Numerical Methods for Computing Eigenvectors

Eric Zheng

21-241 Final Project

December 6, 2019

Abstract

In this document, I present some background on numerical methods for computing the eigenvectors and singular vectors of matrices. Julia implementations of the power and QR methods are given, and the two algorithms are compared.

Contents

1 Background: Eigenvectors and Eigenvalues

I assume that the reader is familiar with linear algebra at the college introductory level. In this section, I review a little bit of background to motivate the algorithms that will be developed in subsequent sections. We begin by recalling definition 1.1.

Definition 1.1: eigenvectors and eigenvalues

Consider an arbitrary $n \times n$ matrix A. For some $\mathbf{x} \in \mathbb{R}^n$ (with $\mathbf{x} \neq \mathbf{0}$), we say that $\mathbf{x}$ is an *eigenvector* of A iff, for some $\lambda \in \mathbb{R}$, $A\mathbf{x} = \lambda\mathbf{x}$. We denote λ as the corresponding *eigenvalue* of A.

From definition 1.1 follows immediately a somewhat naive way to compute the eigenvalues of a given matrix. Note that

$$A\mathbf{x} = \lambda\mathbf{x} \implies A\mathbf{x} - \lambda\mathbf{x} = \mathbf{0}$$

and $\lambda\mathbf{x} = \lambda I\mathbf{x}$, so we have

$$A\mathbf{x} - \lambda I\mathbf{x} = (A - \lambda I)\mathbf{x} = \mathbf{0}.$$

That is to say, $\lambda \in \mathbb{R}^n$ is an eigenvalue of A if and only if $A - \lambda I$ has a non-trivial null space. A matrix has a non-trivial null space if and only if it is singular, so we require that $\det(A - \lambda I) = 0$. This result is stated in theorem 1.1.

Theorem 1.1: computing eigenvalues as polynomial roots

Some $\lambda \in \mathbb{R}^n$ is an eigenvalue of the $n \times n$ matrix A if and only if $\det(A - \lambda I) = 0$. We call $\det(A - \lambda I)$ the *characteristic polynomial* for A. The problem then becomes identifying the roots of this polynomial.

Once the eigenvalues have been computed, we can find the corresponding eigenvectors by finding the null space of $A - \lambda I$, for example by using the reduced row echelon form. The key drawback of this method is that polynomial root-finding is sensitive to small numerical errors [6].

However, the equivalence of eigenvalue-finding and polynomial root-finding gives some insight into how we could approach the problem with more advanced methods. It is known that polynomials of degree five and higher do not in general have an exact solution by radicals [14], although we can use iterative methods to get arbitrarily good approximations of these roots. In the following sections, we will apply similarly iterative methodologies to find the eigenvectors and eigenvalues of matrices.

2 The Power Method

2.1 Mathematical Formulation

Definition 2.1: dominant eigenvector

Let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of the matrix A, and let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be corresponding eigenvalues. We call $\mathbf{x}_i$ a *dominant eigenvector* if $|\lambda_i| > |\lambda_j|$ whenever $i \neq j$. The corresponding λ_i is called the *dominant eigenvalue*.

The power method for computing eigenvectors takes successive powers of the matrix A^k until some stopping criterion is reached. As k grows large, the columns of A will approach the dominant eigenvector of A. This is stated in theorem 2.1.

Theorem 2.1: the power method

If A is an $n \times n$ diagonalizable matrix with a dominant eigenvector $\mathbf{x}$, then the columns of A^k approach a multiple of $\mathbf{x}$ as k grows arbitrarily large. (More precisely, at least one column does so.)

Proof. Since A is diagonalizable, let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be a basis of eigenvectors for $\mathbb{R}^n$, where we order the eigenvectors so that $\mathbf{x}_1$ is a dominant eigenvector. Now since the $\mathbf{x}_i$'s form a basis, any $\mathbf{v} \in \mathbb{R}^n$ can be expressed as the linear combination

$$\mathbf{v} = c_1\mathbf{x}_1 + \cdots + c_n\mathbf{x}_n.$$

Then by linearity, we have

$$\begin{aligned} A^k\mathbf{v} &= A^k(c_1\mathbf{x}_1 + \cdots + c_n\mathbf{x}_n) \\ &= A^k c_1\mathbf{x}_1 + \cdots + A^k c_n\mathbf{x}_n \\ &= \lambda_1^k c_1\mathbf{x}_1 + \cdots + \lambda_n^k c_n\mathbf{x}_n. \end{aligned}$$

But since $|\lambda_1| > |\lambda_i|$ for all $i \geq 2$, we see that the first term $\lambda_1^k c_1\mathbf{x}_1$ dominates as k grows very large (as long as $c_1 \neq 0$). Hence for large k, $A^k\mathbf{v} \approx \lambda_1^k c_1\mathbf{x}_1$, if $c_1 \neq 0$. Taking $\mathbf{v}$ to be the standard basis vectors then produces the desired result, since at least one of the $\mathbf{e}_i$'s must have a component along $\mathbf{x}_1$. □

One issue with the power method is that it assumes that A is both diagonalizable and has a dominant eigenvector. These flaws are highlighted in examples 2.1 and 2.2. Another example of the power method's failure to converge is a rotation matrix [1], which does not typically have real eigenvalues.

