PHYS 514 - Computational Physics

Solution Set 3

Author:
Volkan Aydıngül (Id: 0075359)

Date: November 6, 2020

Contents

1 Problem VIII

1.1 Solution of Linear System vs. Minimization Problem

We are given the following function:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathsf{T} A\mathbf{x} - \mathbf{b}\mathbf{x} \tag{1}$$

Moreover, we are asked to investigate relation between the minimization of above equation and solving below linear system.

$$A\mathbf{x} = \mathbf{b} \tag{2}$$

First, let's consider the minimization problem. So as to find the $\mathbf{x}$ value that minimizes the $f(\mathbf{x})$, we need to look up its derivative.

$$\frac{\partial f}{\partial \mathbf{x}} = A\mathbf{x} - \mathbf{b}$$

Given that Eqn. 3 is true when the A is symmetric.

$$\frac{\partial \mathbf{x}^\mathsf{T} A\mathbf{x}}{\partial \mathbf{x}} = 2A\mathbf{x} \tag{3}$$

At this point, one need to consider the following relation to be able to infer minimum value of $\mathbf{x}$.

$$\frac{\partial f}{\partial \mathbf{x}} = \mathbf{0} = A\mathbf{x} - \mathbf{b} \tag{4}$$

Finally, the solution of the Eqn. 4 is equivalent of the solution of the Eqn. 2. Therefore, we can conclude that minimization of the Eqn. 1 is same as with the solution of Eqn. 2 if the A is symmetric and positive definite.

1.2 Finding the Optimal Learning Rate in Gradient Descent Algorithm

For a given function $f(\mathbf{x})$ to be minimized, the gradient descent algorithm can be written in a following way:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \tau \nabla f(\mathbf{x}_n)$$

From Eqn. 4, $\nabla f(\mathbf{x}_n)$ is known for our case:

$$\nabla f(\mathbf{x}_n) = A\mathbf{x}_n - \mathbf{b}$$

However, there emerges a question that what is the optimal value of learning rate (τ)? To provide an answer for this question, we need to investigate the rate of change of next estimation of the function ($f(\mathbf{x})$) with respect to learning rate (τ), which is formulated below:

$$\frac{\partial f(\mathbf{x}_{n+1})}{\partial \tau} = \frac{\partial f(\mathbf{x}_{n+1})}{\partial \mathbf{x}_{n+1}} \frac{\mathbf{x}_{n+1}}{\partial \tau} \tag{5}$$

To obtain a stable algorithm, Eqn. 5 should be zero, which means that the estimation of function should not be dependent on learning rate. One can easily compute that:

$$\frac{\partial f(\mathbf{x}_{n+1})}{\partial \mathbf{x}_{n+1}} = \nabla f(\mathbf{x}_{n+1})$$

$$\frac{\mathbf{x}_{n+1}}{\partial \tau} = -\nabla f(\mathbf{x}_n)$$

To sum up, we obtain:

$$\nabla f(\mathbf{x}_{n+1})^{\mathsf{T}} \left(-\nabla f(\mathbf{x}_n)\right) = 0$$

Let's dive in this equation for much more detail.

$$\left(A\mathbf{x}_{n+1} - \mathbf{b}\right)^{\mathsf{T}} \left(A\mathbf{x}_n - \mathbf{b}\right) = 0$$

Let's say $\alpha_n = A\mathbf{x}_n - \mathbf{b}$, and continue with this notation for simplicity, without attributing a meaning.

$$\left(A\left(\mathbf{x}_n - \tau\alpha_n\right) - \mathbf{b}\right)^{\mathsf{T}} \alpha_n = 0$$

$$\left(A\mathbf{x}_n - A\tau\alpha_n - \mathbf{b}\right)^{\mathsf{T}} \alpha_n = 0$$
$$\left(A\mathbf{x}_n - \mathbf{b}\right)^{\mathsf{T}} \alpha_n - \left(A\tau\alpha_n\right)^{\mathsf{T}} \alpha_n = 0$$

$$\left(A\mathbf{x}_n - \mathbf{b}\right)^{\mathsf{T}} \alpha_n = \tau \left(A\alpha_n\right)^{\mathsf{T}} \alpha_n$$

$$\alpha_n^{\mathsf{T}}\alpha_n = \tau \left(A\alpha_n\right)^{\mathsf{T}} \alpha_n$$
$$\alpha_n^{\mathsf{T}}\alpha_n = \tau\alpha_n^{\mathsf{T}} \left(A\alpha_n\right)$$
$$\boxed{\tau = \frac{\alpha_n^{\mathsf{T}}\alpha_n}{\alpha_n^{\mathsf{T}}A\alpha_n}}$$

1.3 Gauss-Seidel Method

As we have done in *Jacobi Method*, again, we decompose the A matrix into parts. In *Gauss-Seidel Method*, the A matrix can be expressed as following:

$$A = L_* + U$$

where L_* is the lower triangular part of the A matrix, and U is the strictly upper triangular part of the A matrix.

