COMPUTING THE EIGENDECOMPOSITION AND THE SINGULAR VALUE DECOMPOSITION

PART 5 IN A SERIES ON PRINCIPAL COMPONENT ANALYSIS

30 Dec 2022 · code · part 1 · 2 · 3 · 4

We have looked at what PCA is, at how to understand it, and we have spent a full post 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.*

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 part 1. There, we searched, using projected gradient descent, for the unit vector that gave us the best least-squares 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 part 4.*

Our job in this part 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.

COMPUTING EIGENVECTORS

Our starting point will be the fact that we derived in part 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 part 4: that the principal components are the *singular vectors* of the *data* matrix $\mathbf{X}$, and we'll build on our algorithms for computing eigenvalues to develop three algorithms along the same lines for computing the SVD.

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.

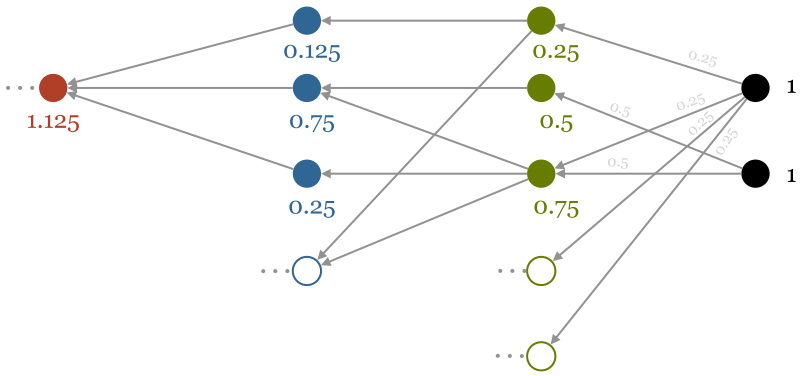
Back then, the main problem facing search engines was the number of webpages trying to game the system. Authors would include fake keywords, large swathes of invisible text, everything they could think of to get as high in the results for as many different search terms as possible. This would inevitably 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.

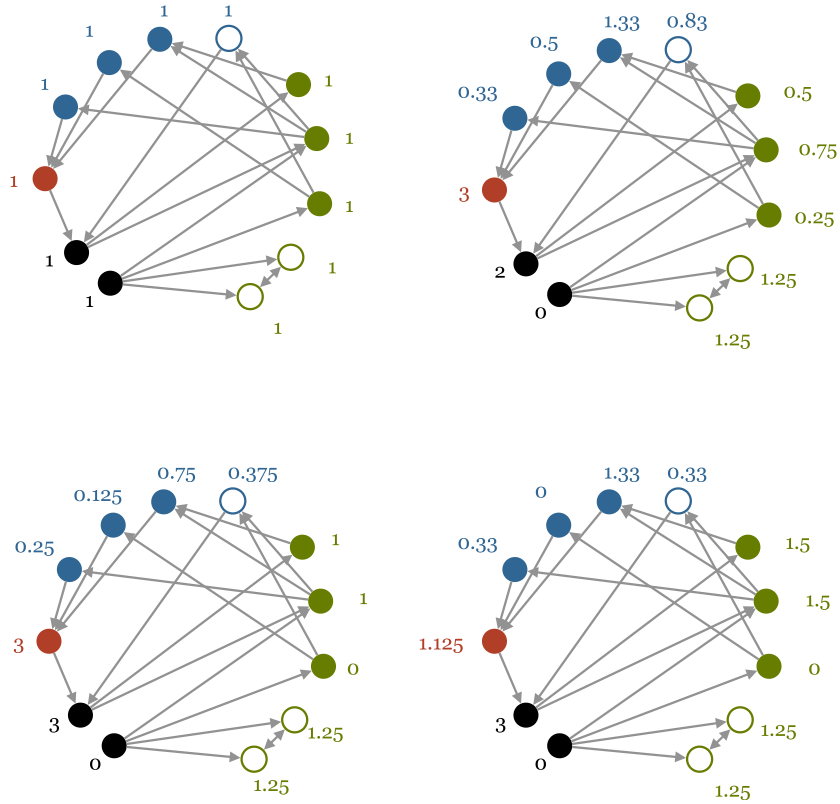
Theoretically, we could keep going forever, always avoiding the question of how we define the reputation of a website, by simply deferring the question

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 determines 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 unltimately 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.

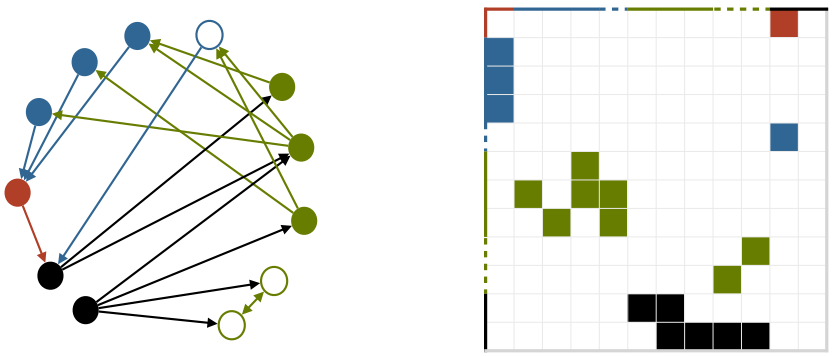
> *The two open green nodes in the graph above are an example of this. They claim a bit of the reputation at the start, but they aren't linked to*

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 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.

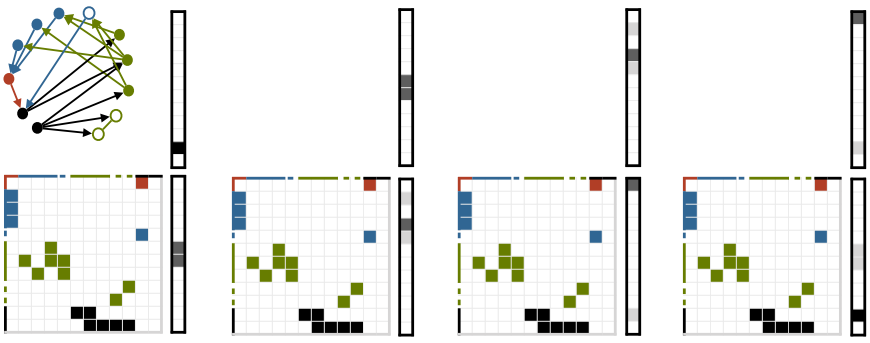
We can model the way website i distributes its initial unit of reputation, by starting with a *one-hot vector* indicating that website. This is a vector of length n, for n websites, with zeroes everywhere, except at the index i, corresponding to the website i, where it contains a 1. This represents our 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 multiplication: 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 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 A, normalize it, multiply the result by 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:

$$\begin{aligned}\frac{1}{\lambda}\boldsymbol{A}\boldsymbol{v} &= \boldsymbol{v} \\ \boldsymbol{A}\boldsymbol{v} &= \lambda\boldsymbol{v}\,.\end{aligned}$$