Example 2.1: power method on a non-diagonalizable matrix

The power method can fail when a matrix is not diagonalizable. Consider the matrix

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

The only real eigenvalue of A is $\lambda = 1$, with corresponding eigenvalue $\mathbf{x} = (0, 1, 1)$. But consider the powers of A:

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

Since $A^4 = I$, higher powers will cycle and we never converge.

Example 2.2: power method without a dominant eigenvalue

The power method can fail when a matrix does not have a dominant eigenvalue, even if it is diagonalizable. Consider the matrix

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

Now A is clearly diagonalizable (it's symmetric), but its eigenvalues are $\lambda = \pm 1$, so A does not have a dominant eigenvalue. If we attempt the power method, we find that $A^2 = I$, so the successive powers A^k will oscillate between A (when k is odd) and I (when k is even). Neither of these contain the eigenvectors of A, which are $(1, 1)$ and $(1, -1)$.

Note that diagonalizability is a sufficient condition for the power method to converge (when combined with a dominant eigenvalue), but it is not necessary. Example 2.3 gives a matrix that is not diagonalizable yet converges under the power method.

Example 2.3: power method despite non-diagonalizability

Consider the matrix

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

This matrix has only one eigenvalue $\lambda = 1$ with eigenvectors $(0, 1)$ and $(0, -1)$, which are not independent. But an easy induction reveals that

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

whose columns indeed converge to the eigenvector $(0, 1)$ as k grows large.

2.2 Convergence Analysis

The power method is a particularly elegant method for computing the dominant eigenvector of a matrix, but how efficient is it? In other words, as we take successive powers A^k, how quickly does λ_1 dominate the result? What matters here is comparatively how much larger $|\lambda_1|$ is than $|\lambda_2|$: the power method converges proportional to $|\lambda_1/\lambda_2|$ [7, p. 529].

More formally, we can prove this by an argument taken from [2]: if A is diagonalizable, then we can express any $\mathbf{v} \in \mathbb{R}^n$ as

$$\mathbf{v} = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n$$

where we assume $c_1 \neq 0$ for the algorithm to converge. Now after k steps, we have

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

where $|\lambda_i/\lambda_1| < 1$ whenever $i \geq 2$ since λ_1 is the dominant eigenvalue. Now as $k \to \infty$, the (λ_2/λ_1) term will dominate the error, so we expect the algorithm to converge proportionally to $|\lambda_2/\lambda_1|^k$.

We can test this numerically with the matrix

$$A = \begin{bmatrix} 23 & 5 & 2 \\ 5 & 23 & 2 \\ 2 & 2 & 26 \end{bmatrix}$$

which has eigenvalues $\lambda_1 = 30$, $\lambda_2 = 24$, and $\lambda_3 = 18$; the corresponding eigenvectors are $\mathbf{x}_1 = (1, 1, 1)$, $\mathbf{x}_2 = (1, 1, -2)$, and $\mathbf{x}_3 = (1, -1, 0)$. For the first twenty iterations of the power method, we get something like figure 1, which is fairly close to the expected $(24/30)^k = (4/5)^k$ decay rate. The Julia code for generating the plot is included in the appendix.

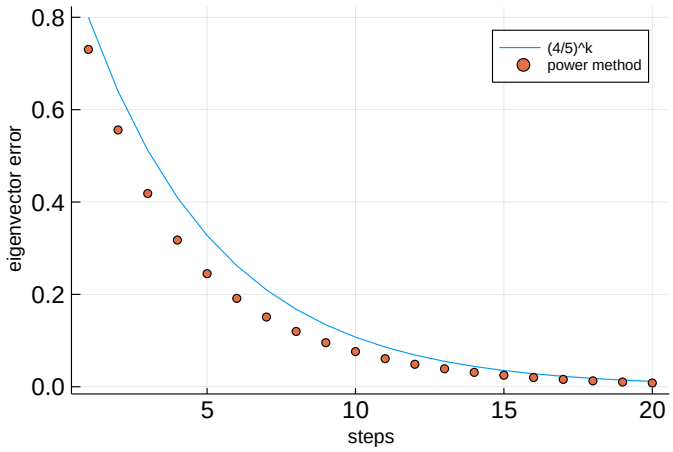

Figure 1: Power method error for twenty steps of the algorithm, plotted against the expected error given by $|\lambda_2/\lambda_1|^k$

2.3 Rayleigh Quotients

Up until now, we have developed a method for finding the eigenvectors of a matrix, but what about the eigenvalues? Based on definition 1.1, it is tempting to just take an eigenvector $\mathbf{x} \in \mathbb{R}^n$, compute $A\mathbf{x} = \lambda\mathbf{x}$, and then see how much the components of $\mathbf{x}$ were scaled to find λ.

The issue is that our algorithm only generates an approximation to $\mathbf{x}$, not $\mathbf{x}$ itself [1]. Hence each of the components will not be exactly scaled by λ: some might be a little larger than they should be, and some might be a little smaller. The next thing that comes to mind is to let μ_i be the amount by which each component of $\mathbf{x}$ was scaled and then just average the μ_i's. Unfortunately, this approach is sensitive to small errors, particularly around 0 [2], as demonstrated in example 2.4.