Suppose that the $\mathbf{x}^*$ is the solution of the linear system of $A\mathbf{x} = \mathbf{b}$. Then, one can write the following relation:

$$L_*\mathbf{x}^* = \mathbf{b} - U\mathbf{x}^*$$

Finally, the iterative scheme can be obtained as following:

$$\mathbf{x}^{(k+1)} = L_*^{-1}\left(\mathbf{b} - U\mathbf{x}^{(k)}\right)$$

1.4 Time Comparison of Different Methods and Matrix Types

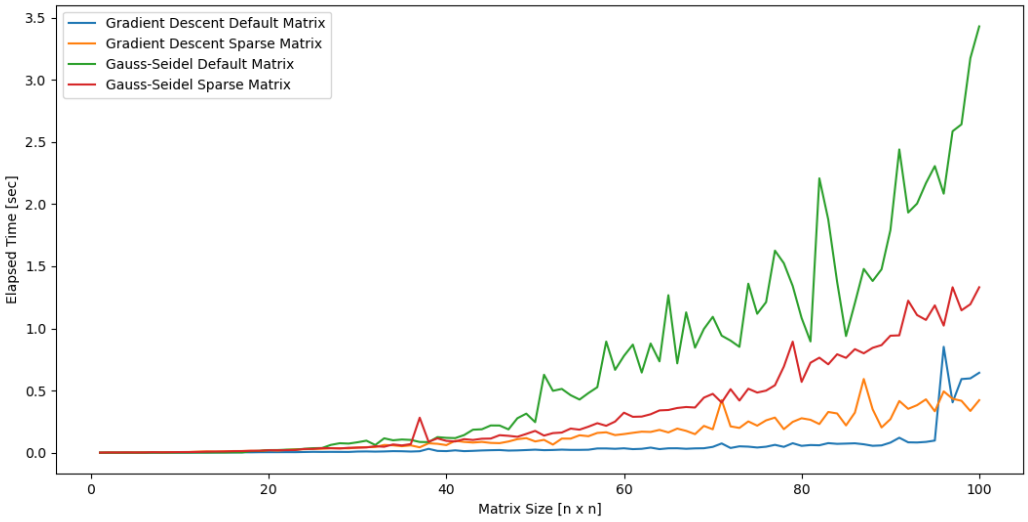

Figure 1: Time Comparison

As can be seen from Figure 1, when we use the *default matrix* constructor of the *NumPy*, we can obtain better performance. If want to analyze Figure 1 in terms of solution methods, we can easily conclude that *Gradient Descent Method* shows superiority over *Gauss-Seidel Method*.

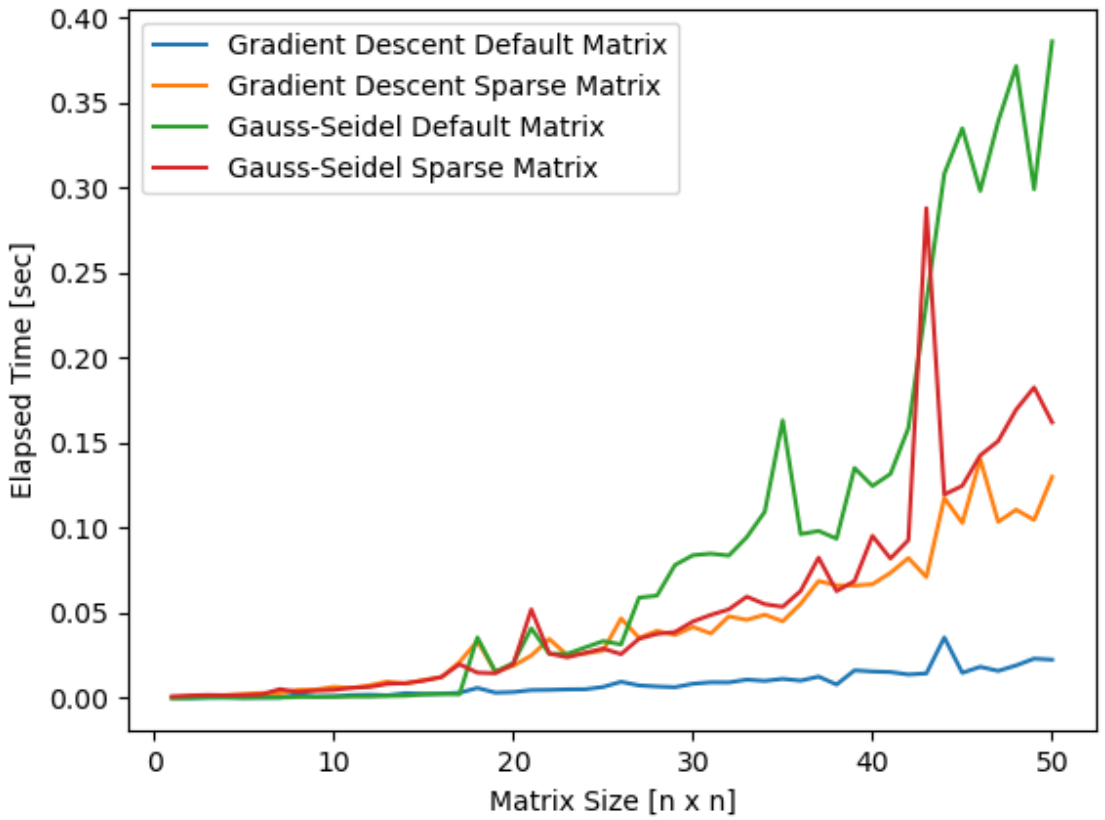

Figure 2: Time Comparison

Moreover, from Figure 2, we can observe when we simulate the time comparisons up to 50×50 matrix, the similar trend with Figure 1 can be displayed.

2 Problem IX

2.1 Time Comparisons of Lambert W Function Evaluations

In the question/code template, we are given the main algorithm in a element-by-element manner. We can display two evidence for that:

- In the inline function definition, broadcast operator (.) is not used.
- Initial value of x determined as a scalar value.

These factors should definitely reduce the performance of the algorithm. To avoid from this draw-back, we can process all algorithm in a element-wise manner. Also, for example, as an ending criteria of the algorithm, we can use the difference of the norm of the solution

vectors rather than using the difference between two scalar guesses. Below in Figure 3, the time comparisons of different algorithms can be observed.

