Eigen-Stuff via Power and QR Methods

By Charisse Qin and Jojo Li

November 2024

Introduction

In Pittsburgh, weather is not difficult to predict. If it is winter, it is snowing. If it is fall or spring, it is raining. If it is summer it... may be sunny. Perhaps.

However, in other places, where the sky does not spew hate upon its underdwellers 365/24/7, weather is more unpredictable.

In such places, eigenvalues and eigenvectors are crucial to provide important information that aids the development of predictive models that tell us what to expect from the clouds on a given day. The next time you check the weather report and find an abrupt thunderstorm in your future, thank the eigen-stuff. Without it, the Pittsburgh rain would have hammered you to shreds.

Eigenvectors and eigenvalues are mobilized on great scale for all sorts of data analysis and processing. As such, it is important to minimize their time and space complexity.
The *power method* and the *QR method* are two leading ways for the computer to stably and efficiently construct singular value decompositions (SVD), and compute eigenvalues and eigenvectors.

1 Power Method

The ***power method*** iteratively approximates the dominant singular vectors and eigenvectors and their corresponding singular value or eigenvalue for a matrix.

1.1 Basic Procedure of Power Method

Let A_0 be the original matrix with size $m \times n$, where $m, n \in \mathbb{N}^+$. Let A_0 have rank $r \in \mathbb{N}^+$.
By definition, its singular value decomposition (SVD) is represented as $A = U\Sigma V^T$.
$U = [\mathbf{u_1}\ \mathbf{u_2}\ ...\ \mathbf{u_m}]$ is an $m \times m$ matrix where $\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_m} \in \mathbb{N}^m$, and $\{\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_m}\}$ are orthonormal eigenvectors of AA^T. They are also called the *left singular vectors*.
$V = [\mathbf{v_1}\ \mathbf{v_2}\ ...\ \mathbf{v_n}]$ is an $n \times n$ matrix where $\mathbf{v_1}, \mathbf{v_2}, ..., \mathbf{v_n} \in \mathbb{N}^n$, and $\{\mathbf{v_1}, \mathbf{v_2}, ..., \mathbf{v_n}\}$ are orthonormal eigenvectors of A^TA. They are also called the *right singular vectors*.
Σ is an $m \times n$ matrix whose only nonzero entries (*singular values*) $\sigma_1, \sigma_2, ..., \sigma_r$ are on its diagonal from the left top corner, appearing in decreasing order, i.e., $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r$.

Let the linearly independent unit eigenvectors of A_0 be $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_s}$ (where $s \in \mathbb{N}$ is the total number of them), with corresponding eigenvalues $\lambda_1, \lambda_2, ..., \lambda_s$ in decreasing order.

1.1.1 To find the dominant singular vectors $\mathbf{v_1}$ and $\mathbf{u_1}$

First, set $\mathbf{x_0}$ to a random nonzero vector in $\mathbb{R}^n$.

Then, let $B = A_0^T A_0$. Then, $\mathbf{v_1}$ is approximated by multiplying B iteratively to the left of $\mathbf{x_0}$. In other words, after k iterative steps, $\mathbf{x_k} = B^k \mathbf{x_0}$.
In this way, normalizing $\lim_{k\to\infty} \mathbf{x_k}$ gives $\mathbf{v_1}$. Which is, $\mathbf{v_1} = \dfrac{\lim_{k\to\infty} \mathbf{x_k}}{||\lim_{k\to\infty} \mathbf{x_k}||}$.
By the construction of SVD introduced in class, $\mathbf{u_1} = \dfrac{A_0\mathbf{v_1}}{\sigma_1}$.

1.1.2 To find all singular vectors for nontrivial singular values

For $i \in \mathbb{N}$ and $1 \leq i \leq n$, let A_i be the matrix after i modifications on A_0.

After finding $\mathbf{v_i}$, $\frac{B\mathbf{v_i}\cdot\mathbf{v_i}}{\mathbf{v_i}\cdot\mathbf{v_i}}$ is the corresponding eigenvalue of B,
so $\sigma_i = \sqrt{\frac{B\mathbf{v_i}\cdot\mathbf{v_i}}{\mathbf{v_i}\cdot\mathbf{v_i}}}$ (by definition of SVD), and $\mathbf{u_i} = \frac{A_0\mathbf{v_i}}{\sigma_i}$ will be the i^{th} left singular vector.

Then, modify A_{i-1} into $A_i = A_{i-1} - \sigma_i \mathbf{u_i}\mathbf{v_i}^T$, and find $\mathbf{v_{i+1}}$ and $\mathbf{u_{i+1}}$ (the dominant singular vectors of A_i) by **1.1.1** with $B = A_i^T A_i$.

Repeat this process until the σ_i found is less than 10^{-5}, which indicates that the corresponding singular vectors are not significant enough.

1.1.3 To find the dominant unit eigenvector $\mathbf{p_1}$

Suppose A_0 is a square matrix, its eigenvalues are real numbers, and corresponding eigenvectors belong to $\mathbb{R}^n$.

First, set $\mathbf{y_0}$ to a random nonzero vector in $\mathbb{R}^n$.

Then, $\mathbf{p_1}$ is approximated by multiplying A_0 iteratively to the left of $\mathbf{y_0}$. In other words, after k iterative steps, $\mathbf{y_k} = A_0^k \mathbf{y_0}$.
In this way, normalizing $\lim_{k\to\infty} \mathbf{y_k}$ gives $\mathbf{p_1}$. Which is, $\mathbf{p_1} = \dfrac{\lim_{k\to\infty} \mathbf{y_k}}{||\lim_{k\to\infty} \mathbf{y_k}||}$.

The corresponding eigenvalue would be $\lambda_1 = \frac{A_0\mathbf{p_1}\cdot\mathbf{p_1}}{\mathbf{p_1}\cdot\mathbf{p_1}}$.

1.1.4 To find all unit eigenvectors for nontrivial eigenvalues

Here, we restrict the input A_0 to symmetric matrices.

Then, as proved in class, the SVD of A_0 will also be its diagonalization, which shows that its eigenvectors will be the same as its left (and right) singular vectors calculated above.

1.2 The Math Behind Power Method

1.2.1 Calculation of the dominant singular vectors

Consider the matrix A_0 defined in **1.1**.

$$
\begin{aligned}
A_0 &= U\Sigma V^T \\
&= \begin{bmatrix} \mathbf{u_1} & \mathbf{u_2} & \cdots & \mathbf{u_m} \end{bmatrix}
\begin{bmatrix}
\sigma_1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & \sigma_2 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & 0 & \sigma_3 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\
0 & 0 & 0 & \cdots & \sigma_r & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0
\end{bmatrix}
\begin{bmatrix} \mathbf{v_1}^T \\ \mathbf{v_2}^T \\ \vdots \\ \mathbf{v_n}^T \end{bmatrix} \\
&= \sum_{i=1}^{r} \sigma_i \mathbf{u_i} \mathbf{v_i}^T \qquad \text{(by matrix multiplication)}
\end{aligned}
$$

Correspondingly,

$$
\begin{aligned}
A_0^T &= (U\Sigma V^T)^T \\
&= (V^T)^T \Sigma^T U^T && \text{(by property of transpose)} \\
&= V\Sigma^T U^T && \text{(by property of transpose)} \\
&= \begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \cdots & \mathbf{v_n} \end{bmatrix}
\begin{bmatrix}
\sigma_1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & \sigma_2 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & 0 & \sigma_3 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\
0 & 0 & 0 & \cdots & \sigma_r & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0
\end{bmatrix}
\begin{bmatrix} \mathbf{u_1}^T \\ \mathbf{u_2}^T \\ \vdots \\ \mathbf{u_m}^T \end{bmatrix} \\
&= \sum_{j=1}^{r} \sigma_j \mathbf{v_j} \mathbf{u_j}^T && \text{(by matrix multiplication)}
\end{aligned}
$$

Note that by orthogonality, for every $a, b \in [m]$ such that $a \neq b$, $\mathbf{u_a} \cdot \mathbf{u_b} = \mathbf{0}$,
and for every $c, d \in [n]$ such that $c \neq d$, $\mathbf{v_c} \cdot \mathbf{v_d} = \mathbf{0}$.