Example 2.4: sensitivity of averaging for finding λ

Consider the matrix

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

which has $\mathbf{x}_1 = (1, 0)$ and $\mathbf{x}_2 = (0, 1)$ with corresponding $\lambda_1 = 2$ and $\lambda_2 = 1$. Now suppose we have an eigenvector approximation $\hat{\mathbf{x}}_1 = (1, 0.00001)$. This is very close to the true dominant eigenvector:

$$\|\hat{\mathbf{x}}_1 - \mathbf{x}_1\| = 0.00001,$$

[1] In fact, even if we gave the algorithm infinite time to run, it is mathematically impossible for a computer with a finite alphabet to represent arbitrary reals, since $\mathbb{R}$ is uncountable.

yet we have

$$A\hat{\mathbf{x}}_1 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0.00001 \end{bmatrix} = \begin{bmatrix} 2 \\ 0.00001 \end{bmatrix}.$$

If we try to approximate λ_1 by diving component-wise and then averaging, we get:

$$\mu_1 = 2/1 = 2$$
$$\mu_2 = 0.00001/0.00001 = 1$$

which would give $\lambda_1 \approx 3/2$, which is significantly off from the true value $\lambda_1 = 2$, despite the very good eigenvector approximation.

So how should we compute eigenvalues given a corresponding eigenvector? A common way to do this is via the Rayleigh quotient, given in definition 2.2.

Definition 2.2: Rayleigh quotient [6]

If we have an approximation $\mathbf{x} \in \mathbb{R}^n$ for an eigenvector of A, then the *Rayleigh quotient* approximation for the corresponding eigenvalue λ is given by

$$\lambda \approx \frac{\mathbf{x}^T A\mathbf{x}}{\mathbf{x}^T\mathbf{x}}.$$

This can be thought of as the average of the μ_i's weighted by the square of each component [2], so errors around small components affect the overall eigenvalue approximation very little. For example, using the same numbers as example 2.4, we get

$$\lambda_1 \approx \frac{\hat{\mathbf{x}}_1^T A\hat{\mathbf{x}}_1}{\hat{\mathbf{x}}_1^T\hat{\mathbf{x}}_1} = \frac{2.0000000001}{1.0000000001} \approx 1.999999999$$

which is much closer to the true value of $\lambda_1 = 2$.

2.4 Deflation

One limitation is that the power method, as presented in theorem 2.1, only finds the dominant eigenvector of a matrix. For many physical systems, the behavior is determined mostly by the dominant eigenvector, so this is sufficient [6]. However, in the special case where A is symmetric, we can actually do better. By the spectral theorem (restated in theorem 4.1), we can take the spectral decomposition of symmetric A into orthogonal X and diagonal Λ, as in

$$A = X\Lambda X^T = \lambda_1\mathbf{x}_1\mathbf{x}_1^T + \cdots + \lambda_n\mathbf{x}_n\mathbf{x}_n^T$$

where we sort the eigenvalues in order of magnitude. Now applying the power method will yield $\mathbf{x}_1$ (which in turn can be used to find λ_1). But observe that

$$B_1 = A - \lambda_1\mathbf{x}_1\mathbf{x}_1^T = \lambda_2\mathbf{x}_2\mathbf{x}_2^T + \cdots + \lambda_n\mathbf{x}_n\mathbf{x}_n^T$$

is a symmetric matrix that has eigenvalues $\lambda_2, \ldots, \lambda_n$ and corresponding $\mathbf{x}_2, \ldots, \mathbf{x}_n$! ($B_1$ also has a zero eigenvalue, which is not important because it is the least in magnitude.) Hence we can apply the power method to B_1 to find $\mathbf{x}_2$ and λ_2. We can keep on going with

$$B_2 = B_1 - \lambda_2 \mathbf{x}_2 \mathbf{x}_2^T = \lambda_3 \mathbf{x}_3 \mathbf{x}_3^T + \cdots + \lambda_n \mathbf{x}_n \mathbf{x}_n^T$$

to which we can apply the power method to pick off $\mathbf{x}_3$ and λ_3. We keep on going until we have found all n eigenvectors, a technique known as *deflation*.

In fact, even if A is not symmetric, we can apply a similar deflation technique for finding all the eigenvalues of A, although we don't get any additional eigenvectors beyond the dominant one. The reader is referred to [3] for a full explanation, but a more general deflation procedure known as Hotelling's deflation is given in theorem 2.2. In the case when we take unit eigenvectors of a symmetric matrix, this procedure becomes the previously described deflation procedure. There exist other techniques that are even more robust against, for instance, rounding errors [9], but they are beyond the scope of this paper.

Theorem 2.2: Hotelling's deflation

Suppose A is a matrix with eigenvalues $\lambda_1, \ldots, \lambda_k$ and corresponding eigenvalues $\mathbf{x}_1, \ldots, \mathbf{x}_k$. Then the matrix

$$B = A - \frac{\lambda_1}{\mathbf{x}_1^T \mathbf{x}} \mathbf{x}_1 \mathbf{x}_1^T$$

has eigenvalues $\lambda_2, \ldots, \lambda_k$, although it does not necessarily have the same eigenvectors $\mathbf{x}_2, \ldots, \mathbf{x}_k$.

2.5 Optimization for Sparse Matrices

As we have developed it thus far, the power method involves computing large powers A^k. While we can make this somewhat efficient via tricks like exponentiation by squaring, matrix multiplication is an expensive thing to do; the best known algorithms run in roughly $O(n^{2.373})$ [4] for an $n \times n$ matrix. Can we do better?

A faster approach is to reframe the problem as a bunch of matrix-vector multiplications, which can be computed naively in $O(n^2)$. In fact, if A is a sparse matrix (i.e. most entires are zero), matrix-vector multiplication can become *extremely* efficient [1]. We can tweak theorem 2.1 slightly into theorem 2.3, whose proof is much the same as theorem 2.1.

