# Eigenvalue and Eigenvector Calculations Using a Neural Network

Raghvendra Mishra (ME21B1075) and Ayush Agarwal (ME21B1076)

May 16, 2023

## 1 Introduction

Eigenvalue and eigenvector calculations are essential in many areas of science and engineering, from physics and chemistry to computer science and finance. These calculations can be used to solve a wide range of problems, such as determining the stability of a system, analyzing data, and understanding the structure of a network. While there are many methods for calculating eigenvalues and eigenvectors, most of them are iterative and can be computationally expensive, especially for large matrices.

In this paper, we are considering two of these methods that are **Power Method** and **Jacobi algorithm** to understand how well do they work while scaling and how accurate they are. Along with that, the objective of the paper is to analyze a new way of calculating eigenvalues and eigenvectors, thus testing it's advantages and disadvantages against iterative methods.

Recently, there has been growing interest in using **Neural Networks** to calculate eigenvalues and eigenvectors. Neural networks are powerful machine learning tools that can learn complex functions from data and generalize to new data. By training a neural network on a set of matrices and their corresponding eigenvalues and eigenvectors, it is possible to create a model that can predict the eigenvalues and eigenvectors of new matrices with high accuracy and efficiency.

## 2 Literature Review

### 2.1 Eigenvalues and Eigenvectors

There's usually much blur about what are eigenvalues and eigenvectors and as it is the centerpiece of our topic of discussion, we have to start with a clear understanding of what those terms mean. For this, we need to have a solid understanding of matrices being used as **Linear Transformations** .

#### 2.1.1 Matrices as Linear Transformations

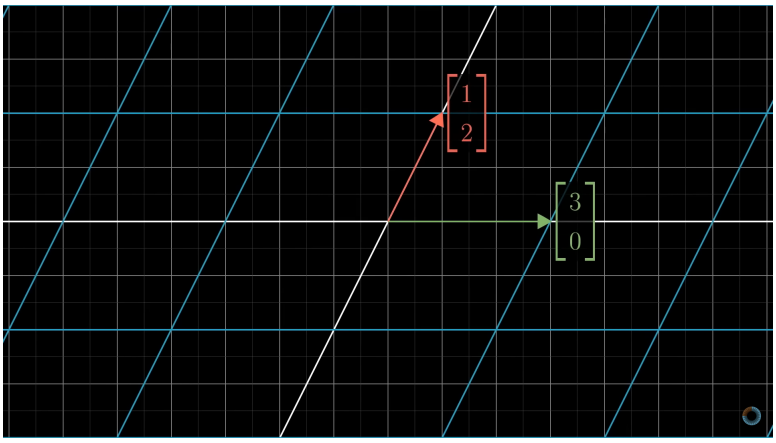



Figure 1: Vector Space before L.T.

Here we see a vector space with the basis vectors $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ which is also the Cartesian vector space that we mostly use. Now, let's say this vector space undergoes a linear transformation which changes this space to the

one below.

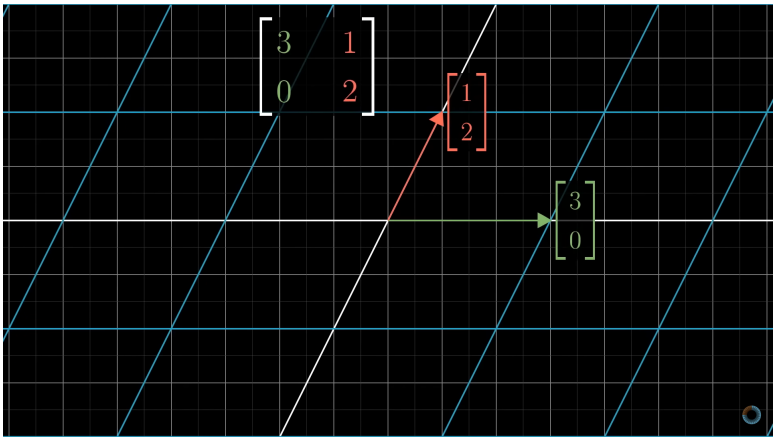



Figure 2: Vector space after L.T.