This should look familiar. It's the definition of an *eigenvector*, which we first saw in part 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 *principal* 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 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 being linked to by i, 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 $\boldsymbol{A}$ 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 $\boldsymbol{S}$, 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:

$$\boldsymbol{x} \leftarrow \frac{\boldsymbol{A}\boldsymbol{x}}{\|\boldsymbol{A}\boldsymbol{x}\|_1} .$$

That is, we multiply $\boldsymbol{A}$ by $\boldsymbol{x}$ 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

$$\mathbf{x} \leftarrow \frac{\mathbf{A}\mathbf{x}}{\|\mathbf{A}\mathbf{x}\|}.$$

If we removed the normalization step, and made the iteration $\mathbf{x} \leftarrow \mathbf{A}\mathbf{x}$, it would be very easy to see that after a number of iterations, say four, the resulting vector $\mathbf{x}_4$ would be $\mathbf{A}\mathbf{A}\mathbf{A}\mathbf{A}\mathbf{x}_0$, where $\mathbf{x}_0$ is the vector we started with. Put simply, we would have $\mathbf{x}_k = \mathbf{A}^k\mathbf{x}_0$.

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 $\|\mathbf{x}c\| = \|\mathbf{x}\|c$.

Therefore, we can say that

$$\mathbf{x}_2 = \frac{\mathbf{A}\mathbf{x}_1}{\|\mathbf{A}\mathbf{x}_1\|} = \frac{\mathbf{A}\frac{\mathbf{A}\mathbf{x}_0}{\|\mathbf{A}\mathbf{x}_0\|}}{\|\mathbf{A}\frac{\mathbf{A}\mathbf{x}_0}{\|\mathbf{A}\mathbf{x}_0\|}\|} = \frac{\mathbf{A}\mathbf{A}\mathbf{x}_0 \ \frac{1}{\|\mathbf{A}\mathbf{x}_0\|}}{\|\mathbf{A}\mathbf{A}\mathbf{x}_0\|\frac{1}{\|\mathbf{A}\mathbf{x}_0\|}} = \frac{\mathbf{A}^2\mathbf{x}_0}{\|\mathbf{A}^2\mathbf{x}_0\|}.$$

Or, more generally, the k-th vector in our iteration is just the vector $\mathbf{A}^k\mathbf{x}_0$, 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 $\mathbf{A}^k\mathbf{x}_0$.

First, we know from the spectral theorem that if $\mathbf{A}$ is $n \times n$ and symmetric, it has n real eigenvalues λ_i—including multiplicities—and n corresponding eigenvectors $\boldsymbol{v}_i$. 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 parts, that the eigenvectors form a basis: any vector in $\mathbb{R}^m$ can be written as a linear combination of the eigenvectors. That means that we can write $\mathbf{x}_0$ as $c_1\boldsymbol{v}_1 + \ldots + c_n\boldsymbol{v}_n$, for some values

$c_1 \ \dots \ c_n$. If we choose $\mathbf{x}_0$ 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 $\mathbf{x}_0$ written like this, and compute $\mathbf{x}_k$. First, let's look at the unnormalized vector $\boldsymbol{A}^k\mathbf{x}_0$

$$\begin{aligned}\boldsymbol{A}^k\mathbf{x}_0 &= \boldsymbol{A}^k(c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \ \dots \ + c_n\boldsymbol{v}_n) \\ &= c_1\boldsymbol{A}^k\boldsymbol{v}_1 + c_2\boldsymbol{A}^k\boldsymbol{v}_2 + \ \dots \ + c_n\boldsymbol{A}^k\boldsymbol{v}_n \\ &= c_1{\lambda_1}^k\boldsymbol{v}_1 + c_2{\lambda_2}^k\boldsymbol{v}_2 + \ \dots \ + c_n{\lambda_n}^k\boldsymbol{v}_n \\ &= c_1{\lambda_1}^k\left(\boldsymbol{v}_1 + \frac{c_2}{c_1}\frac{{\lambda_2}^k}{{\lambda_1}^k}\boldsymbol{v}_2 + \ \dots \ + \frac{c_n}{c_1}\frac{{\lambda_n}^k}{{\lambda_1}^k}\boldsymbol{v}_n\right) .\end{aligned}$$

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

$$\boldsymbol{A}^k\mathbf{x}_0 \to c_1{\lambda_1}^k\boldsymbol{v}_1 .$$

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.*

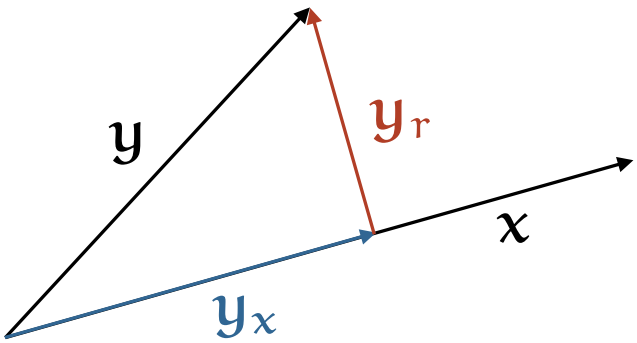
This is the method of **power iteration**. A very simple way of computing eigenvectors and a very powerful one. All you need is a way to compute matrix/vector products and a way to normalize your vectors. Even on something as large as the web adjacency matrix, this is a feasible computation, (if you store the matrix in the right way).

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 $\mathbf{x}$ converge to the dominant eigenvector, but at the same time, we also perform the power iteration on a second vector $\mathbf{y}$ orthogonal to $\mathbf{x}$. After each iteration, $\mathbf{x}$ and $\mathbf{y}$ won't necessarily be orthogonal anymore, so we change $\mathbf{y}$ to be orthogonal to $\mathbf{x}$ after every iteration.