Theorem 2.3: more efficient power method

Suppose A is an $n \times n$ diagonalizable matrix with a dominant eigenvalue λ and corresponding eigenvector $\mathbf{x}$. Then there exists some $\mathbf{v}_0 \in \mathbb{R}^n$ such

that

$$\mathbf{v}_k = A^k \mathbf{v}_0 = A\mathbf{v}_{k-1}$$

approaches a multiple of $\mathbf{x}$ as k grows arbitrarily large.

Now if A is diagonalizable, then any $\mathbf{v}_0 \in \mathbb{R}^n$ can be expressed in the basis of eigenvectors as

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

As long as $\mathbf{v}_0$ has some component $c_1 \neq 0$ along $\mathbf{x}_1$, then $A^k\mathbf{v}_0$ will converge to $\mathbf{x}_1$ as k grows large. This is great, since $A^k\mathbf{v}_0$ can be recursively computed as $A(A^{k-1}\mathbf{v}_0)$, which is k matrix-vector multiplications instead of k matrix-matrix multiplications.

But what happens if we choose an initial vector $\mathbf{v}_0$ with $c_1 = 0$? In this case, it is possible that this method will fail to converge. An example is given in example 2.5. However, if we just choose a random $\mathbf{v}_0 \in \mathbb{R}^n$, we get $c_1 \neq 0$ with very high probability [5, p. 53], so this is not particularly worrisome. Additionally, even if we are incredibly unlucky and choose a $\mathbf{v}_0$ that is deficient in $\mathbf{x}_1$, rounding errors in the computations will likely save us and push so that $c_1 \neq 0$ [1].

Example 2.5: efficient power method with bad $\mathbf{v}_0$

Consider the matrix

$$A = \begin{bmatrix} 11 & 5 & 2 \\ 5 & 11 & 2 \\ 2 & 2 & 14 \end{bmatrix}$$

which has eigenvalues $\lambda_1 = 18$, $\lambda_2 = 12$, $\lambda_3 = 6$ with corresponding eigenvectors $\mathbf{x}_1 = (1, 1, 1)$, $\mathbf{x}_2 = (1, 1, -2)$, and $\mathbf{x}_3 = (1, -1, 0)$. If we choose a good initial vector like $\mathbf{v}_0 = (1, 2, 3)$, we indeed get

$$\mathbf{v}_k \to a\mathbf{x}_1 \text{ as } k \to \infty.$$

However, if we instead choose a bad initial vector like $\mathbf{v}_0 = (1, 0, -1)$ (which is orthogonal to $\mathbf{x}_1$), we instead get $\mathbf{v}_k \to b\mathbf{x}_2$. In this case, since A was diagonalizable, we still ended up with an eigenvector of A (just not the one we expected). If we instead had a non-diagonalizable matrix like

$$B = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix},$$

which has real eigenvalue $\lambda = 2$ and $\mathbf{x} = (1, 0, 0)$, we could fail to converge to anything at all if we choose $\mathbf{v}_0 = (0, 1, 1)$. However, a good choice of $\mathbf{v}_0 = (1, 1, 1)$ will still converge to a multiple of $\mathbf{x}$, highlighting the importance of the initial vector selection.

2.6 Implementation

Based on the power method (as presented in theorem 2.1), we present algorithm 2.1 to compute the dominant eigenvector and eigenvalue of a given diagonalizable matrix. The algorithm may also converge for a non-diagonalizable matrix, but we do not guarantee this. The referenced subroutine `FirstNonzeroCol` gets the first nonzero column of the given matrix (or the first whose norm exceeds some tolerance), and `SquareAndNorm` takes a given matrix A and then returns A^2, normalized by the magnitude of its first nonzero column.[2]

In this implementation of the power method, we choose to compute $A^{2k} = A^k A^k$ rather than $A^{k+1} = A^k A$ at step k because this is just as efficient (i.e. an $n \times n$ matrix multiplication) yet produces larger powers and therefore converges faster.

Algorithm 2.1: `DominantEigen1`

input : real diagonalizable $n \times n$ matrix A
parameter: tolerance $\varepsilon > 0$, max iterations $N > 0$
output : dominant eigenpair $(\mathbf{x}, \lambda)$

$A' \leftarrow A$;
$i \leftarrow 0$; `// number of iterations so far`
$\mathbf{x} \leftarrow$ `FirstNonzeroCol(`A'`)`;
$A' \leftarrow$ `SquareAndNormalize(`A'`)`;
while $\|\mathbf{x} -$ `FirstNonzeroCol(`A'`)`$\| > \varepsilon$ and $i < N$ **do**
  $\mathbf{x} \leftarrow$ `FirstNonzeroCol(`A'`)`;
  $A' \leftarrow$ `SquareAndNormalize(`A'`)`;
  $i \leftarrow i + 1$;
$\mathbf{x} \leftarrow$ `FirstNonzeroCol(`A'`)`;
$\lambda \leftarrow (\mathbf{x}^T A\mathbf{x})/(\mathbf{x}^T\mathbf{x})$;

Additionally, based on theorem 2.3, we implement the more efficient power method as algorithm 2.2.

[2] If A^k does not have any nonzero columns, then A is nilpotent; its only eigenvalue is $\lambda = 0$.