With the linear transformation we see that the $\hat{i}$ brought to co-ordinates $(3, 0)$ and $\hat{j}$ brought to co-ordinates $(1, 2)$. The linear transformation that the vector space undergoes can thus be represented as:

$$\begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$$

Therefore, any square matrix of $n \times n$ dimensions can be said to be the linear transformation of a vector space of $n$ dimensions from the basis

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

To the basis

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Where each column is the new co-ordinates of the initial axes. This new definition of linear transformation really helps visulaize how matrices are linked to vectors and geometry and is key to understanding what are eigenvalues and eigenvectors.

### 2.1.2 Vector undergoing Linear Transformations

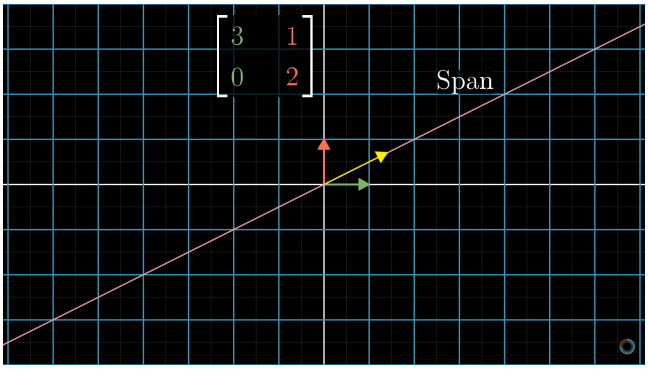



Figure 3: Vector Space before L.T.

Now consider a random vector on the initial vector space with basis $\hat{i}$ and $\hat{j}$, moreover considering the entire **span** of the vector on the vector space. For $\mathbb{R}^2$, the span of a vector is just the straight line that the vector lies on. So now, we let the vector space go through the desired linear transformation.

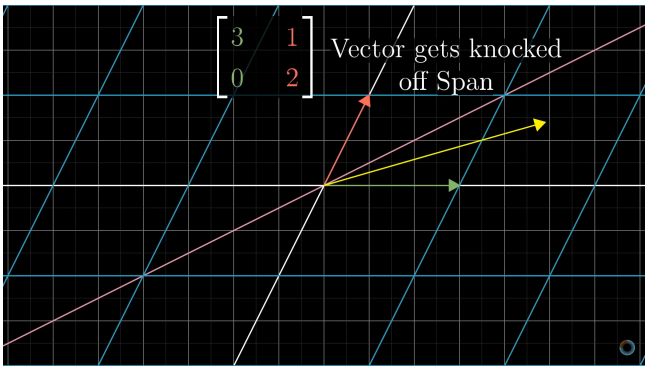



Figure 4: Vector space after L.T.

Only for it to be "knocked off it's span", which would seem to be the obvious case. It would seem coincidental for a vector(as follows):

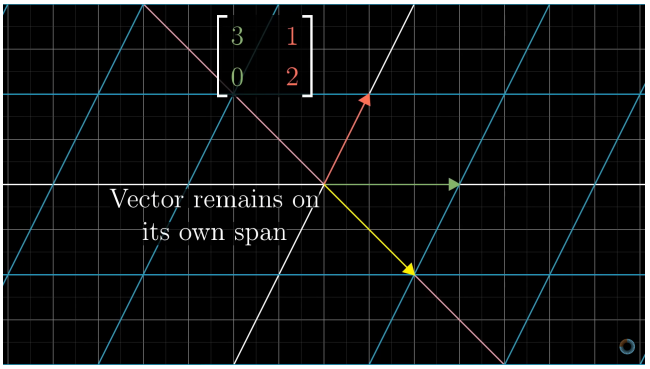



Figure 5: Vector space before L.T.

To still have it's direction and thus stay on it's span even after the space undergoing a linear transformation.

These special vectors that thus survive a linear transformation of a vector space with only a change in scale, like getting multiplied with a scalar, are known as **Eigenvectors** and the change in scale that they undergo is known as the **Eigenvalue** of the corresponding eigenvector.