How do we force one vector $\mathbf{y}$ to be orthogonal to another $\mathbf{x}$? The simplest way to achieve this, is to first project $\mathbf{y}$ onto $\mathbf{x}$, call the result $\mathbf{y}_\mathbf{x}$, and then to assume that $\mathbf{y}$ consists of two components: the part $\mathbf{y}_\mathbf{x}$, that points in the same direction as $\mathbf{x}$ and the remainder $\mathbf{y}_\mathbf{r}$. This tells us that $\mathbf{y} = \mathbf{y}_\mathbf{x} + \mathbf{y}_\mathbf{r}$, and thus that $\mathbf{y}_\mathbf{r} = \mathbf{y} - \mathbf{y}_\mathbf{x}$.



The vector $\mathbf{y}_\mathbf{r}$, $\mathbf{y}$ minus $\mathbf{y}$ projected onto $\mathbf{x}$ is called the *rejection* of $\mathbf{y}$ from $\mathbf{x}$. We'll use this to build our new power iteration. Here is the new algorithm:

loop:

$\mathbf{x}, \mathbf{y} \leftarrow A\mathbf{x}, A\mathbf{y}$

$\mathbf{y} \leftarrow \mathbf{y} - \mathbf{y}_\mathbf{x}$ make $\mathbf{y}$ orthogonal to $\mathbf{x}$

$\mathbf{x}, \mathbf{y} \leftarrow \mathbf{x}/\|\mathbf{x}\|,\ \mathbf{y}/\|\mathbf{y}\|$ normalize both

Note that the operations we apply to $\mathbf{x}$ are exactly the same as before. All we've done is to add another vector to the iteration. Our earlier proof that $\mathbf{x}$

converges to the dominant eigenvector v_1 still applies to this algorithm. The question is, what does $\mathbf{y}$ converge to?

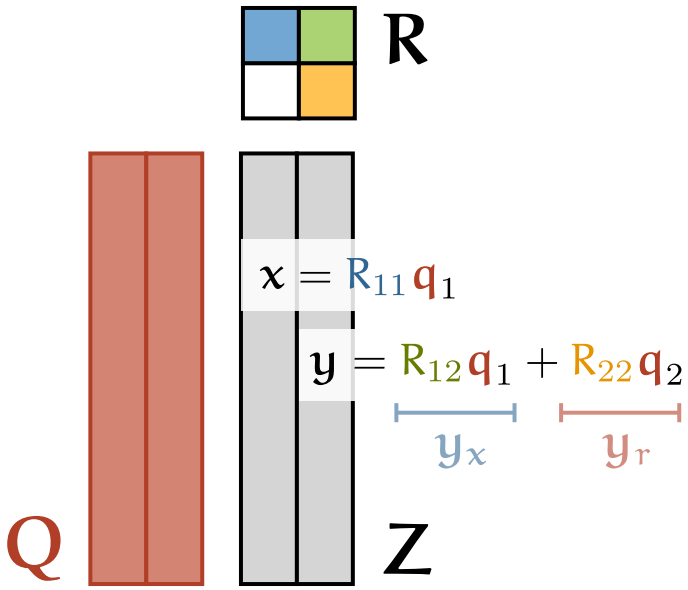
Intuitively, it seems like a good bet that $\mathbf{y}$ will convergence to the second most dominant eigenvector. To help us analyze this question, and to build a foundation for later algorithms, we'll first take this algorithm and rewrite it in matrix operations.

The first line is easy: if we create a new $n \times 2$ matrix $[\,\mathbf{x}\ \mathbf{y}\,]$ with vectors $\mathbf{x}$ and $\mathbf{y}$ as columns, then the matrix multiplication $\mathbf{A}[\,\mathbf{x}\ \mathbf{y}\,]$ 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

$$\mathbf{Q} \leftarrow \begin{bmatrix} \mathbf{x} & \mathbf{y} \end{bmatrix}$$

loop:

$$\mathbf{Z} \leftarrow \mathbf{AQ}$$

$$\mathbf{Q}, \mathbf{R} \leftarrow \text{qr}(\mathbf{Z})$$

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 $\mathbf{v}_1, \dots, \mathbf{v}_n$ which form a basis for $\mathbb{R}^n$. That means that for our starting vector $\mathbf{x}_0$, there are scalars $c_1, \dots, c_n$ so that

$$\mathbf{x}_0 = c_1\mathbf{v}_1 + \dots + c_n\mathbf{v}_n$$

and similarly, for our starting vector $\mathbf{y}_0$ there are scalars $d_1, \dots, d_n$ so that:

$$\mathbf{y}_0 = d_1 \boldsymbol{v}_1 + \dots + d_n \boldsymbol{v}_n \,.$$

Imagine that we set $\mathbf{x}_0 = \boldsymbol{v}_1$, 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

$$\begin{aligned} \mathbf{y}_x &= d_1 \boldsymbol{v}_1 &&+ 0 &&+ \dots &&+ 0 \\ \mathbf{y}_r &= 0 &&+ d_2 \boldsymbol{v}_2 &&+ \dots &&+ d_n \boldsymbol{v}_n \,. \end{aligned}$$

> *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 $(d_1, \dots, d_n)$, so its projection onto the first eigenvector $\boldsymbol{v}_1$ is $(d_1, 0, \dots, 0)$ and the corresponding rejection is $(0, d_2, \dots, d_n)$.*

Once we've rejected $\boldsymbol{v}_1$, we can see that multiplication by $\mathbf{A}$ will never result in a vector with a non-zero component in the $\boldsymbol{v}_1$ direction:

$$\mathbf{A}\left(0\boldsymbol{v}_1 + d_2\boldsymbol{v}_2 + \dots + d_n\boldsymbol{v}_n\right) = 0\lambda_1\boldsymbol{v}_1 + d_2\lambda_2\boldsymbol{v}_2 + \dots + d_n\lambda_n\boldsymbol{v}_n \,.$$

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 $d_1 = 0$.

$$\begin{aligned} \mathbf{A}^k\mathbf{y}_0 &= \mathbf{A}^k(d_1\boldsymbol{v}_1 + d_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3 + \dots + c_n\boldsymbol{v}_n) \\ &= 0 + d_2\mathbf{A}^k\boldsymbol{v}_2 + d_2\mathbf{A}^k\boldsymbol{v}_2 + \dots + d_n\mathbf{A}^k\boldsymbol{v}_n \\ &= d_2{\lambda_2}^k\boldsymbol{v}_2 + d_3{\lambda_3}^k\boldsymbol{v}_3 + \dots + d_n{\lambda_n}^k\boldsymbol{v}_n \\ &= d_2{\lambda_2}^k\left(\boldsymbol{v}_2 + \frac{d_3}{d_2}\frac{{\lambda_3}^k}{{\lambda_2}^k}\boldsymbol{v}_3 + \dots + \frac{d_n}{d_2}\frac{{\lambda_n}^k}{{\lambda_2}^k}\boldsymbol{v}_n\right) \end{aligned} \,.$$

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

$$y \to d_2 \lambda_2 v_2 \,.$$

Essentially, we are performing the same algorithm as before, but by projecting away from v_1, we are ensuring that v_1 can't dominate. The next most dominant eigenvector, v_2 automatically pops out.