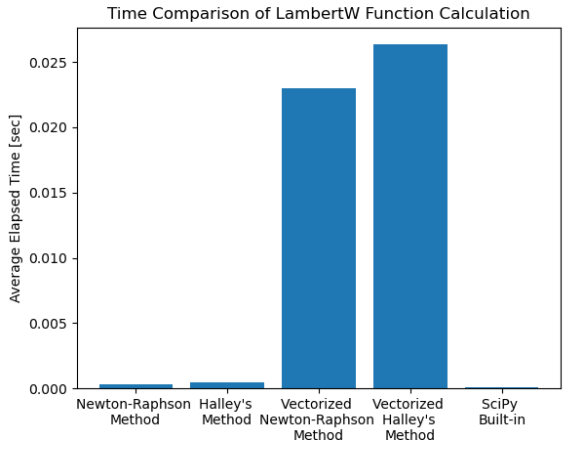

Figure 3: Time Comparison of Lambert W Function Evluations

As can be seen from Figure 3, the vectorized operation results in dramatic performance reduction.

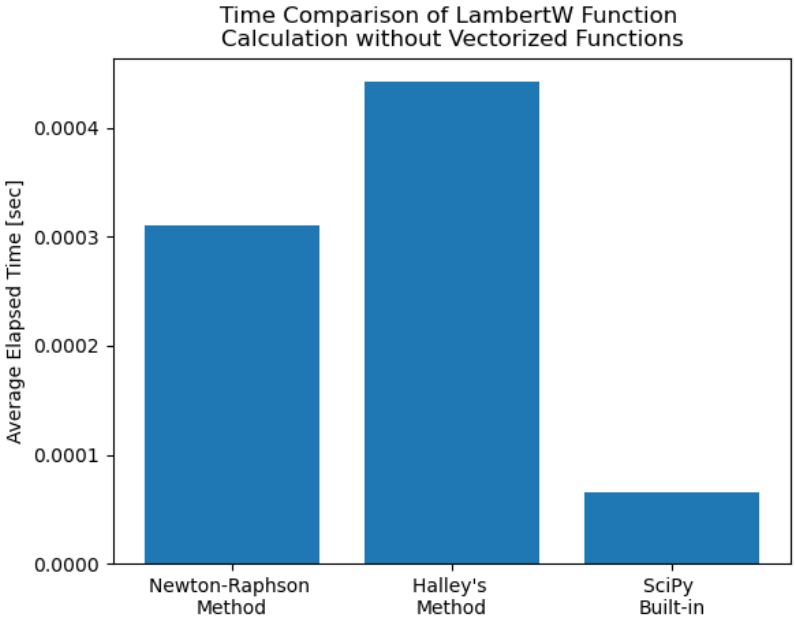

Figure 4: Time Comparison of Lambert W Function Evluations in Detail

In Figure 4, the time comparisons of algorithms (vectorized operation is omitted) can be displayed. Here, we can observe the similarity between the algorithms we implemented and the built-in NumPy function.

2.2 Function Evaluations

From Figure 5, we can evaluate the *Newton-Raphson Method* and *Halley's Method* are able to approximate *Lambert W* function with a great accuracy. It can be observed from the Figure 5 that nearly all the time, the difference between *Newton-Raphson Method* and *Halley's Method* and NumPy is zero.

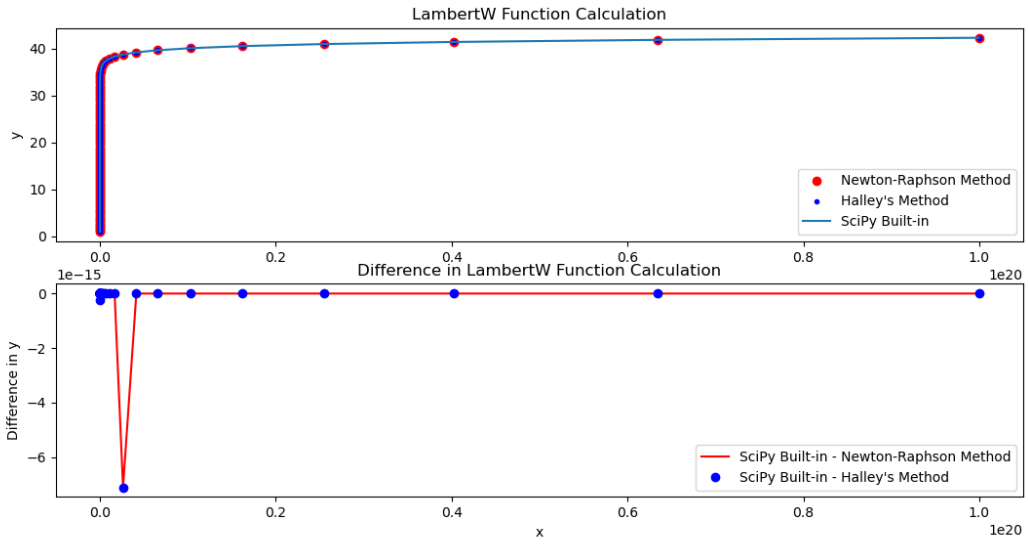

Figure 5: Function Evaluations and Differences w.r.t. NumPy

2.3 Time Comparisons of Koc W Function Evaluations

The same discussion in Section 2.1 and Section 2.2 is also valid here.