## 2.2 Methods of Eigenvector and Eigenvalue calculation

There are quite a few methods of calculation of the two such as Power Method, QR Algorithm, Jacobi Algorithm and Singular Value Decomposition (SVD) but for this paper we would be looking at Power Method and Jacobi Algorithm to compare with the proposed way of calculation of the same.

### 2.2.1 Power Method

Let $A$ be a $n \times n$ matrix with $n$ distinct eigenvalues and we order them as,

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$$

with their corresponding eigenvectors,

$$e_1, e_2, \ldots, e_n$$

Now since $e_1, e_2, \ldots, e_n$ are eigenvectors of A, they are by definition linearly independent of each other. Therefore, any $n \times 1$ column vector can be written in terms of a linear combination of the $n$ eigenvectors

$$x_0 = c_1e_1 + c_2e_2 + \cdots + c_ne_n = \Sigma_{i=1}^{n} c_ie_i$$

$$x_1 = Ax_0 = A\Sigma_{i=1}^{n} c_ie_i = \Sigma_{i=1}^{n} c_iAe_i$$

Now since $e_i$ is an eigenvector of $A$, it means that $A \cdot e_i = \lambda_i e_i$

$$\therefore x_1 = \Sigma_{i=1}^{n} c_i \lambda_i e_i$$

So, if we do this $p$ times, i.e. we calculate the $x_p$ vector, it'll been

$$x_p = A^p x_0$$

Which will just be

$$x_p = A^p x_0 = \Sigma_{i=1}^{n} c_i \lambda_i^p e_i$$

Since, $\lambda_1$ is a much more dominant value than the others, $\lambda_1^p$ will just factor out $\lambda_2^p$, $\lambda_3^p$, ..., etc. Thus reducing the term $x_p$ to,

$$x_p = \lambda_1^p c_1 e_1$$

Now taking $x_{p+1}$ as,

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

We can get the eigenvalue $\lambda_1$ and eigenvector $e_1$ as,

$$\lambda_1 = \frac{x_p^T x_{p+1}}{x_p^T x_p} \qquad e_1 = x_p$$

And to prevent underflow or overflow issues, we can **normalize** $x_p$ as,

$$x_p = \frac{x_p}{x_p^T x_p^{\frac{1}{2}}}$$

### 2.2.2 Jacobi Algorithm

Let's say we have a $3 \times 3$ square, symmetric matrix

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

We note the largest non-diagonal entry in the matrix $A$ as $a_{ij}$, which here makes it $a_{13} = 4$. With this, we are trying to create an orthogonal matrix which Jacobi chose to be as, So, for the new matrix $T$ we fill the $a_{ij}$ position of the matrix with $\sin(x)$ and accordingly place the other values on $a_{ii}$, $a_{jj}$ and $a_{ji}$. Rest of the entries other than the diagonal entries are to 0 and the diagonal entries other than at both ends of the diagonal are to be 1. So for this particular example we get,

$$T = \begin{bmatrix} \cos(x) & 0 & -\sin(x) \\ 0 & 1 & 0 \\ \sin(x) & 0 & \cos(x) \end{bmatrix}$$

To get the values of this we need to know the value of the angle $x$. For that, we use the equation

$$x = \frac{1}{2} \tan^{-1}\left(\frac{2a_{ij}}{a_{ii} - ajj}\right)$$

For the current example, we get the value of $x = \frac{\pi}{4}$, thus making the matrix $P$ as

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Now, we find the inverse of $P$ but since P is an orthogonal matrix

$$T_1^{-1} = T_1^T = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Further we compute

$$B_1 = T_1^{-1}AT_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & -2 & 4 \\ -2 & 5 & -2 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 5 & -2\sqrt{2} & 0 \\ -2\sqrt{2} & 5 & 0 \\ 0 & 0 & -3 \end{bmatrix}$$