Hence,

$$
\begin{aligned}
B &= A_0^T A_0 \\
&= (\sum_{j=1}^{r} \sigma_j \mathbf{v_j} \mathbf{u_j}^T)(\sum_{i=1}^{r} \sigma_i \mathbf{u_i} \mathbf{v_i}^T) \\
&= \sum_{i,j=1}^{r} \sigma_i \sigma_j \mathbf{v_j} (\mathbf{u_j}^T \mathbf{u_i}) \mathbf{v_i}^T \\
&= \sum_{i=1}^{r} \sigma_i^2 \mathbf{v_i} \mathbf{v_i}^T && \text{(by matrix multiplication)}
\end{aligned}
$$

Therefore, $B^k = \sum_{i=1}^{r} \sigma_i^{2k} \mathbf{v_i} (\mathbf{v_i}^T \mathbf{v_i})^{k-1} \mathbf{v_i}^T = \sum_{i=1}^{r} \sigma_i^{2k} \mathbf{v_i} \mathbf{v_i}^T$ for $k \in \mathbb{N}^+$.

For any n-dimensional vector $\mathbf{x_0}$, it can be written as $\sum_{i=1}^{n} c_i \mathbf{v_i}$ for some $c_1, c_2, ..., c_n$ since the columns of V are orthonormal.
Then, $B^k \mathbf{x_0} = (\sum_{i=1}^{r} \sigma_i^{2k} \mathbf{v_i} \mathbf{v_i}^T)(\sum_{j=1}^{n} c_j \mathbf{v_j}) = \sum_{i=1}^{r} c_i \sigma_i^{2k} \mathbf{v_i} (\mathbf{v_i}^T \mathbf{v_i}) = \sum_{i=1}^{r} c_i \sigma_i^{2k} \mathbf{v_i}$.

As k approaches infinity, σ_1 and $\mathbf{v_1}$ dominates the value of $B^k \mathbf{x_0}$ since the singular values are arranged in decreasing order (σ_1 is the largest among them).
In other words, $\lim_{k \to \infty} B^k \mathbf{x_0} = c_1 \sigma_1^{2k} \mathbf{v_1}$. Thus, normalizing this value would give $\mathbf{v_1}$.

1.2.2 Calculation of all singular vectors for nontrivial singular values

Every subsequent matrix A_i after a modification is the result of removing the weighted $\mathbf{v_i}$ (i.e. its dominance) from A_{i-1}. After this, $\mathbf{v_{i+1}}$ would become the dominant right singular vector for A_i. In this way, one can find all nontrivial singular vectors for A_0 iteratively based on the same process and math described in **1.2.1**.

1.2.3 Calculation of the dominant unit eigenvector

The calculation of the dominant unit eigenvector is similar to that of the singular vectors.

For any n-dimensional vector $\mathbf{y_0}$, it is a combination of A_0's eigenvectors, so it can be written as $\sum_{i=1}^{s} c_i \mathbf{p_i}$ for some $c_1, c_2, ..., c_s$, since the eigenvectors are linearly independent. Then, $A_0^k \mathbf{y_0} = \sum_{i=1}^{s} c_i A_0^k \mathbf{p_i} = \sum_{i=1}^{s} c_i \lambda_i^k \mathbf{p_i}$ (by property of eigenvalues).

As k approaches infinity, λ_1 and $\mathbf{p_1}$ dominates the value of $A_0^k \mathbf{y_0}$ since the eigenvalues are arranged in decreasing order (λ_1 is the largest among them).
In other words, $\lim_{k\to\infty} A_0^k \mathbf{y_0} = c_1 \lambda_1^k \mathbf{p_1}$. Thus, normalizing this value would give $\mathbf{p_1}$.

1.2.4 Calculation of the corresponding eigenvalue for an eigenvector

Given eigenvector $\mathbf{v}$ of matrix A, the corresponding eigenvalue λ is calculated by the **Rayleigh quotient**:

$$\lambda = \frac{A\mathbf{v} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}$$

This formula gives the correct calculation because:

$$\frac{A\mathbf{v} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} = \frac{\lambda \mathbf{v} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} = \frac{\lambda(\mathbf{v} \cdot \mathbf{v})}{(\mathbf{v} \cdot \mathbf{v})} = \lambda.$$

It is used to calculate the eigenvalues for B and A_0 after every calculation of dominant singular vectors and eigenvector.

1.3 Code

The complete version of the codes is attached as Appendix A.

1.3.1 Functions for SVD and eigenvectors

`pow_singular_dom`: calculates dominant right singular vector

Input: Matrix $A \in \mathbb{R}^{m \times n}$; its number of columns n
Output: Approximated dominant right singular vector of A

Set $B = A^T A$
Initialize a random n-dimensional vector $\mathbf{x_0}$
Normalize $\mathbf{x_0}$: set $\mathbf{x} = \frac{\mathbf{x_0}}{\|\mathbf{x_0}\|}$
Set $\mathbf{v_0} = \mathbf{x_0}$
Set $\mathbf{v_1} = \mathbf{x_0}$
Set $k = 0$ (to keep track of number of iterations for efficiency analysis)
while true **do**
    Set $k = k + 1$
    Compute $\mathbf{x} = B\mathbf{x}$
    Normalize $\mathbf{x}$: $\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$
    Set $\mathbf{v_1} = \mathbf{x}$
    Check convergence: if $\|\mathbf{v_1} - \mathbf{v_0}\| < 10^{-5}$, **print** k and **return** $\mathbf{v_1}$
    Set $\mathbf{v_0} = \mathbf{x}$
end while

Changes on the original algorithm:

Since it is impossible for Julia to make k approach infinity, we set a threshold $\epsilon = 10^{-5}$ such that if the Frobenius distance between $\mathbf{v_i}$ and $\mathbf{v_{i-1}}$, $||\mathbf{v_i} - \mathbf{v_{i-1}}||_F$, is less than ϵ, it signals that the program has found a good approximation for singular vectors or eigenvectors and the loop would stop at this point

We normalize $B^k\mathbf{x_0}$ every iteration as k increases to prevent overflow in Julia. By the nature of normalization, this would not affect the final results, which are all unit vectors.

`pow_singular_all`: calculates all nontrivial singular values and their singular vectors

Input: Matrix $A \in \mathbb{R}^{m \times n}$; its number of columns n
Output: No return value

Set $A_0 = A$ to store the original matrix
Set $B = A^T A$
Set $i = 1$ to keep track of the number of singular values calculated
Set $\mathbf{v_i} = \textit{pow\_singular\_dom}(A, n)$
Set $\sigma_i = \sqrt{\frac{B\mathbf{v_i} \cdot \mathbf{v_i}}{\mathbf{v_i} \cdot \mathbf{v_i}}}$
Set $\mathbf{u_i} = \frac{A_0 \mathbf{v_i}}{\sigma_i}$
while true **do**
    print $\mathbf{v_i}, \mathbf{u_i}, \sigma_i$
    Check number of singular values: if $i == n$, exit loop
    Set $i = i + 1$
    Compute $A = A - \sigma_i \mathbf{u_i} \mathbf{v_i}^T$
    Compute $B, \mathbf{v_i}, \sigma_i, \mathbf{u_i}$ as above
    Check if nontrivial: if $\sigma_i < 10^{-5}$, $i = i - 1$ and exit loop
end while
print i, the number of nontrivial singular values found

`pow_eigen_dom`: calculates the dominant eigenvector

Input: Matrix $A \in \mathbb{R}^{n \times n}$; its number of columns n
Output: Approximated dominant eigenvector of A

Initialize a random n-dimensional vector $\mathbf{y_0}$
Normalize $\mathbf{y_0}$: set $\mathbf{y} = \frac{\mathbf{y_0}}{\|\mathbf{y_0}\|}$
Set $\mathbf{p_0} = \mathbf{y_0}$
Set $\mathbf{p_1} = \mathbf{y_0}$
Set $k = 0$ (to keep track of number of iterations for efficiency analysis)
while true **do**
    Set $k = k + 1$
    Compute $\mathbf{y} = A\mathbf{y}$
    Normalize $\mathbf{y}$: $\mathbf{y} = \frac{\mathbf{y}}{\|\mathbf{y}\|}$
    Set $\mathbf{p_1} = \mathbf{y}$
    Check convergence: if $\|\mathbf{p_1} - \mathbf{p_0}\| < 10^{-5}$, **print** k and **return** $\mathbf{p_1}$
    Set $\mathbf{p_0} = \mathbf{y}$
end while

1.4 Example

This section provides an example of approximating the dominant unit eigenvector for a square matrix A using the power method.