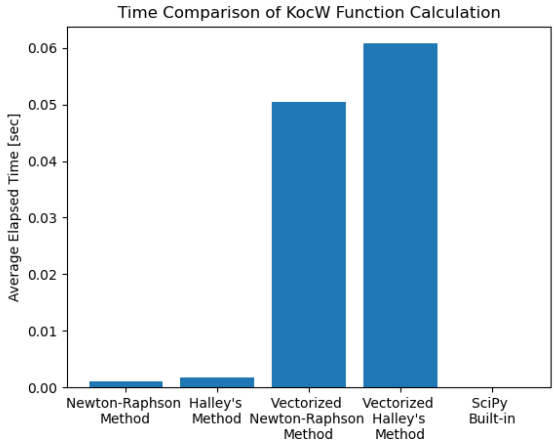

Figure 6: Time Comparison of Koc W Function Evluations

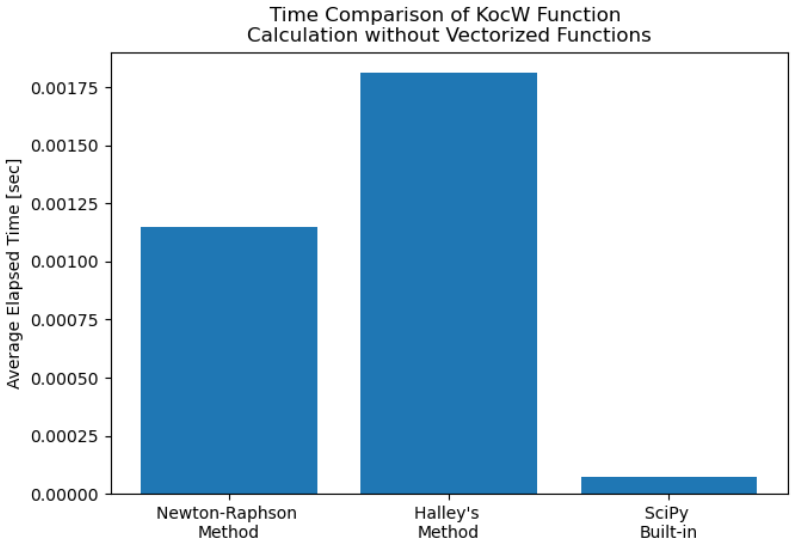

Figure 7: Time Comparison of Koc W Function Evluations in Detail

2.4 Function Evaluations

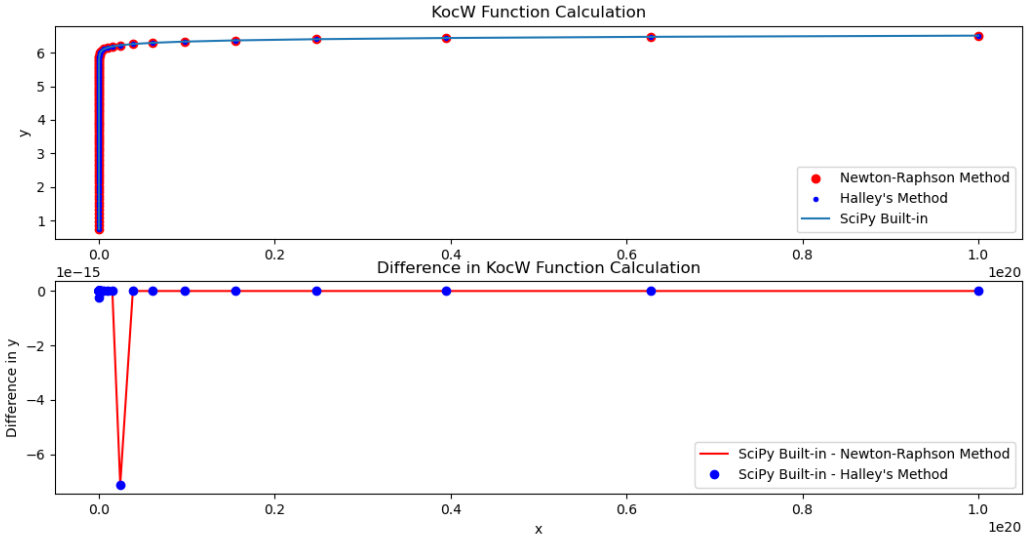

Figure 8: Function Evaluations and Differences w.r.t. NumPy

3 Problem VII

3.1 Hotelling's Deflation

We are given an A matrix to find its eigenvalues and eigenvectors. When we apply the *Power Iteration* method, we can find the largest eigenvalue (in absolute value) and the corresponding eigenvector of the A matrix. To be able to find the other eigenvalues and eigenvectors, one can apply the *Hotelling's Deflation* method. First, let's appyl eigendecomposition to the A matrix, then A matrix can be written as following:

$$A = Q\Lambda Q^{-1}$$

Now, suppose that A is a symmetric matrix, then following relation holds:

$$Q^{-1} = Q^{\mathsf{T}} \tag{6}$$

$$A = Q\Lambda Q^{\mathsf{T}}$$

where Λ is the diagonal matrix whose elements are the eigenvalues of the A matrix, and Q is the orthogonal matrix whose columns are the eigenvectors of A matrix.Then, the above relation can be written for $N \times N$ symmetric matrix in this way:

$$A = \lambda_1 \mathbf{v_1} \mathbf{v_1}^{\mathsf{T}} + \lambda_2 \mathbf{v_2} \mathbf{v_2}^{\mathsf{T}} \dots \lambda_N \mathbf{v_N} \mathbf{v_N}^{\mathsf{T}} \tag{7}$$

where λ is the eigenvalue of A matrix and $\mathbf{v}$ is the corresponding eigenvector. Eigenvalues are oriented such that the relation of $(|\lambda_1| > |\lambda_2| > \dots |\lambda_N|)$ holds.

Let's construct a **new** matrix in following way:

$$A^{(1)} = A - \lambda_1 \mathbf{v_1} \mathbf{v_1}^{\mathsf{T}} \tag{8}$$

where λ_1 is the largest eigenvalue which is obtained from power iteration, and $\mathbf{v_1}$ is the corresponding eigenvector. Then, if we plug the Eqn. 7 into Eqn. 8, we obtain the following relation:

$$A^{(1)} = \lambda_2 \mathbf{v_2} \mathbf{v_2}^{\mathsf{T}} + \lambda_3 \mathbf{v_3} \mathbf{v_3}^{\mathsf{T}} \dots \lambda_N \mathbf{v_N} \mathbf{v_N}^{\mathsf{T}}$$

where where λ_2 is the largest eigenvalue of $A^{(1)}$ and, apparently, the second largest eigenvalue of A which are obtained from power iteration. Finally, we can conclude that when we apply the operation in Eqn. 8, the new matrix we obtained has the largest eigenvalue such that it is the second largest eigenvalue of the matrix to which we applied operation. As a side note, if the A matrix is not a symmetrix matrix, then we could not use the relation in the Eqn. 6, and we were not able to construct the whole algorithm.

