The Power Method, QR Method, and Deflation

Eli Slothower

December 9, 2022

1 Introduction

It is well known that to find the eigenvalues, eigenvectors, singular values, and singular vectors of an $n \times n$ matrix A, one can start by calculating the roots of its characteristic polynomial, namely those of $A - \lambda I$, and assigning each root as an eigenvalue of A. From there, the eigenvectors, singular values, and singular vectors can be found through various other methods. However, finding the roots of the characteristic polynomial is far too slow of a process, even for computers, and, in fact, this method is impossible when $n > 4$.[6] Iterative methods, such as the Power Method and the QR Method, are much faster at calculating such things, and there aren't any restrictions how small or large n must be. Even so, however, these methods can still be improved upon. The Power Method, for example, falls short in that it only computes the dominant eigenvalue, eigenvector, singular value, and singular vectors of a matrix, rather than *all* eigenvalues, eigenvectors, et cetera. This paper and its accompanying software will explore these methods, as well as methods of improving the Power Method, and uncover how they work.

2 The Power Method

As stated above, the Power Method is an iterative method of calculating the dominate eigenvalue, eigenvector, singular value, and singular vectors of a matrix A. We will discuss later on how to find all values (rather than just the dominant values) of a matrix with the Power Method. Before looking at how the Power Method works, let us first define what it means to be an *iterative* method. According to the Oxford Dictionary, iteration is defined as "the action of iterating or repeating, or process of being iterated."[3] Thus we can conclude that an iterative method is one that repeats its process until it comes to its result.

The Power Method tells us that if A is an $n \times n$ symmetric matrix with n real, distinct eigenvalues, then the Power Method will return λ_1 where $|\lambda_1| > |\lambda_2| > ... > |\lambda_n|$.[5, 6] There are still two main questions left to answer regarding this method though, namely (1) "How does this method work?", and (2) "Why does this method work?". Let us continue by answering the first question.

To begin, create a vector $\vec{x}_0$ as an initial approximation of the dominant eigenvector of A.[4] Next, find $\vec{x}_1$ by computing $A\vec{x}_0$. Next, find $\vec{x}_2$ by computing $A\vec{x}_1$. This is the point at which the *iterative* process of the Power Method begins to show itself; continue this process to find $\vec{x}_i$ for $0 \leq i \leq p$ where p is an integer such that $A\vec{x}_p$ converges to a point. Given a symmetric matrix A, as specified, the Power Method will always converge to a point. Once $\vec{x}_p$ has been calculated, the dominant eigenvalue, λ_1, can be found by calculating $\frac{\vec{x}_p^T \vec{x}_{p+1}}{\vec{x}_p^T \vec{x}_p}$.[5] The dominant eigenvalue has now been found, and the eigenvectors, singular values, and singular vectors can now be found from here, which we will discuss how to do later.

Why does this method work? When we look at the proof, it becomes clear.

Lemma 1: $\lim_{p\to\infty} \sum_{i=2}^{n} c_i(\frac{\lambda_i}{\lambda_1})^p e_i = 0$ because as p increases, $|\lambda_1|^p > |\lambda_i|^p$ since λ_1 is the leading eigenvalue of A, so $\lim_{p\to\infty}(\frac{\lambda_i}{\lambda_1})^p = 0$.

Theorem 1:[5, 6] If A is an $n \times n$ symmetric matrix with n real, distinct eigenvalues, then the Power Method will return the dominant eigenvalue of A, denoted as λ_1.

Let $\vec{x}_0 \in R^n$ as an initial approximation of the dominant eigenvector. Because A has n distinct eigenvalues, we know that A has n linearly independent eigenvectors. Thus, let us represent $\vec{x}_0$ as a linear combination of the eigenvectors, e_i, with constants c_i.

$$\begin{aligned}
\text{Let } \vec{x}_0 &= c_1e_1 + c_2e_2 + ... + c_ne_n \\
&\text{(i.e. an approximation of one of the dominant eigenvectors of } A) \\
&= \sum_{i=1}^{n} c_ie_i \qquad \text{(by summation form)}
\end{aligned}$$

$$\begin{aligned}
\text{Let } \vec{x}_1 &= A\vec{x}_0 \\
&= A\sum_{i=1}^{n} c_ie_i && \text{(by summation form)} \\
&= \sum_{i=1}^{n} c_iAe_i && \text{(by linearity)} \\
&= \sum_{i=1}^{n} c_i\lambda_ie_i && \text{(by } Ae_i = \lambda_ie_i)
\end{aligned}$$

$$\begin{aligned}
\text{Let } \vec{x}_p &= A^p \vec{x}_0 \\
&= \sum_{i=1}^{n} c_i \lambda_i^p e_i \\
&\qquad \text{(by powers of a diagonalizable matrix)} \\
&= c_1 \lambda_1^p e_1 + \sum_{i=2}^{n} c_i \lambda_i^p e_i \\
&\qquad \text{(by factorization)} \\
&= \lambda_1^p (c_1 e_1 + \sum_{i=2}^{n} c_i (\frac{\lambda_i}{\lambda_1})^p e_i) \\
&\qquad \text{(by factorization)}
\end{aligned}$$

$$\implies \vec{x}_p = \lambda_1^p (c_1 e_1 + \sum_{i=2}^{n} c_i (\frac{\lambda_i}{\lambda_1})^p e_i) \approx \lambda_1^p c_1 e_1 \qquad \text{(by Lemma 1)}$$