And since $B_1$ is not an orthogonal matrix we move to 2nd iteration where we repeat the process from selecting the largest non-diagonal element and using it to calculate the new matix $T_2$, which comes out to be of this format,

$$T_2 = \begin{bmatrix} \cos(x) & -\sin(x) & 0 \\ \sin(x) & \cos(x) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When the iteration ends and we have achieved a orthogonal, diagonal matrix the values on the diagonal are the eigenvalues of the matrix $A$ and subsequently, the eigenvector can be calulcated also

$$T = T_1T_2 \dots T_n$$

#### 2.2.3 Neural Network

The proposed neural network model:

$$\frac{dx(t)}{dt} = -x(t) + f(x(t))$$

for $t \geq 0$, where

$$f(x) = [x^T xA + (1 - x^T Ax)I]x$$

and $x = (x_1, x_2, \dots x_n)^T \in R^n$ represent the state of the network.
This is a class of **recurrent neural network**.
When $x(t)$, the output of the neural network, is fully converged then

$$\dot{x}(t) = 0$$

$$\therefore x(t) = f(x(t))$$

Thus, when the neural network has converged with its output which is the trial eigenvector, $x$, it is equal to $f(x)$. Now, the paper suggests that there is a predetermined way to guarantee the algorithm searches for the smallest or largest eigenvalues, this is not found to work in practice for this implementation. Instead, to ensure that the smallest or the largest eigenvalues can always be found a change was made to the loss function, which will be described below.

$$MSE(x, y) = \Sigma_i(x_i - y_i)^2$$

And once the trial eigenvector $x(0)$, which will be a vector of random numbers, converges to a true eigenvector of the matrix, we can also get the eigenvector using the **Rayleigh quotient** .

$$\lambda(t \to \infty) = \frac{x(t \to \infty)^T Ax(t \to \infty)}{x(t \to \infty)^T x(t \to \infty)}$$

# 3 Methodology

## 3.1 Neural Network

#### 3.1.1 f (x)

We have a method $f(x)$ that defines the function while taking the arguments $x$ and $A$.

#### 3.1.2 Set-up

The first section of code defines the identity matrix, I, and the initial trial eigenvector, $x_0$. These must then be converted to tensors in order to be used with Tensorflow.

### 3.1.3 Neural Network Set-up

Afterwards, the neural network is established by utilizing the nn-structure parameter, which is a list of numerical values. The length of this list determines the number of hidden layers in the network architecture, while each number within the list represents the quantity of hidden neurons in that particular layer. Additionally, a final output layer is appended at the conclusion of the hidden layers, with its dimension being equivalent to the size of the eigenvector.

### 3.1.4 Loss Function

The loss function is formulated as the mean squared error between the neural network's output, denoted as x, and f(x), while incorporating an additional component to regulate the identification of the eigenvector. The eigen-guess parameter governs the determination of the desired eigenvector. Providing a guess greater than the matrix's largest eigenvalue guarantees the discovery of the largest eigenvector, while a guess smaller than the smallest eigenvalue ensures the identification of the smallest eigenvector. In practical scenarios where the approximate eigenvalues are unknown, substituting the guesses with a very large positive value for the largest eigenvector or a very large negative value for the smallest eigenvector can be done. The eigen-lr parameter controls the contribution of this component to the overall loss function. Setting eigen-lr to zero implies that the neural network will find any eigenvector of the given matrix, disregarding the additional term.

### 3.1.5 Training

The optimizer is defined and the neural network having being instructed to minimize the loss function, for each training iteration of the neural network the trial eigenvalue is calculated from the trial eigenvector. The change of this eigenvalue from the last eigenvalue is calculaed to end the training process with sequential eigenvalues are sufficiently close.

### 3.1.6 Testing

The neural network is then tested for multiple symmtric, positive definite matrices which are generated using the formulated

$$A = \frac{Q + Q^T}{2}$$

where Q is a square matrix with random entries.

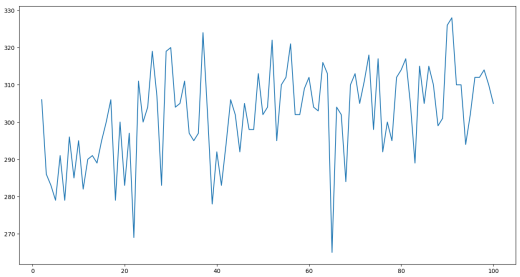
## 3.2 Power Method and Jacobi Algorithm

The alogrithm is placed in a for loop while using numpy to do the necessary martix manipulations with a set threshold and max iterations value.