3.2 Time Comparisons

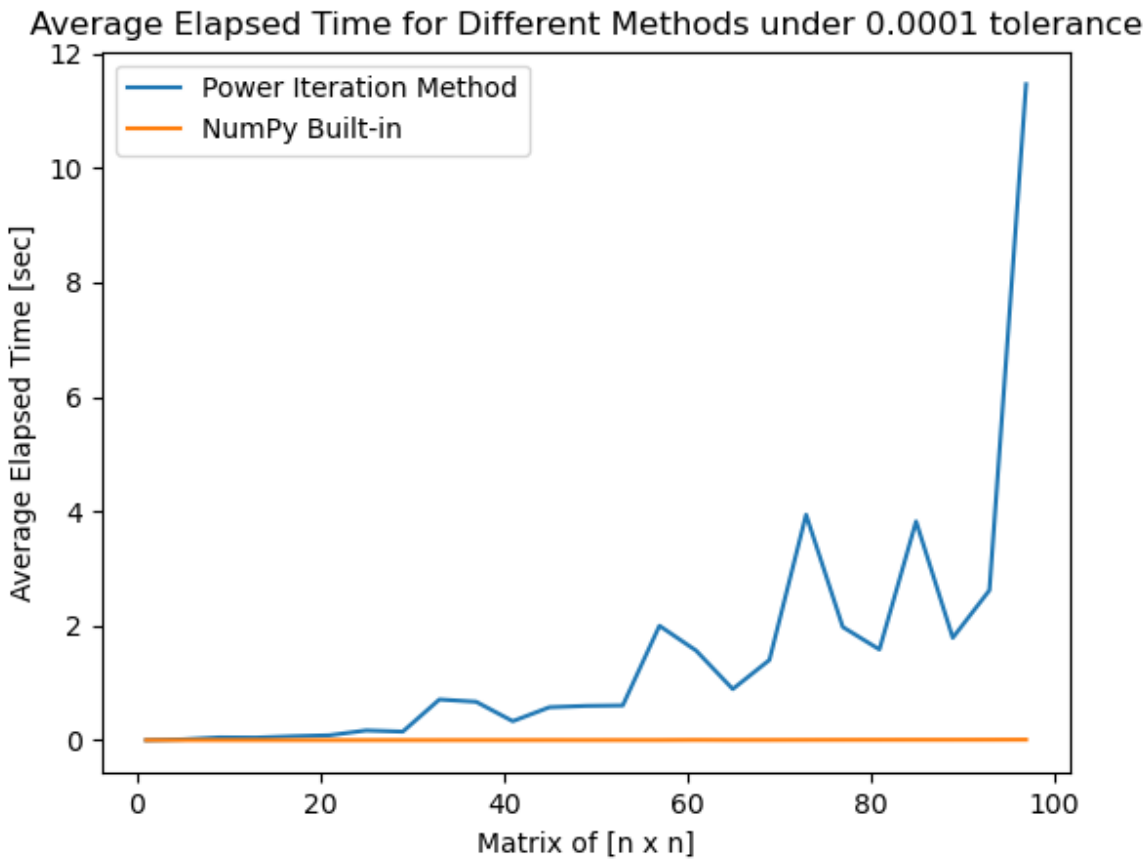

Figure 9: Time Comparison Power Iteration Method and Built-in NumPy *eig()* Function

As can be seen from the Figure 9, the pure power iteration algorithm performs very poorly compared to the built-in NumPy *eig()* function. Generally, the reason for the low performance is the time requirement for the convergence. When we set a low value for accuracy or algorithm termination criteria, the performance is tended to increase, however, in this case, the accuracy is not desirable. For example, below in Figure 10, the improvements on the algorithm can be observed, when the accuracy is decreased.

