Iterative Solvers

Gustav Cedergrund and Kevin Stull
APPM 4600 - Numerical Analysis and Scientific Computing
Department of Applied Mathematics, University of Colorado - Boulder

Abstract:
Finding the solution to linear systems and the eigenvalue problem is a fundamental challenge in modern numerical mathematics. Given the high expense of direct solving methods, iterative solvers can be applied in specific cases at a fraction of the computational cost. For linear systems, intrinsic techniques like Richardson's Iteration and contemporary techniques like Generalized Minimal Residual Iteration can outperform Direct Inversion under the proper conditions. In this paper, we investigate those conditions as well as the advantages and shortcomings of these two methods. These investigations are then expanded with a deeper look into modern eigenvalue-based iterative techniques in the Power method and QR Iteration. Finally, this paper delves into real-world applications, showcasing the instrumental role of iterative solvers in addressing challenges faced by industry giants such as Google and Boeing. Through exploration, this paper aims to frame and contextualize the intricate problem space, establishing the conditional significance of iterative solvers in theoretical and practical domains.

1. Introduction

In the wide field of computational mathematics, linear systems of the form $\mathbf{Ax} = \mathbf{b}$ are ubiquitous. They exist as a foundational problem in contexts as varied as physics, engineering, data analysis, and general optimization [1]. As such, finding efficient ways to solve these linear systems is a fundamental challenge with profound implications across many domains.

The most intrinsic method for finding the solution $\mathbf{x}^*$ of these systems is by Direct Inversion, that is, $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$. The issue with this straightforward approach is that inverting a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ performs very poorly as size n increases; specifically, the algorithm has a run-time of $O(n^3)$ [2].

To solve this issue, 'Iterative Solver' methods are often implemented. These methods operate by refining an initial guess $\mathbf{x}_0$ over a sequence of successive approximations $\{\mathbf{x}_n\}$. This sequence of approximations, or iterations, aims to converge to the true solution such that $\{\mathbf{x}_n\} \to \mathbf{x}^*$. In principle, iterative methods require infinitely many iterations for exact convergence; however, they should also give a good approximation after some finite number. Therefore, we construct each iterative solver with a halting condition based on some criterion. This criterion may be when the accuracy of the approximation has reached some tolerance, when each successive iteration changes only a negligible amount, or simply when a maximum iteration count is reached. To measure the performance of an iterative method, we commonly refer to the *rate of convergence*. This term refers to the speed at which the approximation error changes at each successive iteration $\mathbf{x}_k$. We say that the method converges with order α and asymptotic error constant μ if

$$\lim_{n\to\infty} \frac{\|\mathbf{x}^* - \mathbf{x}_{k+1}\|}{\|\mathbf{x}^* - \mathbf{x}_k\|^\alpha} = \mu. \tag{1}$$

The iteration converges linearly if $\alpha = 1$ and $\mu < 1$. If $\alpha = 2$, it is quadratically convergent [3].

The Introductory Material portion of this report delves into the applications of two specific iterative

solvers - Richardson's Iteration and Generalized Minimal Residual (GMRES) - which exhibit promise in efficiently handling large-scale linear systems.

Richardson's Iteration is one of the most straightforward iterative solver techniques; still, it performs very well for certain matrices. It was initially proposed in 1910 by Lewis Fry Richardson [4]. The technique recasts the iterative approach for a linear system problem into a fixed point iteration. In this way, we repeatedly apply a function $g : \mathbb{R}^n \to \mathbb{R}^n$ with a fixed point at $\mathbf{x}^*$ (i.e., $g(\mathbf{x}^*) = \mathbf{x}^*$) such that $\{g(\mathbf{x}_n)\} \to \mathbf{x}^*$. By definition in (1), and from the Fixed Point Theorem [5], Richardson's Iteration will converge if and only if the ratio of error at each successive iteration is less than 1, that is,

$$\frac{\|\mathbf{x}^* - \mathbf{x}_{k+1}\|}{\|\mathbf{x}^* - \mathbf{x}_k\|} < 1.$$