Algorithm 2.2: `DominantEigen2`

input : real diagonalizable $n \times n$ matrix A
parameter: tolerance $\varepsilon > 0$, max iterations $N > 0$
output : dominant eigenpair $(\mathbf{x}, \lambda)$

$\mathbf{x} \leftarrow$ choose random vector in $\mathbb{R}^n$;
$\mathbf{x} \leftarrow$ `Normalize(x)`;
$i \leftarrow 0$; `// number of iterations so far`
while $\|A\mathbf{x} - \mathbf{x}\| > \varepsilon$ and $i < N$ **do**
  $\mathbf{x} \leftarrow$ `Normalize(`$A\mathbf{x}$`)`;
  $i \leftarrow i + 1$;
$\lambda \leftarrow (\mathbf{x}^T A\mathbf{x})/(\mathbf{x}^T\mathbf{x})$;

Finally, we can use the deflation method from theorem 2.2 to write a general power method routine (algorithm 2.3) that computes all the eigenvectors and eigenvalues of a symmetric matrix.

Algorithm 2.3: `EigenPowerSymmetric`

input : real symmetric $n \times n$ matrix S
parameter: tolerance $\varepsilon > 0$, max iterations $N > 0$
output : list xs of eigenvectors and λs of eigenvalues

$xs \leftarrow$ empty list;
$\lambda s \leftarrow$ empty list;
for $i \in [n]$ **do**
  $\mathbf{x}, \lambda \leftarrow$ `DominantEigen(`S`)`;
  $S \leftarrow S - (\lambda/(\mathbf{x}^T\mathbf{x}))\mathbf{x}\mathbf{x}^T$;
  append $\mathbf{x}$ to xs;
  append λ to λs;

3 The QR Method

3.1 Mathematical Formulation

Another iterative method to compute the eigenvalues of a matrix is known as the QR method. The key insight behind this method is that similar matrices have the same eigenvalues, a fact proven in theorem 3.1.

Theorem 3.1: similar matrices have the same eigenvalues

Suppose A and C are similar square matrices. Then if $\lambda \in \mathbb{R}$ is an eigenvalue of A if and only if it is an eigenvalue of C.

Proof. Suppose $A = BCB^{-1}$. Now consider an eigenvalue λ of A with eigenvector $\mathbf{x}$. Then

$$A\mathbf{x} = \lambda\mathbf{x} \implies BCB^{-1}\mathbf{x} = \lambda\mathbf{x}.$$

But B is invertible, so we can multiply on the left by B^{-1} to obtain:

$$B^{-1}BCB^{-1} = B^{-1}(\lambda\mathbf{x}) \implies C(B^{-1}\mathbf{x}) = \lambda(B^{-1}\mathbf{x}).$$

Now since B^{-1} is invertible, $B^{-1}\mathbf{x} \neq \mathbf{0}$ if $\mathbf{x} \neq 0$ (i.e. B^{-1} has a trivial null space), which means that we have found an eigenvector $B^{-1}\mathbf{x}$ of C with eigenvalue λ. An identical argument in the other direction shows that any eigenvalue of C is also an eigenvalue of A, which concludes the proof. □

Additionally, recall that the eigenvalues of a triangular matrix lie on its diagonal [7, p. 294]. (This follows from the fact that $\det(A - \lambda I) = 0$ and the determinant of triangular A is the product of the diagonal entries.) This fact, combined with theorem 3.1, suggests the following method [7, p. 530] of computing the eigenvalues of A:

1. Somehow transform A into a similar triangular matrix B.
2. Read the eigenvalues off of the diagonal entires of B.

This is not exactly the QR method, but it captures much of the intuition behind the QR method, which is presented in theorem 3.2.

Theorem 3.2: the QR method [7, p. 530]

Suppose we have the matrix A. We form the sequence

$$\begin{aligned} A_0 &= A, \quad A_0 = Q_0R_0 \\ A_1 &= R_0Q_0, \quad A_1 = Q_1R_1 \\ A_2 &= R_1Q_1, \quad A_2 = Q_2R_2 \\ &\vdots \end{aligned}$$

which essentially takes the QR decomposition of A_k at each step and reverses the factors to find the next A_{k+1}. In many cases, as $k \to \infty$, the A_k's approach an upper triangular matrix that is similar to A.

Proof. It is not always true that the A_k's tend to a triangular matrix; in fact, according to [8], it is not known in general that the QR method converges for non-symmetric matrices, although it usually works very well. However, it *is* always true that A_k is similar to A. To show this, let $A = QR$ and $A_1 = RQ$ for orthogonal Q and triangular R. But note that

$$QA_1Q^T = QRQQ^T = QR = A$$

so there exists a matrix B such that $A = BA_1B^{-1}$, namely $B = Q^T$. At each step, the new A_{k+1} is similar to the previous A_k, as desired. □

The basic QR method, as presented in theorem 3.2, does not always converge. An example of this, taken from [8], is given in example 3.1. There are more advanced techniques for shifting the A_k's to improve the algorithm's convergence; the reader is referred to [7, p. 530] for details. Even this basic QR algorithm already has several advantages over deflation-based techniques such as the power method; it yields all the eigenvalues without suffering the rounding errors that deflation introduces [9].

Example 3.1: QR algorithm failure

Consider the matrix

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

which permits the QR decomposition

$$Q = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

But $A' = RQ = A$, so the algorithm gets stuck and does not converge to a triangular matrix at all.