# 4 Results

## 4.1 Neural Network

### 4.1.1 Iterations vs dimensions



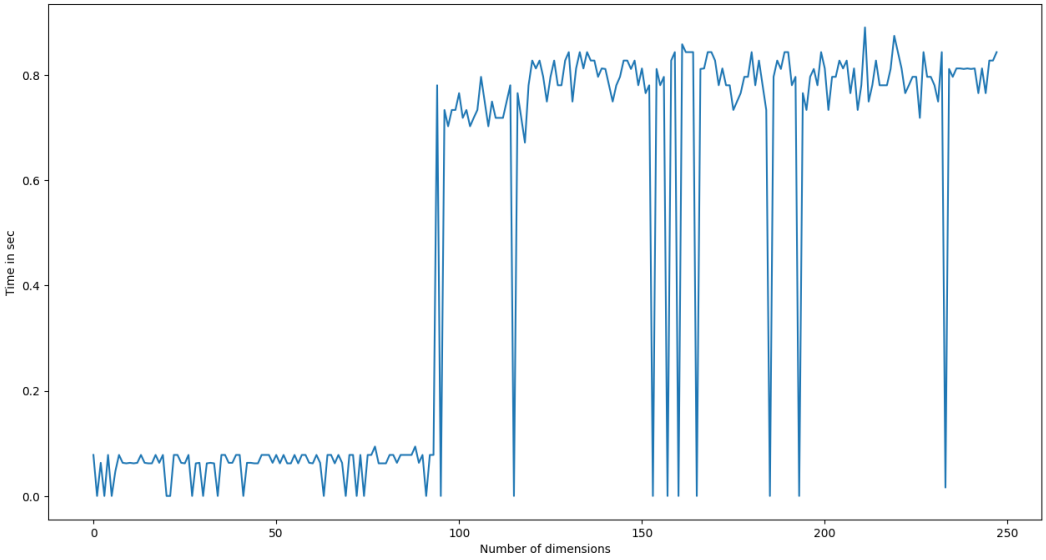

Neural network algorithm is pretty efficient in learning to produce the eigenvectors and eigenvalues even for $100 \times 100$ matrices way below the set limit of iterations which is 5000 for each of the methods. These are the values for the number of steps ranging from $2 \times 2$ to $100 \times 100$

### 4.1.2 Error Analysis and Convergence


The above give the error analysis of calculating the eigenvalues using a neural network for dimensions ranging from a $2 \times 2$ matrix to a $126 \times 126$ matrix. As you can see, from the graphs and the threshold that is set to be 1e-16 for the neural network, the convergence is achieved in under 400 steps till 130th dimensional matrix. After that, even though the number of steps don't increase, the time taken by the neural network increases exponentially.