In practice, we don't need to set $x = v_1$. The iteration on x is entirely the same as what we saw in the power iteration, so we know that x will converge to 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.

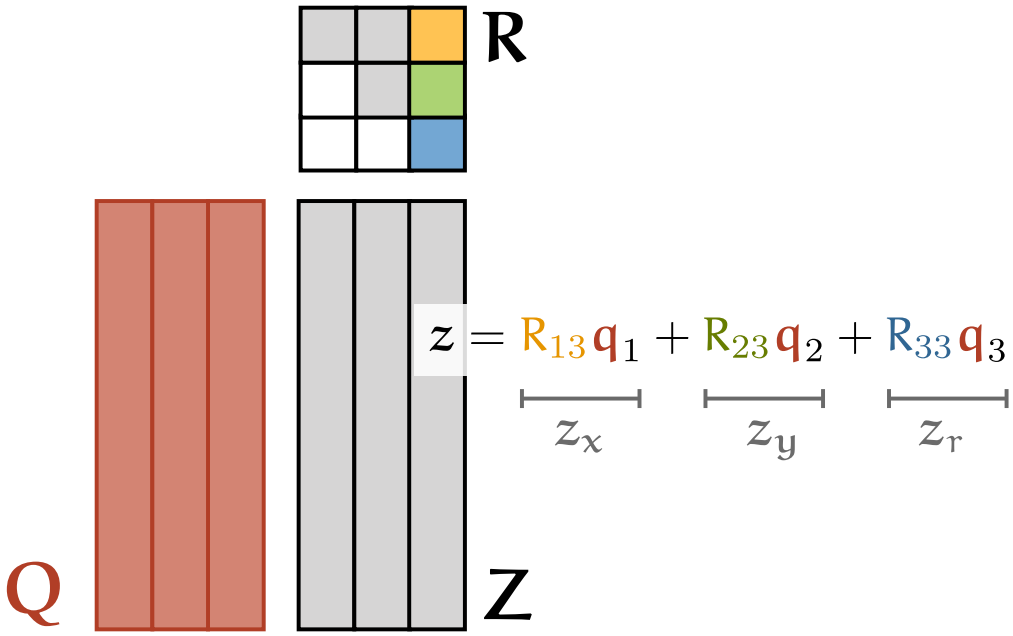
Adding even more eigenvectors

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:

$$Q \leftarrow \begin{bmatrix} x_1 & \dots & x_k \end{bmatrix}$$

loop:

$$Z \leftarrow AQ$$

$$Q, R \leftarrow \text{qr}(Z)$$

If we use this algorithm to compute the principal components, 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 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 A be any $n \times n$ matrix. A represents a map on $\mathbb{R}^n$. Now imagine that we have another basis on $\mathbb{R}^n$, represented by the invertible matrix 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 B.

What is the relation between A and B? We know that the map A should be equivalent to mapping to the basis P, applying the map B and mapping back to the standard basis. Composing these operations gives us

$$A = PBP^{-1}.$$

Any two square matrices A and B for which this relation holds—for some 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 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 A can be decomposed as

$$A = PDP^T$$

with D diagonal, and P orthogonal (implying $P^T = P^{-1}$). 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

$$A = QRQ^T.$$

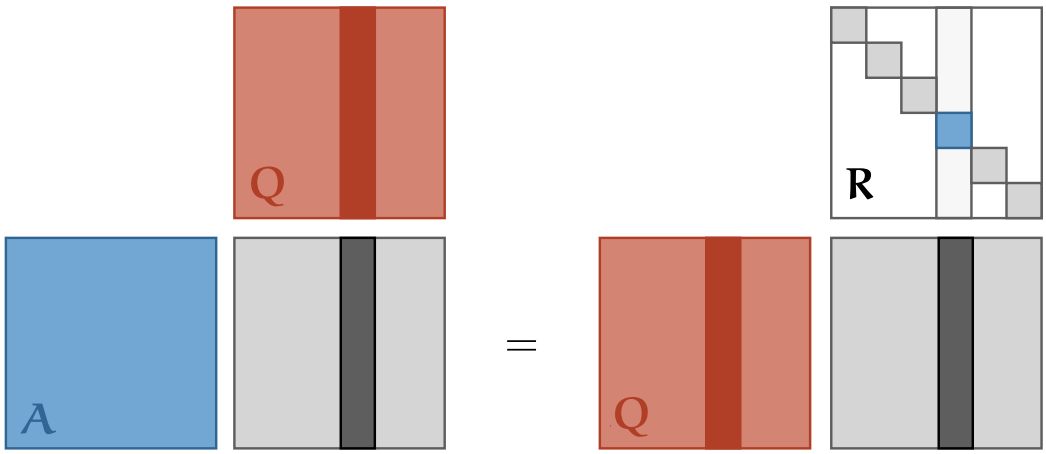
This tells us that A is similar to some triangular matrix R. Triangular matrices, like diagonal ones, have their eigenvalues along the diagonal, so we can read the eigenvalues of A off the diagonal of R.

To show that $\mathbf{Q}$ contains the eigen*vectors* of $\mathbf{A}$ we need another assumption: that $\mathbf{A}$ is symmetric: $\mathbf{A} = \mathbf{A}^\mathsf{T}$. If we know that, then we have $\mathbf{R} = \mathbf{Q}^\mathsf{T}\mathbf{A}\mathbf{Q}$ from rewriting the similarity relationship, and $\mathbf{Q}^\mathsf{T}\mathbf{A}^\mathsf{T}\mathbf{Q} = \mathbf{R}^\mathsf{T}$ from transposing both sides. Putting these together, we get:

$$\mathbf{R} = \mathbf{Q}^\mathsf{T}\mathbf{A}\mathbf{Q} = \mathbf{Q}^\mathsf{T}\mathbf{A}^\mathsf{T}\mathbf{Q} = \mathbf{R}^\mathsf{T}.$$

Which tells us that if $\mathbf{A}$ is symmetric, $\mathbf{R}$ is symmetric too, so it must be diagonal.