Of course, the QR algorithm is not perfect. Although there are advanced tricks to speed up the process, each step in the basic version presented in theorem 3.2 takes $O(n^3)$ time, and converge can be slow [11]. This is especially true for very large systems where we only care about a few eigenvectors; in this case, we are often better served by the power method [9]. Despite this, the QR method and its variants are some of the most popular eigenvalue algorithms [7, p. 530] and have been for the past fifty years [8], especially for dense matrices [13].

3.2 Going from Eigenvalues to Eigenvectors

The QR method, as presented in theorem 3.2, will find all of the eigenvalues of A, but now we are faced with the opposite problem that we had with the power method: how do we go from eigenvalues to the corresponding eigenvectors? Of course, by theorem 1.1, $\mathbf{x} \in N(A - \lambda I)$, so we could use elimination to get the reduced row echelon form of $A - \lambda I$ and thus the null space, but is there a more sophisticated technique we could employ?

One way we could do it is called the inverse power method [8]. This method relies on a few key observations:

- A and $A - \mu I$ have the same eigenvectors, for $\mu \in \mathbb{R}$.
- If the λ_i's are the eigenvalues of A, then $(\lambda_i - \mu)$'s are the eigenvalues of $A - \mu I$.
- $(A - \mu I)$ and $(A - \mu I)^{-1}$ have the same eigenvectors but reciprocal eigenvalues. (In general, as long as μ is not an eigenvalue of A, $A - \mu I$ will be invertible, since $\det(A - \mu I) \neq 0$ if μ is not an eigenvalue.)

Suppose we have the eigenvalues $\lambda_1, \ldots, \lambda_n$ of A, and we wish to find the eigenvector $\mathbf{x}_i$ corresponding to λ_i. We can take some μ that is closer to λ_i than it is to all the other eigenvalues but is not an eigenvalue itself. Then $(A - \mu I)$ will have the same eigenvalues as A itself. We apply the power method on $(A - \mu I)^{-1}$—we know that the eigenvalues of $(A - \mu I)^{-1}$ are

$$\frac{1}{\lambda_1 - \mu}, \ldots, \frac{1}{\lambda_n - \mu}.$$

But we selected $\mu \in \mathbb{R}$ to be close to λ_i, such that $|\lambda_i - \mu| < |\lambda_j - \mu|$ for $i \neq j$. Then the dominant eigenvalue of $(A - \mu I)^{-1}$ is $1/(\lambda_i - \mu)$, so when we apply the power method to $(A - \mu I)^{-1}$, we will get back $\mathbf{x}_i$, the corresponding eigenvector.

Besides cheating a bit by requiring the power method, one obvious objection to using the inverse power method to get eigenvectors is that it involves the computation of a matrix inverse. However, a clever observation can make this much more efficient [10]: an iteration $B^{-1}\mathbf{x} = \mathbf{x}'$ is the same as numerically solving $B\mathbf{x}' = \mathbf{x}$, which can be done efficiently many times over by, for example, taking the LU decomposition of B.

3.3 Implementation

The QR method from theorem 3.2 can be more or less directly transcribed into algorithm 3.1, whose Julia implementation is given in the appendix. The referenced subroutine `QRDecomposition` can be accomplished using a variety of methods, such as the Gram-Schmidt algorithm [7, p. 327]. In practice, more advanced techniques are favored due to numerical stability with floating-point arithmetic [15].

Algorithm 3.1: `EigenQR`

input : real $n \times n$ matrix A
parameter: tolerance $\varepsilon > 0$, max iterations $N > 0$
output : eigenvalues λs

$Q, R \leftarrow$ `QRDecomposition(`A`)`;
$A' \leftarrow RQ$;
$i \leftarrow 0$; `// number of iterations so far`
while $\|A' - A\| > \varepsilon$ and $i < N$ **do**
  $Q, R \leftarrow$ `QRDecomposition(`A'`)`;
  $A \leftarrow A'$;
  $A' \leftarrow RQ$;
  $i \leftarrow i + 1$;
$\lambda s \leftarrow$ diagonal entries of A';

4 Connection to Singular Vectors

Up until now, we have been almost exclusively concerned with finding the eigenvectors and eigenvalues of a matrix. But what about finding the singular vectors of a matrix?

Note that the singular vectors of any $m \times n$ matrix A are just the eigenvectors of $A^T A$, so by finding the eigenvectors we already have a procedure for finding the singular vectors. To make the connection even better, note that $A^T A$ is symmetric, so by theorem 4.1 it is in fact diagonalizable. We can thus reliably use both the power and QR methods to find both the eigenvectors and eigenvalues of $A^T A$, which give us the singular vectors and values of A. We therefore do not develop a separate algorithm for finding the singular value decomposition of a matrix.

Theorem 4.1: spectral theorem for real symmetric matrices

If A is a real symmetric $n \times n$ matrix, then A has n orthonormal eigenvectors. The reader is referred to [12] for a full proof of this theorem.

5 Algorithm Comparison

In this section, I summarize the advantages and disadvantages of the power and QR algorithms. I also present some data on their respective convergences on randomly generated matrices.