$$\implies x_{p+1} \approx \lambda_1^{p+1} c_1 e_1$$

$$\begin{aligned}
\text{Then, } \frac{\vec{x}_p^T \vec{x}_{p+1}}{\vec{x}_p^T \vec{x}_p} &= \frac{(c_1 \lambda_1^p e_1^T)(c_1 \lambda_1^{p+1} e_1)}{(c_1 \lambda_1^p e_1^T)(c_1 \lambda_1^p e_1)} \\
&\qquad \text{(by substitution)} \\
&= \frac{e_1^T \lambda_1 e_1}{e_1^T e_1} \quad \text{(by division of constants)} \\
&= \frac{\lambda_1 e_1^T e_1}{e_1^T e_1} \qquad \text{(by linearity)} \\
&= \lambda_1 \\
&\text{(by division of constants (dot product of } e_1^T e_1 \text{ results in a constant))}
\end{aligned}$$

$$\therefore \text{By the Power Method, } \lambda_1 = \frac{\vec{x}_p^T \vec{x}_{p+1}}{\vec{x}_p^T \vec{x}_p}$$

□

Now let us discuss how to find the dominant eigenvector, dominant singular value, and dominant singular vectors from the Power Method.

1). Let s be the maximum component in $\vec{x}_p$. Then, from the Power Method, the dominant eigenvector of A is $\vec{x}_p$ by definition of $\vec{x}_p$.

2). Let λ_1 be the dominant eigenvalue of A returned from the Power Method. Then the dominant singular value of A is $\sqrt{\lambda_1}$, by definition of singular value.

3). Let $\vec{x}_{p,A^TA}$ be the dominant eigenvector of A^TA returned from the Power Method. Then the dominant right singular vector of A is λ_{1,A^TA}, by definition of right singular vector.

4). Let $\vec{x}_{p,AA^T}$ be the dominant eigenvector of AA^T returned from the Power Method. Then the dominant left singular vector of A is λ_{1,AA^T}, by definition of left singular vector.

Now that we know both how the Power Method works and why the Power Method works, let us take a look at the accompanied software to gain some more intuition about the code, as well as the provided examples.

Note: Before continuing, please refer to the appendix and review all necessary functions and examples. Note that all of our Power Method functions take in a matrix A that abides by the preconditions, as well as an $n \times 1$ initial approximation x.

Example PM.1: This is a trivial case. The purpose of this example is to show that our function works on trivial cases. We know that the dominant eigenvalue is 2 since A is a diagonal matrix. This is what our getEvaluePM(A,x) returns. We know that the corresponding eigenvector is [1, 0], which is what our getEvectorPM(A, x) returns. Next, we know the corresponding singular value should be $\sqrt{2}$, which is what our getSingularValuePM(A,x) returns. Lastly, our getSingularVectorAtAPM(A,x) and getSingularVectorAAtPM(A,x) correctly return the left and right singular vectors, which we can verify with a singular vector calculator.

Example PM.2: This is a non-trivial case, although it is not an extremely difficult one. The purpose of this example is to show that our function works on normal-difficulty cases. Using an eigenvalue calculator, we can see that the dominant eigenvalue of A is 17.034, which our getEvaluePM(A,x) returns. Similarly, we know that the dominant eigenvector of A is [0.577; 0.547; 0.606], which our getEvectorPM(A,x) returns. Our getSingularValuePM(A,x) properly returns $\sqrt{17.034}$, which we would expect. Lastly, our getSingularVectorAtAPM(A,x) and getSingularVectorAAtPM(A,x) properly returns the dominant singular vector that a singular vector calculator returns.

Example PM.2.1: This is an extension of Example PM.2. The purpose of this example is to show that our intitial approximation $\vec{x}$ can truly be arbitrary (with the appropriate dimensions of course), and we still get the expected results with all of our functions. As we can see, all results in this example are the same as the results in that of Example PM.2, which we would expect.

Example PM.3: This is a non-trivial, difficult case. The purpose of this example is to show that our functions work on large matrices. To be said succinctly, all five of our Power Method functions properly return the expected answers for this large matrix, which we can verify with various calculators.

We have now shown that our Power Method functions work on trivial and non-trivial cases.

3 The QR Method

In its current form, we can see that the Power Method isn't as useful as we want it to be; we would like to be able to find *all* eigenvalues, eigenvectors, singular values, and singular vectors of a matrix. It is here at which the QR Method becomes useful. The QR Method tells us that if we have an $n \times n$ real matrix A, then we will come to a (nearly) upper-triangular matrix A_k that contains approximations to the eigenvalues of A on the diagonal.[2] We say that it is "nearly" upper-triangular because the components below the diagonal will either converge to 0, or become negligible ($\sim >0.1$) Again, we are left with two main questions to answer regarding this method, namely (1) "How does this method work?", and (2) "Why does this method work?". Let us continue by answering the first question.

To start, use QR decomposition to factor A into Q_1R_1. Then let $A_1 = R_1Q_1$. Next, use QR decomposition to factor A_1 into Q_2R_2. Then let $A_2 = R_2Q_2$. It is at this point that the iterative process becomes clear; continue this process until A_k seems to converge, as described above. Once it does, the elements on the diagonal of A_k will be the eigenvalues of A.[2, 6]

Why does this method work? Let us take a look at the proof.

Theorem 2:[6] If A is an $n \times n$ nonsingular matrix, then the QR Method will return a matrix with its eigevalues equal to those of the eigenvalues of A.

Note: We will assume that the matrix the QR Method returns is an upper-triangular matrix, meaning that its eigenvalues are the components on its diagonal. We will not be proving in this paper why the QR Method returns a matrix that converges into such a matrix; we are instead simply proving that its eigenvalues are equal to those of A.