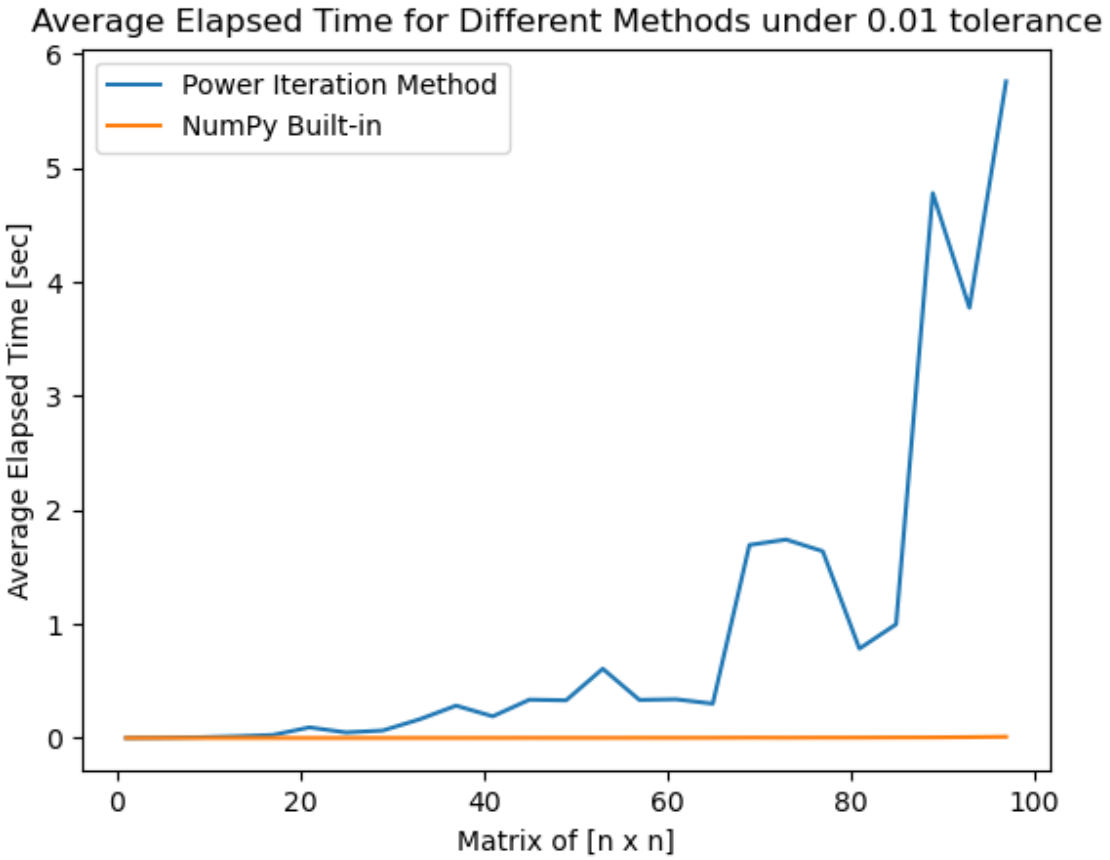

Figure 10: Time Comparison Power Iteration Method and Built-in NumPy *eig()* Function with Low Accuracy

3.3 Accuracies Compared to NumPy Built-in

Now, we will examine the accuracy of the eigenvalues and eigenvectors we found. To be able to do this, we will analyze the plots whose x-axis and y-axis are the power iteration solution, and NumPy solution, respectively. As a side note, all following figures are plotted for a 100×100 random symmetric matrix. In Figure 11, the accuracies of the eigenvalues estimated by *Power Iteration* method can be observed. At Figure 11 and Figure 12, the condition to terminate the iteration is to have a proximity of 10^{-4} between successive iterations.

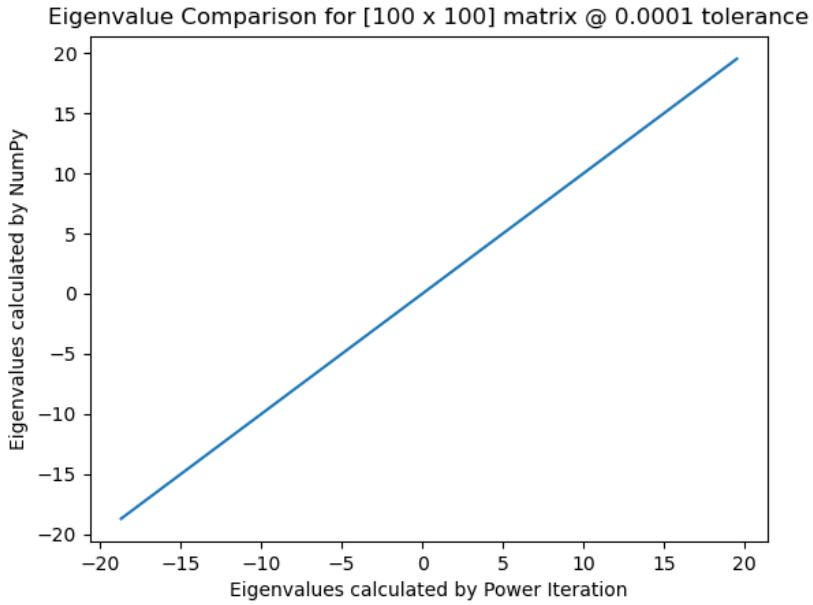

Figure 11: Accuracy of Eigenvalue Estimation Compared to NumPy - Termination Condition=10^{-4}

In Figure 12, the accuracies of the eigenvectors estimated by *Power Iteration* method can be observed. Again, in this Figure 12, the x-axis represents MATLAB estimation, while y-axis represents Power Iteration estimates.