Consider the matrix:

$$A = \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix}.$$

Set the initial vector $\mathbf{x_0}$ as $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

Iteration 1:

$$\mathbf{x_1} = A\mathbf{x_0} = \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 5 \end{bmatrix}$$

Normalize $\mathbf{x_1}$:

$$\mathbf{x_1} = \frac{\begin{bmatrix} 7 \\ 5 \end{bmatrix}}{\sqrt{7^2 + 5^2}} = \frac{\begin{bmatrix} 7 \\ 5 \end{bmatrix}}{\sqrt{74}} = \begin{bmatrix} \frac{7}{\sqrt{74}} \\ \frac{5}{\sqrt{74}} \end{bmatrix}.$$

Iteration 2:

$$\mathbf{x_2} = \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} \frac{7}{\sqrt{74}} \\ \frac{5}{\sqrt{74}} \end{bmatrix} = \begin{bmatrix} \frac{45}{\sqrt{74}} \\ \frac{27}{\sqrt{74}} \end{bmatrix}.$$

Normalize $\mathbf{x_2}$:

$$\mathbf{x_2} = \frac{\begin{bmatrix} \frac{45}{\sqrt{74}} \\ \frac{27}{\sqrt{74}} \end{bmatrix}}{\sqrt{(\frac{45}{\sqrt{74}})^2 + (\frac{27}{\sqrt{74}})^2}} \approx \frac{\begin{bmatrix} \frac{45}{\sqrt{74}} \\ \frac{27}{\sqrt{74}} \end{bmatrix}}{6.10051} \approx \begin{bmatrix} 0.85749 \\ 0.51450 \end{bmatrix}.$$

Iteration 3:

$$\mathbf{x_3} = A\mathbf{x_2} = \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 0.85749 \\ 0.51450 \end{bmatrix} = \begin{bmatrix} 5.31645 \\ 2.91549 \end{bmatrix}.$$

Normalize $\mathbf{x_3}$:

$$\mathbf{x_3} = \frac{\begin{bmatrix} 5.31645 \\ 2.91549 \end{bmatrix}}{\sqrt{(5.31645)^2 + (2.91549)^2}} \approx \frac{\begin{bmatrix} 5.31645 \\ 2.91549 \end{bmatrix}}{6.06339} \approx \begin{bmatrix} 0.87681 \\ 0.48083 \end{bmatrix}.$$

Iteration 4:

$$\mathbf{x_4} = A\mathbf{x_3} = \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 0.87681 \\ 0.48083 \end{bmatrix} = \begin{bmatrix} 5.34571 \\ 2.80013 \end{bmatrix}.$$

Normalize $\mathbf{x_4}$:

$$\mathbf{x_4} = \frac{\begin{bmatrix}5.34571\\2.80013\end{bmatrix}}{\sqrt{(5.34571)^2+(2.80013)^2}} \approx \frac{\begin{bmatrix}5.34571\\2.80013\end{bmatrix}}{6.03468} \approx \begin{bmatrix}0.88583\\0.46401\end{bmatrix} \approx \begin{bmatrix}0.89\\0.46\end{bmatrix}.$$

Iteration 5:

$$\mathbf{x_5} = A\mathbf{x_4} = \begin{bmatrix}5 & 2\\1 & 4\end{bmatrix}\begin{bmatrix}0.88583\\0.46401\end{bmatrix} = \begin{bmatrix}5.35717\\2.74187\end{bmatrix}.$$

Normalize $\mathbf{x_5}$:

$$\mathbf{x_5} = \frac{\begin{bmatrix}5.35717\\2.74187\end{bmatrix}}{\sqrt{(5.35717)^2+(2.74187)^2}} \approx \frac{\begin{bmatrix}5.35717\\2.74187\end{bmatrix}}{6.01807} \approx \begin{bmatrix}0.89018\\0.45561\end{bmatrix} \approx \begin{bmatrix}0.89\\0.46\end{bmatrix}.$$

Convergence:

As shown from the results of iterations 4 and 5, as more iterations are conducted, the resulting unit vector converges to approximately $\begin{bmatrix}0.89\\0.46\end{bmatrix}$.

Hence, the estimated dominant eigenvector $\mathbf{p_1}$ is $\begin{bmatrix}0.89\\0.46\end{bmatrix}$.

Then, its corresponding eigenvalue λ_1 can be calculated by

$$\lambda_1 = \frac{A\mathbf{p_1}\cdot\mathbf{p_1}}{\mathbf{p_1}\cdot\mathbf{p_1}} = \frac{\begin{bmatrix}5 & 2\\1 & 4\end{bmatrix}\begin{bmatrix}0.89\\0.46\end{bmatrix}\cdot\begin{bmatrix}0.89\\0.46\end{bmatrix}}{\begin{bmatrix}0.89\\0.46\end{bmatrix}\cdot\begin{bmatrix}0.89\\0.46\end{bmatrix}} = \frac{\begin{bmatrix}5.37\\2.73\end{bmatrix}\cdot\begin{bmatrix}0.89\\0.46\end{bmatrix}}{\begin{bmatrix}0.89\\0.46\end{bmatrix}\cdot\begin{bmatrix}0.89\\0.46\end{bmatrix}} = \frac{6.0351}{1.0037} \approx 6.01.$$

In conclusion, A's approximated dominant eigenvalue and corresponding eigenvector are:

$$\lambda_1 \approx 6.01, \quad \mathbf{p_1} \approx \begin{bmatrix}0.89\\0.46\end{bmatrix}.$$

One can find its dominant singular vectors similarly, by substituting A^TA with A.
Its other singular vectors can be found iteratively using the same process.

1.5 Efficiency Analysis

1.5.1 Why power method is fast

Originally, as introduced in class, to find a matrix's eigenvalues and eigenvectors, one needs to solve roots of characteristic polynomials and find null spaces involving this matrix.

On the other hand, when using the power method, since each iteration results in a unit vector, it repeatedly computes simple multiplications of an $n \times n$ matrix and an n-dimensional unit vector, which is an easier, faster, and more stable procedure for the computer.

1.5.2 Speed of convergence

Let the proportional gap between the largest singular value (or eigenvalue) and the second-largest one be $d = \dfrac{\sigma_1 - \sigma_2}{\sigma_1}$.

Generally, the larger d is, the faster (fewer iterations) the procedure of finding the dominant eigenvector or singular vectors is, as observed from the following experiment with different matrices.

The power method is run on the following matrices respectively:

$$\begin{bmatrix} 12 & 3 & 5 & 7 & 2 & 9 & 4 & 1 & 11 & 6 \\ 2 & 15 & 3 & 7 & 6 & 5 & 8 & 9 & 1 & 10 \\ 4 & 1 & 16 & 8 & 7 & 5 & 9 & 3 & 12 & 2 \\ 3 & 6 & 9 & 14 & 4 & 11 & 13 & 7 & 10 & 15 \\ 5 & 7 & 6 & 4 & 18 & 3 & 2 & 9 & 1 & 13 \\ 11 & 8 & 7 & 5 & 12 & 17 & 3 & 2 & 6 & 14 \\ 1 & 2 & 3 & 4 & 5 & 6 & 19 & 8 & 11 & 10 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 6 & 5 & 3 & 4 & 1 & 2 & 7 & 8 & 19 & 20 \\ 8 & 4 & 3 & 12 & 9 & 1 & 6 & 11 & 10 & 7 \end{bmatrix}, \quad \begin{bmatrix} 4 & 2 & 3 & 0 & 1 & 0 & 0 & 0 \\ 0 & 5 & 4 & 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 6 & 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & 0 & 7 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 3 & 8 & 4 & 1 & 0 \\ 0 & 0 & 0 & 1 & 4 & 9 & 3 & 2 \\ 0 & 0 & 0 & 0 & 1 & 3 & 10 & 4 \\ 0 & 0 & 0 & 0 & 0 & 2 & 4 & 11 \end{bmatrix},$$

$$\begin{bmatrix} 2 & 3 & 4 & \cdots & 12 \\ 3 & 4 & 5 & \cdots & 13 \\ 4 & 5 & 6 & \cdots & 14 \\ \vdots & \vdots & \vdots & & \vdots \\ 12 & 13 & 14 & \cdots & 22 \end{bmatrix}, \quad \begin{bmatrix} 4 & 2 & 3 & 1 \\ 2 & 5 & 1 & 0 \\ 3 & 1 & 6 & 2 \\ 1 & 0 & 2 & 7 \end{bmatrix}, \quad \begin{bmatrix} 5 & 4 & 2 \\ 0 & 3 & -1 \\ 0 & 0 & 1 \end{bmatrix}.$$

For every matrix and every modification, the number of iterations to find the dominant right singular vector is recorded. The initial vector $\mathbf{x_0}$ is fixed.
The correct singular values of each matrix are calculated by the Julia function `svd()`.