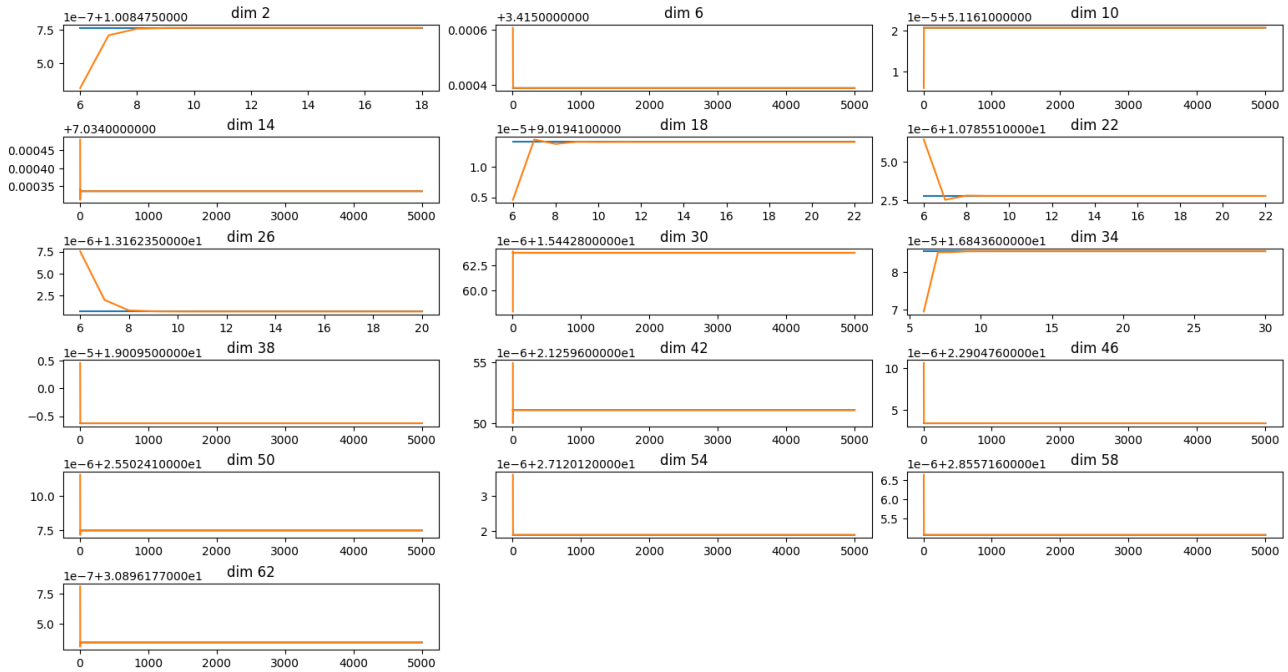
## 4.2 Power Method

### 4.2.1 Dimensions vs Time



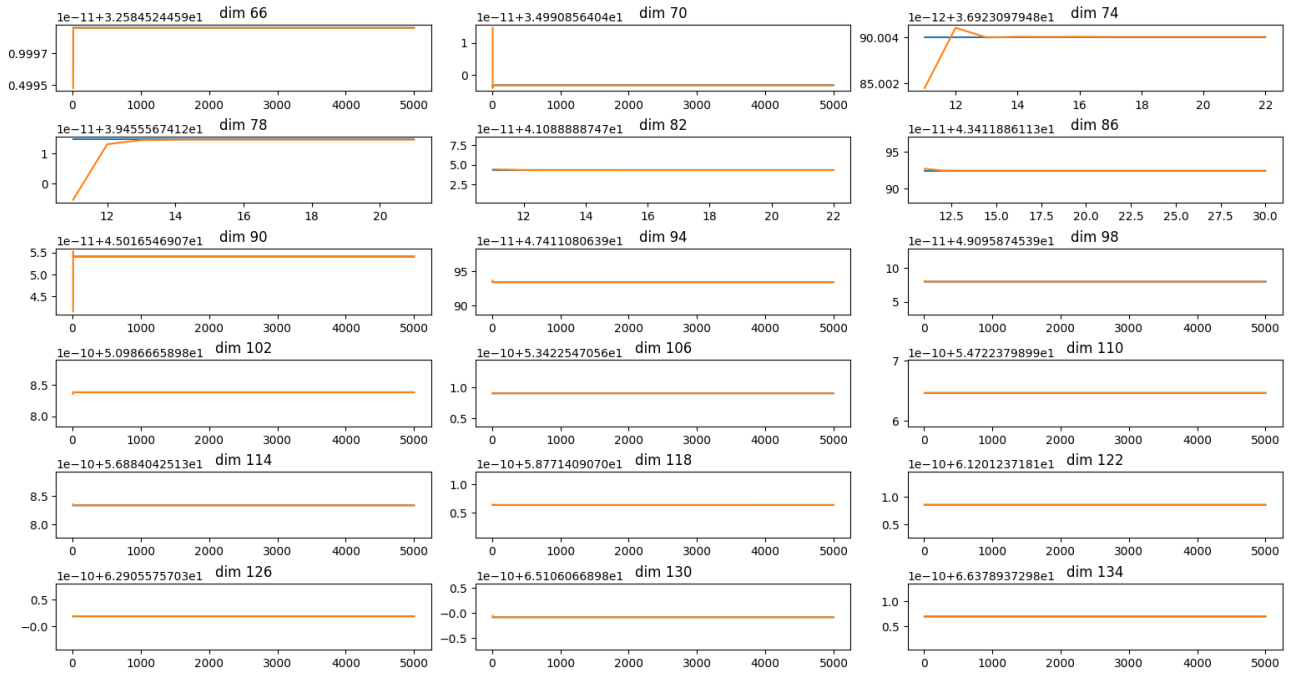

From the above graph we can see Power Method requiring less than a second to get the eigenvalues of a given matrix, even for a matrix of $250 \times 250$ size.