Let A be an $n \times x$ nonsingular matrix.

$$A = Q_1R_1 \qquad \text{(by QR decomposition)}$$

$$\text{Let } A_1 = R_1Q_1$$

Note: A_1 has the same eigenvalues as A because $A = Q_1R_1$ is similar to $A_1 = Q^{-1}AQ$. We know Q^{-1} exists because, by Gram-Schmidt, Q is an orthogonal matrix.

$$A_1 = Q_2R_2 \qquad \text{(by QR decomposition)}$$

$$\text{Let } A_2 = R_2Q_2$$

Note: A_2 has the same eigenvalues as A_1 due to the same logic as explained above.

$$\text{Let } A_p = R_pQ_p \ \ \forall\, p \in \mathbb{N}$$

Note: A_p has the same eigenvalues as A due to the same logic as explained above.

Recall that A_p is assumed to be an upper-triangular matrix.

$\therefore$ By the QR Method, the eigenvalues of A_p are equal to those of the eigenvalues of A.

$\square$

Now that we know both how the QR Method works and why the QR Method works, let us take a look at the accompanied software to gain some more intuition about the code, as well as the provided examples.

Note: Before continuing, please refer to the appendix and review all necessary functions and examples. Note that all of our QR Method functions take in a matrix A that abides by the preconditions.

Example QR.1: This is the trivial case. The purpose of this example is to show that our function works on trivial cases. We know that since A is a diagonal matrix, its eigenvalues are on the diagonal. Luckily, our getEvaluesQR(A) correctly returns 2 and 1 as our eigenvalues. We also know that the eigenvectors of A are [1,0] and [0,1], which our getEvectorsQR(A) correctly returns. Furthermore, our getSingularValuesQR(A) correctly returns $\sqrt{2}$ and $\sqrt{1}$, which makes sense since 2 and 1 are the eigenvalues of A. Lastly, our getLeftSingularVectorsQR(A) and getRightSingularVectorsQR(A) correctly return the left and right singular vectors, which we can verify with a singular vector calculator.

Example QR.2: This is a non-trivial case, although it is not an extremely difficult one. The purpose of this example is to show that our function works on normal-difficulty cases. Using an eigenvalue calculator, we can see that the eigenvalues of A are 17.034, -2.561, and 0.527. Our getEvaluesQR(A) returns these same results. Similarly, our getEvectorsQR(A) returns the same results as an eigenvector calculator. Next, our getSingularValuesQR(A) correctly returns the square roots of each of the eigenvalues we calculated. Our getLeftSingularVectorsQR(A) and getRightSingularVectorsQR(A) both return correct values as well, according to singular vector calculators.

Example QR.3: This is a non-trivial, difficult case. The purpose of this example is to show that our functions work on large matrices. To be said succinctly, all five of our QR functions properly return the expected answers for this large matrix, which we can verify with various calculators..

We have now shown that our QR functions work on trivial and non-trivial cases.

4 Deflation

Recall that the Power Method only helps us find the dominant eigenvalue, dominant eigenvector, dominant singular value, and dominant left/right singular vectors of an $n \times n$ symmetric matrix A. Usually though, we want to find *all* of these values, not just the dominant ones. Luckily, this is possible by improving upon the Power Method with a process known as "deflation."

The purpose of deflation is to "modify a matrix to eliminate the influence of a given eigenvector, typically by setting the associated eigenvalue to zero."[1] Applying this method to the Power Method, we can easily find all eigenvalues of a matrix A that satisfies the preconditions of the Power Method (and the following eigenvectors, singular values, and singular vectors).

How does this method work? Begin with an $n \times x$ matrix A that satisfies the preconditions of the Power Method. Use the Power Method to calculate the dominant eigenvalue (denoted as λ_1), dominant eigenvector (denoted as v_1), dominant singular value, and dominant singular vectors. Next, let $A_1 = A - \frac{\lambda_1}{(\vec{v}^T\vec{v})^2}\vec{v}\vec{v}^T$. This eliminates the influence of the dominant eigenvector, $\vec{v}_1$, and zeros out its associated eigenvalue from the matrix. Now, call the Power Method on A_1, which will return its dominant eigenvalue (denoted as λ_2), dominant eigenvector (denoted as v_2), dominant singular value, and dominant singular vectors. Create A_2 using the same formula, but with A_1, λ_2, and $\vec{v}_2$. It is at this point that the iterative process becomes clear; continue this process up to and including A_{n-1}, for which at this point all desired values will have been computed.[7]

Now that we know how deflation with the Power Method works, let us take a look at the accompanied software to gain some more intuition about the code, as well as the provided examples. Note that all of our Power Method and Power Method Deflation functions take in a matrix A that abides by the preconditions, as well as an $n \times 1$ initial approximation x.

Note: Before continuing, please refer to the appendix and review all necessary functions and examples.

Example PMD.1: This is the trivial case. The purpose of this example is to show that our function works on trivial cases. We expect that our Power Method

Deflation functions would return the same results as our QR Method functions would (even though both methods use completely different algorithms) since the QR Method and the Power Method with deflation return the same things (i.e. all eigenvalues, eigenvectors, singular values, and singular vectors). Luckily, all five of our Power Method Deflation functions return the same thing as our QR Method functions in Example QR.1.

Example PMD.2: This is a non-trivial case, although it is not an extremely difficult one. The purpose of this example is to show that our function works on normal-difficulty cases. Knowing that our five Power Method Deflation functions return the same results as our QR Method functions in Example QR.2, and using the reasoning we explained above in Example PMD.1, we know our Power Method Deflation functions work properly for this example.

Example PMD.3: This is a non-trivial, difficult case. The purpose of this example is to show that our functions work on large matrices. Knowing that our five Power Method Deflation functions return the same results as our QR Method functions in Example QR.3, and using the reasoning we explained above in Example PMD.1, we know our Power Method Deflation functions work properly for this example.

