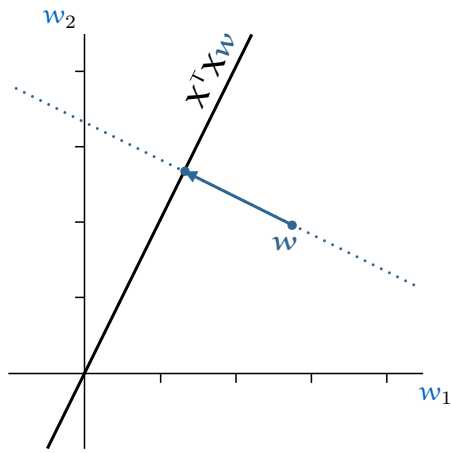
We can now rewrite our modified normal equation as

$$\mathbf{X}^T\mathbf{X}w = \mathbf{X}^T\mathbf{y} - \epsilon w\,.$$

To get some insight into what the solutions to this equation look like, let's imagine that we have an $n \times 2$ data matrix (n instances, 2 features) and that the first column of $\mathbf{X}$ happens to be a unit vector, and the second column is twice the first column. This gives us

$$\mathbf{X}^\mathsf{T}\mathbf{X} = \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}.$$

¡p¿This is a singular matrix: the second column is twice the first. Its operation is to squeeze the whole plane into a single line. Any w is mapped to a point $(x, 2x)$, with $x = w_1 + 2w_2$.



The low-rank matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ projects multiple points w onto the same point in its column space. In this case, a line of points is projected to a single point on the line $w_1 = 2w_2$.

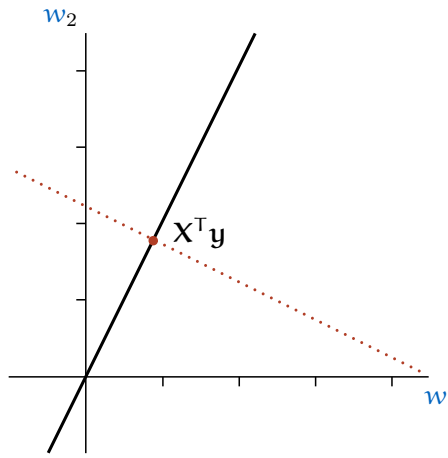
The line of all points mapped to $(x, 2x)$, its *pre-image*, is orthogonal to the line that forms the column space. In this specific instance, we can rewrite $w_2 = -\frac{1}{2}w_1 + \frac{1}{2}x$. That is, a line with slope $-\frac{1}{2}$, which we can tell by inspection is orthogonal to our line with slope 2. But this is not just true in this instance, it holds more generally. For any matrix $\mathbf{M}$, the pre-image of some point in col $\mathbf{M}$ is orthogonal to the column space itself.

See the appendix for a proof.

The original normal equation states that we want to choose w so that $X^T X w$ coincides with the point $X^T y$. Since the second column of X is twice the first, we have

$$X^T y = \begin{pmatrix} y' \\ 2y' \end{pmatrix}$$

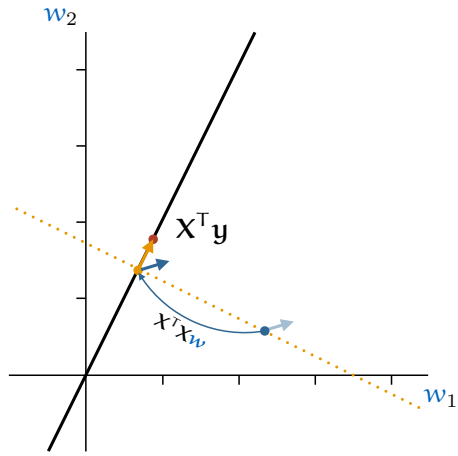
for some y'. This is on the line that corresponds to col $X^T X$. Every w for which $w_1 + 2w_2 = y'$, will be mapped onto this point giving us a solution for $X^T X w = X y$.



Given our target y, we note that $x^T y$ is in the column space of $X^T X$, and its pre-image forms an orthogonal line. Here we see the effect of an overdetermined problem: we have multiple solutions to the normal equation.

Now, when we introduce the term ϵw to the left-hand-side. This moves our target by a small multiple of w. Where formerly we got from w to the target $X^T y$ in a single matrix multiplication, we now need two steps: first we multiply $X^T X w$ and then we add ϵw. The first step is guaranteed to put us in the column space of $X^T X$, which is where the target is as well. Therefore, if we are going to reach the target, the second step should keep us

in the column space. In other words, w needs to point along the column space.



After we change the normal equation to $\mathbf{X}^T\mathbf{X}w + \epsilon w = \mathbf{X}^T\mathbf{y}$, we need an extra step to get to the target. The projection onto the column space should end slightly below $\mathbf{X}^T\mathbf{y}$ so that the term $+\epsilon w$ can take us to the target. We then see that only one point on the dotted line will do the trick: if we use the blue point, the second step will push us out of the column space. Only the w corresponding to the direction of the column space will allow the final step to actually reach $\mathbf{X}^T\mathbf{y}$.

This is how the term ϵw moves us from a problem with multiple solutions a problem with a single solution. We've also claimed that this solution is the one with the smallest norm. How do we see this in the image? The trick is to note that the the line P from which we choose w is orthogonal to the column space, which crosses the origin. This means that if we choose the point where the two cross, we are essentially *projecting the origin onto* P. This means we've chosen the point on P that is *closest* to the origin, or, in other words, the solution with the smallest norm.

We've ignored the fact that we changed the target by the term $\epsilon\mathbf{I}$, so we've slightly changed the problem. But we can justify

that by making ϵ very small. The whole derivation still holds, so long as it's non-zero. More technically, as we let ϵ go to zero, our solution converges to the one with the lowest L2 norm.

This is a very neat principle, which we'd like to work into our definition of the pseudo-inverse. To this end, we can define $\mathbf{X}^\dagger$ as the matrix

$$\mathbf{X}^\dagger = \lim_{\epsilon \to 0} \left(\mathbf{X}^T\mathbf{X} + \epsilon\mathbf{I}\right)^{-1}\mathbf{X}^T.$$

That is, if the inverse of $\mathbf{X}^T\mathbf{X}$ exists, we simply use it as before, but if it doesn't, we instead use the matrix that is the limit of $\mathbf{X}^T\mathbf{X} + \epsilon\mathbf{I}$ for ever smaller ϵ. In the context of our least squares problem, the limit adds an extra constraint so that we converge to the approximation to the inverse that will give us the solution with the smallest norm.

That was a long story, so let's recap:

- The pseudo-inverse lets us solve matrix equations. Whenever you feel like moving a matrix to the other side by taking its inverse, but the inverse doesn't exist, you can use the pseudo-inverse instead.
- This will give you an exact solution if one exists, and a least squares solution if it doesn't.
- If multiple solutions exist, you'll get the one with the smallest norm.

So, all in all, the pseudo-inverse is a pretty powerful tool. How do we compute it? Obviously, if we know that $\mathbf{X}$ has full column rank, we can just compute $\mathbf{X}^T\mathbf{X}$, take its inverse, and multiply by $\mathbf{X}^T$. But that isn't always the nicest approach. For one thing, computing the inverse is a numerically unstable business. In fact you usually avoid it by computing the pseudo-inverse instead, even if the inverse exists. Another problem is that this only works for $\mathbf{X}$ with full column-rank. Ideally, we would have a single, simple algorithm that returns the limit version of the pseudo-inverse if $\mathbf{X}^T\mathbf{X}$ isn't invertible, without us having to check.

As you may have guessed, one of the best ways of achieving this is by the singular value decomposition.

The algorithm is simple, so let's start there, and show why it's correct later. Given $\mathbf{X}$, we first compute its SVD

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T .$$

On the diagonal of $\mathbf{\Sigma}$, replace all non-zero values σ by their inverse $1/\sigma$, and call the resulting matrix $\mathbf{\Sigma}^\dagger$. We then reverse the basis transformations, and the result is the pseudo-inverse:

$$\mathbf{X}^\dagger = \mathbf{V}\mathbf{\Sigma}^\dagger\mathbf{U}^T .$$

Note that this also directly provides us with an SVD of $\mathbf{X}^\dagger$.

If we don't want to compute the full SVD, we get the same result with the truncated SVD, so long as we include all singular values.

This is because the $\mathbf{v}$ and $\mathbf{u}$ vectors corresponding to the $\mathbf{0}$ block of $\mathbf{\Sigma}$ can be safely removed without changing the value of the decomposition. We haven't modified these vectors when we computed $\mathbf{X}^\dagger$, so the result isn't changed if we don't include these.

The main reason that the SVD is the preferred way of computing the pseudo-inverse is one we've seen before: noise. In fact, noise can be particularly disastrous when we compute the pseudo-inverse. A small amount of noise turns zero singular values into very small non-zero singular values. We can see above that they become very *big* singular values in the pseudo-inverse because we take their inverse, when actually they should be ignored. In short, a small amount of noise in $\mathbf{X}$ creates a huge error in $\mathbf{X}^\dagger$. The SVD makes this process explicit, and allows us to simply treat very small singular values as zero, eliminating the problem.

Now, why does this algorithm give us the pseudo-inverse? Let's first look at the case where $\mathbf{X}$ has full column rank m. In that case, the pseudo-inverse is simply:

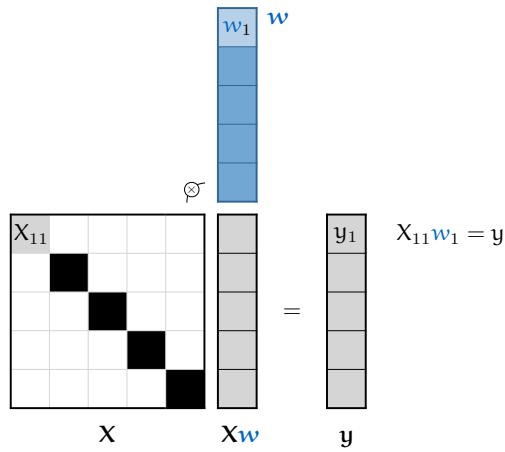
$$\mathbf{X}^\dagger = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T$$

Replacing each $\mathbf{X}$ by its SVD, truncated to m and omitting the subscripts to simplify the notation, we get

$$\begin{aligned} X^\dagger &= \left(V\Sigma U^T U\Sigma V^T\right)^{-1} V\Sigma U^T \\ &= \left(V\Sigma^2 V^T\right)^{-1} V\Sigma U^T \\ &= V\Sigma^{-2} V^T V\Sigma U^T \\ &= V\Sigma^{-1} U^T . \end{aligned}$$

Which is the result of the algorithm described above. Note that because we truncated at m, Σ and V are $m \times m$.

To make it intuitive exactly how SVD is solving our problem here, we can return to an idea we used before when we were developing eigenvectors: everything is a lot easier when your matrices are diagonal. Let's try that here. Imagine that X is a diagonal $n \times n$ matrix and we are looking for the vector w that minimizes the distance between Xw and a given $y \in \mathbb{R}^n$.

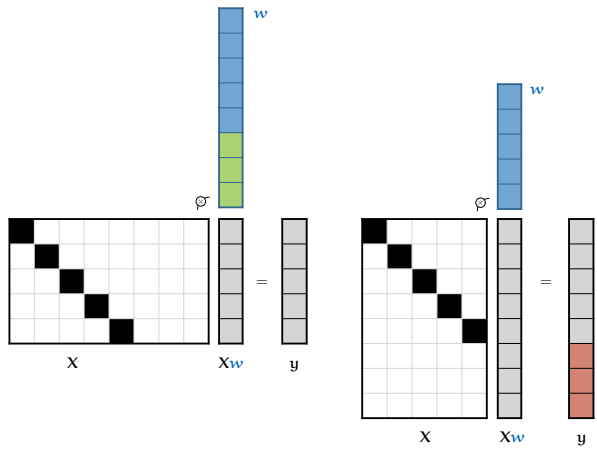


> The very simplest setting for the problem $Xw = y$ is when X is square and diagonal. In that case, each w_i can be solved as a simple scalar equation.

What we see here is that for a diagonal matrix, the choices for each w_i become in a sense *independent*. The only requirement for

w_1 is that it makes $X_{11}w_1$ as close as possible to y_1. We can make them equal by setting $w_1 = y_1/X_{11}$, without worrying about the other parts of $\mathbf{X}$ or w. We can do the same thing for every w_i.

If $\mathbf{X}$ is not square, but it is still diagonal, we get some extra zero rows or extra zero columns. In the first case, we can't control the elements of $\mathbf{X}w$ corresponding to the zero rows, so we may not be able to produce y exactly. The best solution is to optimize for the values we can control. In short, the solution in the previous paragraph will give us the orthogonal projection onto the column space of $\mathbf{X}$. In the second case, when $\mathbf{X}$ is wide, we get some extra elements of w, which we can set to whatever we like, because they correspond to zero columns. Setting these to zero will give us the minimal-norm solution.



If we have a diagonal, non-square $\mathbf{X}$, two situations can arise. If $\mathbf{X}$ is wide, we get some entries in w corresponding to zero-columns. We have an underdetermined problem with multiple possible solutions. We get the solution with the smallest norm by setting these entries to zero. If we have a tall $\mathbf{X}$, we have some entries in y corresponding to zero rows. This means we can never make $\mathbf{X}w$ exactly equal to y. We have an overdetermined problem, and the best we can do is to minimize $\|\mathbf{X}w - y\|$. We do this by ensuring that the elements of $\mathbf{X}w$ we *can* control are equal to the corresponding elements of y.

The important thing, in both cases, is that we can work out the

optimal solution simply by solving the scalar equation $X_{ii}w_i = y_i$ for every w_i individually.

When we studied eigenvectors, after we saw how easy things were for a diagonal matrix, all we needed to do to deal with non-diagonal matrices was to develop a *diagonalization*: a way to transform space to a basis where the transformation expressed by our matrix *becomes* expressed by a diagonal matrix.

That was the eigendecomposition. The singular value decomposition does the same thing for us here. It takes any non-diagonal matrix, and it gives us *two* orthogonal basis transformations U and V, one for the input space and one for the output space so that the remainder of the transformation can be expressed as a (rectangular) diagonal matrix.

We can see this happen more clearly if we rewrite directly from the minimization objective. Note first that if we take the full SVD and move the basis transformation to the other side, we get $U^T X V = \Sigma$: with two changes of basis, X becomes a diagonal matrix. This suggests that if we rewrite the data X and the target values y in the right bases, the minimization becomes a simple diagonal objective. We can work out the correct transformations with some simple rewriting:

$$\begin{aligned} \|Xw - y\| &= \|U\Sigma V^T w - UU^T y\| \\ &= \|\Sigma V^T w - U^T y\| \\ &= \|\Sigma w' - y'\| \end{aligned}$$

with $w' = V^T w$ and $y' = U^T y$.

And with that, we have a diagonal minimization objective. If we compute the SVD of X, we can transform y to y' easily. Once we find a solution for w', we can easily transform it back to the desired w by reversing the basis transformation: $w = Vw'$.

For the w' that solves the diagonal problem, there are a few different possible situations.

For now, let's stick with the assumption that X is full-rank. In that case, Σ has non-zero elements all along its diagonal. We simply set w'_i equal to $y_i \Sigma_{ii}$ and we exactly solve our problem. If Σ is taller than it is wide, we have some left-over values of w' with no corresponding elements of Σ. We can set these to

whatever we like, so if we want the w' with the smallest norm, we should set them to 0.

> *As before, we only get a perfect solution if Σ is square or wide (assuming it's full rank). If it's tall, we have a bunch of rows we can't control, and we have to settle for the orthogonal projection.*

Now, what do we do if $\mathbf{X}$ is rank deficient? In that case, some of the diagonal values of Σ will be zero. If we follow the simple definition of the pseudo-inverse, and fill in the SVD, we get stuck at:

$$\mathbf{X}^\dagger \stackrel{?}{=} \left(V\Sigma^2 V^T\right)^{-1} V\Sigma U^T .$$

Only the first k diagonal values of Σ^2 are nonzero, so $V\Sigma^2V^T$ does not have an inverse. The limit definition tells us to instead take the inverse of $V\Sigma^2V^T + \epsilon\mathbf{I}$, and to let ϵ go to zero. Why does this lead to the algorithm we described earlier? To me, it's not immediately clear that adding these ϵs will result in us ignoring the zeros on the diagonal of Σ, which is what the algorithm tells us to do.

Luckily, it's easier to see in the minimization objective. We start with the modified normal equation

$$(\mathbf{X}^T\mathbf{X} + \epsilon\mathbf{I})w = \mathbf{X}^T\mathbf{y} .$$

Since $\mathbf{X}$ is rank deficient, this doesn't have an exact solution. Instead, we can minimize the norm of the difference between the left and right sides. With a little rewriting, we get:

$$\begin{aligned}
\|(\mathbf{X}^T\mathbf{X} + \epsilon\mathbf{I})w - \mathbf{X}^T\mathbf{y}\| &= \|(V\Sigma^T U^T U\Sigma V^T + \epsilon VV^T)w - V\Sigma^T U^T\mathbf{y}\| \\
&= \|(\Sigma^T\Sigma V^T + \epsilon V^T)w - \Sigma^T U^T\mathbf{y}\| \\
&= \|(\Sigma^T\Sigma + \epsilon\mathbf{I})V^T w - \Sigma^T U^T\mathbf{y}\| \\
&= \|(\Sigma^2 + \epsilon\mathbf{I})w' - \mathbf{y}'\|
\end{aligned}$$

where we've set $\mathbf{y}' = \Sigma^T U^T\mathbf{y}$ and $w' = V^T w$. It's a bit more complex than what we had before, but the principle is the same.

In the final line, we see that we again have a simple linear problem with a diagonal matrix: $\Sigma^2 + \epsilon\mathbf{I}$.

The solution to this problem is to solve, for each w'_i, the scalar equation $({\Sigma_{ii}}^2 + \epsilon)w'_i = y'_i$, which gives us

$$w'_i = \frac{y'_i}{{\Sigma_{ii}}^2 + \epsilon}.$$

From the definition of $\mathbf{y}'$, we see that $y'_i = \Sigma_{ii}\mathbf{u}_i{}^T\mathbf{y}$ where $\mathbf{u}_i$ is the i-th column of $\mathbf{U}$. Filling this in, we get

$$w'_i = \frac{\Sigma_{ii} \times \mathbf{u}_i{}^T\mathbf{y}}{{\Sigma_{ii}}^2 + \epsilon}.$$

Here, we can see clearly where the problem of invertibility occurs: if Σ_{ii} is zero, and we don't add ϵ, we get a division by zero, which is undefined. The small ϵ added to the diagonal of $\mathbf{X}^T\mathbf{X}$ ends up being added to the denominator, so we are preventing this division by zero.

> *Since Σ_{ii} also occurs in the numerator, we know that these divisions by zero will always be instances of $0/0$. The addition of ϵ in the denominator turns this value into 0.*

The addition of ϵ also affects the result for w'_i where Σ_{ii} is nonzero, but we eliminate this effect by letting ϵ go to zero. In that case, the solution goes to $w_i = {\Sigma_{ii}}^{-1}\mathbf{u}_i{}^T\mathbf{y}$

Putting all this together, we can derive the algorithm we already know. Given $\mathbf{X}$, we compute its SVD $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. We transform our targets $\mathbf{y}$ to a new basis $\mathbf{y}' = \mathbf{\Sigma}^T\mathbf{U}^T\mathbf{y}$. We solve the diagonal problem

$$\underset{\mathbf{w}'}{\operatorname{argmin}} \left\| \left(\mathbf{\Sigma}^2 + \epsilon\mathbf{I}\right)\mathbf{w}' - \mathbf{y}' \right\|$$

which we've shown above gives us

$$w'_i = {\Sigma_{ii}}^{-1}y'_i = {\Sigma_{ii}}^{-1}\mathbf{u}_i{}^T\mathbf{y}$$

if $\Sigma_{ii} \neq 0$ and $w'_i = 0$ otherwise. In matrix notation, we can write

$$\mathbf{w}' = \mathbf{\Sigma}^\dagger\mathbf{U}^T\mathbf{y}.$$

Finally, we translate w' back to w by multiplying by V:

$$w = Vw' = V\Sigma^{\dagger}U^{T}y = X^{\dagger}y$$

which is the algorithm we started out with.

As always, there are other ways of computing the pseudo-inverse, but we like the SVD method, because we can very cleanly eliminate any noise by simply taking the singular values that are close to zero and treating them as zero.