GMRES is a newer and more complex iterative solver developed jointly by Yousef Saad and Martin H. Schultz in 1986 [6]. The technique works by implementing an iteration scheme in relation to a Krylov subspace, a concept created by Alexei Krylov in 1931 [7]. The Krylov subspace $\mathcal{K}_m$ generated by matrix $\mathbf{A}$ and vector $\mathbf{b}$ is the linear subspace spanned by the images of $\mathbf{b}$ under the first m powers of $\mathbf{A}$ starting at 0 [8]. This is shown below in (2).

$$\mathcal{K}_m = \{\mathbf{b}, \mathbf{Ab}, \mathbf{A}^2\mathbf{b}, \ldots, \mathbf{A}^{m-1}\mathbf{b}\} \tag{2}$$

GMRES approximates the solution by finding a vector $\mathbf{x}_m$ in the image of the Krylov subspace $\mathcal{K}_m$ with minimal residual ($\|\mathbf{Ax}_m - \mathbf{b}\|$) [6]. In this, the method becomes a minimization problem at each iteration. We utilize the Arnoldi method to help find this vector $\mathbf{x}_m$. The Arnoldi method is its own iterative technique we implement solely for the purpose of vector approximation; thus, we will not cover its entire workings in detail. For the complete formulation of this method, we reference the original publication by W. E. Arnoldi in 1951 [9].

This paper will first cover the mathematical derivation of both methods in Section 2 and then discuss the numerical results of method implementation in Section 3. In sections 4 - 7, we further the talk of iterative solvers in the Independent Extension portion of the paper. The original code formulated for each section can be found in the Appendix in Section 9. By providing a comprehensive overview of these iterative solvers, their underlying principles, and their applicability to different types of linear systems, this paper aims to provide valuable insight to those interested in learning more about numerical analysis and computational mathematics.

2. Mathematical Formulation for Introductory Material

In this section, we develop the mathematical formulation behind each of the two iterative methods - Richardson's Iteration and GMRES - for solving linear systems.

2.1. **Richardson's Iteration.**

Recall from the introduction that Richardson's Iteration recasts the linear system into a fixed point iteration. First, we will investigate this recasting of the problem.

Let $\mathbf{x}^*$ represent the exact solution to a system so that $\mathbf{Ax}^* = \mathbf{b}$. Using this definition, the system

can be re-written as follows for some $\alpha \in \mathbb{R}$:

$$\begin{aligned} \alpha \cdot \mathbf{A}\mathbf{x}^* &= \alpha \cdot \mathbf{b} \\ \mathbf{x}^* - \mathbf{x}^* + \alpha\mathbf{A}\mathbf{x}^* &= \alpha\mathbf{b} \\ \mathbf{x}^* - (\mathbf{I} - \alpha\mathbf{A})\mathbf{x}^* &= \alpha\mathbf{b} \\ \mathbf{x}^* &= (\mathbf{I} - \alpha\mathbf{A})\mathbf{x}^* + \alpha\mathbf{b} \end{aligned} \tag{3}$$

Now, let $g : \mathbb{R}^n \to \mathbb{R}^n$ be a function defined as:

$$g(\mathbf{x}) = (\mathbf{I} - \alpha\mathbf{A})\mathbf{x} + \alpha\mathbf{b}. \tag{4}$$

Combining the two definitions, it is clear that function g as defined in (4) has a fixed point at $\mathbf{x}^*$ as $g(\mathbf{x}^*) = \mathbf{x}^*$ by (3). Thus, we define Richardson's Iteration according to the iteration scheme below:

$$\mathbf{x}_{k+1} = g(\mathbf{x}_k) = (\mathbf{I} - \alpha\mathbf{A})\mathbf{x}_k + \alpha\mathbf{b}$$