We have now shown that our Power Method Deflation functions work on trivial and non-trivial cases.

References

[1] *Deflation methods for sparse PCA.* deflation Methods for Sparse PCA - statwiki. (n.d.). Retrieved December 9, 2022, from https://wiki.math.uwaterloo.ca/statwiki/index.php?title=deflation_Methods_for_Sparse_PCA

[2] Gilbert Strang - MIT. (2019). *Computing Eigenvalues and Singular Values. YouTube.* Retrieved December 6, 2022, from https://youtu.be/d32WV1rKoVk.

[3] *Iteration.* Home : Oxford English Dictionary. (n.d.). Retrieved December 6, 2022, from https://www.oed.com/view/Entry/100312?redirectedFrom=iteration

[4] Jeffrey Chasnov. (2021). *Eigenvalue Power Method (Example) — Lecture 31 — Numerical Methods for Engineers. YouTube.* Retrieved December 6, 2022, from https://youtu.be/nKd0lu3yThg.

[5] Jeffrey Chasnov. (2021). *Eigenvalue Power Method — Lecture 30 — Numerical Methods for Engineers. YouTube.* Retrieved December 6, 2022, from bit.ly/3Y2SfvC.

[6] Strang, G. (2021). *Introduction to linear algebra.* Wellesley-Cambridge Press.

[7] *YouTube. Lecture 47 Matrix Eigenvalue Problems - 2 Power Method - 2.* Retrieved December 9, 2022, from https://youtu.be/WVA6pVtMFgs.

5 Appendix

The Power Method, QR Method, and Deflation

By Eli Slothower, CMU SCS class of 2025

The purpose of this document is to showcase the Power Method, the QR Method, and improvements upon these methods to calculate eigenvalues, eigenvectors, singular values, and singular vectors of a matrix. All theory and reasoning behind the following code can be found in the paper that accompanies this code.

In [193…

```
#Load in LinearAlgebra package
using LinearAlgebra
```

The Power Method

Below is our PowerMethod() function for a square, symmetric matrix A, which returns a tuple of, in this order, the dominant eigenvalue, the dominant eigenvector, and the dominant singular value. We use the below helper functions to extract these values from this tuple individually, as well as to compute the dominant left singular vector and the dominant right singular vector.

Note that if the inputted matrix does not meet the above preconditions, PowerMethod() will not run, which it would notify you of.

In [193…

```
function PowerMethod(A, x) #takes in a matrix A that meets the preconditions,
                           #and an initial approximation x
    m, n = size(A) #gets dimensions of A
    if (m == n) #checks that inputted matrix is square
        for i in 1:200
            x = ((A*x)/sqrt(dot((A*x), (A*x)))) #creates the next value
                                                #of x (and divides by its
                                                #magnitude for overflow
                                                #purposes)
        end
        nextX = A*x #calculates one next x for eigenvalue computation
        lambda1 = dot(x, nextX)/dot(x, x) #calculates lambda_1
        eigenvector = x #finds corresponding eigenvector
        singularValue = sqrt(abs(lambda1)) #calculates dominant singular
                                           #value by taking square root
                                           #of lambda_1
        return(lambda1, eigenvector, singularValue) #returns tuple as
                                                    #outlined above
    else
        print("A does not meet the preconditions") #tells client if
                                                   #preconditions are
                                                   #not met
        return
    end

end
```

Out[193…

```
PowerMethod (generic function with 1 method)
```

In [193…

```
function getEvaluePM(A, x)
    return PowerMethod(A, x)[1]
end
```

Out[193…

```
getEvaluePM (generic function with 1 method)
```

In [193…

```
function getEvectorPM(A, x)
    return PowerMethod(A, x)[2]
end
```

Out[193…

```
getEvectorPM (generic function with 1 method)
```

In [193…

```
function getSingularValuePM(A, x)
    return PowerMethod(A, x)[3]
end
```

Out[193…

```
getSingularValuePM (generic function with 1 method)
```

In [193…

```
function getSingularVectorAtAPM(A, x)
    PMresult = PowerMethod(transpose(A)*A, x) #calculates PowerMethod()
                                               #on AtA
    eigenvector = PMresult[2]
    return eigenvector
end
```

Out[193…

```
getSingularVectorAtAPM (generic function with 1 method)
```

In [193…

```
function getSingularVectorAAtPM(A, x)
    PMresult = PowerMethod(A*transpose(A), x) #calculates PowerMethod()
                                               #on AAt
    eigenvector = PMresult[2]
    return eigenvector
end
```

Out[193…

```
getSingularVectorAAtPM (generic function with 1 method)
```

Example PM.1

In [193…

```
A = [2 0; 0 1]
x = [1;1]
result = getEvaluePM(A,x)
```

Out[193…

```
2.0
```

In [194…

```
A = [2 0; 0 1]
x = [1;1]
result = getEvectorPM(A,x)
```

Out[194…

```
2-element Vector{Float64}:
 1.0
 6.223015277861142e-61
```

In [194…

```
A = [2 0; 0 1]
x = [1;1]
result = getSingularValuePM(A,x)
```

Out[194…

```
1.4142135623730951
```

In [194…

```
A = [2 0; 0 1]
x = [1;1]
result = getSingularVectorAtAPM(A,x)
```

Out[194…
```
2-element Vector{Float64}:
 1.0
 3.8725919148493183e-121
```

In [194…
```
A = [2 0; 0 1]
x = [1;1]
result = getSingularVectorAAtPM(A,x)
```

Out[194…
```
2-element Vector{Float64}:
 1.0
 3.8725919148493183e-121
```

Example PM.2

In [194…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getEvaluePM(A,x)
```

Out[194…
```
17.034137096355504
```

In [194…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getEvectorPM(A,x)
```