### 4.2.2 Error Analysis and Convergence



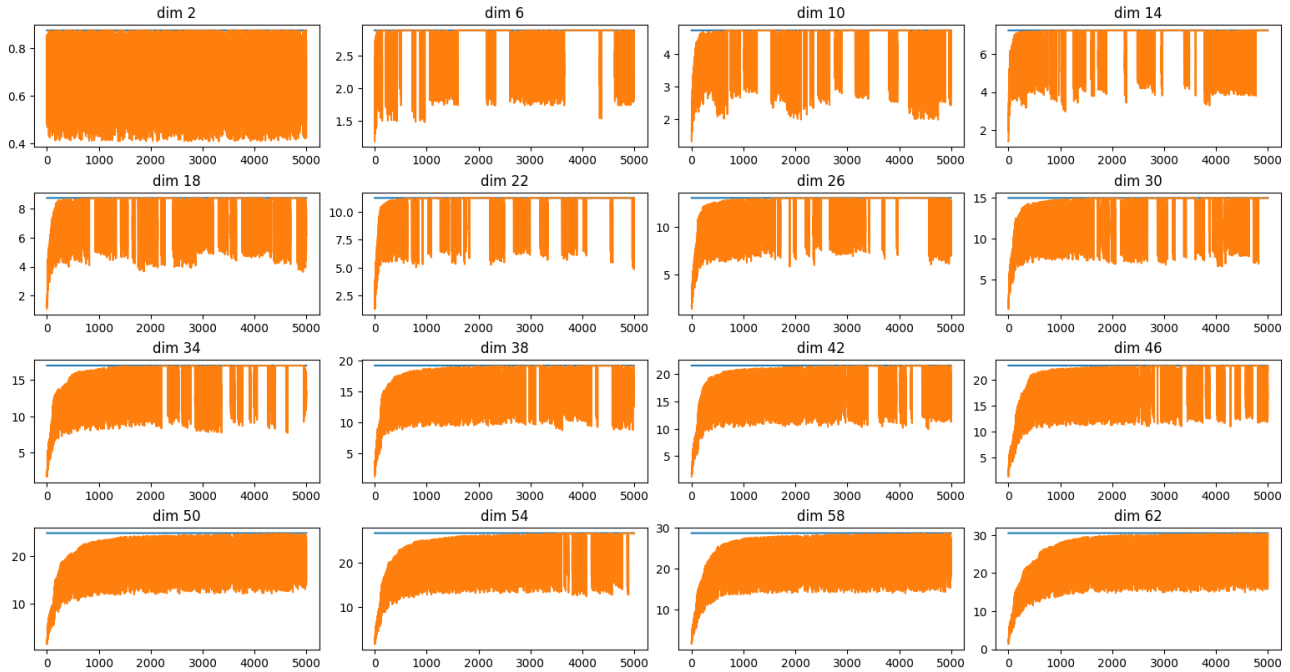

For a better presentation all the graphs are split into 2 groups of one from 2 - 62 and another 66 - 134.

The graphs give the error analysis of calculating the eigenvalues for the power method algorithm for dimensions ranging from a $2 \times 2$ matrix to a $134 \times 134$ matrix. Power Method almost instantly achieves convergence but it never touches the threshold value of 1e-16.