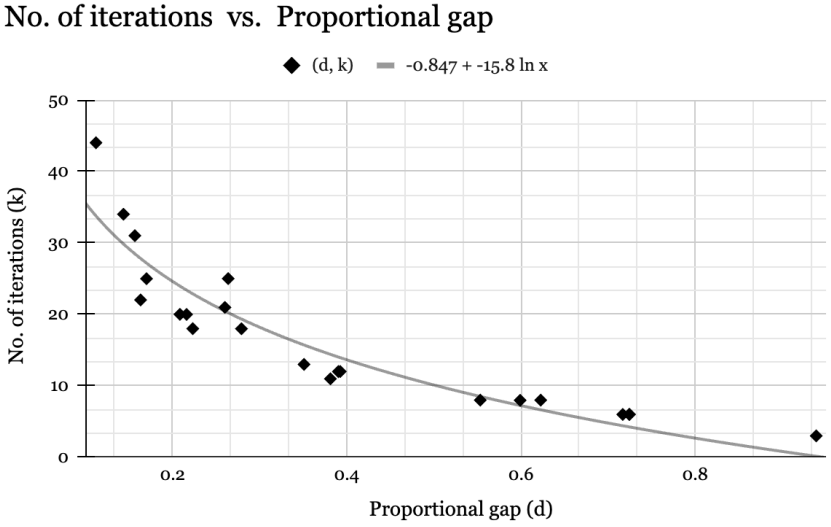
A scattered plot is built with:
(1) The x-axis being the proportional gaps d;
(2) The y-axis being the corresponding numbers of iterations (k).

Data:

σ_1	σ_2	Gap	k
83.2933	22.9649	0.72429	6
22.9649	19.2140	0.16333	22
19.2140	17.0598	0.11212	44
17.0598	14.6074	0.14375	34
14.6074	11.3465	0.22324	18
11.3465	8.98187	0.20840	20
8.98187	6.64550	0.26012	21
6.64550	4.05179	0.39030	12
4.05179	1.11630	0.72449	6
140.606	8.60563	0.93880	3
17.1651	12.3756	0.27903	18

σ_1	σ_2	Gap	k
12.3756	9.70012	0.21619	20
9.70012	7.13966	0.26396	25
7.13966	6.02022	0.15679	31
6.02022	3.90830	0.35080	13
3.90830	3.24349	0.17010	25
3.24349	1.44912	0.55322	8
6.90767	2.76935	0.59909	8
2.76935	0.78412	0.71686	6
10.3516	6.28859	0.39250	12
6.28859	3.89130	0.38121	11
3.89130	1.46855	0.62261	8

Graph:



In the above graph, the estimated trend line shows a negative correlation between $\ln d$ and the number of iterations. In other words, the power method is faster when the second-largest singular value is smaller compared to the largest one.

Since the process for eigenvalues is similar, this is the same case for calculating the dominant eigenvector.

1.6 Discussion

1.6.1 Results

Using power method, this study demonstrates a way to approximate:
(1) All nontrivial singular values with corresponding singular vectors for *any* matrices;
(2) The dominant eigenvalue and corresponding eigenvector for *square* matrices;
(3) Nontrivial eigenvalues and eigenvectors for *symmetric* matrices.

The *efficiency* of the power method is correlated with the proportional gap between the largest singular value (or eigenvalue) and the second-largest one, namely, $\frac{\sigma_1-\sigma_2}{\sigma_1}$ or $\frac{\lambda_1-\lambda_2}{\lambda_1}$. The larger this ratio is, the faster the method is. Specifically, when the natural logarithm of this ratio increases, the number of iterations decreases.

1.6.2 Limitations

(a) Regarding the calculation of eigenvectors other than the dominant one, this study restricts the input type to symmetric matrices, for the following reason:
In order to find all unit eigenvectors for nontrivial eigenvalues for non-symmetric matrices, one needs to find the corresponding left eigenvector in order to remove the influence of the dominant eigenvector from the original matrix (i.e. *deflate* A_0).
Yet, due to the complexity in iterations in Julia, it is hard to keep the approximation process accurate and efficient, thus decreasing the reliability of the results on non-symmetric matrices.

(b) By the efficiency analysis, the power method would become very slow when the largest two singular values (or eigenvalues) are very close to each other.

(c) It is difficult for the power method to compute the corresponding singular vectors (or eigenvectors) for singular value (or eigenvalue) zero, since the multiplication of zero and any vector results in the zero vector, making them not dominant.

2 QR Method

The QR method finds the singular value decomposition of a matrix via QR factorization of a matrix A.

2.1 What is QR factorization?

Any m by n matrix A can be decomposed into the product of two matrices: Q and R. Q denotes an orthogonal m by m matrix, while R is an upper triangular m by n matrix.

2.1.1 The Gram-Schmidt Process

Q is a orthogonal matrix, meaning it is square with orthonormal columns.

To derive Q from A, we follow the Gram-Schmidt process in order to break down the columns of A into orthonormal columns of Q.

Let the ith columns of A be vectors $\mathbf{a}_i$ so that

$$A = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{bmatrix}$$

Then, let $\mathbf{b}_i$ denote the ith orthogonal column corresponding to each column $\mathbf{a}_i$ of A.

We find $\mathbf{a}_i$ as follows.

First, we some column a of A to begin our process. For sake of simplicity, let this be the first column of A, $\mathbf{a}_1$.

Then, simply let

$$\mathbf{b}_1 = \mathbf{a}_1$$

For the next column $\mathbf{a}_2$ of A, we want $\mathbf{b}_2$ to be orthogonal to $\mathbf{b}_1$. One way to find an orthogonal vector is to take the error vector of the projection of $\mathbf{a}_2$ onto $\mathbf{b}_1$.

Then,

$$\mathbf{b}_2 = \mathbf{a}_2 - \text{proj}_{\mathbf{b}_1}\mathbf{a}_2 = \mathbf{a}_2 - \frac{\mathbf{a}_2 \cdot \mathbf{b}_1}{\mathbf{b}_1 \cdot \mathbf{b}_1}\mathbf{b}_1$$

For the next column $\mathbf{a}_3$ of A, we now need $\mathbf{b}_3$ to be orthogonal to both $\mathbf{b}_1$ and $\mathbf{b}_2$. Then, we first find the error vector of the projection of $\mathbf{a}_3$ onto $\mathbf{b}_1$, and then find the error vector of the projection of that error vector onto $\mathbf{b}_2$.

Then,

$$\mathbf{b}_3 = \mathbf{a}_3 - \text{proj}_{\mathbf{b}_1}\mathbf{a}_3 - \text{proj}_{\mathbf{b}_2}\mathbf{a}_3$$

Generally, for all $\mathbf{b}_i$, $\mathbf{b}_i$ must be orthogonal to all other $\mathbf{b}_j$ for $i \neq j$. So, we continuously find the error vector of $\mathbf{a}_i$ to all existing orthogonal vectors.

In general,

$$\mathbf{b}_i = \mathbf{a}_i - \text{proj}_{\mathbf{b}_1}\mathbf{a}_i - \text{proj}_{\mathbf{b}_2}\mathbf{a}_i - ... - \text{proj}_{\mathbf{b}_{(i-1)}}\mathbf{a}_i$$

Finally, we have obtained an orthogonal set of vectors $\mathbf{b}_i$. However, we want Q to be an orthogonal matrix, requiring the vectors $\mathbf{q}_i$ of Q to also be orthonormal, in addition to orthogonal. So, we find the unit vectors $\mathbf{q}_i$ by dividing $\mathbf{b}_i$ by its magnitude.