So, what have we learned? Firstly, that the least squares linear regression problem is solved by forming the pseudo-inverse, and secondly, that the pseudo-inverse can be computed very robustly by taking the SVD, inverting the nonzero values of Σ, multiplying it back together and transposing it.

But don't think that all this is good for is linear regression. What we've shown here is the basis behind pretty much any linear learning method. Imagine that you have a large space of points $\mathbb{R}^n$ and some target y anywhere in that space. You are looking for a point $x \in \mathbb{R}^n$ as close to y as possible but under the constraint that x is in some linear subspace P of dimension n. In that case the pseudo-inverse will give you your solution, quickly and precisely. This may sound a bit abstract, but it covers a huge swathe of search problems in machine learning, optimization and computer vision.

Even if your problem isn't linear, for instance if you're training a neural network, this can be a worthwhile perspective. Such models are usually trained by assuming that they are *locally* linear, computing the linear solution to this local approximation, and then taking a small step towards this solution. This is, for instance, how gradient descent operates. Even if you are training a non-linear model, the linear perspective can tell you a lot.

Finally, it's worth taking a minute to consider what all this boils down to. The best approximation theorem at heart is nothing more than an application of the Pythagorean theorem. When I first learned that theorem, long ago, I didn't really get what the big deal was, why this theorem was so much more famous than any other one in Euclidean geometry. Now, I see that pretty much everything we do in statistics and machine learning comes down to the best approximation theorem, which in turn is built on the Pythagorean theorem.

4.4 Making use of the SVD

To finish up, let's see what else the pseudo-inverse and SVD can do for us. I promised that the basic framework of the best approximation theorem popped up in a lot of places, so let's make good on that promise with a few examples. The last example will, once again, bring us back to principal component analysis.

4.4.1 Compression and noise removal

Let's start by looking a little closer at something we've glossed over in the story so far: the idea of *noise*. We've said that one of the main benefits of the SVD is that it easily allows us to remove noise. If we have a low-rank matrix that contains some data that we're interested in, but through some process like measurement error or floating point computation, a little noise has been added, we saw that the singular values that were zero before will now be small non-zero values. Setting these back to zero removes a large part of the noise.

The idea of noise is an important and very deep subject in machine learning and statistics. If you look up an official definition of noise in data analysis, you may find some assertion that the noise is the "random part" of the data. The pattern is caused by a mechanism, and the noise is a random signal "without cause" that is added on top. The pattern and the noise together make the data we observe.

This is an interesting perspective, but in practice it doesn't quite work like that. What constitutes noise is usually a *subjective* choice. Two people may look at the same data, and consider different parts noise. To explain, imagine that you've decided to weigh yourself regularly and to analyse the resulting data.

If you're looking to lose weight, you're probably only interested in a rough weekly trend. The daily fluctuations won't mean much, but over a whole week, you can tell whether your current diet and exercise regime is working. In this view, the first-order trend is all you're interested in. The rest you can treat as random noise. It's not random in the sense that there is no cause behind it, it's just that we are interested in looking at the data at a more granular level.

If you are interested in something else, you may want to re-

tain more of the data. For instance, you might wonder whether it's true that eating starchy food causes you to retain water, increasing your weight the next day. In that case, you are interested in the difference between successive days, based on what you ate the day before. You may even be interested in the way in which your metabolism changes throughout the day, in which case, you'd want to retain all the fine detail of the data over the course of the day (assuming you weigh yourself often enough).

In short, what you consider noise and what you consider pattern is a subjective choice, depending on what you do want to do with the data.

In some sense this is true of almost anything we call random. In theory, we could predict quite accurately how a flipped coin will land if we had a good idea of the state of the atmosphere around the coin, and the amount of force impacting the coin at the start of the flip. In practice, we usually treat all of this as unknown: we call it noise or randomness, and we say that the coin will land either side with equal probability. It's not random because there's no cause and effect, it's random because we don't care about that part of the process.

> *In machine learning, this problem manifests as over- and underfitting. Overfitting is when we model too much of the noise as pattern and we end up remembering parts of the data that we should be ignoring. Underfitting is when we ignore too much of the noise and apply too simple a model to the data. In machine learning, the test set is the ultimate arbiter of what we should consider pattern and what we should consider noise.*

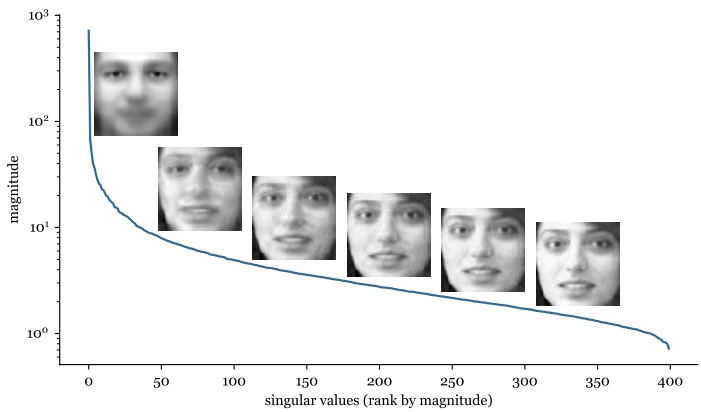
All this is just to make the point that what we filter out as noise is mostly determined by how we want to model the world. If we're interested in losing weight, we will treat the data as roughly linear (on the scale of a week), and ignore any finer detail. We will treat this as random noise when we fit a line through our data, not because it's truly random, but because the causes are too complex to model, and because they are irrelevant to our goal.

The relevance to SVD is that when we set the noise threshold in SVD, we are implicitly making this choice. So far we've

said that there will be an obvious difference between the true singular vectors and those caused by noise, and often this is true, but it would be more accurate to say that we are *choosing* which parts to treat as noise.

If we wanted to, we could ramp up the threshold, and also zero out some of the more substantial singular vectors. The ones that were definitely not introduced by simple measurement error or floating point computations. This is essentially what happens when we compute a truncated SVD. We can either choose the number of singular vectors to retain, or set the noise threshold.

We've seen an example of this already, although we didn't know it at the time. In Chapter 1, we reconstructed our face data step by step, by adding more principal components. The principal component are the singular vectors, so this process is equivalent to taking an SVD decomposition, truncated at k, and looking at the reconstructed data.



A plot of the singular values of the Olivetti faces data, in order of magnitude. The insets show the reconstructions using 10, 75, 140, 205, 270 and 335 singular values and vectors.

We see that at the high end, we get singular values with a magnitude between 0.1 and 1.0. This corresponds to things like camera

noise, compression artifacts and floating point errors in the computation of the SVD itself. Reconstructing the data from only the principal components with singular values above this noise level results in a barely noticeable smoothing of the images but little loss in identifiability.

If we set the noise threshold higher, around 10, we begin to lose some of the detailed facial features but the basic identity of each person is still retained. If we go even higher than that, to around 100, only a handful of principal components remain. We see that the faces are so smoothed out that only basic properties like gender, age and lighting direction are retained.

There may not be many use cases for reconstructing the data from this few components, but there are certainly use cases for analysing the data where we may care about a subject's gender and age without caring much about what their exact face looks like.

4.4.2 **Compressing single images**

Representing images with their principal components in a larger dataset is a kind of image compression "in context". We can work out that age and gender are important attributes because they are important causes of variation in the images in the set. In practice, however, image compression happens on one image at a time, without explicit reference to a larger set of images they belong too.

In this setting, we can also apply the SVD. Instead of flattening the image into a vector, we keep the image as a grid of pixels, and treat that grid as out matrix M. We'll assume we have a grayscale image with pixel values encoded as numbers between 0 (for black) and 1 (for white). What can we expect to happen if we apply an SVD to this matrix?

$$M = U\Sigma V^T$$

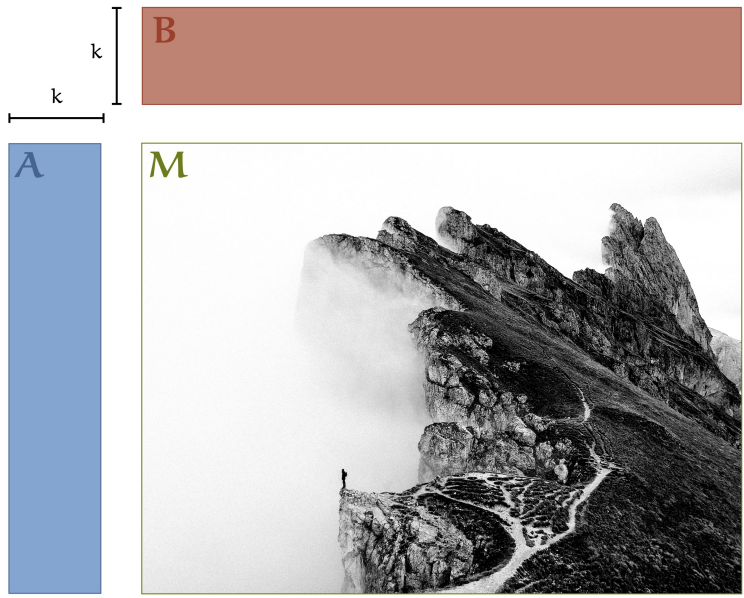
The first thing to note is that we get a perfect reconstruction (at least in theory) if we compute a full SVD. But then the three matrices U, Σ, V contain many more values than the matrix M,

so we're not exactly achieving compression.

What happens if we use a compact SVD? That way we still have all the information, but we don't use as many numbers to represent $\mathbf{M}$ as the full SVD does. In his case, it depends on the image. For some images, the whole thing may be described with only a handful of singular vectors and values. In that case, the compact SVD will be much more efficient. This happens if the image itself is a low-rank matrix.

In most cases, if we want compression, we'll need to use a *truncated* SVD. That is, we will throw away information, but we will hopefully end up with an image that still looks to us like the original.

As we illustrate what this looks like, we can also show a link to the rank decomposition, we introduced earlier. This is a decomposition of the $n \times m$ matrix $\mathbf{M}$ into the product $\mathbf{AB}$ of an $n \times k$ matrix $\mathbf{A}$ and a $k \times m$ matrix $\mathbf{B}$.



Taking a grayscale photograph, and interpreting it as a matrix, we can find a compression by applying a rank decomposition. Photograph by Tom Verdoot.

There are plenty of interesting patterns in the above photograph, but we can also tell immediately that the matrix M is either low-rank or very close to it. On the left there are several columns of pixels that are almost entirely white, save for a gentle gradient into light gray. If just one of these columns is an exact scalar multiple of another, or a linear combination of several others, the matrix is rank deficient. More generally, because these columns are all close to being the same, we can treat them as such to compress them.

Now, how do we find the rank decomposition? It may not surprise you to learn that the SVD can help us here.

If we can find two matrices such that they multiply to form something close to M, and we can keep k small enough, we will have achieved compression. We can easily turn the truncated SVD into an approximate rank decomposition: the matrices U_k and ${V_k}^T$ have the right dimensions already. All we need to do is add the singular values. The simplest thing to do is to take the squares of the first k singular values, arrange the in a diagonal matrix $\Sigma^{\frac{1}{2}}$, so that $\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}} = \Sigma_k$ and write

$$M \approx AB = U_k \Sigma^{\frac{1}{2}} \Sigma^{\frac{1}{2}} V_k^T = U_k \Sigma_k V_k^T$$

with $A = U\Sigma^{\frac{1}{2}}$ and $B = \Sigma^{\frac{1}{2}} V^T$.

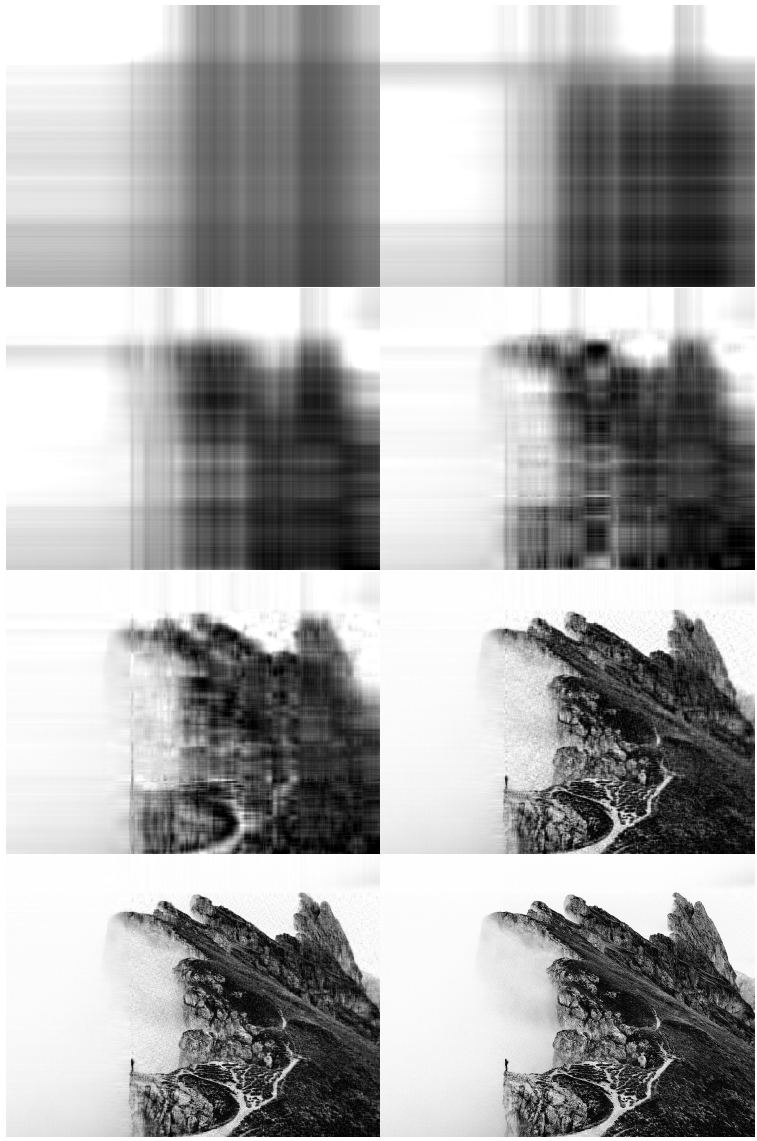
This gives us a quick way to compute a decomposition of an image matrix. Here's what this looks like at different levels of truncation.

To play around with this algorithm in color, see this demo[1] *by Tim Baumann.*

4.4.3 Computing rank decompositions

How good is this as an approximation of M using a rank decomposition? It turns out it's optimal. The **Eckart-Young-Mirsky theorem** tells us that no two rank-k matrices can multiply to produce a matrix that is closer than AB. There are others that are equally close, but none are better.

[1] http://timbaumann.info/svd-image-compression-demo



Reconstructions of our image using 1, 2, 3, 5, 10, 50, 100 and 500 singular values and vectors.

This sounds like another best-approximation theorem. The proof of this fact is pretty straightforward, but it doesn't illustrate how this relates to the best approximation results we've seen already. To make that link clear, we'll look at something a little short of a formal proof, which will hopefully provide more intuition into what is happening when we approximate a matrix this way.

First, let's state the problem formally. We are looking for two matrices $\mathbf{A}$ and $\mathbf{B}$ with sizes $n \times k$ and $k \times m$ so that

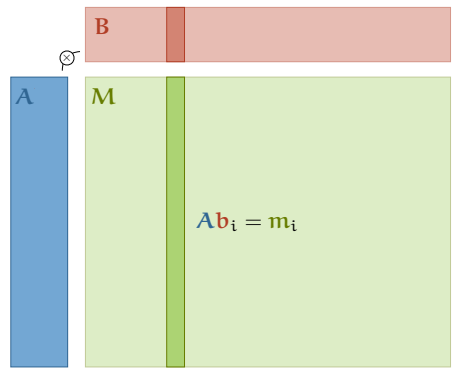
$$\|\mathbf{AB} - \mathbf{M}\|$$

is as small as possible.

> *We're using a matrix norm here called the Frobenius norm. That sounds fancy, but we're just talking about the Euclidean distance between $\mathbf{AB}$ and $\mathbf{M}$ that you would get if you flattened both matrices into vectors.*

This is a *bi-linear* minimization objective. If we fix $\mathbf{A}$, we can use what we already know about linear problems to find the optimal $\mathbf{B}$ and vice versa. To illustrate, let's first assume that the optimal $\mathbf{A}$ is given and we only need to find $\mathbf{B}$.

To apply what we've learned about the pseudo-inverse, we can proceed column by column. Looking at a matrix multiplication, we know that whatever $\mathbf{B}$ we choose, the i-th column of $\mathbf{B}$ multiplies by $\mathbf{A}$ to make the i-th column of $\mathbf{M}$: $\mathbf{A}\mathbf{b}_i = \mathbf{m}_i$.



This is entirely independent of what we choose for the other

columns: if we change something in b_i, only m_i is affected. Thus, we can treat this as a separate optimization problem for each column, and apply the pseudo-inverse to each. The solution is

$$b_i = A^\dagger m_i \,.$$

If we combine the columns b_i back into a matrix, we get

$$B = [\, b_1 \,\ldots\, b_m \,] = \left[\, A^\dagger m_1 \,\ldots\, A^\dagger m_m \,\right] = A^\dagger M \,.$$

In short, solving $AB = M$ for B, we find that $B = A^\dagger M$. We take A to the other side using the pseudo-inverse, analogous to the way we would solve a scalar equation.

If we reverse the setting, and assume that B is given, we can take the transpose of all matrices so that our objective becomes

$$\|B^T A^T - M^T\| \,.$$

We can then follow the same derivation as above, finding that the *rows* of A multiply by B^T to make the rows of M. Ultimately, the optimal choice of A becomes

$$A = MB^\dagger \,.$$

This gives us two equations that should hold at the optimum. We can now show that the SVD solution to the rank decomposition problem we described above satisfies these equations.

First, we take the SVD of M truncated at k:

$$M \approx U\Sigma V^T \,.$$

We've ommitted the subscripts for clarity, but note that U is $n \times k$, Σ is $k \times k$ and V is $m \times k$.

If we define $A = U\Sigma^{\frac{1}{2}}$ and $B = \Sigma^{\frac{1}{2}} V^T$ then they combine into a low rank approximation of M as we already showed. What we can also conclude is that this gives us the SVD decompositions of A and B. All we need is to add a $k \times k$ identity matrix:

$$A = U\Sigma^{\frac{1}{2}} I^\top$$
$$B = I\Sigma^{\frac{1}{2}} V^T \,.$$

This in turn, tells us what the pseudo-inverses of $\mathbf{A}$ and $\mathbf{B}$ are.

$$\begin{aligned}\mathbf{A}^\dagger &= \mathbf{I}\boldsymbol{\Sigma}^{-\frac{1}{2}}\mathbf{U}^\mathsf{T} \\ \mathbf{B}^\dagger &= \mathbf{V}\boldsymbol{\Sigma}^{-\frac{1}{2}}\mathbf{I}^\top .\end{aligned}$$

We can now simply fill in these definitions to show that the required equations hold for this choice of $\mathbf{A}$ and $\mathbf{B}$. For $\mathbf{A}$ we get

$$\begin{aligned}\mathbf{A} &= \mathbf{U}\boldsymbol{\Sigma}^{\frac{1}{2}}\mathbf{I}^\top \\ &= \mathbf{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{-\frac{1}{2}}\mathbf{I}^\top \\ &= \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T}\mathbf{V}\boldsymbol{\Sigma}^{-\frac{1}{2}}\mathbf{I}^\top \\ &= \mathbf{M}\mathbf{B}^\dagger .\end{aligned}$$

And for $\mathbf{B}$

$$\begin{aligned}\mathbf{B} &= \mathbf{I}\boldsymbol{\Sigma}^{\frac{1}{2}}\mathbf{V}^\mathsf{T} \\ &= \mathbf{I}\boldsymbol{\Sigma}^{-\frac{1}{2}}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T} \\ &= \mathbf{I}\boldsymbol{\Sigma}^{-\frac{1}{2}}\mathbf{U}^\mathsf{T}\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathsf{T} \\ &= \mathbf{A}^\dagger\mathbf{M} .\end{aligned}$$

This isn't a complete proof. We have only shown that one necessary condition holds. To complete the proof, we should show that this condition is also sufficient. This is possible, but a bit technical. Instead, we'll wait until we tie this back into principal component analysis, in the next subsection. There, we will see that the results we already know will serve as a proof of the Eckart-Young-Mirsky theorem.

So, in summary, the SVD gives us a very efficient way to approximate matrices by compressed representations. We started with image compression as a use case, because it's a visual example. But there are many other situations where our data forms a matrix, and in any such case, the rank decomposition, computed by SVD, provides us with an efficient mechanism to separate pattern from noise. Either for compression, analysis or for prediction.