Investigating the error of this fixed point iteration at iteration $k+1$, one can derive:

$$\begin{aligned} \|\mathbf{x}^* - \mathbf{x}_{k+1}\| &= \|((\mathbf{I} - \alpha\mathbf{A})\mathbf{x}^* + \alpha\mathbf{b}) - ((\mathbf{I} - \alpha\mathbf{A})\mathbf{x}_k + \alpha\mathbf{b})\| \\ &= \|(\mathbf{I} - \alpha\mathbf{A})(\mathbf{x}^* - \mathbf{x}_k) + \alpha\mathbf{b} - \alpha\mathbf{b}\| \\ &= \|(\mathbf{I} - \alpha\mathbf{A})(\mathbf{x}^* - \mathbf{x}_k)\| \\ &\leq \|(\mathbf{I} - \alpha\mathbf{A})\| \, \|(\mathbf{x}^* - \mathbf{x}_k)\| \end{aligned}$$

such that

$$\frac{\|\mathbf{x}^* - \mathbf{x}_{k+1}\|}{\|\mathbf{x}^* - \mathbf{x}_k\|} \leq \|\mathbf{I} - \alpha\mathbf{A}\| \, . \tag{5}$$

Therefore, as discussed in the introduction, α needs to be chosen such that $\|(\mathbf{I} - \alpha\mathbf{A})\| < 1$ for Richardson's Iteration to converge as a fixed point method. If this is the case, the error decreases at each successive step, and the rate of convergence is at least linear.

Assume now that $\mathbf{A}$ is either Positive Definite (PD) or Positive Semi-Definite (PSD). For a PD matrix, all eigenvalues are positive; for a PSD matrix, all eigenvalues are greater than or equal to 0. PD and PSD matrices are also symmetric [1]. Thus, $\mathbf{A} = \mathbf{A}^\mathsf{T}$ which indicates $\mathbf{I} - \alpha\mathbf{A} = (\mathbf{I} - \alpha\mathbf{A})^\mathsf{T}$ by extension.

For all symmetric matrices $\mathbf{X}$, the norm $\|\mathbf{X}\|$ is equal to the spectral radius $\rho(\mathbf{X})$ [1]. Thus, as $\mathbf{I} - \alpha\mathbf{A}$ is symmetric, $\|(\mathbf{I} - \alpha\mathbf{A})\| = \rho(\mathbf{I} - \alpha\mathbf{A})$. Connecting this to the requirement for convergence given by (5), the iteration converges if and only if the spectral radius $\rho(\mathbf{I} - \alpha\mathbf{A}) < 1$; that is, $\forall i \in \{1, 2, \ldots, n\}$, $|\lambda_i| < 1$ where each λ_i is an eigenvalue of $\mathbf{I} - \alpha\mathbf{A}$.

We continue formulation with a few definitions. For $i \in \{1, 2, \ldots, n\}$ we let λ_i be the i-th largest eigenvalue of $\mathbf{A}$. As $\mathbf{A}$ is PD or PSD, the eigenvalues can be defined as

$$0 \leq \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$$

By extension, the eigenvalues of $\mathbf{I} - \alpha\mathbf{A}$, assuming an $\alpha \geq 0$, are given by

$$1 \geq 1 - \alpha\lambda_1 \geq 1 - \alpha\lambda_2 \geq \cdots \geq 1 - \alpha\lambda_n$$

which means that

$$\rho(\mathbf{I} - \alpha\mathbf{A}) = \max\{|1 - \alpha\lambda_1|, |1 - \alpha\lambda_n|\}. \tag{6}$$

Defining α as

$$\alpha = \frac{2}{\lambda_1 + \lambda_n}, \tag{7}$$

then $\alpha \geq 0$ as assumed. Plugging α back into (6), with the current definitions of λ_1, gives:

$$\rho(\mathbf{I} - \alpha\mathbf{A}) = \max\left\{\left|1 - \frac{2}{\lambda_1 + \lambda_n}\lambda_1\right|, \left|1 - \frac{2}{\lambda_1 + \lambda_n}\lambda_n\right|\right\} \tag{8}$$

$$\leq \max\{|1 - 0|, |1 - 2|\} = 1. \tag{9}$$

As $\rho(\mathbf{I} - \alpha\mathbf{A}) \leq 1$, convergence of Richardson's is not guaranteed. However, requiring instead in (8) that $\mathbf{A}$ is exclusively PD so that $\lambda_1 > 0$, then the equality in (9) becomes strict - shown below in (10).

$$\rho(\mathbf{I} - \alpha\mathbf{A}) < \max\{|1 - 0|, |1 - 2|\} = 1 \tag{10}$$

Thus, given that our matrix $\mathbf{A}$ is Positive Definite, we can define α as in (7). This means our fixed point iteration conditions are satisfied, and Richardson's Iteration converges.