So,

$$\mathbf{q}_i = \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|}$$

Then, we have found the orthogonal matrix Q of $A = QR$ where

$$Q = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_n \end{bmatrix}$$

2.1.2 Finding R

Finally, we find the upper triangular matrix R. We calculate R vai $R = Q^T A$.

Since $A = QR$, $Q^{-1}A = R$. Then since Q is orthogonal, $Q^{-1} = Q^T$.

We know the resulting R is upper triangular because Q ins constructed so that each column q of Q is orthonormal to the other columns. Then, for entry $q_{i,j}$ of Q and entry $a_{i,j}$ of A, $r_{i,j} = q_{i,j}^T a_{i,j} = q_{i,j} \cdot a_{i,j}$. If $j > i$, then $q_{i,j}$ is orthogonal to $a_{i,j}$ by the Gram-Schmidt process. So, our ensuing R is upper triangular.

2.2 What is the QR Method?

The QR method is based on the idea that for matrix $A = QR$, matrix $A_1 = RQ$ will have the same eigenvalues as A. This is true because, multiplying both sides of $A = QR$ by Q^{-1}, $Q^-1A = R$, and $A_1 = RQ = Q^{-1}AQ$, which then implies that A_1 is similar to A and has the same eigenvalues.

Then, A_1 can then be factored into $A_1 = Q_1R_1$, and there exists $A_2 = R_1Q_1$ that again has the same eigenvalues as A, by virtue of being similar to A_1.

Eventually, after repeating this process some k times, $A_k = R_{k-1}Q_{k-1}$ will have converged to a triangular matrix. This matrix is known as Schur form of A, where the eigenvalues of A are listed on the diagonal of A_k.

As this process occurs, we continuously multiply an initial identity matrix, named AQ in this paper, by the Q_i we obtain at each ith QR factorization, so that $AQ = \prod_i Q_i$. Then, the columns of AQ will contain the eigenvectors of A.

There are some limitations to this QR method. First of all, our code is limited to symmetric matrices. This is because the QR Method needs extensive modifications in order to handle complex eigenvalues, which occur in non-symmetric matrices.

2.3 Why does this work?

The QR method works by reducing the off-norm (off-diagonal, and therefore non-eigenvalue) values of the matrix.

It preserves the eigenvalues of the matrices along the way, as shown above, while causing off-diagonal values to decay and reduce to approximately zero, obtaining a diagonalized matrix that can be used to determine the eigenvalues.

This is based off of Jacobi's idea of rotations. QR factorization uses Q to rotate the matrix so that non-zero values are pushed towards the diagonal and reducing A to a diagonal matrix A_k.

Now, we examine how AQ is calculated.

$AQ = \prod_i Q_i$ contains the eigenvectors of A due to the similarity of A_i to A.

As we showed before, $A_1 = Q^{-1}AQ$. At this iteration our matrix $AQ = \prod_1 Q_1 = Q$

Then, $A_2 = Q_1^{-1}A_1Q_1$, where Q_1 is the orthogonal matrix of A_1. Plugging in our formula for A_1 from before, $A_2 = Q_1^{-1}Q^{-1}AQQ_1 = (QQ_1)^{-1}A(QQ_1)$. At this time $AQ = \prod_2 Q_2 = QQ_1$

At every subsequent step, A_i can be expressed as $A_i = (QQ_1...Q_{i-1})^{-1}A(QQ_1...Q_{i-1})$ and $AQ = \prod_i Q_i = QQ_1...Q_{i-1}$

Then, when we have obtained our matrix A_k, $A_k = (QQ_1...Q_{k-1})^{-1}A(QQ_1...Q_{k-1})$ and $AQ = \prod_k Q_k = QQ_1...Q_{k-1}$

We can rearrange this equation in terms of A.

$$\begin{aligned} & A_k = (QQ_1...Q_{k-1})^{-1}A(QQ_1...Q_{k-1}) \\ \implies & A_k(QQ_1...Q_{k-1})^{-1} = (QQ_1...Q_{k-1})^{-1}A \\ \implies & (QQ_1...Q_{k-1})A_k(QQ_1...Q_{k-1})^{-1} = A \end{aligned}$$

So

$$A = (QQ_1...Q_{k-1})A_k(QQ_1...Q_{k-1})^{-1}$$

Let $P = QQ_1...Q_{k-1}$. Since P is the product of many orthogonal matrices, P is also orthogonal. So, $A = PA_kP^{-1}$.

Since A is symmetric, by the spectral theorem, there must be an orthogonal diagonalization of A. Let this be $A = XDX^{-1}$ where X is a orthogonal matrix of eigenvectors of A and D is a diagonal matrix of eigenvalues of A.

Then, as A_k is a diagonal matrix containing the eigenvalues of A, and P is an orthogonal matrix, $A = (QQ_1...Q_{k-1})A_k(QQ_1...Q_{k-1})^{-1} = PA_kP^{-1} = XDX^{-1}$, and $X = P = QQ_1...Q_{k-1}$ where P is a matrix containing the eigenvectors of A.

So, $AQ = P$, and $AQ = \prod_k Q_k$ is a matrix with columns containing the orthonormal eigenvectors of A.

2.4 Code

See Appendix 4.1 for pure Julia Code

To implement the QR method, we needed to take a symmetric matrix A.

Then, we knew we had to store the product of each Q_i obtained at each ith step of the QR method.

So, we create an identity matrix AQ as a storage point for $\prod_i Q_i$, setting `AQ = I`

Then, we know that the QR method relies on repeating itself until an upper triangular matrix is reached. So, we construct a loop with a break condition on whether or not A_i is upper triangular.

We let this while loop be

```
while NOT is upper triangular (A_i)
```

so that when A_i is upper triangular our loop will exit.

Then, we used the Julia function `qr` to find Q and R decomposition of A_i.

We then multiplied AQ by Q every single time so that

```
Q, R = qr(A_i)
AQ = AQ * Q
```

Finally, we obtained a new A_{i+1} by multiplying R and Q together again.

For the sake of efficiency, our code needs to ignore sufficiently small values. After this, we check every entry of A_{i+1} so that if that entry is less than a threshold of 10^{-5}, we can discard it.

We implemented this by using Julia to check every entry so that

```
if absolute value(A_i+1[index]) < 10^-5
    then A_i+1[index] = 0
```

Finally we check to see that all off-diagonal values of A_{i+1} are zero. We can do this by finding the magnitude of the matrix, excluding its diagonal. We use Julia's `norm` function for this that calculates magnitude, so that if

```
norm(A - diagonals(A)) < 10^-5
```

, we `break`.

2.5 Examples

For example, take the example matrix

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

Our code first finds its QR decomposition using the generic `qr()` command in Julia. Performing the QR decomposition, we get the following matrices Q and R:

$$Q = \begin{bmatrix} -0.9428 & 0.2722 & 0.1925 \\ -0.2357 & -0.9526 & -0.1925 \\ -0.2357 & -0.1361 & 0.9623 \end{bmatrix}$$

$$R = \begin{bmatrix} -4.24264 & -2.12132 & -2.12132 \\ 0 & -3.67423 & -1.22474 \\ 0 & 0 & 3.4641 \end{bmatrix}$$

Now, we compute $A_1 = RQ$:

$$A_1 = \begin{bmatrix} -4.24264 & -2.12132 & -2.12132 \\ 0 & -3.67423 & -1.22474 \\ 0 & 0 & 3.4641 \end{bmatrix} \begin{bmatrix} -0.9428 & 0.2722 & 0.1925 \\ -0.2357 & -0.9526 & -0.1925 \\ -0.2357 & -0.1361 & 0.9623 \end{bmatrix}$$

After performing the matrix multiplication, we get:

$$A_1 = \begin{bmatrix} 5.0 & 1.1547 & -0.816497 \\ 1.1547 & 3.66667 & -0.471405 \\ -0.816497 & -0.471405 & 3.33333 \end{bmatrix}$$

and A_1 has a QR decomposition of

$$A_1 = \begin{bmatrix} -0.9638 & -0.3470 & -0.2131 \\ -0.2317 & 0.8974 & -0.3737 \\ 0.1346 & 0.2674 & 0.9555 \end{bmatrix} \begin{bmatrix} -5.19615 & -2.0 & 1.41421 \\ 0.0 & -3.31662 & 0.426401 \\ 0.0 & 0.0 & 3.1334 \end{bmatrix}$$

where

$$Q_1 = \begin{bmatrix} -0.9638 & -0.3470 & -0.2131 \\ -0.2317 & 0.8974 & -0.3737 \\ 0.1346 & 0.2674 & 0.9555 \end{bmatrix}$$

and

$$R_1 = \begin{bmatrix} -5.19615 & -2.0 & 1.41421 \\ 0.0 & -3.31662 & 0.426401 \\ 0.0 & 0.0 & 3.1334 \end{bmatrix}$$