4.4.4 Recommendation

One particularly relevant and popular example is that of **recommendation**. This is where we see that the SVD can be a powerful tool for *prediction*.

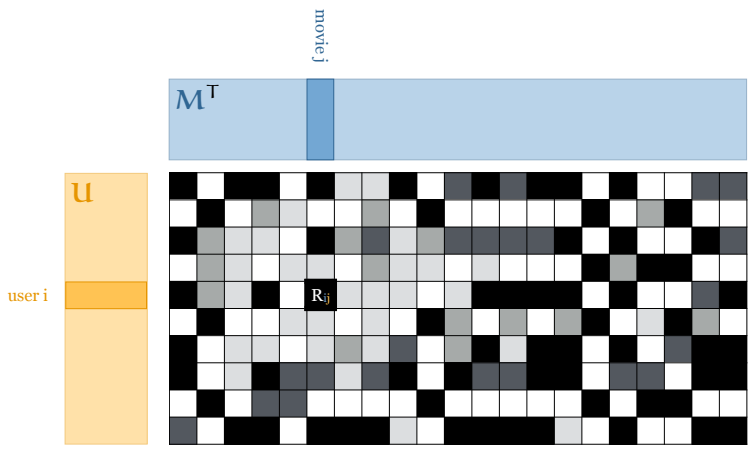
Imagine that you are running a movie streaming service. You have a large collection of movies and a large population of users.

Each user has given you a few ratings for a small subset of the movies. This rating consists of a high positive number if they like the movie, a high negative number if they dislike the movie and something near zero if they were ambivalent.

> *That's not usually how users rate things, but it's the simplest setting for us. We can worry about mapping a real user rating to this scale later.*

If we have n users, and m movies, we can put these ratings in a big $n \times m$ matrix $\mathbf{R}$. The element R_{ij} tells us what user i thought of movie j. For some pairs i, j, we don't have a rating (if we knew the whole matrix, we wouldn't need a recommender). For now, we'll set these elements of the matrix to zero, and see what we can do.

Let's see what happens if we apply a rank decomposition to the matrix $\mathbf{R}$. We'll get one $n \times k$ matrix U, and one $m \times k$ matrix M such that $\mathbf{R} \approx UM^T$.



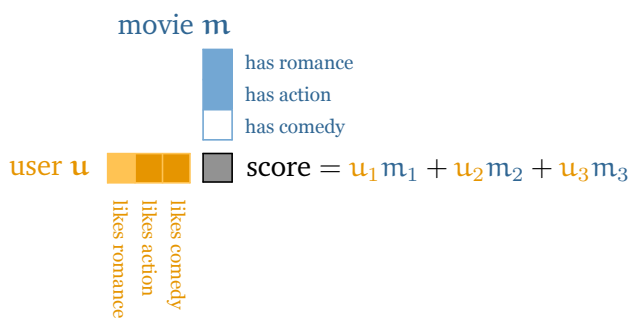
We can think of the i-th row of U as a representation of the i-th user and the j-th row of M as a representation of the j-th movie. Their dot product is the approximation to the rating R_{ij}. The basic idea of this kind of recommendation is that if R_{ij} approximates the known ratings well, then it may approximate the unknown ratings too. We can think of R_{ij} as a *prediction* for whether user i will like movie j.

> *Of course, we're now also explicitly optimizing for the prediction to be* 0 *for these entries, so something slightly counter-intuitive is happening here, but in practice the way the algorithm deviates from these* 0*'s is still predictive. We'll see a fix for this later.*

What might a good solution to this problem look like? Here's one way to think about it. Imagine that we were to fill in the user representation u and the movie representation m by hand, in such a way that their dot product would be positive if the user likes the movie and negative if the user dislikes the movie.

One way to do this is to come up with k features a movie can have, like comedy, drama and romance. You can score the movie on each aspect, giving it for instance, a large negative score if for comedy if the movie is highly un-comedic and a zero for romance if the movie is neither romantic nor unromantic.

We can then score the user in a similar way based on how much they like these aspects. If the used loves comedy, we give them a large score for comedy. If they hate romance, we give them a large negative score for romance, and if they are ambivalent about drama, we give them a zero for drama.

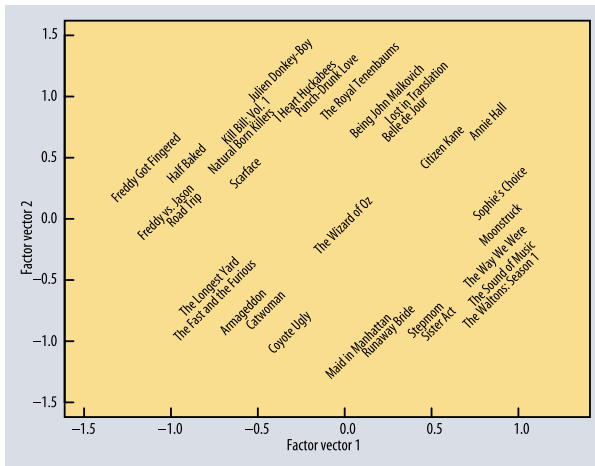


If we have all this information for the user and the movie, then we can see that the dot product is a great predictor for how much the user will like the movie.

Note that if two aspects have the same sign: the user likes romance and the movie contains it, or the user hates romance and the movie is unromantic, both lead to a higher score. Similarly, if two aspects have opposite signs the score is lowered. Finally, note that if one aspect is near zero, that aspect does not affect the score. For instance, if the user is ambivalent about the amount of romance, it doesn't matter for the score whether or not the movie contains a lot or romance or is totally unromantic.

In practice, we don't have the means to collect all this information for our users and movies. Instead, we collect ratings, and we *learn* the representations for our users and movies by decomposing the rating matrix.

The surprising thing is that matrix decomposition still learns intepretable features. Here are the first two dimensions in this learned space from a simple matrix decomposition of a movie rating dataset.



The first two dimensions of a movie representation learned by matrix decomposition, from Koren et al. (2009).

This is an interesting takeaway of the recommender system use case: it shows us how we can interpret the rows of the two matrices we've decomposed $\mathbf{R}$ into. In general, if we decompose $\mathbf{M}$ into $\mathbf{AB}$, then the i-th row of $\mathbf{A}$ is a vector representation of the element represented in the i-th row of $\mathbf{M}$. In movie recommendation, this is the i-th user, and in image compression, it's something more abstract like the i-th row of pixels. Likewise, the j-th column of $\mathbf{B}$ is a learned representation of whatever the j-th column of $\mathbf{M}$ represents.

The hyperparameter k gives us a classic under/overfitting tradeoff. For low k, we have little space to model many aspects of our users and movies. We won't fit much of the pattern in the data, but we can be sure that we are ignoring any noise. If k gets higher, we can model the existing ratings more accurately, but at some point we will also be capturing more and more of the noise. At this point we are fitting the original ratings too accurately, and the representations may no longer generalize to user/movie pairs for which the rating is unknown.

One interesting aspect about the movie recommendation problem is that the matrix that we decompose is *incomplete*. As a first line of attack, we just set the missing values to 0 and applied the matrix decomposition anyway. In practice, this tells the algorithm to optimize for a lot of values that we actually don't want to optimize for. What we can do instead, is compute the minimization objective *only for the known values*.

It's easy to see how this should be done if we open up the minimization objective a little. Focusing on the square of the norm, we see that

$$\|\mathbf{R} - \mathbf{UM}\|^2 = \sum_{i,j} \left(R_{ij} - \mathbf{u}_i{}^T \mathbf{m}_j\right)^2 .$$

If we replace the sum on the right, over all elements i, j by a sum over only the known elements of the matrix $\mathbf{R}$, we get an objective that optimizes only for the known values. The downside is that in this case we can no longer solve the problem analytically (by computing an SVD). We need to use methods like gradient descent or alternating least squares to find a good solution.

But even if the traditional SVD doesn't allow us to compute

the optimal decomposition here, we can take inspiration from this use case, for how to define the SVD if our matrix $\mathbf{M}$ is only partly known. Our original derivation of the SVD started with us multiplying unit vectors by $\mathbf{M}$ and seeing which caused the largest resulting vector. This derivation doesn't translate to the case where we don't know some elements of $\mathbf{M}$, but we now have another perspective to build on.

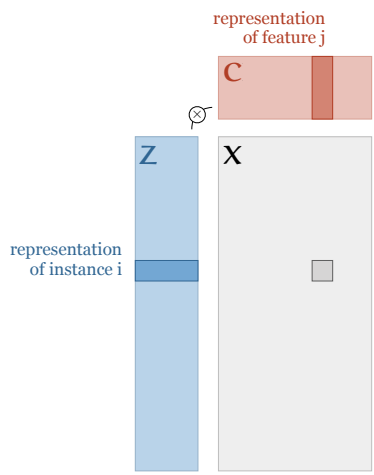
We can define the k-truncated SVD as the solution to the low-rank approximation problem. This problem we *can* solve with a partial matrix, so we can use it to define a partial SVD, even if we'll need to solve with gradient descent.

We can define this more precisely, but, first, we'll look at one more use case, which will tie everything together. It will give us a complete proof of the Eckhart-Young-Mirsky theorem, show us how to do SVD on partially known matrices, and most importantly, bring us back to familiar territory.

4.4.5 PCA as matrix decomposition

It's only fitting that we end back where we started, at principal component analysis. Now that we have added matrix decompositions to our toolbelt, we can view PCA from this perspective.

Let's start with a given PCA, using k components. If we place the principal components of the data in an $n \times k$ matrix $\mathbf{Z}$, with each row representing an instance, and the reconstruction vectors in an $m \times k$ matrix $\mathbf{C}$, then we can illustrate the whole PCA analysis with single matrix multiplication.



Here we see that PCA is nothing but a low-rank approximation of our data matrix $\mathbf{X}$, albeit with the extra constraint that the columns of $\mathbf{Z}$ and $\mathbf{C}$ should be mutually orthogonal unit vectors.

One particularly useful aspect of this perspective is that it shows that we've already proved the Eckart-Young-Mirsky theorem.

In Chapter 2, we showed that PCA provides an optimal reconstruction from a lower dimensional representation, under the constraints that the reconstruction is linear and consists of orthogonal vectors.

First, note that the second constraint does not actually limit the quality of the solution. If we find a solution with non-orthogonal vectors, we can always convert them to an orthogonal set of vectors, keeping the reconstruction the same.

The conclusion is that if we have any matrix $\mathbf{M}$ and ask for a rank k approximation, we can interpret $\mathbf{M}$ as a data matrix and apply PCA. This will give us two matrices $\mathbf{Z}$ and $\mathbf{C}$ which we have already shown in previous chapters to be an optimal approximation to $\mathbf{M}$. As we already derived earlier, if we have a full SVD $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, then the PCA can be computed as $\mathbf{Z} = \mathbf{U}\mathbf{\Sigma}$ and $\mathbf{C} = \mathbf{V}$.

> *Earlier, we distributed $\mathbf{\Sigma}$ equally over the two matrices. In PCA, it's more common to multiply it only by the principal components.*

So, Eckart-Young-Mirsky properly proved. However, we can do a little better. EYM doesn't say that the SVD solution is the *only* optimal low-rank approximation to $\mathbf{M}$. There is a large set of solutions. What makes the SVD an especially useful member of this set is that it satisfies some extra requirements.

We've been here before with the PCA solution. We characterized the PCA objective originally as a reconstruction objective, and we noted that there we many solutions, even if we constrained them to be orthogonal. The PCA solution was special because even if we optimize for k principal components, the PCA solution is the one for which we can isolate the first r principal components and get a PCA solution for the problem with r.

This is what makes the SVD unique among the solutions to the low rank approximation problem. A full SVD can be truncated at every k to give a solution to the rank k decomposition problem. A k-truncated SVD can be truncated at $r < k$, to give a solution to the rank r problem.

There are some trivial variations, but apart from that, the singular values and their vectors are uniquely determined.

What we can now do is take this as a *definition* of the SVD. Then we can generalize to a definition for SVD under missing values. The first singular value σ, and its two singular vectors $\mathbf{u}$, $\mathbf{v}$, we choose in such a way that they minimize the reconstruction objective:

$$\|\mathbf{M} - \mathbf{u}\sigma\mathbf{v}^T\| = \sum_{i,j} \left(M_{ij} - \left(\mathbf{u}\sigma\mathbf{v}^T\right)_{ij}\right)^2$$

subject to the constraint that $\mathbf{u}$ and $\mathbf{v}$ are unit vectors. If some values of $\mathbf{M}$ are unknown, we sum only over the known values.

This is not how we defined first the singular vectors above. Instead, we looked for the unit vector that yielded the largest vector when multiplied by $\mathbf{M}$. This definition doesn't work with missing values, of course, because we can't compute the matrix multiplication unless all values of $\mathbf{M}$ are known. We know that the two definitions are equivalent, but only because we know that the SVD computes the PCA, and the first principal component can be defined this way.

If that feels like going the long way round, here is a more direct proof of the equivalence. We start with the reconstruction objective.

$$\begin{aligned} \underset{\mathbf{u}\ \mathbf{v}\ \sigma}{\text{argmin}} \quad & \|\mathbf{M} - \mathbf{u}\sigma\mathbf{v}^T\|^2 \\ \text{such that} \quad & \|\mathbf{u}\| = \|\mathbf{v}\| = 1\,. \end{aligned}$$

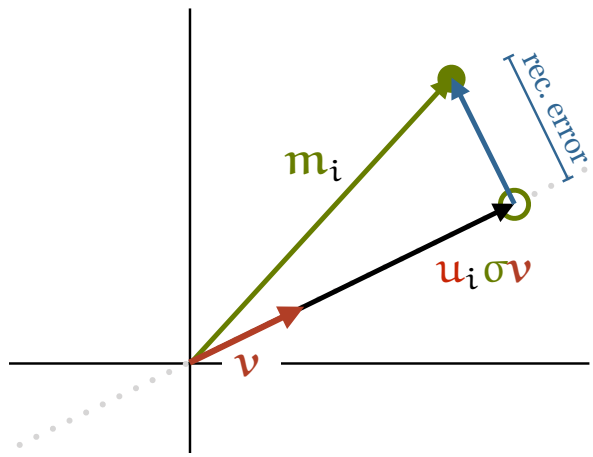
Opening up the norm in the objective, we get

$$\begin{aligned}\|\mathbf{M} - \mathbf{u}\sigma\mathbf{v}^T\|^2 &= \sum_{ij} (M_{ij} - u_i \sigma v_j)^2 \\ &= \sum_i \|\mathbf{m}_i - u_i \sigma \mathbf{v}\|^2\end{aligned}$$

where $\mathbf{m}_i$ is the i-th row of $\mathbf{M}$. If $\mathbf{M}$ is a data matrix, you can think of this as breaking up the reconstruction error over the whole matrix into a sum over the reconstruction error per instance.

Now, we apply the central trick that we first saw in Chapter 2: that reconstruction error is related to variance by the Pythagorean theorem (it always comes back to Pythagoras).

Looking at the following diagram



The vectors $\mathbf{m}_i$ and $u_i \sigma \mathbf{v}$ form a right-handed triangle at the optimimum. However we set $\mathbf{v}$, $u_i \sigma \mathbf{v}$ will be an orthogonal projection of $\mathbf{m}_i$ onto $\mathbf{v}$ at the point where the reconstruction error is minimized.

we see that we have

$$\|\mathbf{m}_i\|^2 = \|\mathbf{m}_i - u_i \sigma \mathbf{v}\|^2 + \|u_i \sigma \mathbf{v}\|^2$$

or, over all i

$$\sum_i \|\mathbf{m}_i\|^2 = \sum_i \|\mathbf{m}_i - u_i \sigma \mathbf{v}\|^2 + \sum_i \|u_i \sigma \mathbf{v}\|^2$$

The left hand side is a constant, so minimizing one term on the right is equivalent to maximizing the other. With that, our objective *becomes*

$$\begin{aligned} &\underset{\mathbf{v}\,\mathbf{u}\,\sigma}{\text{argmax}} \sum_i \|u_i \sigma \mathbf{v}\|^2 \\ &\text{such that } \|\mathbf{u}\| = \|\mathbf{v}\| = 1 \end{aligned}$$

Now, note that $\mathbf{v} = 1$, so that $\|u_i \sigma \mathbf{v}\|^2 = {u_i}^2 \sigma^2$. This turns the objective into $\sum_i {u_i}^2 \sigma^2 = \sigma^2 \sum_i \mathbf{u}_i = \sigma^2 \|\mathbf{u}\|$, so that we get

$$\begin{aligned} &\underset{\mathbf{v}\,\mathbf{u}\,\sigma}{\text{argmax}} \ \sigma^2 \\ &\text{such that } \|\mathbf{u}\| = \|\mathbf{v}\| = 1 \\ &\quad \text{and } \|\mathbf{M}\mathbf{v}\| = \sigma \mathbf{u} \end{aligned}$$

or, equivalently

$$\begin{aligned} &\underset{\mathbf{v}}{\text{argmax}} \ \mathbf{M}\mathbf{v} \\ &\text{such that } \|\mathbf{v}\| = 1\,. \end{aligned}$$

So, the reconstruction definition is equivalent to the "maximum stretch" definition for the first singular vector. The definition is the same for the other singular vectors, with the exception that there are additional constraints, so the same proof will work for the other singular vectors too.

We can now define an SVD for matrices that are only partially known, for instance, a recommendation matrix, or a data matrix with missing values. We simply use the reconstruction definition, and compute the reconstruction only over the known values.

The downside of this approach is that the problem is no longer easy to solve. The fast and robust algorithms that we have to compute SVDs for complete matrices don't work for incomplete ones. Instead we need to look to iterative approaches. There are a few options, but simple gradient descent is the simplest and probably the most popular. If we want a proper SVD, with the singular vectors separated and arranged by magnitude, we'll need to work them out iteratively. We search for the first singular vector, and once we've found it we search for the second singular vector orthogonal to it. This is like the iterative algorithm for PCA we introduced in Chapter 1. It's slow, but we get a proper SVD with separated singular values.

If we don't want to wait that long, we can also solve the reconstruction problem in one go: find two orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ and a diagonal matrix $\mathbf{\Sigma}$ so that their product approximates our matrix, all truncated to k dimensions. The downside, like the combined approach to solving the PCA problem, is that we don't get a neat separation of the singular vectors. If we truncate at k, we can find a solution that is as good as the iterative solution, but the k vectors we find will be a mixture of the k vectors that the iterative algorithm returns.

So that's the singular value decomposition. The workhorse of linear algebra.

In the context of PCA, you can think of the SVD as a way to compute a principal component analysis efficiently, but you can also think of it as an alternative definition. In Chapter 1, we defined the PCA in term of minimizing reconstruction error, in Chapter 2, we defined it in terms of the eigenvectors of the covariance matrix, and here we have defined it a third time, in terms of the singular values of the data matrix. All of these definitions are, of course, equivalent, and all of them have different things to offer.

The main thing the SVD definition has to offer is that it is the preferred way to compute a principal component analysis. If you're willing to take an SVD routine as a given, you now have everything you need to compute the PCA in almost all settings. And in almost all cases, that is what you should do. There are very few situations where you would want to implement your

own SVD algorithm.

Still, can we really claim to have understood SVD, and by extension PCA, if we cannot implement it from scratch? In the next chapter we will take the final step in our journey, and look at some algorithms for computing both eigenvectors and singular vectors.

Chapter 5 · Computing the eigendecomposition and the singular value decomposition

We have looked at what PCA is, at how to understand it, and we have spent a full chapter on developing the tools necessary to prove the spectral theorem, the particular decomposition that makes it all possible.

What we haven't discussed yet, in any detail, is how to *build it*. And how to build it efficiently. We've looked at the singular value decomposition (SVD) in great detail, and with an algorithm for computing the SVD available, computing the PCA is trivial. But that's kicking the can down the road. How then, do we implement the SVD?

In practice, you'll rarely have to build much of PCA from scratch, and if you do, there are better resources than this one to tell you what to pay attention to.

> *For instance,* Matrix Computations *by Golub and Van Loan (2013).*

So why do we care about building a PCA implementation from scratch ourselves? Because building something is one of the best ways of understanding it. There are many things we don't yet understand, or understand well, about PCA, eigenvectors and singular vectors. By looking at some of the different ways you might implement PCA, we can learn a lot more about what it actually does.

So that will be our aim. We won't focus on the most popular algorithms, and we won't bother with all the tricks required to

make the algorithms faster or more robust. We will simply set ourselves the challenge to implement PCA from scratch in a relatively efficient manner, but we will focus on those algorithms that illustrate most clearly what is happening when PCA is computed.