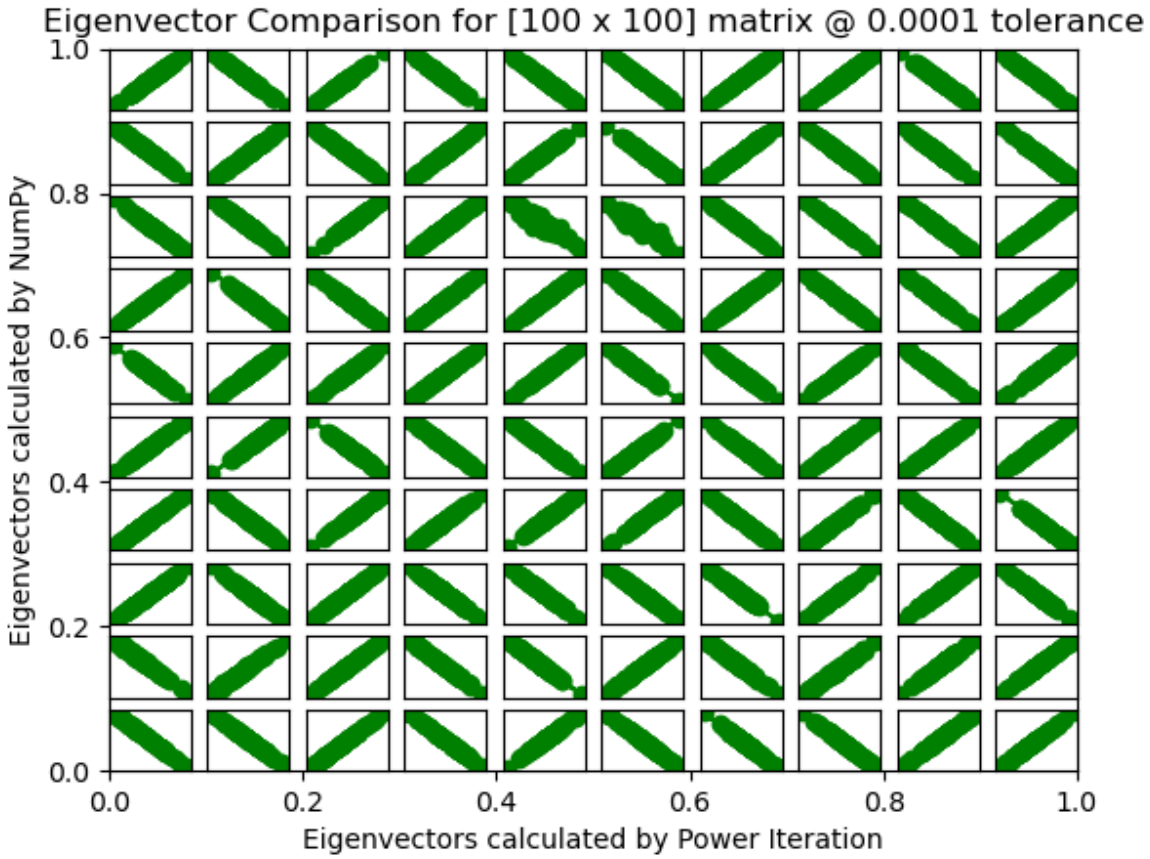

Figure 12: Accuracy of Eigenvector Estimation Compared to NumPy - Termination Condition=10^{-4}

Now we willduplicate the process for a termination condition of 10^{-2}. Having these condition, Figure 13 and Figure 14 can be displayed. All axis representations and matrix conditions are the same.

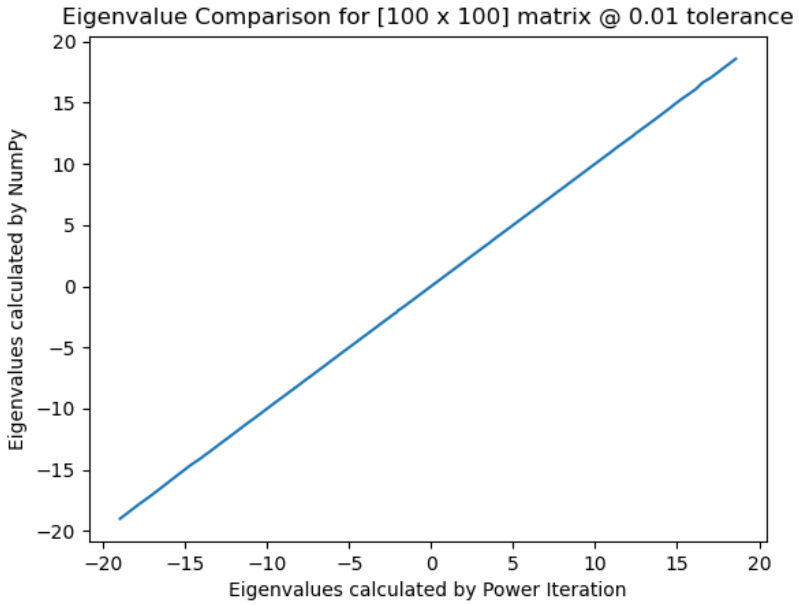

Figure 13: Accuracy of Eigenvalue Estimation Compared to NumPy - Termination Condition=10^{-2}

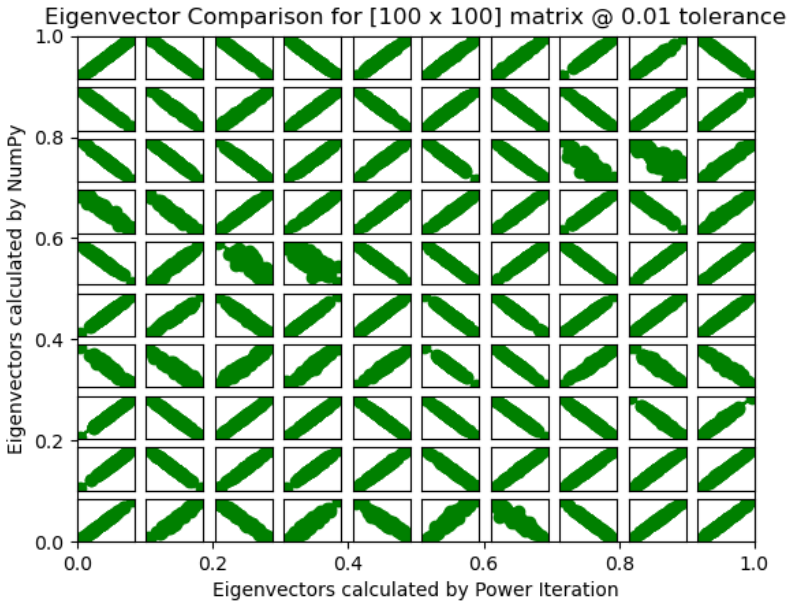

Figure 14: Accuracy of Eigenvector Estimation Compared to NumPy - Termination Condition=10^{-2}

From Figure 13 and Figure 14, we can see that while the eigenvalue results are tended to preserve its accuracy, the eigenvectors start to decompose its accuracy. The intricate

subplots in the Figure 14 shows this loss of accuracy.