As for the rate of convergence of the method, from (5) and (8), it is clear that Richardson's Iteration converges linearly with a constant μ dependent on the ratio of λ_n/λ_1. This ratio is also the definition of the condition number of a matrix, $\kappa(\mathbf{A})$ [2]. Therefore, the larger the condition number, the slower the convergence; if $\kappa(\mathbf{A}) = 1$ such that $\lambda_1 = \lambda_n$, then the method converges quadratically.

Important to note is that the convergence criterion based on eigenvalues of $\mathbf{A}$ is independent of initial guess $\mathbf{x}_0$. As a result, if the conditions for convergence are met, Richardson's Iteration converges regardless of the initial guess.

With all parts and steps defined, we formalize the method in Algorithm 1 below:

Algorithm 1 - Richardson's Iteration

$\mathbf{x_0} \leftarrow \mathbf{0}$ // arbitrary initial guess
for $k = 0, 1, 2, \ldots$ **do**
  $\mathbf{x_{k+1}} \leftarrow (\mathbf{I} - \alpha\mathbf{A})\mathbf{x_k} + \alpha\mathbf{b}$
end for

We continue our investigation of Richardson's Iteration in Section 3.1, where we apply this formulated method to various matrices and test its performance.

2.2. **GMRES Iteration.**

This section considers the mathematical formulation of GMRES - a Krylov subspace method.

The general convergence of Krylov subspace methods is an open question [10]. Naively, it should not be surprising that searching along the span of linear combinations of $\mathbf{b}$ and $\mathbf{A} \in \mathbb{R}^{n\times n}$ should be useful in yielding a solution. However, expecting an exact formulation of $\mathbf{x}^*$ to appear may be overly ambitious. A least squares solution can be found instead.

Let residual vector $\mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b}$ for some initial guess $\mathbf{x}_0$. The Krylov subspace of $\mathbf{A}$ and $\mathbf{r}_0$ can then be formulated as

$$\mathcal{K}_m = \{\mathbf{r}, \mathbf{A}\mathbf{r}, \mathbf{A}^2\mathbf{r}, \ldots, \mathbf{A}^{m-1}\mathbf{r}\}. \tag{11}$$

The theoretical formulation of GMRES relies on an iterative minimization - finding vector $\mathbf{x}_m \in \mathbf{x}_0 + \mathcal{K}_m$ that minimizes $\mathbf{r}_m = \|\mathbf{A}\mathbf{x}_m - \mathbf{b}\|$. However, the columns of $\mathcal{K}_m$ in (11) may be close to linear dependence. Thus, one can apply Arnoldi iteration to enforce linear independence, forming an orthonormal basis for $\mathcal{K}_m$. Define the vectors of this basis as $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m$ such that $\mathbf{Q}_k \in \mathbb{R}^{n\times k}$ is the first k $\mathbf{q}$ vectors. Also, let the first vector $\mathbf{q}_1 = \frac{\mathbf{r}_0}{\|\mathbf{r}_0\|}$. Arnoldi iteration can then be defined via a recurrence relationship of the form

$$\mathbf{A}\mathbf{Q}_k = \mathbf{Q}_{k+1}\hat{\mathbf{H}}_k. \tag{12}$$

with $\mathbf{H} \in \mathbb{R}^{k+1\times k}$ a constructed upper Hessenberg matrix [9].