Remember that we've seen one algorithm already in Chapter 1. There, we searched, using projected gradient descent, for the unit vector that gave us the best reconstruction of the data from a single number. This gave us the first principal component, and from that we could search for the second one, and so on.

> *We also briefly mentioned a version of this algorithm for the singular value decomposition in Chapter 4.*

Our job in this chapter is to do a little better. We'll still require iterative algorithms with approximate solutions, but we'll try to get away from approximating the principal components one by one. Finally, we'll do our best to show that our algorithms are guaranteed to converge, and if possible, give an idea of how fast they converge.

5.1 Computing eigenvectors

Our starting point will be the fact that we derived in Chapter 2: the principal components are the eigenvalues of the sample covariance matrix $\mathbf{S}$ of our data. All we need to do is to compute $\mathbf{S}$, and then figure out what its eigenvectors are. Along the way, we'll try to develop a little more insight into what eigenvectors are, and what they tell us about a matrix.

We'll look at three algorithms for computing eigenvectors that each build on one another: **power iteration**, **orthogonal iteration** and **QR iteration**.

Then, we'll switch to the alternative perspective we developed in Chapter 4: that the principal components are the *singular vectors* of the *data* matrix $\mathbf{X}$, and we'll build on our three algorithms for computing eigenvalues to develop three algorithms along the same lines for computing the SVD.

5.1.1 **Power iteration: computing one eigenvector**

Before we get to the business of computing eigenvectors, allow me a little diversion. It will help us to build some more intuition for what eigenvectors are, and this intuition will become the foundation for all the algorithms that follow.

As ancient as it may make the rest of us feel, some people reading this will have become internet users only after Google was invented. If this is you, you will have no memory of how useless search engines were before Google came along. You won't have experienced the watershed that the introduction of Google was. You may not even believe that their almost-complete market dominance can be traced back to one simple idea.

The modern Google engine combines a vast array of methods and philosophies, but the original idea that set it so far apart from the competitors was singular, and simple.

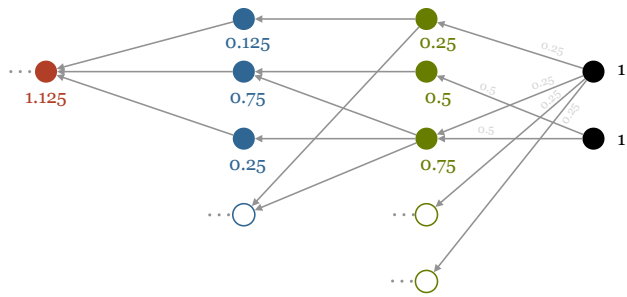
Back then, the main problem facing search engines was the number of webpages trying to game the system. Authors would include fake keywords, large swathes of invisible text, everything they could think of to get as high in the results for as many different search terms as possible. This would inevitably lead to useless, nonsense-filled websites cluttering up search results.

What Google wanted to do was to develop a measurement of *reputation*: a single number that could capture to what extent a website was a respectable source of information playing by the rules, versus a cheap ad-laden swindle, trying to attract clicks. Their basic idea was a *social* one: if somebody, somewhere on the web chooses to link to you, they must trust you, and this should serve as a signifier of your reputation. The more people link to you the better your reputation.

By itself, this notion of reputation is easy enough to game. Just set up a load of websites that all link to each other. But the idea can be applied *recursively*: the better the reputation of the sites that link to you, the more they add to your reputation. And *their* reputation is in turn determined by the reputation of the sites that link to *them* and so on.

Theoretically, we could keep going forever, always avoiding the question of how we define the reputation of a website, by simply deferring the question to the reputation of the websites

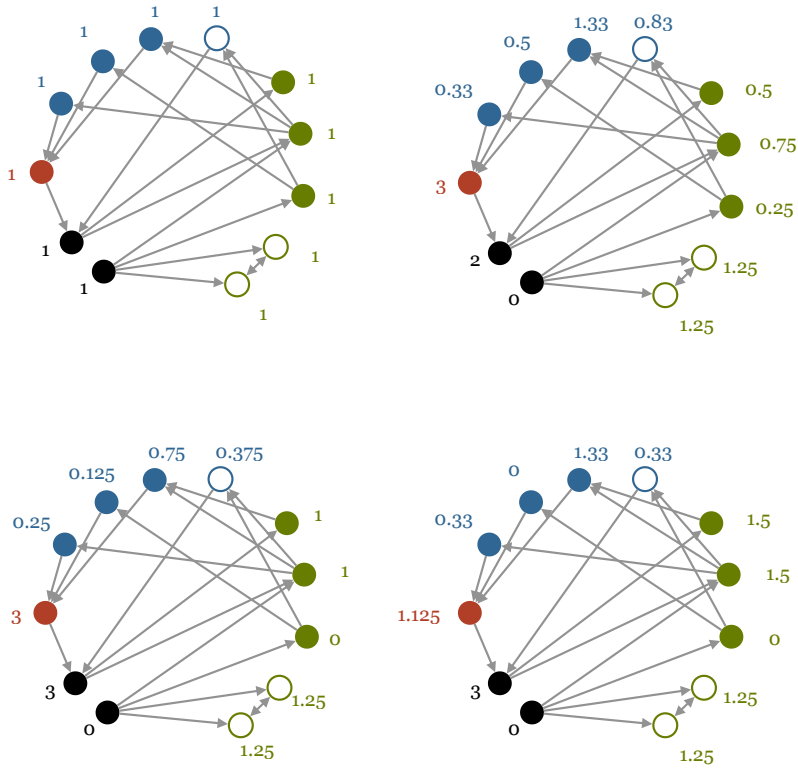
that link to it. In practice, we'll need to stop somewhere, and define the base reputation, independent of who links to it. It turns out however, that if we defer the question for a large number of steps, it doesn't much matter what we do when the recursion stops. We can simply assign all websites we find at that point a reputation of 1, and the main thing that will determine the reputation of the site we started with will be the structure of the graph of links, not the constant reputation we assigned to the websites at which we stopped.



Computing the reputation for a site (the red node on the far left). We follow all incoming links to a certain depth. At that point we assign every node we find a reputation of 1. We can then use this to determine the reputation of the nodes to which they link. We do this by distributing the 1 unit of reputation equally over all outgoing links (including the ones, indicated by open discs, that don't ultimately lead to the site we want to compute the reputation for). This gives us the reputations of the green nodes, from which we can compute the reputations of the blue nodes, which finally gives us the reputation of the red node.

This may seem like a slightly mind-bending idea at first, but that is mostly because we're working backwards. We can define the same idea forwards. Let's say that we start out by giving every website one unit of reputation for free. Then, at every step, every website takes all its current reputation, divides it up equally over

all websites it links to and gives it all away. If the website receives no incoming links from others it is now out of reputation and stays at 0. If, however, it gets some incoming links as well as outgoing links, it also gets some new reputation. We then iterate this process: every step each website divvies up all reputation it has an gives it all away.



The forward computation of the reputation. We start with a reputation of 1 for every site, and have each site distribute all its reputation equally to all sites it links to. Note that after three steps, we have the same reputation for the red site (the leftmost node in each graph).

In this example, the reputations still fluctuate, and we get different values, depending on how long we continue the algorithm. However, as we will show later, for most graphs, this process eventually converges to a stable state. Each site ends up sending out as much reputation as it receives. This is the amount that we ultimately take as the reputation of the website.

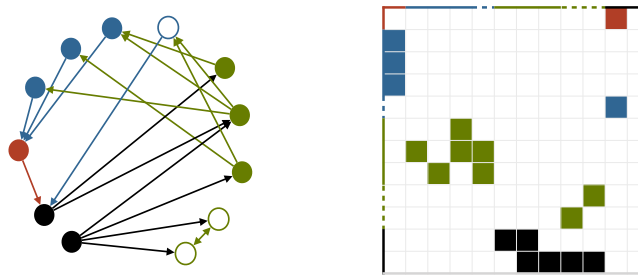
Why is this hard to game? Imagine setting up a bunch of websites that all link to each other. Say you have the resources to host 10 sites. That means that at the start of the process, you get ten units of reputation. The best you can do to maintain this reputation is never to link out to legitimate websites, so no reputation flows out. But then, unless you get people to link to you, there's no reputation flowing in either.

> *The two open green nodes in the graph above are an example of this. They claim a bit of the reputation at the start, but they aren't linked to by the rest of the network, where there is much more reputation flowing around.*

Compare this to a site like Wikipedia, or BBC News. Each of their pages will be linked to by hundreds of sites, each of which will themselves have high reputation. You'd need to set up at least hundreds of thousands of websites to equal that amount of reputation.

What does any of this have to do with eigenvectors? The link becomes clear when we try to figure out what the stable state of this process is. Given a particular graph of websites and who links to whom, what's the ultimate amount of reputation each site ends up with?

First we need to translate the problem setting to linear algebra. Let's say we have a set of n websites. We can represent the directed graph of which site links to which other site (the *web graph*) as a large $n \times n$ matrix $\mathbf{A}$: the *adjacency matrix of the web graph*.



The adjacency matrix for our graph. Colored elements are valued 1, and white elements are valued 0. The colors only indicate the mapping to the graph, the matrix itself is just a square, binary matrix.

We're simplifying the problem by ignoring the fact that there can be many links between two sites and that a website (i.e. a domain) can have many different pages. If you're building a search engine, you can use all this information to make your method more powerful and more complex, but we're only interested in the basic idea here.

We can model the way website i distributes its initial unit of reputation, by starting with a *one-hot vector* indicating that website. This is a vector of length n, for n websites, with zeroes everywhere, except at the index i, corresponding to the website i, where it contains a 1. This represents our starting point for website i, where it has one unit of reputation, and hasn't distributed anything yet.

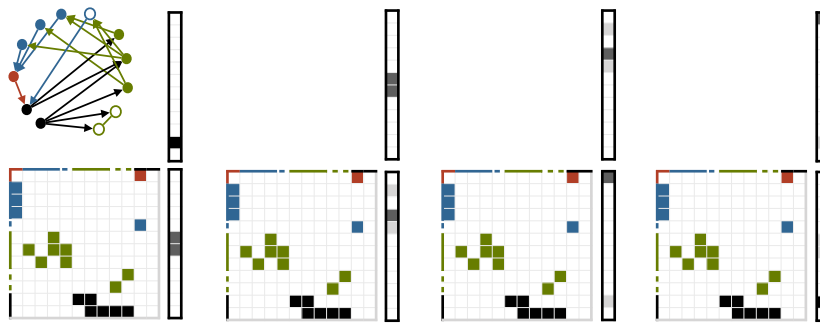
If we multiply this vector by our adjacency matrix, we end up with a new vector that contains 1's for all the websites that i linked to.

However, this doesn't keep the total amount of reputation fixed. To keep the sum total amount of reputation in the system constant, the website should break its one unit of reputation into k equal parts, and give each to one of the k sites it links to. As a simple solution, we *normalize* the vector after the multipli-

cation: we compute the sum of all the elements in the vector and then divide each by the sum.

> *We could also normalize the adjacency matrix* $\mathbf{A}$ *over the rows, but separating the two steps will serve to illustrate an important point later.*

Now, we can just repeat the process. We multiply the vector by $\mathbf{A}$, normalize it, multiply the result by $\mathbf{A}$ again, and so on. Every time we iterate this algorithm, the reputation *diffuses* across the graph a little more.



Multiplying a one-hot vector by the adjacency matrix, and normalizing, allows us to see how one node's unit of reputation diffuses across the network. After four iterations, most of its reputation has circled back, but some of it has spread to other nodes. Eventually this iteration will converge to a stable vector.

What ultimately happens to this sequence of vectors depends on the properties of the graph. However, for most graphs, especially the slightly messy ones like the web graph, the process will converge to a fixed state. For most almost all starting vectors, this multiplication will lead ultimately to a single vector, which when multiplied by the adjacency matrix, and normalized, **remains the same**. It doesn't matter what site we start at, we will always end

up with the same distribution of reputation across the graph.

And that provides the link with eigenvectors. If we call the input input vector $\boldsymbol{v}$ and the factor required to normalize the result of the matrix multiplication $\frac{1}{\lambda}$, then we have reached a stable state when:

$$\frac{1}{\lambda}\boldsymbol{A}\boldsymbol{v} = \boldsymbol{v}$$
$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v}\,.$$

This should look familiar. It's the definition of an *eigenvector*, which we first saw in Chapter 2. If we reach a stable state, we have found a vector for which multiplying it by $\boldsymbol{A}$ changes its length, but not its direction. In short, if we try this iterative approach, and it converges, *we have found an eigenvector of our matrix*. In fact, as we will see in a bit, we will have found the *first* eigenvector. The one with the largest eigenvalue.

> *For me, this is probably the key intuition for why eigenvectors are so important. If we iterate the operation of the matrix, the eigenvectors represent stable states (ignoring changes in magnitude). The key to eigenvectors is that when you want to characterize an operation, you start with its stable states.*

In Google's use case, this tells us that after infinitely many redistributions of reputation, the distribution of reputation over the web stabilizes. This is the basic idea behind *pagerank*, Google's main algorithm (at least in the early days). The websites that ultimately end up with the most reputation, are likely to provide quality content and should end up higher in the ranking of the search results.

You may wonder why all the reputation doesn't flow into a single website. This *could* happen if a website is linked to, but doesn't link anywhere else. The real pagerank algorithm includes a few tricks to avoid such situations, but so long as a website has incoming and outgoing links, it will end up with some proportion of the reputation, but not all of it, the same way a bathtub with the tap running and the plug removed will never be fully empty.

We can extend this principle to a lot of other situations. In

any process where some quantity—like reputation, money or people—is distributed between entities according to fixed proportions, the final, stable state of such a system is an *eigenvector* of the matrix describing how the quantitiy is redistributed.

For instance, if you have a number of cities, and some statistics describing what proportion of people move from every city to every other city for each year, you can work out what populations the cities will stabilize to.

> *This explainer of eigenvalues by Victor Powell and Lewis Lehe provides a nice tool to visualize just this scenario.*

In these examples, the quantity being redistributed was reputation, or population. We can also take this quantity to be *probability*. Instead of counting the total number of people in each city, we can take the probability that a single person moving around randomly, ends up in a given city after some fixed, large number of steps n.

Or, in the Google example, imagine a user starting at a given website i, and clicking a random outgoing link. We don't observe which link they click, so the best we can say is that the user is on one of the sites that i links to, with each getting equal probability. That is, we get a uniform distribution over all websites linked to by i. This is exactly the vector that we get from one iteration of our algorithm: we multiply the one-hot vector for i by $\mathbf{A}$ and normalize.

If the user, wherever they've ended up, clicks another random link, our distribution representing their current position diffuses again. If we had a probability of 0.1 for them being on website j, and j links out to two other websites, each of these gets probability 0.05. All we need to do is multiply the probability vector by $\mathbf{A}$ again, and normalize.

This kind of description of a linear redistribution of quantities is called a *Markov process* or a *Markov chain*. It's a very useful branch of mathematics, but we'll not dig into it any deeper. It has provided us with two things. First, another perspective on eigenvectors as the stable states to which the process of repeated matrix multiplication converges. Second, *a way of computing at*

least one eigenvector.

This bring us back to the business at hand. How do we compute eigenvectors?

Let's follow the iteration approach and analyse it a bit more carefully. When we compute the eigenvectors of the covariance matrix S, we have two advantages over Google. First, the matrix we will deal with is probably much smaller than the adjacency matrix of the web graph. We'll assume that we can easily store it in memory and multiply vectors by it. Second, since it's a covariance matrix, *we know that it's symmetric*. This will make several aspects of our analysis a lot simpler.

The algorithm suggested by the story of Google can be summarized by the following iteration:

$$x \leftarrow \frac{Ax}{\|Ax\|_1}.$$

That is, we multiply A by x and then divide the entries of the resulting vector by its sum, which we've denoted here by the L1-norm $\|\cdot\|_1$.

As it turns out, it doesn't matter much what norm we normalize by. We can just as easily use the Euclidean norm, and normalize the vector to be a unit vector after each step. If we do this, it stops being a probability vector, but the algorithm still yields an eigenvector. Since it makes the analysis a little simpler, we'll switch to that approach, and use the iteration

$$x \leftarrow \frac{Ax}{\|Ax\|}.$$

If we removed the normalization step, and made the iteration $x \leftarrow Ax$, it would be very easy to see that after a number of iterations, say four, the resulting vector x_4 would be $AAAAx_0$, where x_0 is the vector we started with. Put simply, we would have $x_k = A^k x_0$.

Luckily, the normalization step doesn't make things much more difficult than this. Note that the norm of a vector is a *linear quantity*: if we multiply all the elements of the vector by 2, the norm is also multiplied by 2. In short $\|xc\| = \|x\|c$.

Therefore, we can say that

$$x_2 = \frac{Ax_1}{\|Ax_1\|} = \frac{A\frac{Ax_0}{\|Ax_0\|}}{\|A\frac{Ax_0}{\|Ax_0\|}\|} = \frac{AAx_0\ \frac{1}{\|Ax_0\|}}{\|AAx_0\|\frac{1}{\|Ax_0\|}} = \frac{A^2x_0}{\|A^2x_0\|}.$$

Or, more generally, the k-th vector in our iteration is just the vector $A^k x_0$, normalized. We can normalize every iteration, every other iteration or every k iterations. The end result will be the same.

> *That is, unless the values get so big they can no longer be stored accurately in floating point representation. That's why in practice, it pays to normalize every iteration.*

Let's see what we can say about this vector $A^k x_0$.

First, we know from the spectral theorem that if A is $n \times n$ and symmetric, it has n real eigenvalues λ_i—including multiplicities—and n corresponding eigenvectors v_i. We'll assume that the eigenvalues are sorted by magnitude, so that λ_1 is the biggest eigenvalue, and λ_2 the second biggest, and so on.

We also know, from previous chapters, that the eigenvectors form a basis: any vector in $\mathbb{R}^n$ can be written as a linear combination of the eigenvectors. That means that we can write x_0 as $c_1 v_1 + \ \ldots\ + c_n v_n$, for some values $c_1\ \ldots\ c_n$. If we choose x_0 randomly, the probability that any of these are exactly 0 will be vanishingly small.

Now, let's see what we get if we start with x_0 written like this, and compute x_k. First, let's look at the unnormalized vector $A^k x_0$

$$\begin{aligned} A^k x_0 &= A^k(c_1 v_1 + c_2 v_2 + \ \ldots\ + c_n v_n) \\ &= c_1 A^k v_1 + c_2 A^k v_2 + \ \ldots\ + c_n A^k v_n \\ &= c_1 {\lambda_1}^k v_1 + c_2 {\lambda_2}^k v_2 + \ \ldots\ + c_n {\lambda_n}^k v_n \\ &= c_1 {\lambda_1}^k \left(v_1 + \frac{c_2}{c_1}\frac{{\lambda_2}^k}{{\lambda_1}^k} v_2 + \ \ldots\ + \frac{c_n}{c_1}\frac{{\lambda_n}^k}{{\lambda_1}^k} v_n \right). \end{aligned}$$

In the last line, we take the factor $c_1{\lambda_1}^k$ out of the brackets. This leaves the term $\boldsymbol{v}_1$ by itself, and divides the rest of the terms by this factor. Note the factors ${\lambda_i}^k/{\lambda_1}^k$ in all terms except the first. We know that λ_1 is the eigenvalue with the greatest magnitude, so λ_i/λ_1 is always in the interval $[-1, 1]$. This means that its k-th power goes to zero with k. For large enough k, all that remains is

$$\boldsymbol{A}^k\boldsymbol{x}_0 \to c_1{\lambda_1}^k\boldsymbol{v}_1 \,.$$

That is, we converge to the first eigenvector. We can now add the normalization back in, but that doesn't affect the direction of the vector we end up with, only its magnitude.

> *This analysis also tells us the rate of convergence. Every iteration, the second biggest term in our sum decays with a factor of $|\lambda_2/\lambda_1|$. This tells us that we are converging geometrically, and that the speed of convergence is determined by the difference in magnitude between the biggest and the second biggest eigenvalue.*

This is the method of **power iteration**. A very simple way of computing eigenvectors and a very powerful one. All you need is a way to compute matrix/vector products and a way to normalize your vectors. Even on something as large as the web adjacency matrix, this is a feasible computation, (if you store the matrix in the right way).

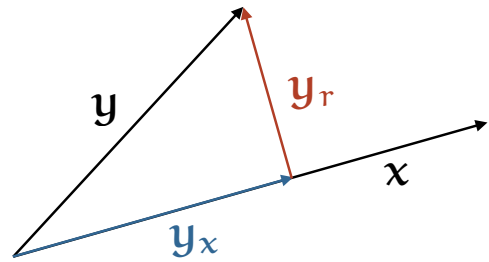
5.1.2 Orthogonal iteration: adding another eigenvector

So, that's a simple way to compute the dominant eigenvector. What do we do if we need more eigenvectors? The simplest trick is to use the fact that **eigenvectors are always orthogonal to one another**.