Why does this prove that $\mathbf{Q}$ contains the eigenvectors? Rewrite the similarity to $\mathbf{A}\mathbf{Q} = \mathbf{Q}\mathbf{R}$. This shows that multiplying $\mathbf{A}$ by the i-th column of $\mathbf{Q}$, is equivalent to multiplying it by the i-th diagonal element of $\mathbf{R}$. This is the i-th eigenvalue, so the corresponding column of $\mathbf{Q}$ must be the corresponding eigenvector.



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:

$$\mathbf{Z} \leftarrow \mathbf{A}$$

loop:

$$\mathbf{Q}, \mathbf{R} \leftarrow \text{qr}(\mathbf{Z})$$
$$\mathbf{Z} \leftarrow \mathbf{R}\mathbf{Q}$$

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 sequence $\mathbf{Z}_0, \mathbf{Z}_1, \ldots$ and similarly for the $\mathbf{Q}$'s and $\mathbf{R}$'s. We then have

$$\begin{aligned}
\mathbf{Z}_i &= \mathbf{R}_i \mathbf{Q}_i && \text{line 4 in iteration } i \\
&= \mathbf{Q}_i{}^\mathsf{T} \mathbf{Q}_i \mathbf{R}_i \mathbf{Q}_i && \text{because } \mathbf{Q}_i{}^\mathsf{T} \mathbf{Q}_i = \mathbf{I} \\
&= \mathbf{Q}_i{}^\mathsf{T} \mathbf{Z}_{i-1} \mathbf{Q}_i && \text{line 3 in iteration } i-1\,.
\end{aligned}$$

Note the key principle: for every $\mathbf{Q}$ and $\mathbf{R}$ in the sequence, $\mathbf{Q}\mathbf{R}$ is the previous $\mathbf{Z}$ and $\mathbf{R}\mathbf{Q}$ 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 $\mathbf{A} = \mathbf{Q}'\mathbf{D}\mathbf{Q}'^{\mathsf{T}}$, or equivalently $\mathbf{D} = \mathbf{Q}'^{\mathsf{T}}\mathbf{A}\mathbf{Q}'$.

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 $\mathbf{D}_i = \mathbf{Q}_i'^{\mathsf{T}}\mathbf{A}\mathbf{Q}_i'$.

Note that this is a different sequence from the intermediate values $\mathbf{R}_i'$ computed by the orthogonal algorithm. There, we had $\mathbf{A} = \mathbf{Q}_i'\mathbf{R}_i'\mathbf{Q}_{i-1}'^{\mathsf{T}}$ or equivalently

$$\mathbf{R}_i' = \mathbf{Q}_i'^{\mathsf{T}}\mathbf{A}\mathbf{Q}_{i-1}'.$$

$\mathbf{R}_i'$ 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 $\mathbf{D}_i$.

$\mathbf{D}_i$ 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 $\mathbf{D}_i$ at 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

$$\begin{aligned} \mathbf{D}_i = \mathbf{Z}_i &= \mathbf{Q}_i{}^\intercal\mathbf{Z}_{i-1}\mathbf{Q}_i = \mathbf{Q}_i{}^\intercal\mathbf{Q}_{i-1}{}^\intercal\mathbf{Z}_{i-2}\mathbf{Q}_{i-1}\mathbf{Q}_i = \dots \\ &= \mathbf{Q}_i^{\Pi\intercal}\mathbf{A}\mathbf{Q}_i^{\Pi} \end{aligned}$$

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

$$\mathbf{Q}_i^\intercal \mathbf{D}_i \mathbf{Q}_i^\intercal \mathbf{T} = \mathbf{A}\,.$$

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.

COMPUTING THE SVD

In the previous blog post of this series, 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.

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 $\mathbf{X}^\intercal\mathbf{X}$ 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 $\mathbf{X}^\intercal\mathbf{X}$.

As we saw in the previous part, the eigenvectors of $\mathbf{X}^\intercal\mathbf{X}$ are the *singular* vectors of $\mathbf{X}$. This immediately gives us an algorithm for computing singular

vectors: take any rectangular matrix $\mathbf{M}$, compute $\mathbf{M}^\mathsf{T}\mathbf{M}$ and apply the eigenvector algorithms we developed above.

We can fill in this $\mathbf{M}^\mathsf{T}\mathbf{M}$ and see what it tells us about the algorithm. Let's start with the power iteration. When we apply this to $\mathbf{M}^\mathsf{T}\mathbf{M}$, we get

$$
\begin{aligned}
&\mathbf{x} \leftarrow \mathbf{x}_0 \\
&\textbf{loop}: \\
&\quad \mathbf{x} \leftarrow \mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{x} \\
&\quad \mathbf{x} \leftarrow \mathbf{x}/\|\mathbf{x}\| \,.
\end{aligned}
$$

If we now separate the two matrix multiplications, by $\mathbf{M}$ and then by $\mathbf{M}^\mathsf{T}$ in two steps, we get

$$
\begin{aligned}
&\mathbf{x} \leftarrow \mathbf{x}_0 \\
&\textbf{loop}: \\
&\quad \mathbf{y} \leftarrow \mathbf{M}\mathbf{x} \\
&\quad \mathbf{x} \leftarrow \mathbf{M}^\mathsf{T}\mathbf{y} \\
&\quad \mathbf{x} \leftarrow \mathbf{x}/\|\mathbf{x}\| \,.
\end{aligned}
$$

This is the same algorithm as before, just with the matrix multiplication separated in two steps. Next, we will add another normalization step. This changes the algorithm, but we will show that the effect is negligable.

$$
\begin{aligned}
&\mathbf{x} \leftarrow \mathbf{x}_0 \\
&\textbf{loop}: \\
&\quad \mathbf{y} \leftarrow \mathbf{M}\mathbf{x} \\
&\quad \mathbf{y} \leftarrow \mathbf{y}/\|\mathbf{y}\| \\
&\quad \mathbf{x} \leftarrow \mathbf{M}^\mathsf{T}\mathbf{y} \\
&\quad \mathbf{x} \leftarrow \mathbf{x}/\|\mathbf{x}\| \,.
\end{aligned}
$$

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 $\mathbf{v}$ is a right singular vector of $\mathbf{M}$ and $\mathbf{u}$ its corresponding left singular vector, with σ the corresponding singular value, then

$$\mathbf{M}\mathbf{v} = \sigma\mathbf{u}\,.$$

So, if $\mathbf{x}$ has converged to a right singular vector of $\mathbf{M}$, then $\mathbf{M}\mathbf{x}$, after normalization, is the corresponding left singular vector. This shows that the algorithm actually gives us both singular vectors. The corresponding singular *value* is $\|\mathbf{M}\mathbf{x}\|$.