3.4 Wigner's Semicircle Law

Wigner's Semicircle Law states that as $N \to \infty$, the probability that an eigenvalue λ is in the interval of $(\sqrt{N}x_1 < \lambda < \sqrt{N}x_2)$ is:

$$P(x_1 < \frac{\lambda}{\sqrt{N}} < x_2) = \frac{1}{2\pi} \int_{x_1}^{x_2} \sqrt{4 - x^2}\, dx$$

When the integration is done, the results can be displayed as following:

$$P(x_1 < \frac{\lambda}{\sqrt{N}} < x_2) = \frac{1}{2\pi} \left[\frac{1}{2} \sqrt{4 - {x_2}^2} x_2 + 2 \arcsin \frac{x_2}{2} - \frac{1}{2} \sqrt{4 - {x_1}^2} x_1 + 2 \arcsin \frac{x_1}{2} \right]$$

When we plot the probability distribution of the eigenvalues that is obtained by NumPy, and the estimate of the Wigner's Semicircle Law for $N = 512,\ 1024,\ 2048$, we obtain the following Figure 15,Figure 16, and Figure 17.

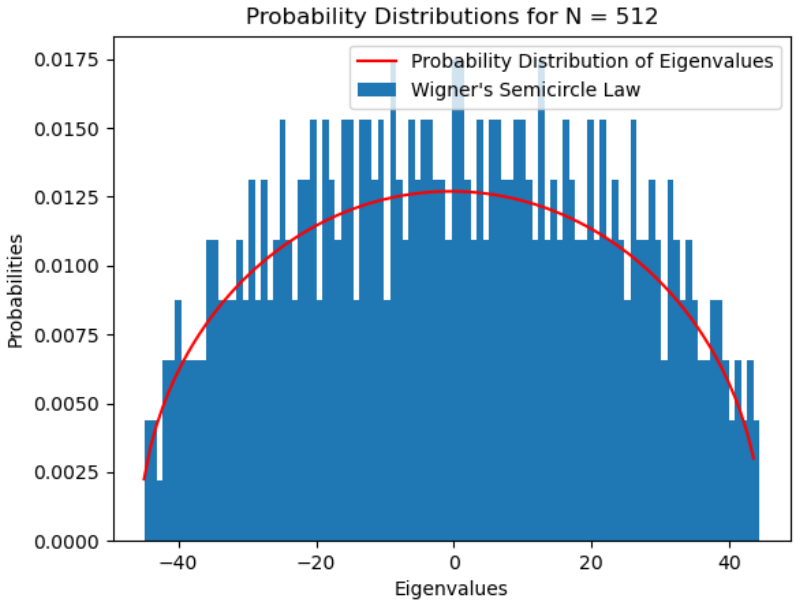

Figure 15: Probability distribution for $N = 512$

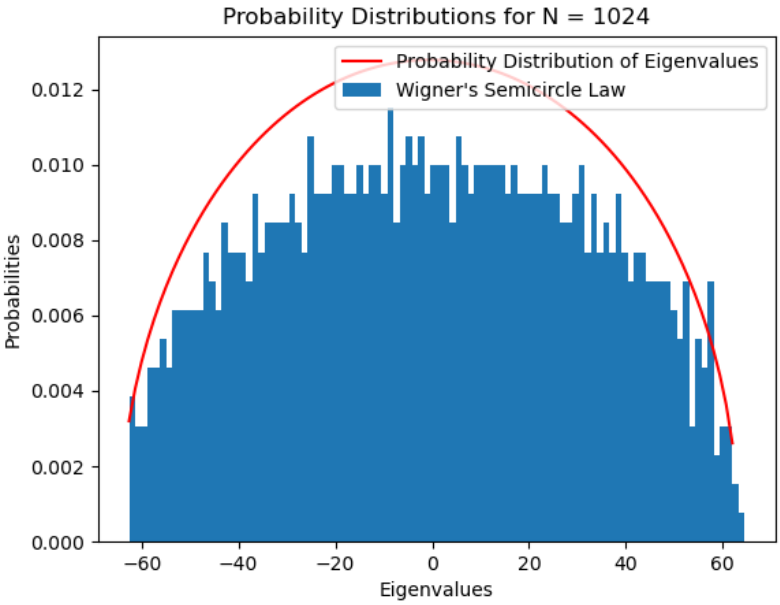

Figure 16: Probability distribution for $N = 1024$

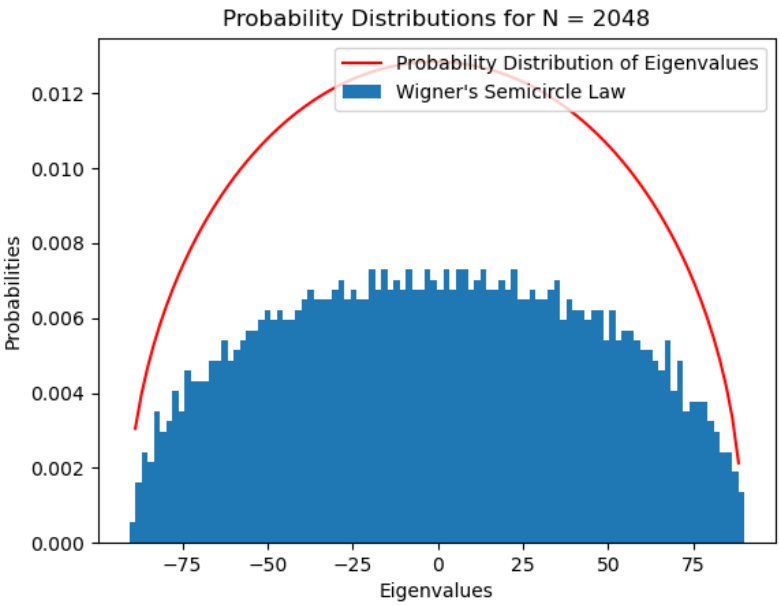

Figure 17: Probability distribution for $N = 2048$