Let's try a simple approach: we perform the power iteration to make our vector $\boldsymbol{x}$ converge to the dominant eigenvector, but at the same time, we also perform the power iteration on a second vector $\boldsymbol{y}$ orthogonal to $\boldsymbol{x}$. After each iteration, $\boldsymbol{x}$ and $\boldsymbol{y}$ won't necessarily be orthogonal anymore, so we explicitly change $\boldsymbol{y}$ to be orthogonal to $\boldsymbol{x}$ after every step.

How do we force one vector $\boldsymbol{y}$ to be orthogonal to another

x? The simplest way to achieve this, is to first project y onto x, call the result y_x, and then to assume that y consists of two components: the part y_x, that points in the same direction as x and the remainder y_r. This tells us that $y = y_x + y_r$, and thus that $y_r = y - y_x$.



The vector y_r, y minus y projected onto x is called the *rejection* of y from x. We'll use this to build our new power iteration. Here is the new algorithm:

loop:

$x, y \leftarrow Ax, Ay$

$y \leftarrow y - y_x$ make y orthogonal to x

$x, y \leftarrow x/\|x\|,\ y/\|y\|$ normalize both

Note that the operations we apply to x are exactly the same as before. All we've done is to add another vector to the iteration. Our earlier proof that x converges to the dominant eigenvector v_1 still applies to this algorithm. The question is, what does y converge to?

Intuitively, it seems like a good bet that y will convergence to the second most dominant eigenvector. To help us analyze this question, and to build a foundation for later algorithms, we'll first take this algorithm and rewrite it in matrix operations.

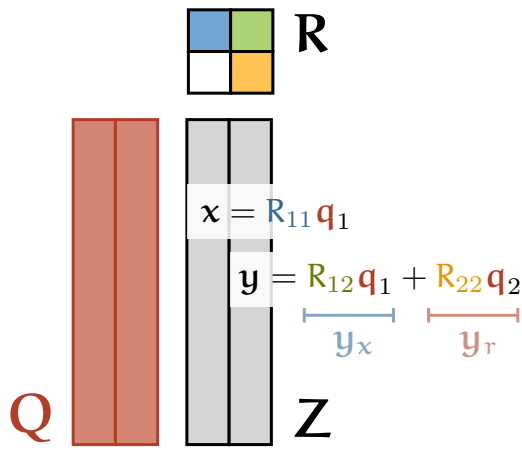
The first line is easy: if we create a new $n \times 2$ matrix $[\ x\ y\ ]$ with vectors x and y as columns, then the matrix multiplica-

tion $\mathbf{A}[\ \mathbf{x}\ \mathbf{y}\ ]$ gives us a new $n \times 2$ matrix whose columns are the new $\mathbf{x}$ and $\mathbf{y}$.

The second and third line, rejection and normalization, we can actually represent in a single matrix operation called a *QR decomposition*. A QR decomposition takes a rectangular matrix $\mathbf{Z}$ and represents it as the product of a rectangular matrix $\mathbf{Q}$ whose columns are all mutually orthogonal unit vectors, and a square matrix $\mathbf{R}$ which is upper triangular (all values below the diagonal are 0).

To apply this to our algorithm, let $\mathbf{Z} \leftarrow \mathbf{A}[\ \mathbf{x}\ \mathbf{y}\ ]$, the $n \times 2$ matrix we create in the first line. If we apply a QR decomposition $\mathbf{Z} = \mathbf{QR}$, then $\mathbf{Q}$ must be an $n \times 2$ matrix with 2 orthogonal unit vectors for columns, and $\mathbf{R}$ must be a 2×2 matrix with the bottom-left element 0.

If we draw a picture, it becomes clear that what lines 2 and 3 of our algorithm are doing is computing a QR decomposition of $\mathbf{Z}$.



The QR decomposition of a matrix with two columns.

Because $\mathbf{R}$ is upper triangular, the first column of $\mathbf{Z} = \mathbf{QR}$ is just the first column of $\mathbf{Q}$ multiplied by some scalar. Because we require the first column of $\mathbf{Q}$ to be a unit vector, this must be the

scalar that normalizes the first column of $\mathbf{Z}$.

The second column of $\mathbf{Z}$ is a linear combination of the two columns of $\mathbf{Q}$. Here the requirement we get from the definition of the QR decomposition is that the second column of $\mathbf{Q}$ is also a unit vector, *and* that it is orthogonal to the first column. As we worked out above, this requires us to subtract the projection of $\mathbf{y}$ onto $\mathbf{x}$ from the original $\mathbf{y}$ before we normalize.

With this perspective, we can rewrite our algorithm as

$\mathbf{Q} \leftarrow [\, \mathbf{x}_0 \; \mathbf{y}_0 \,]$
loop:
  $\mathbf{Z} \leftarrow \mathbf{A}\mathbf{Q}$
  $\mathbf{Q}, \mathbf{R} \leftarrow \mathrm{qr}(\mathbf{Z})$.

Let's see what happens to the second vector $\mathbf{y}$ under our iteration. As before, the key is to note that $\mathbf{A}$ has eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ which form a basis for $\mathbb{R}^n$. That means that for our starting vector $\mathbf{x}_0$, there are scalars $c_1, \ldots, c_n$ so that

$$\mathbf{x}_0 = c_1\mathbf{v}_1 + \ldots + c_n\mathbf{v}_n$$

and similarly, for our starting vector $\mathbf{y}_0$ there are scalars $d_1, \ldots, d_n$ so that:

$$\mathbf{y}_0 = d_1\mathbf{v}_1 + \ldots + d_n\mathbf{v}_n \, .$$

Imagine that we set $\mathbf{x}_0 = \mathbf{v}_1$, and then run the algorithm. In that case the projection and rejection of $\mathbf{y}$ onto and from $\mathbf{x}$ will look very simple. We get

$$\begin{aligned} \mathbf{y}_x &= d_1\mathbf{v}_1 & +0 & \quad + \ \ldots & +0 \\ \mathbf{y}_r &= 0 & +d_2\mathbf{v}_2 & \quad + \ \ldots & + d_n\mathbf{v}_n \, . \end{aligned}$$

This is simplest to see if you think of the eigenvectors as axes in which we express our vector. If we have a vector (x, y, z) ,

then its projection onto the x-axis is simply $(x, 0, 0)$*, and the corresponding rejection is* $(0, y, z)$*. The eigenvectors are a set of mutually orthogonal unit vectors, so we can think of them as just a different coordinate system. In this system,* $\mathbf{y}$ *has coordinates* $(d_1, \ldots, d_n)$*, so its projection onto the first eigenvector* $\boldsymbol{v}_1$ *is* $(d_1, 0, \ldots, 0)$ *and the corresponding rejection is* $(0, d_2, \ldots, d_n)$*.*

Once we've rejected $\boldsymbol{v}_1$, we can see that multiplication by $\boldsymbol{A}$ will never result in a vector with a non-zero component in the $\boldsymbol{v}_1$ direction:

$$\boldsymbol{A}\left(0\boldsymbol{v}_1 + d_2\boldsymbol{v}_2 + \ldots + d_n\boldsymbol{v}_n\right) = 0\lambda_1\boldsymbol{v}_1 + d_2\lambda_2\boldsymbol{v}_2 + \ \ldots\ + d_n\lambda_n\boldsymbol{v}_n\,.$$

Thus, if we start with $\mathbf{x}_0 = \boldsymbol{v}_1$, and $\mathbf{y}_0$ orthogonal to it, we have shown that under the matrix multiplication, $\mathbf{y}$ *stays* orthogonal to $\mathbf{x}$ and we can ignore the rejection step. This means we can use the same derivation as before, except that $d_1 = 0$.

$$\begin{aligned}
\boldsymbol{A}^k\mathbf{y}_0 &= \boldsymbol{A}^k(d_1\boldsymbol{v}_1 + d_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3 + \ldots + c_n\boldsymbol{v}_n) \\
&= 0 + d_2\boldsymbol{A}^k\boldsymbol{v}_2 + d_2\boldsymbol{A}^k\boldsymbol{v}_2 + \ldots + d_n\boldsymbol{A}^k\boldsymbol{v}_n \\
&= d_2{\lambda_2}^k\boldsymbol{v}_2 + d_3{\lambda_3}^k\boldsymbol{v}_3 + \ldots + d_n{\lambda_n}^k\boldsymbol{v}_n \\
&= d_2{\lambda_2}^k\left(\boldsymbol{v}_2 + \frac{d_3}{d_2}\frac{{\lambda_3}^k}{{\lambda_2}^k}\boldsymbol{v}_3 + \ldots + \frac{d_n}{d_2}\frac{{\lambda_n}^k}{{\lambda_2}^k}\boldsymbol{v}_n\right)
\end{aligned}$$

Again, if the eigenvalues are all different, the factors ${\lambda_i}^k/{\lambda_2}^k$ for $i > 2$ all go to zero and we are left with the convergence

$$\mathbf{y} \to d_2\lambda_2\boldsymbol{v}_2\,.$$

Essentially, we are performing the same algorithm as before, but by projecting away from $\boldsymbol{v}_1$, we are ensuring that $\boldsymbol{v}_1$ can't dominate. The next most dominant eigenvector, $\boldsymbol{v}_2$ automatically pops out.

In practice, we don't need to set $\mathbf{x} = \boldsymbol{v}_1$. The iteration on $\mathbf{x}$ is entirely the same as what we saw in the power iteration, so we know that $\mathbf{x}$ will converge to $\boldsymbol{v}_1$. The closer it gets, the more the

v_1 component of y will die out, and the closer the algorithm will behave to what we've described above.

This isn't quite a proof, but it hopefully gives you a sense of how the algorithm behaves when it works as it should. A more rigorous proof will be easier to give when we've extended the algorithm a bit more.

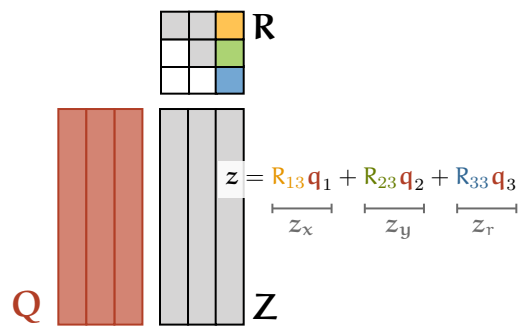
5.1.3 Adding even more eigenvectors

At this point, you can probably guess how to extend this idea to an arbitrary number of eigenvectors. For a third eigenvector, we add another vector z. We multiply it by A as we did with x and y, and then, we project it to be orthogonal to *both* x and y. It turns out this rejection can be computed simply by subtracting from z the projections onto x and onto y:

$$z_r \leftarrow z - z_x - z_y \,.$$

The rest of the algorithm remains as before: we normalize all vectors, and iterate the process.

What we are essentially doing here is computing something called *the Gram-Schmidt process*. For a sequence of vectors, we project the second to be orthogonal to the first. Then we project the third to be orthogonal to both the first and the projected second, and so on until we are out of vectors, normalizing each vector after computing the projections. Extending the logic of the 2 vector case, we see that the Gram-Schmidt process essentially computes a QR decomposition:



The QR decomposition of a matrix with three columns.

The resulting k mutually orthogonal vectors are the columns of $\mathbf{Q}$, and the $k \times k$ matrix $\mathbf{R}$ is the matrix such that $\mathbf{QR}$ results in our original matrix.

In practice, the Gram-Schmidt process isn't the most stable way to compute a QR decomposition. Most modern algorithms use a series of reflections or rotations in place of the projections that the GS process uses.

What we can show, however, is that under the right circumstances, the QR decomposition is *unique*. The proof is a little technical, so we've moved it to the appendix. For now, all we need to know is that if the columns of $\mathbf{Z}$ are linearly independent, it has exactly one QR decomposition for which all the diagonal values of $\mathbf{R}$ are positive (which is what the Gram-Schmidt process provides).

That means it doesn't matter how you compute it, you can analyse it as though you've computed it by the Gram-Schmidt process, which is what we'll do here.

So, to compute the first k eigenvectors, we can use the following algorithm:

$$
\begin{aligned}
&\mathbf{Q} \leftarrow [\mathbf{x}_1 \ \ldots \ \mathbf{x}_k] \\
&\textbf{loop:} \\
&\quad \mathbf{Z} \leftarrow \mathbf{AQ} \\
&\quad \mathbf{Q}, \mathbf{R} \leftarrow \mathrm{qr}(\mathbf{Z})
\end{aligned}
$$

If we use this algorithm to compute the principal components, $\mathbf{A}$ will be our sample covariance, so we know that we have n real eigenvalues, and we can safely set $n = k$. In this setting, it becomes straightforward to show that when the algorithm converges, the columns of $\mathbf{Q}$ converge to the eigenvalues.

> *The proof we used earlier would also work, but it pays to view things from different perspectives.*

To do so, we'll need to introduce a concept called **matrix similarity**. Let $\mathbf{A}$ be any $n \times n$ matrix. $\mathbf{A}$ represents a map on $\mathbb{R}^n$. Now imagine that we have another basis on $\mathbb{R}^n$, represented by

the invertible matrix $\mathbf{P}$: that is, we are representing the same space in a different coordinate system. In this coordinate system, our map can also be represented, but we would need a different matrix. Call this matrix $\mathbf{B}$.

What is the relation between $\mathbf{A}$ and $\mathbf{B}$? We know that the map $\mathbf{A}$ should be equivalent to mapping to the basis $\mathbf{P}$, applying the map $\mathbf{B}$ and mapping back to the standard basis. Composing these operations gives us

$$\mathbf{A} = \mathbf{P}\mathbf{B}\mathbf{P}^{-1}.$$

Any two square matrices $\mathbf{A}$ and $\mathbf{B}$ for which this relation holds—for some $\mathbf{P}$—are said to be *similar*. That is, they represent the same map, just in two different bases.

Similar matrices are a useful concept, because they often share many properties. Most relevant for our case, *similar matrices have the same eigenvalues*. This shouldn't be a big surprise: an eigenvalue is the extent to which an eigenvector is stretched by a map. If we change our coordinate system, the direction of the eigenvector may change, but the amount by which it stretches stays the same.

> *This does require that the origin is in the same place in both coordinate systems (as it is with a change of basis). For instance, losing 10 percent of your weight is the same in pounds or kilograms, but cooling down by 10 percent is a very different proposition in degrees Celsius than it is in kelvin.*

This view of similarity also provides a new perspective on diagonalization. Remember that the spectral theorem tells us that a symmetric matrix $\mathbf{A}$ can be decomposed as

$$\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{\mathsf{T}}$$

with $\mathbf{D}$ diagonal, and $\mathbf{P}$ orthogonal (implying $\mathbf{P}^{\mathsf{T}} = \mathbf{P}^{-1}$). In terms of matrix similarity this simply states that every symmetric matrix is similar to some diagonal matrix. And since we can simply read the eigenvalues off the diagonal in a diagonal matrix, this is useful to know.

We can use the idea of matrix similarity to show that if we set $k = n$ and the orthogonal iteration algorithm converges, it converges to a point where the columns of Q are the eigenvectors.

First, number the sequence of Q-matrices produced by the algorithm Q_1, Q_2, ... At point i in the iteration, we know that: $AQ_{i-1} = Q_i R_i$, since that's the QR decomposition we perform at step i. Because each Q is orthogonal, we can multiply both sides by its transpose, to get

$$A = Q_i R_i {Q_{i-1}}^T.$$

Now, if we assume that the sequence of Q_i's and R_i's converges to some Q and R, in the limit the left and right Q will be the same, giving us

$$A = QRQ^T.$$

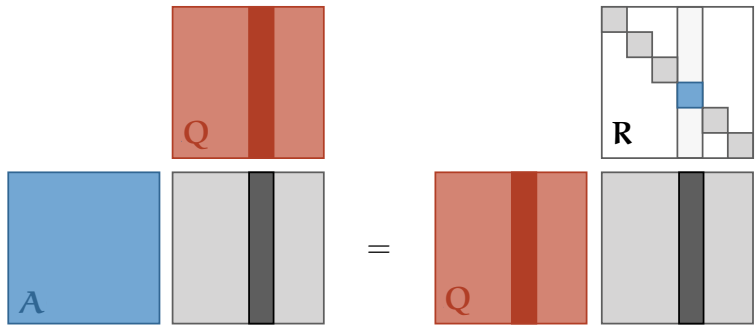
This tells us that A is similar to some triangular matrix R. Triangular matrices, like diagonal ones, have their eigenvalues along the diagonal, so we can read the eigenvalues of A off the diagonal of R.

To show that Q contains the eigen*vectors* of A we need another assumption: that A is symmetric: $A = A^T$. If we know that, then we have $R = Q^T A Q$ from rewriting the similarity relationship, and $Q^T A^T Q = R^T$ from transposing both sides. Putting these together, we get:

$$R = Q^T A Q = Q^T A^T Q = R^T.$$

Which tells us that if A is symmetric, R is symmetric too, so it must be diagonal.

Why does this prove that Q contains the eigenvectors? Rewrite the similarity to $AQ = QR$. This shows that multiplying A by the i-th column of Q, is equivalent to multiplying it by the i-th diagonal element of R. This is the i-th eigenvalue, so the corresponding column of Q must be the corresponding eigenvector.



5.1.4 QR iteration

Our final eigenvector algorithm is *QR iteration*. It looks very similar to orthogonal iteration, but works slightly differently. We'll look at the algorithm first and then dig into how it works, and how it relates to the orthogonal iteration.

Here is the algorithm:

$$
\begin{aligned}
&\mathbf{Z} \leftarrow \mathbf{A} \\
&\textbf{loop:} \\
&\quad \mathbf{Q}, \mathbf{R} \leftarrow \mathrm{qr}(\mathbf{Z}) \\
&\quad \mathbf{Z} \leftarrow \mathbf{R}\mathbf{Q}
\end{aligned}
$$

Note the difference in how $\mathbf{A}$ is used. In the orthogonal iteration, we used $\mathbf{A}$ in every iteration to multiply by. Here, it is only used at the very start. We compute its QR decomposition, multiply $\mathbf{Q}$ and $\mathbf{R}$ in *reverse order*, and iterate.

> *This only works if both* $\mathbf{Q}$ *and* $\mathbf{R}$ *are* $n \times n$. *With the orthogonal iteration, we could choose how many eigenvectors we wanted to compute. With the QR iteration, we are always computing the full set.*

The key idea of this algorithm is that each new matrix $\mathbf{Z}$ is *similar* to the previous one. We can easily show this if we number the se-

quence $\mathbf{Z}_0, \mathbf{Z}_1, \ldots$ and similarly for the $\mathbf{Q}$'s and $\mathbf{R}$'s. We then have

$$\begin{aligned}
\mathbf{Z}_i &= \mathbf{R}_i\mathbf{Q}_i && \text{line 4 in iteration } i \\
&= \mathbf{Q}_i{}^T\mathbf{Q}_i\mathbf{R}_i\mathbf{Q}_i && \text{because } \mathbf{Q}_i{}^T\mathbf{Q}_i = \mathbf{I} \\
&= \mathbf{Q}_i{}^T\mathbf{Z}_{i-1}\mathbf{Q}_i && \text{line 3 in iteration } i-1\,.
\end{aligned}$$

Note the key principle: for every $\mathbf{Q}$ and $\mathbf{R}$ in the sequence, $\mathbf{QR}$ is the previous $\mathbf{Z}$ and $\mathbf{RQ}$ is the next $\mathbf{Z}$.

This tells us that the sequence of $\mathbf{Z}$'s we generate are all similar to one another, including to the first, which is equal to $\mathbf{A}$. If one of them happens to be triangular or diagonal, we can simply read the eigenvalues of $\mathbf{A}$ off the diagonal.

To show that we eventually get such a matrix, we can show that the sequences computed by the QR iteration and the orthogonal iteration are related in a very precise way.

First, let $\mathbf{Q}'_1, \mathbf{Q}'_2, \ldots$ and $\mathbf{R}'_1, \mathbf{R}'_2, \ldots$ be the sequences computed by the *orthogonal* algorithm.

> *Note the prime to indicate that these come from the orthogonal algorithm, not the QR algorithm.*

We know that under the right conditions, the orthogonal algorithm converges to the diagonalization $\mathbf{A} = \mathbf{Q}'\mathbf{D}\mathbf{Q}'^T$, or equivalently $\mathbf{D} = \mathbf{Q}'^T\mathbf{A}\mathbf{Q}'$.