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 $\mathbf{x}'$ along for the iteration, and that for the second vector—in addition to normalizing it every iteration—we make it orthogonal to $\mathbf{x}$.

We're slightly shuffling our variables here. $\mathbf{x}'$ is the vector we previously called $\mathbf{y}$ when we extended the power iteration to multiple eigenvectors. $\mathbf{y}$ is now the intermediate variable we've introduced that results from multiplication by $\mathbf{M}$.

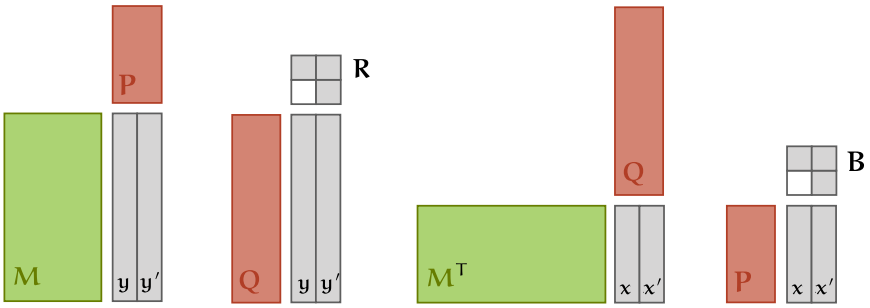
To follow the spirit of the power iteration algorithm for the SVD, we do this twice: we multiply $\mathbf{x}$ and $\mathbf{x}'$ by $\mathbf{M}$, resulting in $\mathbf{y}$ and $\mathbf{y}'$. We project $\mathbf{y}'$ away from $\mathbf{y}$, so that they are orthogonal to each other, and normalize both. Then, we multiply both by $\mathbf{M}^\intercal$, 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 $\mathbf{A}\begin{bmatrix}\mathbf{x}\,\mathbf{x}'\end{bmatrix}$.

For $\mathbf{M}\begin{bmatrix}\mathbf{x}\,\mathbf{x}'\end{bmatrix}$, our logic holds the same as before: we get two orthogonal unit vectors, which can serve as the columns of $\mathbf{Q}$. To transform these by a matrix $\mathbf{R} \in \mathbb{R}^{2\times 2}$ so that $\mathbf{M}\begin{bmatrix}\mathbf{x}\,\mathbf{x}'\end{bmatrix} = \mathbf{QR}$, the first column of $\mathbf{R}$ only needs to stretch $\mathbf{x}$ uniformly (it only needs a value on the diagonal). The second should express $\mathbf{M}\mathbf{x}'$ as two components. One component in the direction of $\mathbf{M}\mathbf{x}$, which gives us a scalar on the first row, and one component orthogonal to $\mathbf{M}\mathbf{x}$ which gives us a scalar on the second row. Therefore, $\mathbf{R}$ is upper triangular.

When we apply the second step of the iteration, we start with two orthogonal unit vectors $\mathbf{y}, \mathbf{y}'$ and we compute $\mathbf{M}^\intercal\begin{bmatrix}\mathbf{y}\,\mathbf{y}'\end{bmatrix}$, 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 $\mathbf{P}$ and the upper triangular one $\mathbf{B}$

$$\mathbf{M}^\intercal\begin{bmatrix}\mathbf{y}\,\mathbf{y}'\end{bmatrix} = \mathbf{PB}$$



The two-column orthogonal algrithm for the SVD in four steps.

We can now, as before, add a third vector $\mathbf{x}''$, a fourth, a fifth, and so on. If we make each orthogonal to all prior vectors, we end up with a matrix $\mathbf{Q}$ with orthogonal columns (or rows). The triangular structure of the matrix $\mathbf{R}$ (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:

$$\mathbf{P} = \begin{bmatrix} \mathbf{x}_0^1 & \dots & \mathbf{x}_0^k \end{bmatrix}$$

loop:

$$\mathbf{Y} \leftarrow \mathbf{M}\mathbf{P}$$
$$\mathbf{Q}, \mathbf{R} \leftarrow \text{qr}(\mathbf{Y})$$
$$\mathbf{X} \leftarrow \mathbf{M}^\intercal \mathbf{Q}$$
$$\mathbf{P}, \mathbf{B} \leftarrow \text{qr}(\mathbf{X})$$

> *We can also think of the second QR decomposition as an LQ decomposition of $\mathbf{Q}^\intercal\mathbf{M}$. 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 $M^\intercal M$, and therefore to the first right singular vector of M.

What we haven't shown yet, is that this also holds true for the other vectors on 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 M. One $m \times m$ matrix V with columns $v_1, \dots, v_m$ that spans the input space of M and one $n \times n$ matrix U with columns $u_1, \dots, u_n$ which spans the output space of M We'll call these the right and left *singular bases* of M respectively.

By definition, for each of these there is a singular value σ_i so that $Mv_i = u_i \sigma_i$.

The key insight here, is that when we have the singular bases, we can express the operation of 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 x in the right singular basis V, we get some an expression like

$$x = c_1 v_1 + \dots + c_m v_k \,.$$

For every right singular vector i, we get come component c_i expressing how much of x projects onto v_i. If we then multiply $y = Mx$, we can express y in the left singular basis as:

$$y = d_1 u_1 + \dots + d_n u_k \,.$$

The key property of these two bases is that the vector $c_i v_i$ is mapped to the vector $d_i u_i$, *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* $\text{rank}(\mathbf{M})$ *right and left singular vectors respectively.*

For what's coming up, we will need to reverse this picture as well. Note first that $\mathbf{M}^\mathsf{T} = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^\mathsf{T}$ (by simply transposing both sides of the SVD decomposition). This is *also* an SVD, but of $\mathbf{M}^\mathsf{T}$ and with right singular vectors $\mathbf{U}^\mathsf{T}$ and left singular vectors $\mathbf{V}^\mathsf{T}$. By definition of the SVD, we see that this implies that $\mathbf{M}^\mathsf{T}\mathbf{u}_i = \mathbf{v}_i\sigma_i$. That is, when we transform back from $\mathbf{y}$ to $\mathbf{x}$, the singular vectors are also mapped onto one another.