Then, the above steps are repeated for A_2 until A_i, when at the ith step our diagonalization process is complete.

$$A_2 = \begin{bmatrix} 5.66667 & 0.80403 & 0.492366 \\ 0.80403 & 3.24242 & 0.148454 \\ 0.492366 & 0.148454 & 3.09091 \end{bmatrix}$$

. . .

$$A_{i-2} = \begin{bmatrix} 6.0 & 2.8032 \times 10^{-5} & -1.61844 \times 10^{-5} \\ 2.8032 \times 10^{-5} & 3.0 & 0.0 \\ -1.61844 \times 10^{-5} & 0.0 & 3.03 \end{bmatrix}$$

$$A_{i-1} = \begin{bmatrix} 6.0 & 1.4016 \times 10^{-5} & 0.0 \\ 1.4016 \times 10^{-5} & 3.0 & 0.0 \\ 0.0 & 0.0 & 3.03 \end{bmatrix}$$

$$A_i = \begin{bmatrix} 6.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 3.0 \end{bmatrix}$$

And we finally arrive at the diagonalized A_i, with the correct eigenvalues across its diagonal.

Throughout this process, we continuously multiply each Q we obtain into the product $\prod_i Q_i$. This gives us the matrix

$$AQ = \prod_i Q_i = \begin{bmatrix} -0.57735 & -0.57735 & -0.57735 \\ -0.57735 & 0.788675 & -0.211325 \\ -0.57735 & -0.211325 & 0.788675 \end{bmatrix}$$

As we proved above, the columns of this matrix will return to us the eigenvectors of A

2.6 Improving the QR Method

We used the technique of shifting to improve our QR Method.

In this technique, we continuously subtract a scalar value μ from the diagonal of our matrix before performing QR decomposition, so that our QR decomposition uses a shifted matrix. Then, when computing the new matrix from RQ, we add back μ to the diagonal.

In other words, first we find the QR decomposition of $A - \mu I$, where I is the identity matrix. Then, we find our new matrix A_1, where $A_1 = RQ + \mu I$.

This happens every iteration of the standard QR method.

2.6.1 What do shifts do?

Shifting is effective because it improves the speed of convergence by modifying the ratio of eigenvalues relative to one another.The convergence speed increases as the ratio of λ_2 and λ_1 increases for some eigenvalues λ_1 and λ_2, where $\lambda_1 > \lambda_2$.

Shifting increases the difference of two eigenvalues, improving convergence speed.

This will be explained in greater detail for one specific type of shift.

2.6.2 Types of Shifting

There are many types of shifts, but three commonly seen shifts are the simple shift, Rayleigh's shifts, and Wilkinson's shift.

Simple shifts are reflected by their name. They simply choose some constant μ can continuously subtract it from a diagonal.

Rayleigh and Wilkinson shifts are more complicated. Rayleigh shifts derive an approximate μ from the Rayleigh quotient.

Wilkinson's shifts are the most commonly used in practice. This is due to its efficiency and that, unlike Rayleigh shifts, they do not have the possibility of failing to converge.

We chose to implement Wilkinson's shift.

2.6.3 Wilkinson's shifts

Wilkinson's shift focuses on the smallest submatrix B of A. This is the bottom-right submatrix of A.

Let us express B as

$$B = \begin{bmatrix} b_{n-1,n-1} & b_{n-1,n} \\ b_{n,n-1} & b_{n,n} \end{bmatrix}.$$

Where B is the smallest submatrix of a n by n matrix.

Then, it computes the eigenvalues of B. Since B is so small, its eigenvalue can be computed in constant time via the equation

$$\lambda^2 - (b_{n-1,n-1} + b_{n,n})\lambda + \left(b_{n-1,n-1}b_{n,n} - b_{n-1,n}b_{n,n-1}\right) = 0.$$

Since B must have two eigenvalues as a two by two matrix, we then select the eigenvalue closest to $b_{n,n}$. By decreasing the size of the bottom-right off-diagonal entry, we accelerate convergence.

We determined this eigenvalue in Julia by comparing the absolute values of the difference between our two eigenvalues of b and $b_{n,n}$

We can take the example of a matrix

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

The eigenvalues of this matrix are -6, -3, and 4.

With bottom-right 2 by 2 submatrix

$$\begin{bmatrix} -3 & 0 \\ 0 & -4 \end{bmatrix}$$

We can use `eigvals` function to find the eigenvalues of this matrix, which turn out to be -3 and -4.

Then, we examine the difference between the bottom-most diagonal entry of A, -4, and each of these eigenvalues.

It turns out the -4 and -4 are closer than -3 and -4, so we choose -4 for our shift value.

Then, we subtract -4 from all diagonal entries.

This gives us

$$A - -4I = \begin{bmatrix} 6 & 0 & 4 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

From this matrix, we get $A_1 = RQ = \begin{bmatrix} 7.84615 & 0.0 & 1.23077 \\ 0.0 & 1.0 & 0.0 \\ 1.23077 & 0.0 & -1.84615 \end{bmatrix}$

Which we then add our shift back to for the matrix

$$\begin{bmatrix} 3.84615 & 0.0 & 1.23077 \\ 0.0 & -3.0 & 0.0 \\ 1.23077 & 0.0 & -5.84615 \end{bmatrix}$$

Which has greatly reduced off diagonal entires and the same eigenvalues as initial matrix A.

2.6.4 Why does this Work?

Wilkinson's shift focuses on the two by two trailing submatrix. As such, it isolates the eigenvalues most likely to converge next, the bottom-most two, which ensures that off-diagonal elements are reduced as quickly as possible.

This is because Wilkinson's shift reduces the ratio of $\frac{\lambda_n}{\lambda_{n-1}}$ by finding a μ value as close to λ_n as possible. Then, the resulting submatrix has an eigenvalue ratio of $\frac{\lambda_n - \mu}{\lambda_{n-1} - \mu}$, where $\lambda_n - \mu$ is minimized as small as possible.

This improves convergence by increasing the difference between the largest and smallest eigenvalues of our submatrix.

2.6.5 Limitations

The QR method can find eigenvalues and eigenvectors for symmetric matrices.

However, while we found that shifts were efficient for smaller matrices, for larger matrix, like 200 by 200 matrices, convergence was notably slower.

There are some sources of error that could account for this.

First, shifts typically need to be used in conjunction to create a practical QR algorithm. Alone, shifts improve the efficiency for code, but also lead to some time loss as time is taken to calculate shifts and eigenvalues.

Along with that, no matter how efficient, it will take longer to reduce matrices of greater size.

2.6.6 Code

See Appendix B 4.1.1 for pure Julia Code.

Shifting modifies the contents of the while loop defined for the QR method in section 2.4

First, we need to prepare for our shift calculation by finding the size of the matrix. We can use the built in size function for this and store it in variable n so that

```
n = size(A, 1)
```

Then, we move into the while loop, where our first step is to calculate our eigenvalues of our bottom right most 2 by 2 submatrix

We do this using the previously calculated size of the matrix n, and the `eigvals` function,

```
eigenvalues = eigvals(A[n-1 to n, n-1 to n]
```

Then, we need to compare it with the bottommost entry at $A[n, n]$

To do this, we compare the absolute value differences of each eigenvalue of our submatrix.

So if

```
abs(eigenvalue one - A[n,n]) < abs(eigenvalue two - A[n,n])
```

we let our "shift" be equal to "eigenvalue one".

If not, then the difference of our eigenvalues is relatively same relative to $A[n, n]$, or eigenvalue one is farther away than eigenvalue two. So we let "shift" be equal to "eigenvalue two".

Then, we prepare to subtract our shift from all diagonal entires of A.

We do this by multiplying our shift by the identity matrix so that

```
SH = shift * I
```

To create a matrix with only diagonal entries of shift.