Now, for every step in the sequence of the orthogonal algorithm, we will define a new matrix called the matrix $\mathbf{D}_i$. Its definition is

$$\mathbf{D}_i = \mathbf{Q}'_i{}^T\mathbf{A}\mathbf{Q}'_i\,.$$

Note that this is a different sequence from the intermediate values $\mathbf{R}'_i$ computed by the orthogonal algorithm. There, we had $\mathbf{A} = \mathbf{Q}'_i\mathbf{R}'_i\mathbf{Q}'_{i-1}{}^T$ or equivalently

$$\mathbf{R}'_i = \mathbf{Q}'_i{}^T\mathbf{A}\mathbf{Q}'_{i-1}\,.$$

$\mathbf{R}'_i$ are the values that the orthogonal algorithm computes. The sequence of $\mathbf{D}_i$s is simply a sequence of new matrices we now

define. We know that the sequences converge to the same point, as the difference between $\mathbf{Q}'_{i-1}$ and $\mathbf{Q}'_i$ vanishes, but early on, $\mathbf{R}'_i$ may be very different from $\mathbf{D}_i$.

Note also that $\mathbf{D}_i$ converges to a diagonal matrix (if $\mathbf{A}$ is symmetric), but the intermediate values won't necessarily be diagonal.

Let's look at the sequence $\mathbf{D}_i$ around time i. For the step prior to that, $i-1$, we know that

$$\begin{aligned}\mathbf{D}_{i-1} &= {\mathbf{Q}'_{i-1}}^{\mathsf{T}}\mathbf{A}\mathbf{Q}'_{i-1} && \text{by the definition of } \mathbf{D}_{i-1}\\ &= \mathbf{Q}'_{i-1}\mathbf{Q}'_i\mathbf{R}'_i && \text{since } \mathbf{A}\mathbf{Q}'_{i-1} \text{ is QR'ed in step } i.\end{aligned}$$

Then, at step i, after the QR decomposition, $\mathbf{R}'_i = {\mathbf{Q}'_i}^{\mathsf{T}}\mathbf{A}\mathbf{Q}'_{i-1}$. We can use this to write:

$$\begin{aligned}\mathbf{D}_i &= {\mathbf{Q}'_i}^{\mathsf{T}}\mathbf{A}\mathbf{Q}'_i\\ &= {\mathbf{Q}'_i}^{\mathsf{T}}\mathbf{A}{\mathbf{Q}'_{i-1}}^{\mathsf{T}}\mathbf{Q}'_{i-1}\mathbf{Q}_i && \text{because } {\mathbf{Q}'_{i-1}}^{\mathsf{T}}\mathbf{Q}'_{i-1} = \mathbf{I}\\ &= \mathbf{R}'_i\mathbf{Q}'_{i-1}\mathbf{Q}'_i && \text{see above.}\end{aligned}$$

So, putting these together, we get

$$\begin{aligned}\mathbf{D}_{i-1} &= \mathbf{Q}'_{i-1}\mathbf{Q}'_i\mathbf{R}_i\\ \mathbf{D}_i &= \mathbf{R}_i\mathbf{Q}'_{i-1}\mathbf{Q}'_i\,.\end{aligned}$$

Note that the factor in green is the product of two orthogonal matrices, so itself an orthogonal matrix. This means that the first line represents a QR decomposition, with $\mathbf{Q} = \mathbf{Q}'_{i-1}\mathbf{Q}'_i$.

In short, if we are given $\mathbf{D}_{i-1}$, we can compute $\mathbf{D}_i$ simply by applying a QR decomposition, and multiplying $\mathbf{Q}$ and $\mathbf{R}$ in reverse order. This is precisely what the QR algorithm does.

In practice, the QR decomposition can be expensive to compute. There are modern versions of this algorithm that only perform the QR step implicitly, to speed up the computation.

So, in the limit, we know that $\mathbf{Z}$ converges to a matrix, which has the eigenvalues along the diagonal, and 0 everywhere else. What about the eigenvectors? You'd be forgiven for guessing that once the algorithm has converged $\mathbf{Q}$ contains these. That isn't the case, however. For one thing, at converge, $\mathbf{Z}$ is diagonal, so its QR decomposition is just the identity matrix times itself.

Note what we showed earlier: that the sequence of $\mathbf{Z}$'s computed by the algorithm are all similar to one another. Making this explicit, we get, at iteration i

$$\begin{aligned}\mathbf{D}_i = \mathbf{Z}_i &= \mathbf{Q}_i{}^T \mathbf{Z}_{i-1} \mathbf{Q}_i = \mathbf{Q}_i{}^T \mathbf{Q}_{i-1}{}^T \mathbf{Z}_{i-2} \mathbf{Q}_{i-1} \mathbf{Q}_i = \dots \\ &= \mathbf{Q}_i^{\Pi}{}^T \mathbf{A} \mathbf{Q}_i^{\Pi}\end{aligned}$$

where $\mathbf{Q}_i^{\Pi}$ is the product of all $\mathbf{Q}$ matrices computed so far. (Note that these are the $\mathbf{Q}$s from the QR algorithm not from the orthogonal iteration.)

If the algorithm has converged to our satisfaction, $\mathbf{D}_i$ is diagonal and we get the required diagonalization

$$\mathbf{Q}_i^{\Pi} \mathbf{D}_i \mathbf{Q}_i^{\Pi} T = \mathbf{A}.$$

The takeaway is that for this algorithm, if all you're interested in is the eigenvalues, you can run the stripped down version we presented above. If you also want the eigenvectors, you'll need to keep a running product of all the $\mathbf{Q}$s you've encountered.

That concludes our three methods for computing eigenvectors of a symmetric matrix. The power iteration is a simple, and highly scalable method to find the dominant eigenvector. The orthogonal iteration is an extension we can use to add additional eigenvectors. Finally, the QR algorithm is a superficially different algorithm, that turns out to compute very a similar sequence of orthonormal bases to the orthogonal iteration.

5.2 Computing the SVD

In the previous Chapter 4 , we did a deep dive into the singular value decomposition (SVD). The main takeaway was that this is a great way to compute the PCA, but more than that, a very versatile operation for dealing with matrices that represent any kind of data.

So, a fitting end to the series would be one or more algorithms for computing the singular value decomposition. Pleasingly, each of the three algorithms we saw above can be adapted to provide us with the singular vectors instead of the eigenvectors. Let's start with the simplest.

5.2.1 Power iteration for the SVD

When we compute the eigenvectors to compute a PCA, we apply the eigenvector algorithm to the matrix $\mathbf{S}$. Normally, we estimate $\mathbf{S}$ by computing $\mathbf{X}^T\mathbf{X}$ and dividing by the number of instances in our dataset. This division affects the eigenvalues, but not the eigenvectors, so we'll ignore that for now. Instead, we'll ask what it means to compute the eigenvectors of the matrix $\mathbf{X}^T\mathbf{X}$.

As we saw in the previous chapter, the eigenvectors of $\mathbf{X}^T\mathbf{X}$ are the *singular* vectors of $\mathbf{X}$. This immediately gives us an algorithm for computing singular vectors: take any rectangular matrix $\mathbf{M}$, compute $\mathbf{M}^T\mathbf{M}$ and apply the eigenvector algorithms we developed above.

We can fill in this $\mathbf{M}^T\mathbf{M}$ and see what it tells us about the algorithm. Let's start with the power iteration. When we apply this to $\mathbf{M}^T\mathbf{M}$, we get

$$
\begin{aligned}
&\mathbf{x} \leftarrow \mathbf{x}_0 \\
&\textbf{loop}: \\
&\quad \mathbf{x} \leftarrow \mathbf{M}^T\mathbf{M}\mathbf{x} \\
&\quad \mathbf{x} \leftarrow \mathbf{x}/\|\mathbf{x}\| \, .
\end{aligned}
$$

If we now separate the two matrix multiplications, by $\mathbf{M}$ and then by $\mathbf{M}^T$ in two steps, we get

$$
\begin{aligned}
&\mathbf{x} \leftarrow \mathbf{x}_0 \\
&\textbf{loop}: \\
&\quad \mathbf{y} \leftarrow \mathbf{M}\mathbf{x} \\
&\quad \mathbf{x} \leftarrow \mathbf{M}^T\mathbf{y} \\
&\quad \mathbf{x} \leftarrow \mathbf{x}/\|\mathbf{x}\| \, .
\end{aligned}
$$

This is the same algorithm as before, just with the matrix multiplication separated in two steps. Next, we will add another nor-

malization step. This changes the algorithm, but we will show that the effect is negligible.

$$
\begin{aligned}
&\mathbf{x} \leftarrow \mathbf{x}_0 \\
&\textbf{loop}: \\
&\quad \mathbf{y} \leftarrow \mathbf{M}\mathbf{x} \\
&\quad \mathbf{y} \leftarrow \mathbf{y}/\|\mathbf{y}\| \\
&\quad \mathbf{x} \leftarrow \mathbf{M}^\mathsf{T}\mathbf{y} \\
&\quad \mathbf{x} \leftarrow \mathbf{x}/\|\mathbf{x}\| \,.
\end{aligned}
$$

This is our power iteration algorithm for the singular vectors of $\mathbf{M}$. Compare it to the power iteration for the eigenvectors. There, $\mathbf{A}$ was a *map* (a square matrix): the input and output space of its transformation were the same. We simply apply the transformation, normalize and repeat. With $\mathbf{M}$, the input and output are different: as a result, we map a vector $\mathbf{x}$ in the input space to a vector $\mathbf{y}$ in the output space, normalize, and then map back again by $\mathbf{M}^\mathsf{T}$.

The only real change we've made from the power iteration on $\mathbf{M}^\mathsf{T}\mathbf{M}$ is the extra normalization on $\mathbf{y}$. As we did before, we can look at the result $\mathbf{x}_2$ after two normalizations, and separate the normalizations and matrix multiplications.

$$
\begin{aligned}
\mathbf{x}_1 &= \frac{\mathbf{M}^\mathsf{T}\mathbf{y}}{\|\mathbf{M}^\mathsf{T}\mathbf{y}\|} = \frac{\mathbf{M}^\mathsf{T}\frac{\mathbf{M}\mathbf{x}_0}{\|\mathbf{M}\mathbf{x}_0\|}}{\|\mathbf{M}^\mathsf{T}\frac{\mathbf{M}\mathbf{x}_0}{\|\mathbf{M}\mathbf{x}_0\|}\|} \\
&= \frac{\mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{x}_0}{\|\mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{x}_0\|}\frac{\frac{1}{\|\mathbf{M}\mathbf{x}_0\|}}{\frac{1}{\|\mathbf{M}\mathbf{x}_0\|}} = \frac{\mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{x}_0}{\|\mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{x}_0\|}
\end{aligned}
$$

Put simply, the extra normalization step doesn't change what the algorithm does: we are still computing the normalized result of $\mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{x}$. All our previous proofs about the algorithm still hold. We are computing the first eigenvector of $\mathbf{M}^\mathsf{T}\mathbf{M}$, and therefore the first (right) singular vector of $\mathbf{M}$.

Where exactly are the singular vectors and values in this algorithm? Recall the definition: if $\boldsymbol{v}$ is a right singular vector of

M and u its corresponding left singular vector, with σ the corresponding singular value, then

$$Mv = \sigma u\,.$$

So, if x has converged to a right singular vector of M, then Mx, after normalization, is the corresponding left singular vector. This shows that the algorithm actually gives us both singular vectors. The corresponding singular *value* is $\|Mx\|$.

5.2.2 Orthogonal iteration for the SVD

The extension to orthogonal iteration follows very straighforwardly. As before, our intuition is that we simply take a second vector x' along for the iteration, and that for the second vector—in addition to normalizing it every iteration—we make it orthogonal to x.

> *We're slightly shuffling our variables here. x' is the vector we previously called y when we extended the power iteration to multiple eigenvectors. y is now the intermediate variable we've introduced that results from multiplication by M.*

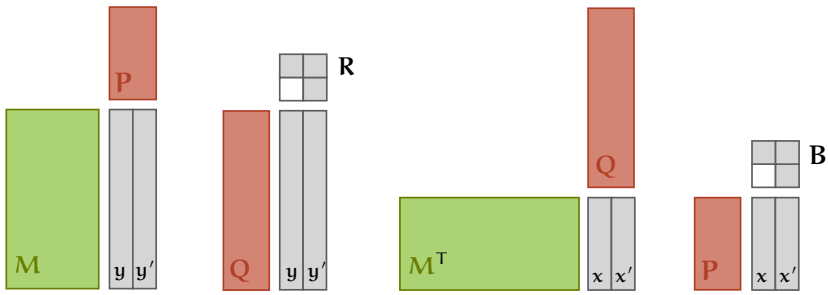
To follow the spirit of the power iteration algorithm for the SVD, we do this twice: we multiply x and x' by M, resulting in y and y'. We project y' away from y, so that they are orthogonal to each other, and normalize both. Then, we multiply both by M^T, and again project and normalize.

The main conclusion we came to before was that we were in effect computing a QR decomposition of the matrix $A\,[\,x\;x'\,]$.

For $M\,[\,x\;x'\,]$, our logic holds the same as before: we get two orthogonal unit vectors, which can serve as the columns of Q. To transform these by a 2×2 matrix R so that $M\,[\,x\;x'\,] = QR$, the first column of R only needs to stretch x uniformly (it only needs a value on the diagonal). The second should express Mx' as two components. One component in the direction of Mx, which gives us a scalar on the first row, and one component orthogonal to Mx which gives us a scalar on the second row. Therefore, R is upper triangular.

When we apply the second step of the iteration, we start with two orthogonal unit vectors y, y' and we compute $M^T [y\ y']$, project and normalize. By the same logic as before, we can interpret this as another QR decomposition. For this one, we'll call the orthogonal matrix P and the upper triangular one B

$$M^T [y\ y'] = PB$$



The two-column orthogonal algorithm for the SVD in four steps.

We can now, as before, add a third vector x'', a fourth, a fifth, and so on. If we make each orthogonal to all prior vectors, we end up with a matrix Q with orthogonal columns (or rows). The triangular structure of the matrix R (which reverses the orthogonalization and normalization) is explained by the fact that the i-th vector has i components: $i-1$ to describe the projections onto the previous vectors, and one to scale the remainder to a unit vector.

Putting all of this together, we get the following algorithm to compute the first k singular vectors:

$$P = \left[x_0^1 \ \ldots \ x_0^k \right]$$

loop:

$$Y \leftarrow MP$$

$$Q, R \leftarrow \mathrm{qr}(Y)$$

$$X \leftarrow M^T Q$$

$$P, B \leftarrow \mathrm{qr}(X)$$

We can also think of the second QR decomposition as an LQ decomposition *of* $\mathbf{Q}^T\mathbf{M}$. *This follows directly from transposing both sides (The LQ decomposition gives us a lower triangular matrix and and orthogonal one). Seeing it as a QR decomposition makes more sense in the way we've explained it, but if you see this algorithm explained elsewhere, it may be with alternating LQ and QR decompositions.*

We know that for the first vector of $\mathbf{P}$, this algorithm behaves the same as the orthogonal algorithm for eigenvectors. It will converge to the first eigenvector of $\mathbf{M}^T\mathbf{M}$, and therefore to the first right singular vector of $\mathbf{M}$.

What we haven't shown yet, is that this also holds true for the other vectors in $\mathbf{P}$, or for that matter, where we can find the singular values, and the left singular vectors.

In our analysis of the orthogonal algorithm for the eigenvectors, we used a simple trick: *express the vectors we're iterating within the eigenbasis of the matrix*. This allowed us to see very neatly what happens as the iteration of the algorithm converges.

In the case of the SVD, we get *two* bases for an $n \times m$ matrix $\mathbf{M}$. One $m \times m$ matrix $\mathbf{V}$ with columns $\mathbf{v}_1 \dots \mathbf{v}_m$ that spans the input space of $\mathbf{M}$ and one $n \times n$ matrix $\mathbf{U}$ with columns $\mathbf{u}_1 \dots \mathbf{u}_n$ which spans the output space of $\mathbf{M}$ We'll call these the right and left *singular bases* of $\mathbf{M}$ respectively.

By definition, for each of these there is a singular value σ_i so that $\mathbf{M}\mathbf{v}_i = \mathbf{u}_i\sigma_i$.

The key insight here, is that when we have the singular bases, we can express the operation of $\mathbf{M}$ *as a mapping between them*. Each right singular basis vector is mapped onto the corresponding left singular basis vector, independent of the others.

To put this more precisely: if we express the input vector $\mathbf{x}$ in the right singular basis $\mathbf{V}$, we get some an expression like

$$\mathbf{x} = c_1\mathbf{v}_1 + \ \dots\ + c_m\mathbf{v}_k\,.$$

For every right singular vector i, we get some component c_i expressing how much of $\mathbf{x}$ projects onto $\mathbf{v}_i$. If we then multiply

$\mathbf{y} = \mathbf{M}\mathbf{x}$, we can express $\mathbf{y}$ in the left singular basis as:

$$\mathbf{y} = d_1\mathbf{u}_1 + \ldots + d_n\mathbf{u}_k\,.$$

The key property of these two bases is that the vector $c_i\boldsymbol{v}_i$ is mapped to the vector $d_i\mathbf{u}_i$, *independently of the other components*. We don't need to know the other values c_j, the i-th component d_i is completely determined only by c_i.

You may note that these sums don't have the same number of terms. What happens, for instance, if $n > m$? In that case the components d_i for $i > m$ will be zero. Effectively, both expressions will have only m terms.

> *If $m > n$, it's possible that the first vector into the algorithm requires all n terms, but after one iteration, both $\mathbf{x}$ and $\mathbf{y}$ can be represented as a linear combination of the first rank $(\mathbf{M})$ right and left singular vectors respectively.*

For what's coming up, we will need to reverse this picture as well. Note first that $\mathbf{M}^T = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^T$ (by simply transposing both sides of the SVD decomposition). This is *also* an SVD, but of $\mathbf{M}^T$ and with right singular vectors $\mathbf{U}^T$ and left singular vectors $\mathbf{V}^T$. By definition of the SVD, we see that this implies that $\mathbf{M}^T\mathbf{u}_i = \boldsymbol{v}_i\sigma_i$. That is, when we transform back from $\mathbf{y}$ to $\mathbf{x}$, the singular vectors are also mapped onto one another.

> *Note that this reverse transformation from $\mathbf{x} = \mathbf{M}^T\mathbf{y}$ is by no means the inverse of $\mathbf{y} = \mathbf{M}\mathbf{x}$, so the fact that they have the same sets of singular vectors is not as obvious as it perhaps sounds.*

Now, let's look at the algorithm. At any point in the iteration, we can express the first vector of $\mathbf{P}$ in the right singular basis of $\mathbf{M}$. For some values $c_1, \ldots, c_m$, we have, for $k = \operatorname{rank}'\mathbf{M}$:

$$\mathbf{p}_1 = c_1\boldsymbol{v}_1 + \ldots + c_m\boldsymbol{v}_k\,.$$

We know that $\mathbf{p}_1$ converges to $\boldsymbol{v}_1$. That means we will get arbitrarily close to the situation where

$$\mathbf{p}_1 = \boldsymbol{v}_1 + 0\boldsymbol{v}_2 + \ldots + 0\boldsymbol{v}_k\,.$$

At that point, when we compute $Y = MP$, in the first line of our iteration, we will get, for the first column y_1 of Y

$$y_1 = Mv_1 = u_1\sigma_1 .$$

In other words, since the input vector is one of the right singular vectors, the output vector is the corresponding left singular vector times the corresponding singular value.

Since this is the first column, the QR decomposition that we then apply in the second line, only serves to normalize $u_1\sigma_1$. Since u_1 is a unit vector by definition, the first column of Q becomes u_1, and the element R_{11} (by which we multiply u_1 to recover y_1) becomes σ_1.

Next comes the multiplication $X = M^T Q$. Focusing only on the first column for now, this is $x_1 = M^T u_1$. From the SVD of the transpose above we see that $M^T u_1 = v_1\sigma_1$. As before, u_1 is a right singular vector of M^T, so the result is the corresponding left singular vector v_1, times a scalar, which is removed in the QR decomposition that follows.