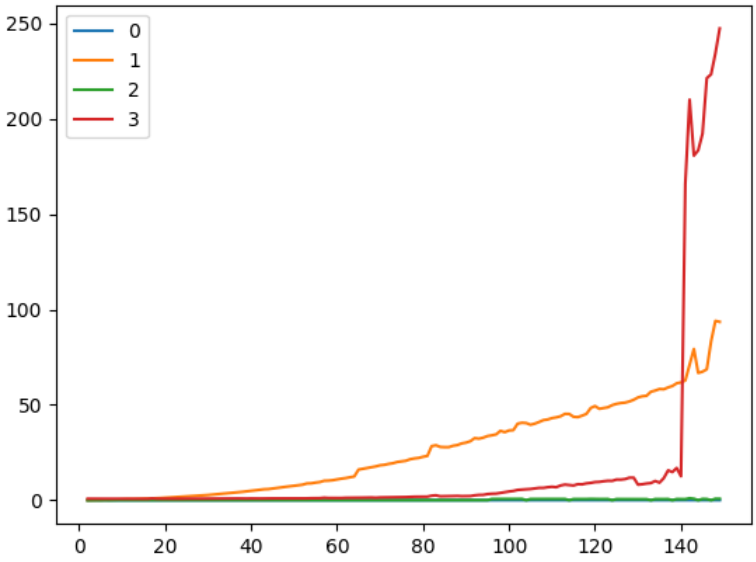



## 4.3 Jacobi Algorithm

### 4.3.1 Error Analysis and Convergence




As you can see, the convergence for Jacobi algorithm is really erratic and varies quite much from the blue line that is the true eigenvalue and doesn't really converge onto it. The convergence rate is slow for matrices with high condition numbers. Additionally, the algorithm may introduce rounding errors due to floating-point arithmetic. Therefore, proper consideration should be given to convergence criteria and precision requirements when implementing the Jacobi algorithm. Beyond dimension 70, the Jacobi algorithm starts taking upto 100 seconds to complete as you will see in the next subsection, hence computing the convergence of dimensions beyond this is a very time taking process.




### 4.4 Time vs dimensions

This is a time vs number of dimensions of symmetric matrix $A$ where time is in seconds on the y-axis and the dimensions are on the x-axis.

0 - Numpy values
1 - Jacobi algorithm
2 - Power Method
3 - Neural Network

As the graph suggests, Jacobi is the slowest of the 3 methods until the matrix size exceeds 140 where Neural net algorithm starts to fail and go way beyond the 200 seconds mark. In terms of time, **Power Method** is the most efficient.

## 5 Discussion

The conclusions can be drawn that Neural Network is an accurate and efficient way of calculating eigenvalues and eigenvectors only upto a certain dimension of less than 150. It may require some changes, like changing the size of the neural network for bigger matrices, to make the convergence faster but for the scope of this paper, the time exceeds a minute as the number of dimensions reach 100.
But, neural network converges to the desired accuracy faster than any other method in under 500 steps for matrices of even 150 dimensions.
So, for faster computations we may use Power Method, with the computational speed of less than a second, but for higher accuraies, we require the power of a neural network to help us.
Meanwhile, Jacobi is just bad.

## 6 Conclusion

To summarize,

- **Power Method** is the fastest of all 3 algorithms that we tested in this report. It can compute the largest dimensions at the fastest rate and is optimal choice for efficient calculations.
- **Neural Network** reaches the desired accuracy fastest, no matter how small. For the scope of the report, we tried with the error acceptance to be 1e-16, which the neural network convergence reached in below 500 iterations, making it the most accurate of the 3 algorithms. It is also pretty efficient until a certain dimension but the algorithms efficiency can may be increased by changing it's neural network structure from a [100, 100] to a much bigger one.

- **Jacobi Algorithm** It requires a large number of iterations to reach convergence, and the convergence rate is slow for matrices with high condition numbers. Additionally, the algorithm may introduce rounding errors due to floating-point arithmetic. Therefore, proper consideration should be given to convergence criteria and precision requirements when implementing the Jacobi algorithm.

# 7 References


- Zhang Yi, Nan Fu and Hua Jin Tang, Neural Networks Based Approach Computing Eigenvectors and Eigenvalues of Symmetric Matrix, Computers and Mathematics with Applications 47 (2004) 1155-1164
- https://butler-julie.medium.com/neural-network-eigenvalue-solver-with-tensorflow-ae18a2ecf45d
- https://www.youtube.com/watch?v=yMbL9TSeAJU
- https://www.youtube.com/watch?v=_PDyi5BVY-E
- https://www.youtube.com/watch?v=PFDu9oVAE-g&t