Then, we find the qr decomposition of SH subtracted from A

```
Q, R = qr(A - SH)
```

And finally add SH back to find A_1

```
A_1 = R * Q + SH
```

To make the modifications to our code necessary to include shifting in our QR algorithm.

2.6.7 Example

Consider the previous example

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

When using shifting, the QR method primarily differs in that before finding the QR decomposition of A_i, we first determine the shift by which we will change the diagonals of our matrix.

We find our shift by examining the bottom right 2 by 2 submatrix,

$$\begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix}$$

This matrix has eigenvalues of 3 and 5, calculated via the `eigvals` function in julia.

Then, we choose which eigenvalue to use. This is done by comparing their relative distance to the bottom-most eigenvalue of this submatrix.

In our case, 3 and 5 are both equal distances away from 4. We account for this by arbitrarily choosing to proceed with the first eigenvalue, 3, when ties like this occur. However, typically, our code compares the absolute difference between the eigenvalues of the submatrix and the bottom most eigenvalue of A to determine which one to proceed with.

Then, we use this eigenvalue of the submatrix and subtract it from all diagonal values of A.

We do this by multiplying 3 by the identity matrix and subtracting from A, like so:

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

Then, we take the QR decomposition of $A - 3I$ to get

$$A = \begin{bmatrix} -0.57735 & -0.57735 & -0.57735 \\ -0.57735 & 0.788675 & -0.211325 \\ -0.57735 & -0.211325 & 0.788675 \end{bmatrix} \begin{bmatrix} -1.73205 & -1.73205 & -1.73205 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Then, we find $A_1 = RQ + 3I$, adding back the shift to maintain our eigenvalues.

Note that

$$RQ = \begin{bmatrix} -1.73205 & -1.73205 & -1.73205 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} -0.57735 & -0.57735 & -0.57735 \\ -0.57735 & 0.788675 & -0.211325 \\ -0.57735 & -0.211325 & 0.788675 \end{bmatrix} = \begin{bmatrix} 3.0 & 0 & 0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Then, adding back $3I$

$$RQ + 3I = \begin{bmatrix} 3.0 & 0 & 0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} + \begin{bmatrix} 3.0 & 0 & 0 \\ 0.0 & 3.0 & 0 \\ 0.0 & 0.0 & 3.0 \end{bmatrix} = \begin{bmatrix} 6.0 & 0 & 0 \\ 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 3.0 \end{bmatrix}$$

Which is a diagonalized matrix of our eigenvalues. So, by using Wilkinson's shift, we have reduced the QR algorithm for

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

to just one shift.

And just as in the process without shifting, we take the products of all Q_i obtained from QR factorization. In this case, there is only one step, and we see that

$$\begin{bmatrix} -0.57735 & -0.57735 & -0.57735 \\ -0.57735 & 0.788675 & -0.211325 \\ -0.57735 & -0.211325 & 0.788675 \end{bmatrix}$$

is the eigenvectors of A.

So, Wilkinson's shift has found accurate eigenvalues and eigenvectors and far fewer steps than the generic QR method.

While this may not have great effect on the time our code takes to run for such a small matrix, for larger matrices Wilkinson's shift greatly improves the efficacy of code and speed of eigenvalue and eigenvector finding.

3 Appendix A (Power Method Codes)

3.1 Singular Vectors

Functions are tested on matrix:

$$\begin{bmatrix} 12 & 3 & 5 & 7 & 2 & 9 & 4 & 1 & 11 & 6 \\ 2 & 15 & 3 & 7 & 6 & 5 & 8 & 9 & 1 & 10 \\ 4 & 1 & 16 & 8 & 7 & 5 & 9 & 3 & 12 & 2 \\ 3 & 6 & 9 & 14 & 4 & 11 & 13 & 7 & 10 & 15 \\ 5 & 7 & 6 & 4 & 18 & 3 & 2 & 9 & 1 & 13 \\ 11 & 8 & 7 & 5 & 12 & 17 & 3 & 2 & 6 & 14 \\ 1 & 2 & 3 & 4 & 5 & 6 & 19 & 8 & 11 & 10 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 6 & 5 & 3 & 4 & 1 & 2 & 7 & 8 & 19 & 20 \\ 8 & 4 & 3 & 12 & 9 & 1 & 6 & 11 & 10 & 7 \end{bmatrix}$$

3.1.1 Dominant Singular Vectors

```
function pow_singular_dom(A, n)
    At = transpose(A)
    B = At * A
    x0 = randn(n)
    x = x0 / norm(x0)
    v0 = x0
    v1 = x0
    itr = 0
    while (true)
        itr += 1
        x = B * x
        x = x / norm(x)
        v1 = x
        if (norm(v1-v0) < 0.00001)
            println("$itr iterations in total for approximation")
            return v1
        end
        v0 = x
    end
end
```

Usage:

```
v_1 = pow_singular_dom(A0, 10)
sigma_1 = sqrt((dot(transpose(A0)*A0*v_1, v_1)) / (transpose(v_1)*(v_1)))
u_1 = (A0 * v_1) / sigma_1
println(sigma_1)
println(v_1)
println(u_1)
```

Sample output:

```
6 iterations in total for approximation
83.29334473649467
[0.22962826154363034, 0.2353185064659048, 0.2549647844903884,
   0.29893338662236135, 0.2943118461306329, 0.2953251378476473,
   0.3404634370987766, 0.29791836900002205, 0.38741939171379086,
   0.45708528697080036]
[0.2249795518114296, 0.24554427595433936, 0.24831123291357401,
   0.3631538243550749, 0.2568776909550959, 0.3191261721028381,
   0.2832041592547104, 0.5217652027248801, 0.32018425243238513,
   0.2697478477155878]
```

3.1.2 All Singular Vectors

```
function print_results(v, u, sigma, i)
    print("Singular value #$i: ")
    println(sigma)

    print("v_$i: ")
    println(v)

    print("u_$i: ")
    println(u)

    println()
end
```

```
function pow_singular_all(A, n)
    i = 1
    A0 = A
    B = transpose(A) * A
    v_cur = pow_singular_dom(A, n)
    sigma_cur = sqrt((dot(B * v_cur, v_cur)) / (transpose(v_cur) * (v_cur)
       ))
    u_cur = (A0 * v_cur) / sigma_cur

    while true
        print_results(v_cur, u_cur, sigma_cur, i)
        if (i == n)
            break
        end
        i += 1
        A = A - (sigma_cur * u_cur * transpose(v_cur))
        B = transpose(A) * A
        v_cur = pow_singular_dom(A, n)
        sigma_cur = sqrt((dot(B * v_cur, v_cur)) / (transpose(v_cur) * (
           v_cur)))
        u_cur = (A0 * v_cur) / sigma_cur
        if (sigma_cur <= 10^(-5))
            i-=1
            break
        end
    end
    println("Found $i nontrivial singular values with corresponding
       singular vectors")
end
```

Usage:

```
pow_singular_all(A0, 10)
```

Sample output:

```
6 iterations in total for approximation
Singular value #1: 83.2933447364945
v_1: [0.2296282452313325, 0.23531843595329605, 0.25496479808615435,
    0.2989333993926933, 0.2943117392054714, 0.2953251182646323,
    0.340463509587474, 0.2979183497085729, 0.3874195029779303,
    0.4570852613061965]
u_1: [0.22497956022230073, 0.2455442586817052, 0.24831124751145514,
    0.3631538330112947, 0.25687765881666874, 0.3191261515326029,
    0.28320417693463934, 0.5217652026632449, 0.32018426983061216,
    0.26974784717879163]

22 iterations in total for approximation
Singular value #2: 22.964906911870287
v_2: [-0.13316507780608208, -0.3461185969960594, 0.02002740685252678,
    0.05730216420726402, -0.5659597018045277, -0.16757230551641278,
    0.4108896229621628, -0.04151971592769828, 0.5748180316898716,
    -0.09707493781109386]
u_2: [0.1117980932076607, -0.2923165902996552, 0.23416648101531878,
    0.16294946582576664, -0.5951832693605938, -0.4444964275110305,
    0.36819876410793767, 0.0029764511104139035, 0.36502288574916025,
    0.004963232556589676]

44 iterations in total for approximation
Singular value #3: 19.21400701734439
v_3: [-0.3887462689005929, 0.24394703058077766, -0.4249482444282847,
    -0.09489209363732985, -0.01925381052471158, -0.45216773883628086,
    0.23617782738870652, 0.4586401823812279, -0.1503330721097477,
    0.3259194316551336]
u_3: [-0.4749094234402444, 0.4003489471442956, -0.4640156045089277,
    -0.012500605242004182, 0.1987250787750045, -0.43743820401070943,
    0.28091432967067287, 0.03844945330790313, 0.2755115917506712,
    0.10757934822107515]

...

2 iterations in total for approximation
Singular value #10: 1.116299138183247
v_10: [-0.48362697280754086, 0.2411481886702638, -0.356556625704616,
    0.19307161343855486, 0.36895600706077925, 0.10853697301843045,
    -0.14202827292716605, -0.3552175338391786, 0.4675280387490968,
    -0.17521536374462354]
u_10: [-0.5629589158724406, 0.06307594423055642, 0.16812947480692722,
    -0.1129701237322431, -0.3776925230759731, 0.5079143716934528,
    0.0426992957547717, -0.22420800253903886, 0.12130522343790794,
    0.4130772914239495]

Found 10 nontrivial singular values with corresponding singular vectors
```