This tells us what we already knew: that v_1 as the first column of P provides a fixed point for our algorithm. Whatever the other columns, this column stays the same under the iteration. We can now look at what happens to the second column of P when the first converges to v_1. First, we express it in the right singular basis of M:

$$p_2 = c_1v_1 + c_2v_2 + \ \ldots\ + c_mv_k .$$

In the first line of the iteration, we multiply M by P. This gives us:

$$\begin{aligned} y_2 = Mp_2 &= c_1Mv_1 + c_2Mv_2 + \ldots + c_kMv_k \\ &= c_1\sigma_1u_1 + c_2\sigma_2u_2 + \ldots + c_k\sigma_ku_k . \end{aligned}$$

Next, the QR decomposition projects this vector away from the first vector (which we have assumed is u_1). This means that the remainder is

$$0 + c_2\sigma_2u_2 + \ldots + c_m\sigma_nu_n$$

which we then normalize to get

$$q_2 R_{22} = 0 + c_2\sigma_2 u_2 + \ldots + c_m\sigma_n u_n \,.$$

The value R_{22} is a normalization factor. The value R_{12} tells us how much of the vector x_2 lies in the direction of v_1. That is, $R_{12} = d_1$.

> *If the algorithm were to converge to the point where all the second vectors are orthogonal to v_1, then R_{12} would be 0. This suggests that the matrix $\mathbf{R}$ slowly becomes diagonal as we converge to the singular vectors.*

Taking R_{22} to the other side, and collecting all scalar multipliers into new multipliers e_i, we get

$$q_2 = 0 + e_2 u_2 + \ldots + e_k u_k \,.$$

In the third line of the algorithm, this vector is multiplied by M^T. This gives us

$$\begin{aligned} M^T q_2 &= 0 + e_2 M^T u_2 + \ldots + e_k M^T u_k \\ &= 0 + e_2\sigma_2 v_2 + \ldots + e_k\sigma_k u_k \,. \end{aligned}$$

Note that the first term remains zero, wether we are in the left or right singular basis of M. This tells us that when the first vector of P has converged to v_1, the rest of the vectors become entirely confined to the subspace orthogonal to v_1. In the left singular basis, we get $q_1 = u_1$ with the remaining columns of Q entirely confined to the subspace orthogonal to u_1.

We can now build an inductive argument to see what happens to the other vectors if we assume that $p_1 = v_1$. Let $P' = [\,p_2 \ \ldots \ p_m\,]$. Assume furthermore that all vectors in P' are orthogonal to v_1 (we have shown above that this is the case after one iteration of the algorithm with $p_1 = v_1$).

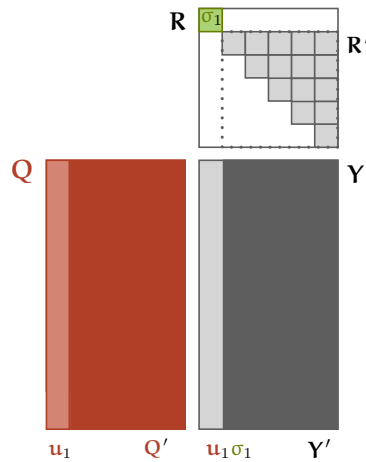
We will go through the algorithm line by line and show that the resulting matrices Q', $\mathbf{R}'$ and B' are submatrices of the matrices Q, $\mathbf{R}$ and $\mathbf{B}$ computed by the iteration on the complete P. From this, we can then conclude that whatever we know about the algorithm applied to P, applies to P' as well.

In the first line of the algorithm, we multiply $\mathbf{Y}' \leftarrow \mathbf{M}\mathbf{P}'$. From the basic definition of matrix multiplication, we can conclude that $\mathbf{Y}'$ consists of the rightmost vectors of $\mathbf{Y}$, without the first one. We can also conclude that all columns of $\mathbf{Y}'$ are orthogonal to $\mathbf{u}_1$.

> *Each column of $\mathbf{P}'$ is orthogonal to $\mathbf{v}_1$, so when we express such a column as a sum of the right eigenvectors, the first term is 0, which means that when we multiply by $\mathbf{M}$, the multiplier for the first left eigenvector $\mathbf{u}_i$ is zero as well.*

Second, we perform the QR decomposition $\mathbf{Q}', \mathbf{R}' \leftarrow \text{qr}(\mathbf{X}')$. We know that span $\mathbf{X}'$ is a subspace orthogonal to $\mathbf{u}_1$, so $\mathbf{Q}'$ will be an orthogonal basis for that subspace. That is $\mathbf{Q} = [\mathbf{v}_1\ \mathbf{q}'_1\ \ldots\ \mathbf{q}'_m]$.

How are $\mathbf{R}$ and $\mathbf{R}'$ related? R_{11} is a scaling factor for $\mathbf{v}_1$, and the remaining R_{1i} show how much of each of the other vectors of $\mathbf{X}$ projects onto $\mathbf{v}_1$. Under our assumptions, these are all orthogonal to $\mathbf{v}_1$, so the top row of R is zero everywhere except the diagonal. This suggests that if we strip the top row and leftmost column from $\mathbf{R}$, we get $\mathbf{R}'$.



The QR decomposition of the whole matrix $\mathbf{Y}$ contains the QR decomposition of the submatrix $\mathbf{Y}'$ (if its first column is a scalar multiple of $\mathbf{u}_1$).

In the third line of the algorithm, we compute $\mathbf{Y}' \leftarrow \mathbf{M}^T\mathbf{Q}'$. Since $\mathbf{Q}'$ spans a subspace orthogonal to $\mathbf{u}_1$, by the same logic we used for line 1, $\mathbf{Y}'$ spans a subspace orthogonal to $\mathbf{v}_1$.

Finally, we repeat the logic of line 2 to conclude that after the fourth line $\mathbf{P} = [\mathbf{v}_1\ \mathbf{p}'_1\ \ldots\ \mathbf{p}'_m]$. And that $\mathbf{B}'$ is the bottom right submatrix of $\mathbf{B}$.

To summarize, for a matrix $\mathbf{P}'$ with columns orthogonal to $\mathbf{v}_1$ and a matrix $\mathbf{P} = [\mathbf{v}_1\ \mathbf{p}'_1\ \ldots\ \mathbf{p}'_m]$, we have just shown that an iteration on $\mathbf{P}'$ produces matrices $\mathbf{Q}'$, $\mathbf{R}'$, $\mathbf{P}'$ and $\mathbf{B}'$, which are submatrices of the corresponding matrices we would get if we ran the iteration on $\mathbf{P}$.

This tells us directly that if $\mathbf{p}_2 = \mathbf{v}_2$ before the iteration, by the argument we have already made above, this is a fixed point, and it will remain so after the iteration.

However, what if it isn't? It seems likely that the iteration with an arbitrary starting value for $\mathbf{p}_2$ will converge to $\mathbf{v}_2$ eventually. What we've shown above is, assuming $\mathbf{v}_1 = \mathbf{p}_1$, that after the first iteration, where $\mathbf{p}_2$ is projected away from $\mathbf{v}_1$, it will remain orthogonal to $\mathbf{v}_1$ forever.

Since $\mathbf{p}_2$ is guaranteed to be orthogonal to $\mathbf{p}_1$ already, all that happens in the QR decomposition is that it is scaled to a unit vector. We can see this in the matrix $\mathbf{R}$, where R_{12} is zero, and R_{22} (or R'_{11}) gives us the required scaling factor.

Under these assumptions, for $\mathbf{p}_2$ one iteration of the algorithm performs in order, a matrix multipliation, a scaling, a matrix multiplication and another scaling. Or, symbolically:

$$\mathbf{p}_2 \leftarrow L_{22}\mathbf{M}^T R_{22}\mathbf{M}\mathbf{p}_2 = L_{22}R_{22}\mathbf{M}^T\mathbf{M}\mathbf{p}_2\,.$$

After k iterations, we get

$$\mathbf{p}_2^k = L_{22}\mathbf{M}^T R_{22}\mathbf{M}\mathbf{p}_2 = (L_{22}R_{22})^k(\mathbf{M}^T\mathbf{M})^k\mathbf{p}_2^0\,.$$

This should look familiar. It shows that all we are doing is iteratively multiplying by $\mathbf{M}^T\mathbf{M}$, and occasionally normalizing. Normally, such an iteration would converge to the *first* eigenvector of $\mathbf{M}^T\mathbf{M}$, and the first right singular vector of $\mathbf{M}$. However, by careful choice of the initial vector, we are constrained to be (and stay) orthogonal to that first eigenvector. As we've already seen

in the orthogonal iteration for eigenvectors, this means that we will converge to the second eigenvector.

Finally, we can repeat the same argument for the other vectors. If we set the first *two* columns of $\mathbf{P}$ equal to v_1 and v_2 respectively, the same argument tells us that the algorithm will behave as though we are just iterating over the remaining columns, and will converge to the dominant singular vector in the remaining subspace. We can continue this process until all columns are exhausted.

In practice, of course, the second column doesn't need to wait until the first has converged. The closer $\mathbf{p}_1$ gets to v_1, the better constrained the iteration on $\mathbf{p}_2$ will be to the correct subspace. By the time $\mathbf{p}_1$ has converged we are likely to see that $\mathbf{p}_2$ has also found its correct value. However, if it hasn't, we have just shown that it is guaranteed to once $\mathbf{p}_1 = v_1$.

5.2.3 The QR algorithm for the SVD

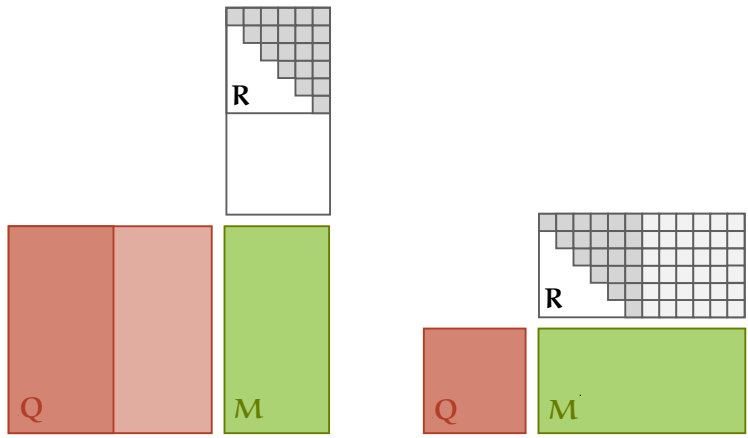
If only for the sake of symmetry, it would be nice if there were an SVD version of the QR iteration algorithm. Happily, it turns out there is, although we need to be careful to translate the logic of the QR iteration in the correct way.

As before, we'll show the algorithm first, and then work out how it relates to the orthogonal iteration.

$$
\begin{aligned}
&\mathbf{X}, \mathbf{Y} \leftarrow \mathbf{M}, \mathbf{M}^\mathsf{T} \\
&\textbf{loop} : \\
&\quad \mathbf{Q}, \mathbf{R} \leftarrow \mathrm{qr}(\mathbf{X}) \\
&\quad \mathbf{P}, \mathbf{B} \leftarrow \mathrm{qr}(\mathbf{Y}) \\
&\quad \mathbf{X} \leftarrow \mathbf{R}\mathbf{P} \\
&\quad \mathbf{Y} \leftarrow \mathbf{B}\mathbf{Q}
\end{aligned}
$$

Compared to the QR iteration for eigenvectors, things have become a little more complex. There, we just took a QR decomposition and multiplied it back in reverse order. Here, we take two QR decompositions, and we don't just multiply them back in reverse order, we also mix up the matrices, multiplying $\mathbf{R}$ with $\mathbf{P}$ and $\mathbf{B}$ with $\mathbf{Q}$.

As before, this algorithm only works if we compute the full QR decomposition. Here's what that looks like for a rectangular matrix.



The key requirement is that Q is square. This means that if the matrix is tall, as it is on the left, we compute as many orthogonal vectors as the matrix has columns, and then extend this to a full basis by choosing arbitrary orthogonal vectors until Q is square. To make the multiplication work, we then extend **R** with rows filled with zeros.

If the matrix is wide, the QR algorithm should provide us with a full basis. We now just need to extend **R** far enough to make the multiplication of Q and **R** reconstruct all columns of the matrix. Since we have a full basis with at least the dimensionality of the matrix, we know that the remaining columns are linear combinations of the columns of Q. We express each remaining column of the matrix as a linear combination of the columns of Q, giving us an additional column of **R** until **R** is as wide as M.

Now, to work out why this is the algorithm that gives us the result we want, we'll need to carefully take the logic from the eigenvector version, and apply it here step-by-step.

In the eigenvector version, two ideas were key:

1. We noted that the sequence of $\mathbf{R}'$ matrices computed by the orthogonal algorithm was defined by $\mathbf{R}'i = \mathbf{Q}'_i\mathbf{A}\mathbf{Q}'_{i-1}$.
2. We defined a sequence of new matrices $\mathbf{D}_i = \mathbf{Q}'^{\mathsf{T}}_i\mathbf{A}\mathbf{Q}'_i$, inspired by the equation that holds when the algorithm has converged.

We then showed that the sequence of $\mathbf{D}_i$'s can be computed by taking the QR decomposition of the previous element, and multiplying it in reverse order, to produce the next.

Translating idea 1 to the SVD version of the orthogonal iteration tells us that

$$\begin{aligned} \mathbf{Q}'_i\mathbf{R}'_i &= \mathbf{M}\mathbf{P}'_{i-1} & \mathbf{R}'_i &= \mathbf{Q}'^{\mathsf{T}}_i\mathbf{M}\mathbf{P}'_{i-1} \\ \mathbf{P}'_i\mathbf{B}'_i &= \mathbf{M}^{\mathsf{T}}\mathbf{Q}'_i & \mathbf{B}'_i &= \mathbf{P}'^{\mathsf{T}}_i\mathbf{M}^{\mathsf{T}}\mathbf{Q}'_i\,. \end{aligned}$$

On the left are simply the two QR decompositions computed in one iteration of the orthogonal algorithm. On the right, we've rewritten them to isolate $\mathbf{R}'_i$ *and* $\mathbf{B}'_i$

To translate idea 2 we note that at convergence, we can increment the index of the rightmost factor by one. That is, in the limit of $i \to \infty$, the following equations hold:

$$\begin{aligned} \mathbf{R}'_i &= \mathbf{Q}'^{\mathsf{T}}_i\mathbf{M}\mathbf{P}'_i \\ \mathbf{B}'_i &= \mathbf{P}'^{\mathsf{T}}_i\mathbf{M}^{\mathsf{T}}\mathbf{Q}'_{i+1}\,. \end{aligned}$$

We now take these equations, and define new matrices $\mathbf{X}_i$ and $\mathbf{Y}_i$ according to them. This is exactly the logic we used to define $\mathbf{D}_i$. In the limit they are equal to $\mathbf{R}'_i$ and $\mathbf{B}'_i$, but for small i there may be a large difference.

$$\begin{aligned} \mathbf{X}_i &= \mathbf{Q}'^{\mathsf{T}}_i\mathbf{M}\mathbf{P}'_i \\ \mathbf{Y}_i &= \mathbf{P}'^{\mathsf{T}}_i\mathbf{M}\mathbf{Q}'_{i+1}\,. \end{aligned}$$

The final step is to show that in these sequences of $\mathbf{X}_i$ and $\mathbf{Y}_i$ we can always compute the element at i by QR decomposing the elements at $i-1$ and multiplying back in reverse order *and mixing up the matrices*.

We'll start with elements $\mathbf{X}_{i-1}$ and $\mathbf{Y}_{i-1}$. We'll need to express both of them as QR decompositions.

$$\begin{aligned}
\mathbf{X}_{i-1} &= {\mathbf{Q}'_{i-1}}^{\mathsf{T}}\mathbf{M}\mathbf{P}'_{i-1} & \mathbf{Y}_{i-1} &= {\mathbf{P}'_{i-1}}^{\mathsf{T}}\mathbf{M}^{\mathsf{T}}\mathbf{Q}'_i && \text{by definition}\\
\mathbf{X}_{i-1} &= {\mathbf{Q}'_{i-1}}^{\mathsf{T}}\mathbf{Q}'_i\mathbf{R}'_i & \mathbf{Y}_{i-1} &= {\mathbf{P}'_{i-1}}^{\mathsf{T}}\mathbf{P}'_i\mathbf{B}'_i && \text{QR's in the orth. alg.}\\
\mathbf{X}_{i-1} &= {\mathbf{Q}'_{i-1}}^{\mathsf{T}}\mathbf{Q}'_i\mathbf{R}_i & \mathbf{Y}_{i-1} &= {\mathbf{P}'_{i-1}}^{\mathsf{T}}\mathbf{P}'_i\mathbf{B}'_i && \text{highlight}.
\end{aligned}$$

In the last line, we haven't changed anything. We've only highlighted that we've ended up expressing both $\mathbf{X}_{i-1}$ and $\mathbf{Y}_{i-1}$ as a QR decomposition. The two factors highlighted in green are both orthogonal matrices, so their product is an orthogonal matrix as well, and the remainders $\mathbf{R}_i$ and $\mathbf{B}_i$ are upper triangular.

Now, we need to show that the next matrices in the sequence, $\mathbf{X}_i$ and $\mathbf{Y}_i$, can be expressed in terms of the factors of these two QR decompositions. Starting with the definitions, we get

$$\begin{aligned}
\mathbf{X}_i &= {\mathbf{Q}'_i}^{\mathsf{T}}\mathbf{M}\mathbf{P}'_i & \mathbf{B}_i &= {\mathbf{P}'_i}^{\mathsf{T}}\mathbf{M}^{\mathsf{T}}\mathbf{Q}'_{i+1} && \text{by definition}\\
&= {\mathbf{Q}'_i}^{\mathsf{T}}\mathbf{M}{\mathbf{P}'_{i-1}}^{\mathsf{T}}\mathbf{P}'_{i-1}\mathbf{P}'_i & &= {\mathbf{P}'_i}^{\mathsf{T}}\mathbf{M}^{\mathsf{T}}{\mathbf{Q}'_i}^{\mathsf{T}}\mathbf{Q}'_i\mathbf{Q}'_{i+1} && \text{insert } \mathbf{I}\\
&= {\mathbf{Q}'_i}^{\mathsf{T}}\mathbf{Q}'_i\mathbf{R}'_i\mathbf{P}'_{i-1}\mathbf{P}'_i & &= {\mathbf{P}'_i}^{\mathsf{T}}\mathbf{P}'_i\mathbf{B}'_i\mathbf{Q}'_i\mathbf{Q}'_{i+1} && \text{from orth. alg.}\\
&= \mathbf{R}'_i\mathbf{P}'_{i-1}\mathbf{P}'_i & &= \mathbf{B}_i\mathbf{Q}'_i\mathbf{Q}'_{i+1}
\end{aligned}$$

¡p¿And there we have the proof that our algorithm converges to the right solution. If we QR decompose $\mathbf{X}_{i-1}$ and $\mathbf{Y}_{i-1}$ and multiply the results back together, mixed up and in reverse order, we get the values $\mathbf{X}_i$ and $\mathbf{Y}_i$. Since we know that these converge to the same values as the sequences $\mathbf{R}'_i$ and $\mathbf{B}'_i$, we see that we must be computing the singular value decomposition of $\mathbf{M}$.

As before, the stripped down version we've given here only gives you the singular values (on the diagonal of $\mathbf{X}$*). If you want the*

singular vectors as well, you'll need to keep a running product of **Q** *and* **P**.

This is where we will end our journey through the magical world of principal component analysis and all its relations. There is much we could still investigate. There are probabilistic and weighted versions of PCA. Regularized and sparse versions. There is a sidepath about non-linear PCA that could lead us into autoencoders, and from there into variational approaches in deep learning. Or, we could take a different track, and look into kernel-based nonlinear versions of PCA.

This is how it is, usually. The more you learn, the more the boundary of the unknown expands with it. It can be disheartening, especially if you're in the habit of focusing on all the things you don't yet know.

So let's allow ourselves a look back instead of forward, to see what we've accomplished. We may not know about all the ways in which PCA may be extended, and given more power by non-linear additions, but constrained to the purely linear domain, I think we can say we've covered the ground pretty thoroughly.

We've seen the basic mechanism in operation, we've uncovered its connection to eigenvalues, we've proved the spectral theorem that the whole thing hinges on, and we've looked at the singular value decomposition, which provides a complementary perspective. Finally, in this last chapter, we've looked at a triad of algorithms for computing the eigendecomposition, and then adapted each in turn to provide us with the singular value decomposition instead.

Looking back, the aim of properly explaining PCA, down to its foundations has set us on a far more formidable journey than that initial challenge implied and, I must admit, than I originally anticipated. Taken together, these five chapters touch on almost all the topics that you might find in a standard linear algebra text book. From bases to (psuedo)inverses, to ranks and determinants. We may have entered the forest at a different point, and with a different goal, but our walk has covered much of the same ground.