Out[194…
```
3-element Vector{Float64}:
 0.5774416595564781
 0.5470106434803808
 0.6060862032152194
```

In [194…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getSingularValuePM(A,x)
```

Out[194…
```
4.127243280490684
```

In [194…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getSingularVectorAtAPM(A,x)
```

Out[194…
```
3-element Vector{Float64}:
 0.5774416595564782
 0.5470106434803808
 0.6060862032152194
```

In [194…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getSingularVectorAAtPM(A,x)
```

Out[194…
```
3-element Vector{Float64}:
 0.5774416595564782
 0.5470106434803808
 0.6060862032152194
```

Example PM.2.1

In [194…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [999;1200000;1456]
result = getEvaluePM(A,x)
```

Out[194…
```
17.034137096355504
```

In [195…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [999;1200000;1456]
result = getEvectorPM(A,x)
```

Out[195…
```
3-element Vector{Float64}:
 0.5774416595564781
 0.5470106434803808
 0.6060862032152194
```

In [195…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [999;1200000;1456]
result = getSingularValuePM(A,x)
```

Out[195…
```
4.127243280490684
```

In [195…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [999;1200000;1456]
result = getSingularVectorAtAPM(A,x)
```

Out[195…
```
3-element Vector{Float64}:
 0.5774416595564782
 0.5470106434803808
 0.6060862032152194
```

In [195…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [999;1200000;1456]
result = getSingularVectorAAtPM(A,x)
```

Out[195…
```
3-element Vector{Float64}:
 0.5774416595564782
 0.5470106434803808
 0.6060862032152194
```

Example PM.3

In [195…
```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getEvaluePM(A,x)
```

Out[195…
```
105077.92749234011
```

In [195…
```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getEvectorPM(A, x)
```

Out[195…
```
7-element Vector{Float64}:
 0.7045593165908005
 0.003167479342962275
 0.05261982026017225
 0.11212454652807066
 0.6698905053676605
 0.0017702795012152963
 0.19871827794675967
```

In [195…
```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getSingularValuePM(A, x)
```

Out[195…
```
324.1572573494848
```

In [195…

```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getSingularVectorAtAPM(A, x)
```

Out[195…

```
7-element Vector{Float64}:
 0.7061387957133922
 0.0031674363493632545
 0.05253617071329216
 0.1119992424967313
 0.6683860178933628
 0.0017710402165370632
 0.19827022843472986
```

In [195…

```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getSingularVectorAAtPM(A, x)
```

Out[195…

```
7-element Vector{Float64}:
 0.7061387957133922
 0.0031674363493632545
 0.05253617071329216
 0.1119992424967313
 0.6683860178933628
 0.0017710402165370632
 0.19827022843472986
```

The QR Method

Below is our QRMethod() function for a square matrix A where det(A) != 0, which returns a tuple of, in this order, a list of the eigenvalues, a matrix of the eigenvectors, and a list of the singular values. We use the below helper functions to extract these values from this tuple individually, as well as to compute the left singular vectors and the right singular vectors.

In [195…

```
function QRMethod(A)
    m, n = size(A) #gets dimensions of A
    if m == n && det(A) != 0 #checks that A is square and det(A) != 0
        Q, R = qr(A) #finds QR decomposition of A
        eigenvectors = Q
        for i in 1:50
            newA = R*Q #iteratively finds newA by swapping Q and R
            Q, R = qr(newA) #finds QR decomposition of newA, following
                            #the QR method
            eigenvectors = eigenvectors*Q #calculates matrix of A's
                                          #eigenvectors
        end
        eigenvalues = diag(R*Q,0) #takes the components on the diagonal
                                  #of RQ, which are the eigenvalues of A

        singularValues = []
        for i in eachindex(eigenvalues)
            append!(singularValues, sqrt(abs(eigenvalues[i])))
            #calculates vector of singular values based off of square
            #roots of eigenvalues of A
        end

        return (eigenvalues, eigenvectors*Q, singularValues)
    else
        print("A does not meet the preconditions") #tells client if
                                                   #preconditions are
                                                   #not met
        return
    end
end
```

Out[195…

```
QRMethod (generic function with 1 method)
```

In [196…

```
function getEvaluesQR(A)
    return QRMethod(A)[1]
end
```

Out[196…

```
getEvaluesQR (generic function with 1 method)
```

In [196…

```
function getEvectorsQR(A)
    return QRMethod(A)[2]
end
```

Out[196…

```
getEvectorsQR (generic function with 1 method)
```

In [196…
```
function getSingularValuesQR(A)
    return QRMethod(A)[3]
end
```

Out[196…
```
getSingularValuesQR (generic function with 1 method)
```

In [196…
```
function getLeftSingularVectorsQR(A)
    QRAtA = QRMethod(transpose(A)*A) #calculates QRMethod() on AtA
    eigenvectorMatrix = QRAtA[2]
    return eigenvectorMatrix
end
```

Out[196…
```
getLeftSingularVectorsQR (generic function with 1 method)
```

In [196…
```
function getRightSingularVectorsQR(A)
    QRAAt = QRMethod(A*transpose(A)) #calculates QRMethod() on AAt
    eigenvectorMatrix = QRAAt[2]
    return eigenvectorMatrix
end
```