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 $c_1, \dots, c_m$, we have, for $k = \text{rank}(\mathbf{M})$:

$$\mathbf{p}_1 = c_1\mathbf{v}_1 + \dots + c_m\mathbf{v}_k\,.$$

We know that $\mathbf{p}_1$ converges to $\mathbf{v}_1$. That means we will get arbitrarily close to the situation where

$$\mathbf{p}_1 = \mathbf{v}_1 + 0\mathbf{v}_2 + \dots + 0\mathbf{v}_k\,.$$

At that point, when we compute $\mathbf{Y} = \mathbf{M}\mathbf{P}$, in the first line of our iteration, we will get, for the first column $\mathbf{y}_1$ of $\mathbf{Y}$

$$\mathbf{y}_1 = \mathbf{M}\mathbf{v}_1 = \mathbf{u}_1\sigma_1\,.$$

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 it to recover y_1) becomes σ_1.

Next comes the multiplication $X = M^\intercal Q$. Focusing only on the first column for now, this is $x_1 = M^\intercal u_1$. From the SVD of the transpose above we see that $M^\intercal u_1 = v_1 \sigma_1$. As before, u_1 is a right singular vector of $M^\intercal$, 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:

$$p_2 = c_1 v_1 + c_2 v_2 + \ \dots \ + c_m v_k \, .$$

In the first line of the iteration, we multiply M by P. This gives us:

$$\begin{aligned} y_2 = M p_2 &= c_1 M v_1 + c_2 M v_2 + \dots + c_k M v_k \\ &= c_1 \sigma_1 u_1 + c_2 \sigma_2 u_2 + \dots + c_k \sigma_k u_k \, . \end{aligned}$$

Next, the QR decomposition projects this vector away from the first vector (which we assume is v_1). This means that the remainder is

$$0 + c_2 \sigma_2 u_2 + \dots + c_m \sigma_n u_n$$

which we then normalize to get

$$q_2 R_{22} = 0 + c_2 \sigma_2 u_2 + \dots + c_m \sigma_n u_n \, .$$

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 $M^\intercal$. This gives us

$$\begin{aligned} M^\intercal q_2 &= 0 + e_2 M^\intercal u_2 + \ldots + e_k M^\intercal u_k \\ &= 0 + e_2 \sigma_2 v_2 + \ldots + e_k \sigma_k u_k \,. \end{aligned}$$

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 $P' = \begin{bmatrix} p_2 & \ldots & p_m \end{bmatrix}$. 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.

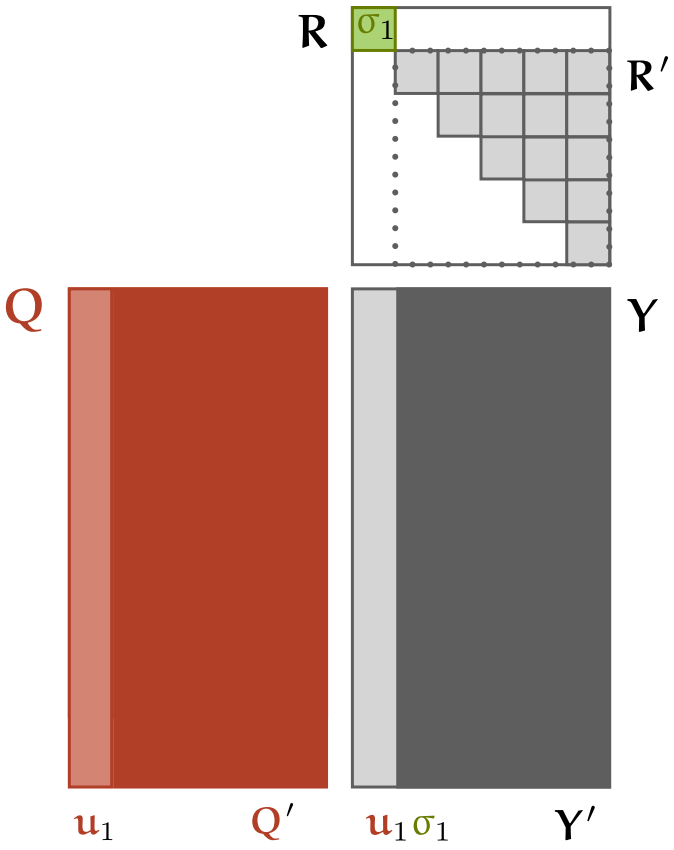
In the first line of the algorithm, we multiply $Y' \leftarrow MP'$. From the basic definition of matrix multiplication, we can conclude that Y' consists of the

rightmost vectors of Y, without the first one. We can also conclude that all columns of Y' are orthogonal to u_1.

> *Each column of P' is orthogonal to 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 M, the multiplier for the first left eigenvector u_i is zero as well.*

Second, we perform the QR decomposition $Q', R' \leftarrow \text{qr}(X')$. We know that span X' is a subspace orthogonal to u_1, so Q' will be an orthogonal basis for that subspace. That is $Q = \begin{bmatrix} v_1 & q'_1 & \dots & q'_m \end{bmatrix}$.

How are R and R' related? R_{11} is a scaling factor for v_1, and the remaining R_{1i} show how much of each of the other vectors of X projects onto v_1. Under our assumptions, these are all orthogonal to 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 R, we get R'



The QR decomposition of the whole matrix Y contains the QR decomposition of the submatrix Y' (if its first column is a scalar multiple of u_1).

In the third line of the algorithm, we compute $\mathbf{Y}' \leftarrow \mathbf{M}^\mathsf{T}\mathbf{Q}'$. 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 $\mathbf{P} = \begin{bmatrix}\mathbf{v}_1 & \mathbf{p}'_1 & \dots & \mathbf{p}'_m\end{bmatrix}$. 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 $\mathbf{P} = \begin{bmatrix}\mathbf{v}_1 & \mathbf{p}'_1 & \dots & \mathbf{p}'_m\end{bmatrix}$, 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:

$$\mathbf{p}_2 \leftarrow L_{22}\mathbf{M}^\mathsf{T}R_{22}\mathbf{M}\mathbf{p}_2 = L_{22}R_{22}\mathbf{M}^\mathsf{T}\mathbf{M}\mathbf{p}_2 \,.$$

After k iterations, we get

$$\mathbf{p}_2^k = L_{22}\mathbf{M}^\mathsf{T}R_{22}\mathbf{M}\mathbf{p}_2 = (L_{22}R_{22})^k(\mathbf{M}^\mathsf{T}\mathbf{M})^k\mathbf{p}_2^0 \,.$$

This should look familiar. It shows that all we are doing is iteratively multiplying by $\mathbf{M}^\mathsf{T}\mathbf{M}$, and occasionally normalizing. Normally, such an

iteration would converge to the *first* eigenvector of $\mathbf{M}^\mathsf{T}\mathbf{M}$, 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 $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ 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 $\boldsymbol{v}_1$, 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 $\mathbf{p}_1 = \boldsymbol{v}_1$.