The power method is perhaps the simplest and most intuitive of all iterative eigenvector algorithms, and its techniques form the basis for many of the more complex algorithms [2]. In its most efficient form, it makes use of matrix-vector multiplications, which are especially efficient on sparse matrices. However, its convergence is dependent on $|\lambda_2/\lambda_1|$; if λ_2 is close to λ_1, the power method can converge very slowly, and if $\lambda_2 = \lambda_1$, the algorithm may not converge at all. Finally, the power method only gives the dominant eigenvector (and by extension, the dominant eigenvalue), which is typically sufficient. While we can use the "inverse" power method to find the smallest eigenvalue, finding any of the other eigenvalues requires the introduction of some deflation techniques, which reduce accuracy by introducing rounding errors at each step.

The QR method is one of the most celebrated advances in numerical linear algebra and, with the addition of some convergence tricks, forms the basis for many real-world eigenvalue solvers. However, in its basic form (as present in theorem 3.2), it can become inefficient for large matrices, and it does not necessarily converge without the introduction of some shifting techniques.

Finally, what's a comparison of algorithms without some head-to-head benchmarking? Figure 2 compares these two algorithms on finding the greatest eigenvalue of a randomly generated 5×5 matrix. Of course, the comparison is not entirely fair; the power and QR algorithms provide different things (dominant

eigenvalue/vector vs. all eigenvalues), but it's as close a direct comparison as can be made. The Julia code for the comparison is given in the appendix.

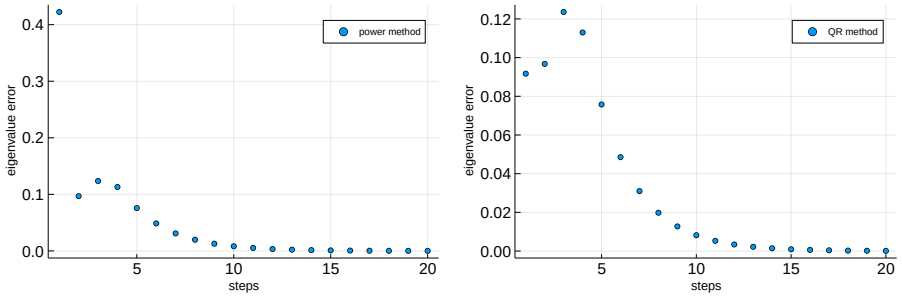

Figure 2: The power method (left) and QR method (right) on the same randomly generated 5×5 diagonalizable matrix

6 Appendix: Julia Code Samples

In this appendix, I provide Julia implementations for the major algorithms presented. The Julia code for generating the plots in this paper is also included. Note that the original source code made use of some Unicode characters that are not easily reproducible with LaTeX, so some variables have been renamed. A direct copy of the original source code can be conveniently obtained from `https://github.com/air-wreck/eigen`.

```
# project.jl
#
# 21-241 Final Project, Fall 2019
# Eric Zheng, Section 4K
#
# This file provides numerical methods for computing the eigenvectors and
# singular vectors of matrices.

using LinearAlgebra;

" DominantEigen(A)

Compute the dominant eigenvector of the diagonalizable matrix A using
the power method. We can opt to use either the column method or the vector
method, although the vector method is used by default.
"
function DominantEigen(A; tol=0.00001, max_iter=1000, method=:vector)
  if method == :column
    return DominantEigen1(A, tol=tol, max_iter=1000)
  elseif method == :vector
    return DominantEigen2(A, tol=tol, max_iter=1000)
  else
    error("method not recognized")
  end
```

```
end

function DominantEigen1(A; tol=0.00001, max_iter=1000)

  function first_nonzero_col(B; e=0.000001)
    for i in axes(B, 1)
      col = B[:,i]
      if norm(col) > e
        return col
      end
    end
    return nothing
  end

  # squares a matrix and then makes the first nonzero column an unit vector
  function square_and_normalize(B; e=0.000001)
    B ^= 2
    return B / norm(first_nonzero_col(B))
  end

  # continue computing powers of A until the Euclidean norm between
  # iterations is less than the tolerance
  A0 = A
  iters::Int = 0
  x = first_nonzero_col(A)
  A = square_and_normalize(A)
  while norm(x - first_nonzero_col(A)) > tol && iters < max_iter
    x = first_nonzero_col(A)
    A = square_and_normalize(A)
    iters += 1
  end
  x = first_nonzero_col(A)
  l = dot(A0 * x, x) / dot(x, x)
  return (x = x, l = l)
end

function DominantEigen2(A; tol=0.00001, max_iter=1000)

  function normalize(x)
    return x / norm(x)
  end

  x = normalize(rand(size(A, 1)))
  iters = 0
  while norm(normalize(A * x) - x) > tol && iters < max_iter
    x = normalize(A * x)
    iters += 1
  end
  l = dot(A * x, x) / dot(x, x)
  return (x = x, l = l)
end

" EigenPowerSymmetric(S)

Compute the eigenvectors of the symmetric matrix S using the power method.
This still works on non-symmetric matrices, but only returns the eigenvalues
and not the correct eigenvectors.
```

```
"
function EigenPowerSymmetric(S; tol=0.00001, max_iter=1000)
  xs = []
  ls = []
  for i in axes(S, 1)  # by Spectral Thm, symmetric S has n eigenvectors
                       # so we don't need to worry about zero, I think
    x, l = DominantEigen2(S)
    S -= l / (x' * x) * x * x'
    push!(xs, x)
    push!(ls, l)
  end
  return (x = hcat(xs...), l = hcat(ls...))
end

" EigenQR(A)

Compute the eigenvalues of the square matrix A using QR decomposition.
"
function EigenQR(A; tol=0.00001, max_iter=1000)
  Q, R = qr(A)
  B = R * Q
  iters = 1
  while norm(A - B) > tol && iters < max_iter
    Q, R = qr(B)
    A = B
    B = R * Q
    iters += 1
  end
  return diag(B)
end

" EigenSolver(A)

Ultimate subroutine to compute the eigenvectors and eigenvalues
of the square matrix A. If no method is given, default to the power method.

Returns: `(x, l)` (for :power), `l` (for :qr)

Examples:
```julia-repl
A = [1 3 ; 3 1];
EigenSolver(A, tol=0.001, method=:qr)
```

Notes:
* The `:power` method can be applied to non-symmetric matrices, as long as they
  are diagonalizable. In this case, the eigenvalues will be correct, but not
  the eigenvectors.
"
function EigenSolver(A; tol=0.00001, max_iter=1000, method=:power)
  if method == :power
    return EigenPowerSymmetric(A, tol=tol, max_iter=max_iter)
  elseif method == :qr
    return EigenQR(A, tol=tol)
  else
    error("method not recognized")
  end
```

```
end

" Singular(A)

Compute the singular vectors and values for the matrix A.

Returns: `(v, s)`

Examples:
```julia-repl
A = [1 2 3 4 ; 5 6 7 8 ; 9 10 11 12];
Singular(A)
```

Notes:
* e > 0 is the tolerance for comparing a singular value to zero.
  The default is e = 0.0000001.
"
function Singular(A; tol=0.00001, max_iter=1000, method=:power, e=0.0000001)
  v, l = EigenSolver(A' * A, tol=tol, max_iter=max_iter, method=method)
  s = map(sqrt, filter(x -> x > e, l))
  return (v = v[:,1:length(s)], s = s)
end
```