Out[196…
```
getRightSingularVectorsQR (generic function with 1 method)
```

Example QR.1

In [196…
```
A = [2 0; 0 1]
result = getEvaluesQR(A)
```

Out[196…
```
2-element Vector{Float64}:
 2.0
 1.0
```

In [196…
```
A = [2 0; 0 1]
result = getEvectorsQR(A)
```

Out[196…
```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

In [196…
```
A = [2 0; 0 1]
result = getSingularValuesQR(A)
```

Out[196…
```
2-element Vector{Any}:
 1.4142135623730951
 1.0
```

In [196…

```
A = [2 0; 0 1]
result = getLeftSingularVectorsQR(A)
```

Out[196…

```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

In [196…

```
A = [2 0; 0 1]
result = getRightSingularVectorsQR(A)
```

Out[196…

```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

Example QR.2

In [197…

```
A = [6 5 6; 5 4 7; 6 7 5]
result = getEvaluesQR(A)
```

Out[197…

```
3-element Vector{Float64}:
 17.034137096355504
 -2.561302422819227
  0.5271653264637243
```

In [197…

```
A = [6 5 6; 5 4 7; 6 7 5]
result = getEvectorsQR(A)
```

Out[197…

```
3×3 Matrix{Float64}:
 0.577442   0.10056   -0.810215
 0.547011   0.689054   0.475377
 0.606086  -0.717699   0.342882
```

In [197…

```
A = [6 5 6; 5 4 7; 6 7 5]
result = getSingularValuesQR(A)
```

Out[197…

```
3-element Vector{Any}:
 4.127243280490684
 1.6004069553770464
 0.7260615169968205
```

In [197…

```
A = [6 5 6; 5 4 7; 6 7 5]
result = getLeftSingularVectorsQR(A)
```

Out[197…

```
3×3 Matrix{Float64}:
 0.577442   0.10056   -0.810215
 0.547011   0.689054   0.475377
 0.606086  -0.717699   0.342882
```

In [197…
```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getRightSingularVectorsQR(A)
```

Out[197…
```
3×3 Matrix{Float64}:
 0.577442   0.10056   -0.810215
 0.547011   0.689054   0.475377
 0.606086  -0.717699   0.342882
```

Example QR.3

In [197…
```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
result = getEvaluesQR(A)
```

Out[197…
```
7-element Vector{Float64}:
  83154.66898108396
 -80972.65976933317
   5664.178317222833
   4090.3044401729617
  -1897.9260490013883
   1067.7078726978568
    131.7262071566637
```

In [197…
```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
result = getEvectorsQR(A)
```

Out[197…
```
7×7 Matrix{Float64}:
 -0.211808     -0.976443    …   0.0317327  -0.0170728   0.0180072
  0.00173457   -0.00265036      0.127985   -0.504825   -0.818175
  0.0600075    -0.0238357      -0.0265957   0.015296   -0.0650124
  0.107956     -0.0636803      -0.799551    0.435794   -0.388956
  0.929261     -0.19642         0.25778     0.158064   -0.0533551
  0.000674297  -0.00167234  …  -0.0184679   0.1339      0.198574
  0.276318     -0.0578394      -0.525197   -0.715399    0.36397
```

In [197…

```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
result = getSingularValuesQR(A)
```

Out[197…

```
7-element Vector{Any}:
 288.3655128150451
 284.5569534721181
  75.26073556126616
  63.95548795977529
  43.565193090371906
  32.67579949592445
  11.477203803917734
```

In [197…

```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
result = getLeftSingularVectorsQR(A)
```

Out[197…

```
7×7 Matrix{Float64}:
 0.780873    -0.623331      0.00760364  …   0.0317327  -0.0170728   0.0180072
 0.00314495   0.000377441  -0.130939        0.127985   -0.504825   -0.818175
 0.0480011    0.043185     -0.733339       -0.0265957   0.015296   -0.0650124
 0.104992     0.0684566     0.0421271      -0.799551    0.435794   -0.388956
 0.588594     0.745429      0.0483079       0.25778     0.158064   -0.0533551
 0.00179785  -0.000138343  -0.663907    …  -0.0184679   0.1339      0.198574
 0.174512     0.221907     -0.0113904      -0.525197   -0.715399    0.36397
```

In [197…

```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getRightSingularVectorsQR(A)
```

```
Out[197…  7×7 Matrix{Float64}:
           0.780873    -0.623331      0.00760364  …   0.0317327  -0.0170728   0.0180072
           0.00314495   0.000377441  -0.130939        0.127985   -0.504825   -0.818175
           0.0480011    0.043185     -0.733339       -0.0265957   0.015296   -0.0650124
           0.104992     0.0684566     0.0421271      -0.799551    0.435794   -0.388956
           0.588594     0.745429      0.0483079       0.25778     0.158064   -0.0533551
           0.00179785  -0.000138343  -0.663907    …  -0.0184679   0.1339      0.198574
           0.174512     0.221907     -0.0113904      -0.525197   -0.715399    0.36397
```

Power Method Using Deflation

Below is our PowerMethodDeflation() function for a square, symmetric matrix A, which returns a tuple of, in this order, a vector of all eigenvalues, a 2D vector (a 2D list) of all eigenvectors, and a list of the singular values. We use the below helper functions to extract these values from this tuple individually, as well as to compute the left singular vectors and the right singular vectors.

In [198…