The QR algorithm for the SVD

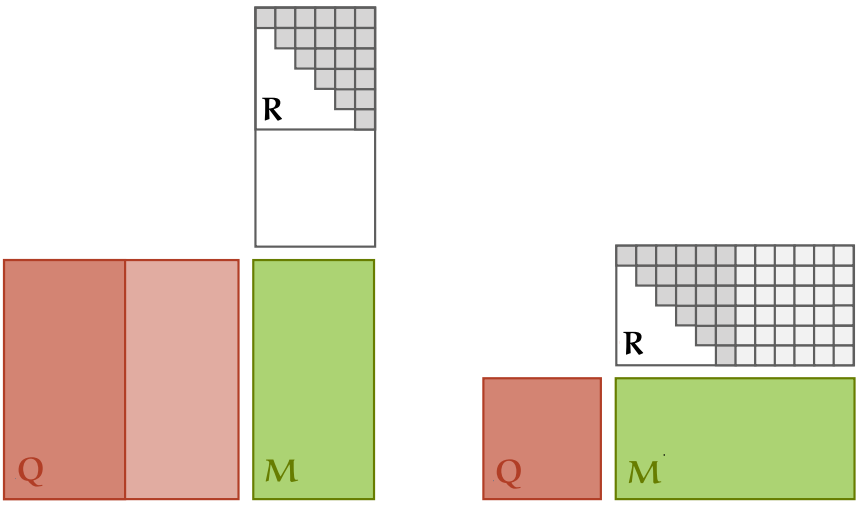
If only for the sake of symmetry, it would be nice if there were an SVD version of the QR iteration algorithm. Happily, it turns out there, is, although we need to be careful to translate the logic of the QR iteration in the correct way.

As before, we'll show the algorithm first, and then work out how it relates to the orthogonal iteration.

$$
\begin{aligned}
&\mathbf{X}, \mathbf{Y} \leftarrow \mathbf{M}, \mathbf{M}^\mathsf{T} \\
&\textbf{loop:} \\
&\quad \mathbf{Q}, \mathbf{R} \leftarrow \text{qr}(\mathbf{X}) \\
&\quad \mathbf{P}, \mathbf{B} \leftarrow \text{qr}(\mathbf{Y}) \\
&\quad \mathbf{X} \leftarrow \mathbf{R}\mathbf{P} \\
&\quad \mathbf{Y} \leftarrow \mathbf{B}\mathbf{Q}
\end{aligned}
$$

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 $\mathbf{Q}$ 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 $\mathbf{Q}$ is square. To make the multiplication work, we then extend $\mathbf{R}$ 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 $\mathbf{R}$ far enough to make the multiplication of $\mathbf{Q}$ and $\mathbf{R}$ 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 $\mathbf{Q}$. We express each remaining column of the matrix as a linear combination of the columns of $\mathbf{Q}$, giving us an additional column of $\mathbf{R}$ until $\mathbf{R}$ is as wide as $\mathbf{M}$.

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

$$\begin{aligned}
\mathbf{X}_i &= \mathbf{Q}'_i{}^{\mathsf{T}}\mathbf{M}\mathbf{P}'_i & \mathbf{B}_i &= \mathbf{P}'_i{}^{\mathsf{T}}\mathbf{M}^{\mathsf{T}}\mathbf{Q}'_{i+1} && \text{by definition}\\
&= \mathbf{Q}'_i{}^{\mathsf{T}}\mathbf{M}\mathbf{P}'_{i-1}{}^{\mathsf{T}}\mathbf{P}'_{i-1}\mathbf{P}'_i & &= \mathbf{P}'_i{}^{\mathsf{T}}\mathbf{M}^{\mathsf{T}}\mathbf{Q}'_i{}^{\mathsf{T}}\mathbf{Q}'_i\mathbf{Q}'_{i+1} && \text{insert } \mathbf{I}\\
&= \mathbf{Q}'_i{}^{\mathsf{T}}\mathbf{Q}'_i\mathbf{R}'_i\mathbf{P}'_{i-1}\mathbf{P}'_i & &= \mathbf{P}'_i{}^{\mathsf{T}}\mathbf{P}'_i\mathbf{B}'_i\mathbf{Q}'_i\mathbf{Q}'_{i+1} && \text{from orth. alg.}\\
&= \mathbf{R}'_i\mathbf{P}'_{i-1}\mathbf{P}'_i & &= \mathbf{B}_i\mathbf{Q}'_i\mathbf{Q}'_{i+1}
\end{aligned}$$

And there we have the proof of our algorithm. 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 $\mathbf{Q}$ and $\mathbf{P}$. See the link at the top for implementations these algorithms, which contain these details.*

Conclusion

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. You more you learn and discover, 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 part, 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 parts 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 part of this

series 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 in to 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.

Appendix

Uniqueness of the QR decomposition

In our discussion, we occasionaly 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 $\mathbf{M}$ is full rank, a QR decomposition with only positive elements on the diagonal of $\mathbf{R}$ 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 $\mathbf{Q}$ and changing the sign of the corresponding row of $\mathbf{R}$. 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 $\mathbf{R}$ where the diagonal element is negative. We'll call this a *positive QR decomposition*.

> For an $n \times m$ matrix $\mathbf{M}$ with linearly independent columns, the positive QR decomposition $\mathbf{QR} = \mathbf{M}$ is unique.
>
> *Proof.* Let $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{P}$, $\mathbf{B}$ be two positive QR decompositions of $\mathbf{M}$. This suggests that $\mathbf{QR} = \mathbf{PB}$ and thus $\mathbf{P}^\intercal\mathbf{Q} = \mathbf{BR}^{-1}$.
>
> > *Note that $\mathbf{R}$ is invertible because $\mathbf{M}$'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 $\mathbf{R}$ 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 $\mathbf{P}^\intercal\mathbf{P} = \mathbf{I}$, 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 $\mathbf{RR}^{-1} = \mathbf{I}$, 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 $\mathbf{A}\mathbf{B}$ 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. It's 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 $\mathbf{R} = \mathbf{B}$. □

COMPUTING THE EIGENDECOMPOSITION AND THE SING…