And that may be the best reason for an exercise like this. Let's

be honest: you don't need to know all this to use PCA effectively. The first chapter of this book alone is more than enough to know what you are doing when you call the standard implementation in some data science library. However, true understanding of a concept comes from seeing it from different angles. So it is with the many building blocks of linear algebra.

The explanations you find in a textbook will provide you, usually, with just one perspective. Enough for a basic grasp of the material, if you're lucky enough to remember what you learned. This kind of exercise, picking one method, and digging into it all the way down to the foundations, invariably requires you to revisit all those concepts you learned from the textbook, but with a new perspective, and a more concrete motivation.

If you're still reading at this point, that is hopefully what we've accomplished. To show how the whole cathedral of linear algebra works together to produce this magical operation that can take a messy, high-dimensional flood of data, and extract clean, low dimensional data, that more often than not, corresponds to the concepts we associate with its domain.

Bibliography

Gilbert, E., Shanmugam, A., and Cavalleri, G. L. (2022). Revealing the recent demographic history of europe via haplotype sharing in the uk biobank. *Proceedings of the National Academy of Sciences*, 119(25):e2119281119.

Golub, G. H. and Van Loan, C. F. (2013). *Matrix computations*. JHU press.

Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37.

Lay, D. C. (1994). Linear algebra and its applications (instructor's ed.).

Novembre, J., Johnson, T., Bryc, K., Kutalik, Z., Boyko, A. R., Auton, A., Indap, A., King, K. S., Bergmann, S., Nelson, M. R., et al. (2008). Genes mirror geography within europe. *Nature*, 456(7218):98–101.

Pearson, K. (1901). Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572.

Young, N. M., Capellini, T. D., Roach, N. T., and Alemseged, Z. (2015). Fossil hominin shoulders support an african ape-like last common ancestor of humans and chimpanzees. *Proceedings of the National Academy of Sciences*, 112(38):11829–11834.

APPENDIX A · SOME HELPFUL LINEAR ALGEBRA PROPERTIES

The following properties may be good to reference when following the proofs and more technical parts in this series. Anything that holds for matrix multiplication holds fort vector/matrix multiplication and for vector/vector multiplication as a special case.

If you want an exercise to test yourself, see which of the following properties you need to show that: for a dataset $\mathbf{x}_1, \ldots, \mathbf{x}_n$ the dot product of the mean of the data with a vector w is the same as the mean of the dot products of each individual instance with w.

- Matrix multiplication does not commute: $\mathbf{AB} \neq \mathbf{BA}$ in general. Matrix multiplication *does* distribute $\mathbf{ABC} = \mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$.
- Any scalar in a matrix multiplication can be moved around freely: $s\mathbf{ABC} = \mathbf{A}s\mathbf{BC} = \mathbf{AB}s\mathbf{C} = \mathbf{ABC}s = \sqrt{s}\mathbf{ABC}\sqrt{s}$ (matrix multiplication is homogeneous).
- Matrix multiplication is additive $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$.
- To take a transposition operator inside a multiplication, flip the order of the multiplication and transpose each element individually: $(\mathbf{ABC})^\mathsf{T} = \mathbf{C}^\mathsf{T}\mathbf{B}^\mathsf{T}\mathbf{A}^\mathsf{T}$. The same holds for matrix inversion, if all matrices ar invertible.
- An invertible matrix $\mathbf{A}$ is a square matrix for which a matrix $\mathbf{A}^{-1}$ exists (the inverse) such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$.
- To move a matrix multiplication "to the other side," multiply (on the same side as the original multiplication) by the inverse:

  $$\mathbf{AB} = \mathbf{C} \Rightarrow \mathbf{A}^{-1}\mathbf{AB} = \mathbf{A}^{-1}\mathbf{C} \Rightarrow \mathbf{IB} = \mathbf{A}^{-1}\mathbf{C} \Rightarrow \mathbf{B} = \mathbf{A}^{-1}\mathbf{C}\,.$$

This requires an invertible matrix.

- Invertable matrices are fine when used in derivations, but in practice, the operation can be numerically unstable, so it's usually avoided by rewriting to a more stable form.
- The **dot product** of two vectors $\mathbf{x}$ and $\mathbf{y}$ is $\mathbf{x}^T\mathbf{y} = \sum_i x_i y_i$. $\mathbf{x}$ and $\mathbf{y}$ are **orthogonal** if $\mathbf{x}^T\mathbf{y} = 0$. This implies that the angle between them (in their shared plane) is 90 deg.
- When we multiply two matrices, $\mathbf{C} = \mathbf{AB}$ we are essentially computing all dot products of a *row* of $\mathbf{A}$ and a *column* of $\mathbf{B}$, and arranging the result in a matrix. C_{ij} is the dot product of row i of $\mathbf{A}$ and column j of $\mathbf{B}$.
- The length of a vector is the square root of its dot product with itself.
- A **unit vector** is a vector of length 1. Since $\sqrt{1} = 1$, we can characterize unit vectors by $\mathbf{x}^T\mathbf{x} = 1$.
- All vectors are column vectors unless otherwise noted. To save whitespace, we may write these inline as $\mathbf{x} = (0, 1, 0)$, but they should still be considered column vectors. The transpose of a column vector is a row vector.

APPENDIX B · PROOFS

B.2 For Chapter 2

Here is the proof that the combined problem for variance maximization is equivalent to the combined problem for reconstruction error.

Equivalence of combined optimization The combined problem for reconstruction error minimization

$$\underset{W}{\operatorname{argmin}} \sum_{i} \|x_i - x_i'\|$$
$$\text{such that } W^T W = I$$

is equivalent to the following variance maximization problem

$$\underset{W}{\operatorname{argmax}} \sum_{i,r} {z_{ir}}^2 \quad \text{with } z_{ir} = {w_r}^T x_i$$
$$\text{such that } W^T W = I\,.$$

Proof. Define the vector z_i as the combination of all the individual z_{ir}'s:

$$z_i = \left({x_i}^T w_1, \ldots, {x_i}^T w_k\right)\,.$$

Note that ${x_i}' = W z_i$. That is, z_i is the latent vector from which we reconstruct x_i'. The length of z is given by:

$$\|z_i\|^2 = {z_{i1}}^2 + \ldots + {z_{ik}}^2$$

so that our objective simplifies to

$$\operatorname*{argmax}_W \sum_i \|z_i\|^2 \quad \text{such that } W^\top W = I\,.$$

The (squared) length of z_i is the same as that of $x_i{}'$, because

$$\|x_i{}'\|^2 = x_i{}'^\top x_i{}' = (Wz_i)^\top Wz_i = z_i{}^\top W^\top Wz_i = z_i{}^\top z_i = \|z_i\|^2$$

so that our objective rewrites to

$$\operatorname*{argmax}_W \sum_i \|x_i'\|^2 \quad \text{such that } W^\top W = I\,.$$

At this point, we can draw another triangle: from the origin, to x_i to $x_i{}'$ and back to the origin. If $x_i{}'$ is the closest point to x_i in the subspace spanned by the columns of W, then the angle at $x_i{}'$ must be orthogonal. Therefore

$$\|x_i\|^2 = \|x_i{}'\|^2 + \|x_i' - x_i\|^2$$

From which we can derive the reconstruction error minimization objective in the same way we did for the iterative problem. □

B.3 For Chapter 3

Euclidean division (simplified) Given a polynomial $p(z)$ of degree n and a linear factor $z - r$, there is a polynomial $q(z)$ of degree $n-1$ and a constant c, called the remainder such that

$$p(z) = (z-r)q(z) + c\,.$$

Proof.

Base case. Let $n = 1$. Assume we are given $p(z) = c_1 z + c_0$, and some r. Define the 0-order polynomial $q(z) = c_1 z$ and the remainder $d = rc_1 + c_0$. This gives us

$$(z-r)q(z) + d = (z-r)c_1 + rc_1 + c_0 = p(z)\,.$$

Induction step. If the theorem holds for $n-1$, then we can show that it holds for n also. Assume we are given an n-th order

polynomial $p(z)$ and some r. Let $p_{\text{tail}}(z)$ consist of all its terms except the highest order one. We can then write:

$$\begin{aligned}
p(z) &= c_n z^n + p_{\text{tail}}(z) \\
p(z) &= c_n z^{n-1} z + p_{\text{tail}}(z) \\
p(z) &= c_n z^{n-1} z - c_n z^{n-1} z + p_{\text{tail}}(z) + c_n z^{n-1} z \\
p(z) &= (z-r) c_n z^{n-1} z + p_{\text{tail}}(z) + c_n z^{n-1} z .
\end{aligned}$$

The last two terms are a polynomial of degree $n-1$. By assumption, we can factor this according to the theorem, which gives us some $q'(z)$ and d' so that

$$\begin{aligned}
p(z) &= (z-r) c_n z^{n-1} + (z-r) q'(z) + d' \\
&= (z-r)(c_n z^{n-1} + q'(z)) + d'
\end{aligned}$$

Where $q(z)$ has degree $n-2$, so that if we set $q(z) = c_n z^{n-1} + q'(z)$ and $d = d'$, we satisfy the theorem. □

Orthogonal vectors. Let $z \in \mathbb{C}^n$. We can choose $n-1$ additional vectors that are orthogonal to z and to each other.

Proof. We'll take it as read that this can be done for real valued vectors.

Let z^m be a real vector containing the magnitudes of the elements of z, and let z^a be a real vector containing the angles. Construct a set of real vectors orthogonal to z^m and to each other.

Then, for the ith element of each of these vectors, including z^m, change the angle from 0 to z^a_i.

Now, note that orthogonality of two of these vectors v and u requires that:

$$0 = \sum_i u_i \overline{v_i} = \sum_i u_i^m \angle u_i^a \ \overline{v_i^m \angle v_i^a} = \sum_i u_i^m \angle u_i^a \ v_i^m \angle -v_i^a$$
$$= \sum_i u_i \angle z_i^a \ v_i^m \angle -z_i^a = \sum_i u_i^m v_i^m \angle 0$$

In short, the angles cancel out, so the dot product reduces to the real-valued dot product of $\mathbf{u}^m$ and $\mathbf{v}^m$, which is 0 by construction. □

As we've seen, for a particular eigenvalue λ many different eigenvectors $\mathbf{x}$ will satisfy $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$. These may well be complex, even if $\mathbf{A}$ and λ are both real. In such cases, however, there is always a real-valued eigenvector as well.

Real eigenvectors. For a real-valued matrix $\mathbf{A}$, with a real eigenvalue λ, if there is a corresponding complex-valued eigenvector, then there is a real-valued eigenvector as well.

Proof. Let $\mathbf{x}$ be a complex eigenvector. We can easily show that its conjugate is an eigenvector too. If $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, then conjugating both sides, we get $\overline{\mathbf{A}}\overline{\mathbf{x}} = \overline{\lambda}\overline{\mathbf{x}}$, and because $\mathbf{A}$ and λ are real, $\mathbf{A}\overline{\mathbf{x}} = \lambda\overline{\mathbf{x}}$.

Next, note that the sum of $\mathbf{x}$ and its conjugate is an eigenvector as well, since

$$\mathbf{A}(\mathbf{x} + \overline{\mathbf{x}}) = \mathbf{A}\mathbf{x} + \mathbf{A}\overline{\mathbf{x}} = \lambda\mathbf{x} + \lambda\overline{\mathbf{x}} = \lambda(\mathbf{x} + \overline{\mathbf{x}}) .$$

The sum of a vector and its conjugate is real, so we have constructed a real-valued eigenvector for λ. □

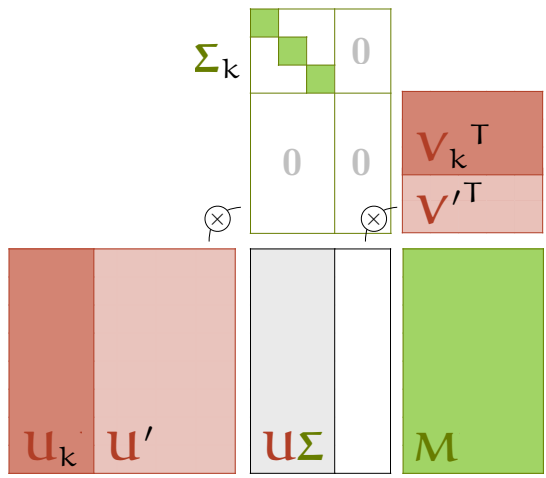
More generally, any linear combination of two eigenvectors for the same eigenvalue λ is an eigenvector for λ.

B.4 For Chapter 4

When we know that a matrix has a given number of singular values, we can truncate the SVD to that number without losing information.

Exact truncated SVD Let M be any matrix with k singular values and let $M = U\Sigma V^T$ be its full SVD. Then, the truncated SVD $M = U_k\Sigma_k V_k{}^T$ holds exactly.

Proof. We denote the different submatrices of the full SVD as follows.



$$M = [U_k\ U'] \begin{bmatrix} \Sigma_k & 0 \\ 0 & 0 \end{bmatrix} [V_k\ V']$$

We can now prove what we want—that U', V', and the zeroes around Σ_k aren't necessary to decompose M—simply by a small number of rewriting steps. To do this, it's useful to know how matrix multiplication distributes over this concatenation operator $[\,]$. It depends on whether the "split" in the matrix is parallel to the dimension we match to multiply or orthogonal to it.

If the split is parallel, as in the multiplication $\begin{bmatrix}\mathbf{A}\\ \mathbf{B}\end{bmatrix}\mathbf{C}$, the result is another concatenated matrix, but with each submatrix multiplied by $\mathbf{C}$:

$$\begin{bmatrix}\mathbf{A}\\ \mathbf{B}\end{bmatrix}\mathbf{C} = \begin{bmatrix}\mathbf{AC}\\ \mathbf{BC}\end{bmatrix}$$

If the split is orthogonal, as in the multiplication $[\mathbf{A}\ \mathbf{B}]\ \mathbf{C}$, we must split the other matrix $\mathbf{C}$ into submatrices $\mathbf{C}_1$, $\mathbf{C}_2$ to match, and the result is the sum:

$$[\mathbf{A}\ \mathbf{B}]\ \mathbf{C} = \mathbf{AC}_1 + \mathbf{BC}_2\,.$$

With these two rules, we can take the full SVD, and simply distribute the matrix multiplications out over the various concatenations.

$$\begin{aligned}
M &= U\Sigma V^T \\
&= [U_k\ U'] \begin{bmatrix}\Sigma_k & 0\\ 0 & 0\end{bmatrix} V^T \\
&= U_k \begin{bmatrix}\Sigma_k & 0\end{bmatrix} V^T + U' \begin{bmatrix}0 & 0\end{bmatrix} V^T \\
&= U_k \begin{bmatrix}\Sigma_k & 0\end{bmatrix} \begin{bmatrix}V_k^T\ V'^T\end{bmatrix} \\
&= U_k \Sigma_k V_k^T + U_k 0 V'^T \\
&= U_k \Sigma_k V_k^T
\end{aligned}$$

□

The pre-image of a point in a column space is orthogonal to the column space.

Proof. Let $Mx = My$, let $z \in \text{col } M$ and let $M = [m_1 \ldots m_m]$. Then, for some $z'_1 \ldots z'_m$ we can write $z = z'_1 m_1 + \ldots + z'_m m_m$. Thus,

$$\begin{aligned} z^T(x-y) &= (z_1'm_1 + \ldots + z_m'm_m)^T(x-y) \\ &= (z_1'm_1x_1 + \ldots + z_m'm_mx_m) - (z_1'm_1y_1 + \ldots + z_m'm_my_m) \\ &= z'Mx - z'My = 0\,. \end{aligned}$$

This tells us that the difference between any two vectors in the pre-image of a point is orthogonal to any vector in the column space of M, so the two are orthogonal. □

B.5 For Chapter 5

In our discussion, we occasionally make the leap from showing that we can derive *a* QR decomposition to taking that to be *the* QR decomposition. Under the right conditions, this is justified: if M is full rank, a QR decomposition with only positive elements on the diagonal of R is unique.

That is, for any QR decomposition, we can always create another valid QR decomposition by flipping the sign on one of the columns of Q and changing the sign of the corresponding row of R. But up to the variations we can create by these sign changes, the decomposition is unique.

This trick can also be used to show that such a QR decomposition always exists. Just take any QR decomposition, and flip the sign for any row of R where the diagonal element is negative. We'll call this a *positive QR decomposition*.

Uniqueness of the positive QR decomposition For an $n \times m$ matrix M with linearly independent columns, the positive QR decomposition $QR = M$ is unique.

Proof. Let Q, R and P, B be two positive QR decompositions of M.

This suggests that $QR = PB$ and thus $P^TQ = BR^{-1}$.

Note that R is invertible because M's columns are linearly independent. A triangular matrix is invertible if and only if its diagonal is nonzero everywhere (the determinant is the diagonal product), and a diagonal element in R is zero if we can express

one column of $\mathbf{M}$ as a linear combination of the other vectors.

Note also that while $\mathbf{P}$ isn't square, so not invertible, the fact that its columns are mutually orthogonal unit vectors still gives us $\mathbf{P}^\mathsf{T}\mathbf{P} = \mathbf{I}$, allowing us to move $\mathbf{P}$ to the left side of the equation.

Now, we can conclude the following:

1. The inverse of $\mathbf{R}$ must be upper triangular as well. By definition $\mathbf{R}\mathbf{R}^{-1} = \mathbf{I}$, and having a non-zero element below the diagonal on $\mathbf{R}^{-1}$ would create a non-zero element below the diagonal in $\mathbf{R}\mathbf{R}^{-1}$ (note that all elements of $\mathbf{R}$ must be non-zero).

2. The product of two upper triangular matrices is itself upper triangular. This follows from the fact that element i, j of the multiplication $\mathbf{AB}$ is the dot product of the i-th row of $\mathbf{A}$ and the j-th column of $\mathbf{B}$. If both are upper triangular, then all elements up to i are zero in this row of $\mathbf{A}$ and all elements after j are zero in the column of $\mathbf{B}$. If we are below the diagonal, them $i > j$, so every term in the dot product is zero.

3. Since $\mathbf{R}$ has a positive diagonal, so does its inverse. We can see from a matrix multiplication diagram that the elements that create the diagonal in the product are only the diagonal elements of $\mathbf{R}^{-1}$, the rest are multiplied by zeroes. Since the resulting diagonal elements need to be zero, the diagonal elements of $\mathbf{R}^{-1}$ must be the inverses of those of $\mathbf{R}$. So if the diagonal elements of $\mathbf{R}$ are positive, so are those of $\mathbf{R}^{-1}$. By similar logic, we can see that $\mathbf{B}\mathbf{R}^{-1}$ has a positive diagonal.

4. $\mathbf{P}^\mathsf{T}\mathbf{Q}$ is an orthogonal matrix. Note that col $\mathbf{P}$ = col $\mathbf{Q}$, so we can express the columns of $\mathbf{Q}$ as linear combinations of those of $\mathbf{P}$, or, for some $\mathbf{S}$ we have $\mathbf{P}\mathbf{S} = \mathbf{Q}$. Multiplying a vector by a matrix with orthonormal columns preserves lengths and dot products, so the columns of $\mathbf{S}$ must be orthonormal too. Since $\mathbf{S}$ is square, it's orthogonal. Finally $\mathbf{P}\mathbf{S} = \mathbf{Q}$ implies $\mathbf{S} = \mathbf{P}^\mathsf{T}\mathbf{Q}$.

In conclusion, $\mathbf{P}^\mathsf{T}\mathbf{Q} = \mathbf{B}\mathbf{R}^{-1}$ tells us two things. From the left hand side, that this is an orthogonal matrix, and from the right hand side, that it is upper triangular with a positive diagonal.

What does an orthogonal upper triangular matrix with a positive diagonal look like? We know that its first column must be a unit vector, with zeros everywhere except the first element. Its second vector must be orthogonal to the first, so its first element must be zero, as must everything below the diagonal to keep our matrix triangular. The only remaining nonzero element is on the diagonal, so it must be 1 to make the column a unit vector.

If we keep going like this, we see that all columns must be zero above the diagonal to keep them orthogonal to the previous columns, zero below the diagonal to keep the matrix triangular, and 1 on the diagonal to keep the column a unit vector.

> *Note that we can't make the diagonal -1 because we've concluded that the diagonal is positive everywhere.*

In short, we have shown that $\mathbf{P}^\mathsf{T}\mathbf{Q} = \mathbf{B}\mathbf{R}^{-1} = \mathbf{I}$. That means that $\mathbf{P}^\mathsf{T}$ is the inverse of $\mathbf{Q}$ and $\mathbf{R}^{-1}$ is the inverse of $\mathbf{B}$. Since the inverse is unique, $\mathbf{P} = \mathbf{Q}$ and $\mathbf{R} = \mathbf{B}$. □

Index