```
function PowerMethodDeflation(A, x) #takes in same input
                                    #as PowerMethod()
    eigenvector = x #dummy vector to initialize variable
    allEvalues = [] #dummy vector to initialize variable
    allEvectors = [] #dummy vector to initialize variable
    m,n = size(A) #gets dimensions of A
    for i in 1:n
        PMResults = PowerMethod(A, x) #runs power method on A
        eigenvalue = PMResults[1] #gets dominant eigenvalue of A
        eigenvector = PMResults[2] #gets dominant eigenvalue of A
        append!(allEvalues, eigenvalue) #appends current dominant
                                        #eigenvalue to list of
                                        #all eigenvalues of A
        push!(allEvectors, eigenvector) #appends current dominant
                                          #eigenvector to list of all
                                          #eigenvectors of A
        A = (A - ((eigenvalue/((dot(eigenvector, eigenvector))^2))
            * eigenvector*transpose(eigenvector)))
        #calculates new A based off of deflation method by getting rid
        #of the current dominant eigenvalue and eigenvector, so when
        #the power method is called on this new A, the next dominant
        #eigenvalue and eigenvector will be found
    end

    singularValues = []
    for i in eachindex(allEvalues)
        append!(singularValues, sqrt(abs(allEvalues[i])))
        #calculates vector of singular values based off of square
        #roots of eigenvalues of A
    end

    return (allEvalues, allEvectors, singularValues)
    #returns tuple as outlined above

    #Note: an inputted matrix A that does not meet the preconditions will
    #halt the program through the call to PowerMethod(), which already
    #checks these preconditions. Because of this, we did not implement
    #checking these preconditions here again, because that would be
    #redundant
end
```

Out[198…

```
PowerMethodDeflation (generic function with 2 methods)
```

In [198…

```
function getEvaluesPMD(A, x)
    return PowerMethodDeflation(A, x)[1]
end
```

Out[198…

```
getEvaluesPMD (generic function with 2 methods)
```

In [198…

```
function getEvectorsPMD(A, x)
    return PowerMethodDeflation(A, x)[2]
end
```

Out[198…

```
getEvectorsPMD (generic function with 2 methods)
```

In [198…

```
function getSingularValuesPMD(A, x)
    return PowerMethodDeflation(A, x)[3]
end
```

Out[198…

```
getSingularValuesPMD (generic function with 2 methods)
```

In [198…

```
function getSingularVectorsAtAPMD(A, x)
    PMDresult = PowerMethodDeflation(transpose(A)*A, x)
    #calculates PowerMethod() on AtA

    eigenvector = PMDresult[2]
    return eigenvector
end
```

Out[198…

```
getSingularVectorsAtAPMD (generic function with 2 methods)
```

In [198…

```
function getSingularVectorsAAtPMD(A, x)
    PMDresult = PowerMethodDeflation(A*transpose(A), x)
    #calculates PowerMethod() on AAt

    eigenvector = PMDresult[2]
    return eigenvector
end
```

Out[198…

```
getSingularVectorsAAtPMD (generic function with 2 methods)
```

Example PMD.1

In [198…

```
A = [2 0; 0 1]
x = [1;1]
result = getEvaluesPMD(A,x)
```

Out[198…

```
2-element Vector{Any}:
 2.0
 1.0
```

In [198…

```
A = [2 0; 0 1]
x = [1;1]
result = getEvectorsPMD(A,x)
```

Out[198…

```
2-element Vector{Any}:
 [1.0, 6.223015277861142e-61]
 [-1.2446030555722283e-60, 1.0]
```

In [198…

```
A = [2 0; 0 1]
x = [1;1]
result = getSingularValuesPMD(A,x)
```

Out[198…

```
2-element Vector{Any}:
 1.4142135623730951
 1.0
```

In [198…

```
A = [2 0; 0 1]
x = [1;1]
result = getSingularVectorsAtAPMD(A,x)
```

Out[198…

```
2-element Vector{Any}:
 [1.0, 3.8725919148493183e-121]
 [-1.5490367659397273e-120, 1.0]
```

In [199…

```
A = [2 0; 0 1]
x = [1;1]
result = getSingularVectorsAAtPMD(A,x)
```

Out[199…

```
2-element Vector{Any}:
 [1.0, 3.8725919148493183e-121]
 [-1.5490367659397273e-120, 1.0]
```

Example PMD.2

In [199…

```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getEvaluesPMD(A,x)
```

Out[199…

```
3-element Vector{Any}:
 17.034137096355504
 -2.5613024228192276
  0.5271653264637228
```

In [199…

```
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getEvectorsPMD(A,x)
```

```
Out[199… 3-element Vector{Any}:
 [0.5774416595564781, 0.5470106434803808, 0.6060862032152194]
 [0.10055966074712493, 0.6890544052397919, -0.7176989488985335]
 [-0.8102153321426852, 0.47537709509274567, 0.3428815145529151]
```

```
In [199…
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getSingularValuesPMD(A,x)
```

```
Out[199… 3-element Vector{Any}:
 4.127243280490684
 1.6004069553770464
 0.7260615169968195
```

```
In [199…
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getSingularVectorsAtAPMD(A,x)
```

```
Out[199… 3-element Vector{Any}:
 [0.5774416595564782, 0.5470106434803808, 0.6060862032152194]
 [0.10055966074712498, 0.6890544052397903, -0.717698948898535]
 [-0.8102153321425966, 0.4753770950928481, 0.3428815145529826]
```

```
In [199…
A = [6 5 6; 5 4 7; 6 7 5]
x = [1;1;1]
result = getSingularVectorsAAtPMD(A,x)
```

```
Out[199… 3-element Vector{Any}:
 [0.5774416595564782, 0.5470106434803808, 0.6060862032152194]
 [0.10055966074712498, 0.6890544052397903, -0.717698948898535]
 [-0.8102153321425966, 0.4753770950928481, 0.3428815145529826]
```

Example PMD.3

```
In [199…
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getEvaluesPMD(A,x)
```

```
Out[199… 7-element Vector{Any}:
            105077.92749234011
           -102898.07681220338
              5664.178317222874
              4090.304440172935
             -1897.9260490013871
              1067.70787269786
               131.7262071566645
```