4 Appendix B (QR Method Codes)

4.1 QR

```
using LinearAlgebra

temp_A = A
AQ = I
while istriu(temp_A) == false
    Q, R = qr(temp_A)
    AQ = AQ * Q
    temp_A = R*Q

    temp_A[abs.(temp_A) .< 10^-5] .= 0
    if norm(temp_A - Diagonal(temp_A)) < 10^-5
        break;
    end
end
display(temp_A)
```

For the below matrix input

$$\begin{bmatrix} 35 & 29 & 32 & 27 & 31 & 30 & 30 & 33 \\ 29 & 46 & 37 & 32 & 37 & 39 & 42 & 42 \\ 32 & 37 & 43 & 30 & 36 & 34 & 32 & 34 \\ 27 & 32 & 30 & 28 & 31 & 31 & 31 & 32 \\ 31 & 37 & 36 & 31 & 41 & 40 & 36 & 39 \\ 30 & 39 & 34 & 31 & 40 & 43 & 36 & 39 \\ 30 & 42 & 32 & 31 & 36 & 36 & 43 & 43 \\ 33 & 42 & 34 & 32 & 39 & 39 & 43 & 46 \end{bmatrix}$$

Sample Output:

```
   283.848    0.0      0.0       0.0       0.0       0.0       0.0        0.0
   0.0     16.1571   0.0       0.0       0.0       0.0       0.0        0.0
   0.0      0.0      9.72915   0.0       0.0       0.0       0.0        0.0
   0.0      0.0      0.0       9.10384   0.0       0.0       0.0        0.0
   0.0      0.0      0.0       0.0       2.91103   0.0       0.0        0.0
   0.0      0.0      0.0       0.0       0.0       2.30394   0.0        0.0
   0.0      0.0      0.0       0.0       0.0       0.0       0.850434   0.0
   0.0      0.0      0.0       0.0       0.0       0.0       0.0        0.0961563
```

Display AQ

```
 0.37077    0.42729     0.460441  ..  -0.565161    0.142886  -0.0472726
 0.398507   0.0937668  -0.0456594      -0.241166   -0.177208  -0.0890704
 0.305786  -0.513515   -0.480015       -0.462569   -0.074499  -0.252332
 0.475749   0.189918   -0.175191        0.375853   -0.631227   0.0611067
 0.320661  -0.572269    0.55009         0.318465    0.153157  -0.323675
 0.378731   0.362463   -0.260031  ..    0.401401    0.514732  -0.295694
 0.257281  -0.160801   -0.28382        -0.0143107   0.492988   0.575678
 0.267256  -0.155358    0.272091        0.0667016  -0.112873   0.631523
```

4.1.1 Wilkinson's Shift

In the below code, `temp_A` is a temporary array meant to store the matrix A we are finding the eigenvectors and eigenvalues of.

`AQ`$= \prod_i Q_i$, which, as proven before, is a matrix containing the eigenvectors of A

```
temp_A = A
n = size(temp_A, 1)
AQ = I
i = 0;

while istriu(temp_A) == false
    i += 1
    phi = eigvals(temp_A[n-1:n, n-1:n])
    REF = temp_A[n,n]
    if abs(phi[1] - REF) <= abs(phi[2] - REF)
        shift = phi[1]
    else
        shift = phi[2]
    end
    SH = shift * I

    Q, R = qr(temp_A - SH)
    AQ = AQ * Q
    temp_A = R * Q + SH

    temp_A[abs.(temp_A) .< 10^-5] .= 0
    if norm(temp_A - Diagonal(temp_A)) < 10^-5
        break;
    end
end
```

For the below matrix input

$$\begin{bmatrix} 35 & 29 & 32 & 27 & 31 & 30 & 30 & 33 \\ 29 & 46 & 37 & 32 & 37 & 39 & 42 & 42 \\ 32 & 37 & 43 & 30 & 36 & 34 & 32 & 34 \\ 27 & 32 & 30 & 28 & 31 & 31 & 31 & 32 \\ 31 & 37 & 36 & 31 & 41 & 40 & 36 & 39 \\ 30 & 39 & 34 & 31 & 40 & 43 & 36 & 39 \\ 30 & 42 & 32 & 31 & 36 & 36 & 43 & 43 \\ 33 & 42 & 34 & 32 & 39 & 39 & 43 & 46 \end{bmatrix}$$

Sample Output:

```
283.848    0.0      0.0       0.0       0.0       0.0       0.0          0.0
  0.0     16.1571   0.0       0.0       0.0       0.0       0.0          0.0
  0.0      0.0      9.72915   0.0       0.0       0.0       0.0          0.0
  0.0      0.0      0.0       9.10384   0.0       0.0       0.0          0.0
  0.0      0.0      0.0       0.0       2.91103   0.0       0.0          0.0
  0.0      0.0      0.0       0.0       0.0       2.30394   0.0          0.0
  0.0      0.0      0.0       0.0       0.0       0.0       0.0961563    0.0
  0.0      0.0      0.0       0.0       0.0       0.0       0.0          0.850434
```

Display AQ

```
 0.306526   -0.45233      0.633406   ..   0.277559     0.182164     0.208796
 0.381342    0.317289    -0.165544         0.272963     0.539108     0.070704
 0.346142   -0.593709    -0.137885         0.0671155   -0.326352    -0.0894484
 0.301764   -0.0929886    0.0402814       -0.753057     0.029325    -0.363893
 0.363696   -0.135222    -0.327679        -0.280368     0.441485     0.316355
 0.365445    0.0170834   -0.52742    ..    0.332524    -0.39593     -0.0298661
 0.367398    0.453949     0.289733        -0.220455    -0.465958     0.535498
 0.385925    0.330064     0.285074         0.197671    -0.0051588   -0.650581
```

References

[1] Blum, A., Hopcroft, J., & Kannan, R. (2020). Best-Fit subspaces and Singular Value Decomposition (SVD). In *Foundations of Data Science* (pp. 29–61). Cambridge: Cambridge University Press.

[2] Strang, G. (2016). *Introduction to linear algebra* (5th ed.) (pp. 529-530). Wellesley-Cambridge Press.

[3] Descloux, J., Fattebert, J., Gygi, F., Gruber, R., & Rappaz, J. (1998). Rayleigh Quotient Iteration, an old recipe for solving modern large-scale eigenvalue problems. *Computers in Physics, 12*(1), 22–27. https://doi.org/10.1063/1.168687

[4] Schilling, R., Nachtergaele, B., & Lankham, I. (2021). *The Gram-Schmidt Orthogonalization Procedure*. Retrieved from https://math.libretexts.org/Bookshelves/Linear_Algebra/Book%3A_Linear_Algebra_(Schilling_Nachtergaele_and_Lankham)/09%3A_Inner_product_spaces/9.05%3A_The_Gram-Schmidt_Orthogonalization_procedure.

[5] Serre, D. (2010). Matrices. In Graduate texts in mathematics. Springer Nature. https://doi.org/10.1007/978-1-4419-7683-3

[6] Barkmeijer, J. (1995). Approximating dominant eigenvalues and eigenvectors of the local forecast error covariance matrix. ¡i¿Tellus A: Dynamic Meteorology and Oceanography¡/i¿, ¡i¿47¡/i¿(4), 495-501. https://doi.org/10.3402/tellusa.v47i4.11536