And this is the source code for all the graphs produced:

```
# project-graph.jl
#
# 21-241 Final Project, Fall 2019
# Eric Zheng, Section 4K
#
# This file makes the pretty plots for the project.

include("project.jl")
using Plots

# we reimlpement DominantEigen to keep
# track of prior values for plotting
function PlotPowerConvergence()
  A0 = [23 5 2 ; 5 23 2 ; 2 2 26]
  x  = [1,1,1] / norm([1,1,1])
  A  = A0 / norm(A0[:,1])

  xs = 1:20  # number of algorithm steps
  ys = Float64[]
  for _ in xs
    push!(ys, norm(A[:,1] - x))
    A *= A0
    A /= norm(A[:,1])
  end
  plot(xs, map(i -> (4/5)^i, xs),
       label="(4/5)^k", xlabel="steps", ylabel="eigenvector error")
  scatter!(xs, ys, label="power method")
  savefig("power-method.pdf")
end

function RandomDiag(n)
  A  = rand(n, n)
```

```
  ls = rand(n)
  l  = sort(ls, rev=true)[1]
  A  = A * diagm(0 => ls) / A
  return A, l
end

function PlotPowerCmp(A0, l, n)
  A  = A0 / norm(A0[:,1])
  xs = 1:n
  ys = Float64[]
  for _ in xs
    x = A[:,1]
    push!(ys, abs(dot(A0 * x, x) / dot(x, x) - l))
    A *= A0
    A /= norm(A[:,1])
  end
  scatter(xs, ys, label="power method",
          xlabel="steps", ylabel="eigenvalue error")
  savefig("power-cmp.pdf")
end

function PlotQRCmp(A0, l, n)
  Q, R = qr(A0)
  A  = R * Q
  ys = Float64[]
  xs = 1:n
  for _ in xs
    guess = sort(diag(A), rev=true)[1]
    push!(ys, abs(guess - l))
    Q, R = qr(A)
    A0 = A
    A  = R * Q
  end
  scatter(xs, ys, label="QR method",
          xlabel="steps", ylabel="eigenvalue error")
  savefig("qr-cmp.pdf")
end

function PlotPowerQRCmp()
  A, l = RandomDiag(5)
  PlotPowerCmp(A, l, 20)
  PlotQRCmp(A, l, 20)
end
```

References

[1] https://www.cs.huji.ac.il/~csip/tirgul2.pdf

[2] https://web.mit.edu/18.06/www/Spring17/Power-Method.pdf

[3] http://www.macs.citadel.edu/chenm/344.dir/08.dir/lect4_2.pdf

[4] https://www.cs.toronto.edu/~yuvalf/Limitations.pdf

[5] Blum, Hopcroft, and Kannan. *Foundations of Data Science*. https://www.cs.cornell.edu/jeh/book.pdf

[6] http://ergodic.ugr.es/cphys/lecciones/fortran/power_method.pdf

[7] Strang. *Introduction to Linear Algebra*, fifth edition.

[8] http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/QRalgorithm.html

[9] http://www.robots.ox.ac.uk/~sjrob/Teaching/EngComp/ecl4.pdf

[10] http://www.ohiouniversityfaculty.com/youngt/IntNumMeth/lecture16.pdf

[11] https://math.nyu.edu/~stadler/num1/material/num1_eigenvalues.pdf

[12] http://www-math.mit.edu/~dav/spectral.pdf

[13] Wang and Gragg. Convergence of the shifted QR algorithm for unitary Hessenberg matrices. *Mathematics of Computation* 70(240).

[14] Rosen. Niels Henrik Abel and equations of the fifth degree. *The American Mathematical Monthly* 102(6).

[15] http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/Householder.html