```
In [199… A = [987 234 6587 12445 98661 89 29374;
              234 600 1 45 73 999 555;
              6587 1 5043 72 800 819 301;
              12445 45 72 20 4444 19 20;
              98661 73 800 4444 0 100 101;
              89 999 819 19 100 4572 0;
              29374 555 301 20 101 0 16;]
         x = [1;1;1;1;1;1;1]
         result = getEvectorsPMD(A, x)
```

```
Out[199… 7-element Vector{Any}:
          [0.7045593165908005, 0.003167479342962275, 0.05261982026017225, 0.11212454652
         807066, 0.6698905053676605, 0.0017702795012152963, 0.19871827794675967]
          [-0.7052142342198308, 3.0234863635354026e-5, 0.03765983138129654, 0.056528442
         07719054, 0.6763768965920753, -0.0003346912991381668, 0.20142824284209992]
          [-0.007603642096408602, 0.13093863677172232, 0.7333394413234289, -0.042127125
         48499737, -0.048307921014386974, 0.6639069440513676, 0.011390414070041556]
          [0.004030465259957338, 0.2054437392950462, -0.6729608506380436, 0.04505471535
         825179, 0.03573612581646158, 0.7081823167680581, 0.008433682198244108]
          [-0.03173272150838021, -0.12798483516662526, 0.026595685049208624, 0.79955108
         68660561, -0.25778049897218774, 0.018467873486066943, 0.5251969129570998]
          [0.017072842650132758, 0.504824853693538, -0.015296005391544923, -0.435794473
         0953685, -0.15806402191177316, -0.13390024837948816, 0.7153992472995199]
          [-0.018007233034550697, 0.8181745897256592, 0.06501242500090056, 0.3889562511
         207561, 0.05335512556226336, -0.19857438300769817, -0.3639697028388841]
```

```
In [199… A = [987 234 6587 12445 98661 89 29374;
              234 600 1 45 73 999 555;
              6587 1 5043 72 800 819 301;
              12445 45 72 20 4444 19 20;
              98661 73 800 4444 0 100 101;
              89 999 819 19 100 4572 0;
              29374 555 301 20 101 0 16;]
         x = [1;1;1;1;1;1;1]
         result = getSingularValuesPMD(A, x)
```

```
Out[199… 7-element Vector{Any}:
          324.1572573494848
          320.7773009615914
           75.26073556126643
           63.95548795977508
           43.56519309037189
           32.6757994959245
           11.477203803917767
```

In [199…

```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getSingularVectorsAtAPMD(A, x)
```

Out[199…

```
7-element Vector{Any}:
 [0.7061387957133922, 0.0031674363493632545, 0.05253617071329216, 0.1119992424
967313, 0.6683860178933628, 0.0017710402165370632, 0.19827022843472986]
 [-0.7068753014426122, 2.2792831075611973e-5, 0.037536497750205164, 0.05626544
776689501, 0.6748083409699729, -0.00033885340616939214, 0.20096294820405255]
 [-0.007603642096409797, 0.13093863677172238, 0.7333394413234289, -0.042127125
48499736, -0.04830792101438463, 0.6639069440513679, 0.011390414070042045]
 [0.004030465259959211, 0.20544373929504609, -0.6729608506380438, 0.0450547153
5825195, 0.035736125816458665, 0.708182316768058, 0.008433682198243576]
 [-0.03173272150838026, -0.12798483516662446, 0.026595685049221617, 0.79955108
68660768, -0.25778049897201083, 0.018467873486067054, 0.5251969129571548]
 [0.01707284264996484, 0.5048248536935429, -0.015296005391517193, -0.435794473
0953335, -0.15806402191136645, -0.1339002483794893, 0.7153992472996321]
 [-0.018007233028058577, 0.8181745897256248, 0.06501242500013753, 0.3889562511
199688, 0.05335512555171472, -0.19857438300773328, -0.3639697028417869]
```

In [200…

```
A = [987 234 6587 12445 98661 89 29374;
     234 600 1 45 73 999 555;
     6587 1 5043 72 800 819 301;
     12445 45 72 20 4444 19 20;
     98661 73 800 4444 0 100 101;
     89 999 819 19 100 4572 0;
     29374 555 301 20 101 0 16;]
x = [1;1;1;1;1;1;1]
result = getSingularVectorsAAtPMD(A, x)
```

Out[200…

```
7-element Vector{Any}:
 [0.7061387957133922, 0.0031674363493632545, 0.05253617071329216, 0.1119992424
967313, 0.6683860178933628, 0.0017710402165370632, 0.19827022843472986]
 [-0.7068753014426122, 2.2792831075611973e-5, 0.037536497750205164, 0.05626544
776689501, 0.6748083409699729, -0.00033885340616939214, 0.20096294820405255]
 [-0.007603642096409797, 0.13093863677172238, 0.7333394413234289, -0.042127125
48499736, -0.04830792101438463, 0.6639069440513679, 0.011390414070042045]
 [0.004030465259959211, 0.20544373929504609, -0.6729608506380438, 0.0450547153
5825195, 0.035736125816458665, 0.708182316768058, 0.008433682198243576]
 [-0.03173272150838026, -0.12798483516662446, 0.026595685049221617, 0.79955108
68660768, -0.25778049897201083, 0.018467873486067054, 0.5251969129571548]
 [0.01707284264996484, 0.5048248536935429, -0.015296005391517193, -0.435794473
0953335, -0.15806402191136645, -0.1339002483794893, 0.7153992472996321]
 [-0.018007233028058577, 0.8181745897256248, 0.06501242500013753, 0.3889562511
199688, 0.05335512555171472, -0.19857438300773328, -0.3639697028417869]
```