Now, return to the desired vector $\mathbf{x}_m$. As $\mathbf{Q}_m$ is a basis of $\mathcal{K}_m$, there is some $\mathbf{y}_m$ [1] so that

$$\mathbf{x}_m = \mathbf{x}_0 - \mathbf{Q}_m\mathbf{y}_m \tag{13}$$

Finally, combining all parts from (12) and (13) in $\mathbf{r}_m$ gives the following:

$$\begin{aligned}
\|\mathbf{r}_m\| &= \|\mathbf{A}\mathbf{x}_m - \mathbf{b}\| \\
&= \|\mathbf{A}(\mathbf{x}_0 - \mathbf{Q}_m\mathbf{y}_m) - \mathbf{b}\| \\
&= \|\mathbf{r}_0 - \mathbf{A}\mathbf{Q}_m\mathbf{y}_m\| \\
&= \left\|\beta\mathbf{q}_1 - \mathbf{Q}_{m+1}\hat{\mathbf{H}}_m\mathbf{y}_m\right\| \\
&= \left\|\mathbf{Q}_{m+1}(\beta\mathbf{e}_1 - \hat{\mathbf{H}}_m\mathbf{y}_m)\right\| \\
\|\mathbf{r}_m\| &= \left\|\beta\mathbf{e}_1 - \hat{\mathbf{H}}_m\mathbf{y}_m\right\|.
\end{aligned} \tag{14}$$

In this equation, $\mathbf{e}_1$ is the first canonical vector (i.e. $e_1 = [1, 0, 0, 0, ..., 0]$) and $\beta = \|\mathbf{r}_0\|$.

Therefore, at iteration k of GMRES, orthonormal vector $\mathbf{q}_k$ is generated using (12) and $\mathbf{y}_k$ is found to minimize $\mathbf{r}_k$, as calculated using (13). Then, least squares approximation $\mathbf{x}_k$ is computed, considered sufficient if $\mathbf{r}_k$ is below some tolerance. This is realized with the following algorithm:

Algorithm 2 - GMRES

$\mathbf{x_0} \leftarrow$ some initial guess
$\mathbf{r}_0 \leftarrow \mathbf{A}\mathbf{x}_0 - \mathbf{b}$
$\mathbf{v_1} \leftarrow \frac{\mathbf{r}_0}{\|\mathbf{r}_0\|}$
for $k = 0, 1, 2, \ldots, n$ **do**
  $h_{i,j} = \langle \mathbf{A}\mathbf{v}_j, \mathbf{v}_j \rangle$ for i = 1,2,. . . ,j
  $\mathbf{v}_{j+1} = \mathbf{A}\mathbf{v}_j - \sum_{i=1}^{j} \mathbf{h}_{i,j}\mathbf{v}_i$
  $h_{j+1,j} = \left\|\mathbf{v}_{j+1}\right\|$
  $\mathbf{v}_{j+1} = \frac{\mathbf{v}_{j+1}}{h_{j+1,j}}$
  $\mathbf{x_k} \leftarrow \mathbf{x}_0 + \mathbf{V}_k\mathbf{y}_k$
  if $\|\mathbf{r}_k\| <$ tol **return** $\mathbf{x}_k$
end for

This algorithm has two termination conditions: when a maximum number of iterations is reached or $\|\mathbf{r}_k\| <$ tol is achieved. Currently, the algorithm does not use $k > n$; instead, once $k = n$ is found without adequately minimizing the residual, $\mathbf{x}_k$ is returned and used as an updated initial guess $\mathbf{x}_0$ in a new GMRES function call. This technique, called restarted GMRES [11], limits memory requirements while allowing for faster convergence of the algorithm. We define an *epoch* as the number of times that GMRES restarts with an updated initial guess $\mathbf{x_0}$. An epoch parameter dictates the number of allowed epochs. The behavior GMRES is expanded in Section 3.2.

3. Numerical Work for Introductory Material

In this section, we include numerical results of the iterative solvers for linear systems. Noteworthy for this section - and for those that follow - is the method by which we create our test matrices. We outline the procedure below.

Let $\mathbf{R} \in \mathbb{R}^{n\times n}$ be a full rank matrix with random entries. Let $\mathbf{Q}$ denote the orthogonal matrix

resulting from the QR factorization of $\mathbf{R}$. Let $\mathbf{D}$ denote a diagonal matrix with $d_1, d_2, \ldots, d_n$ as its diagonal entries. Then, $\mathbf{A} = \mathbf{Q}^*\mathbf{D}\mathbf{Q}$ is a random matrix with eigenvalues of $d_1, d_2, \ldots, d_n$. Thus, using this method, we have full control of the eigenvalues in random matrices.

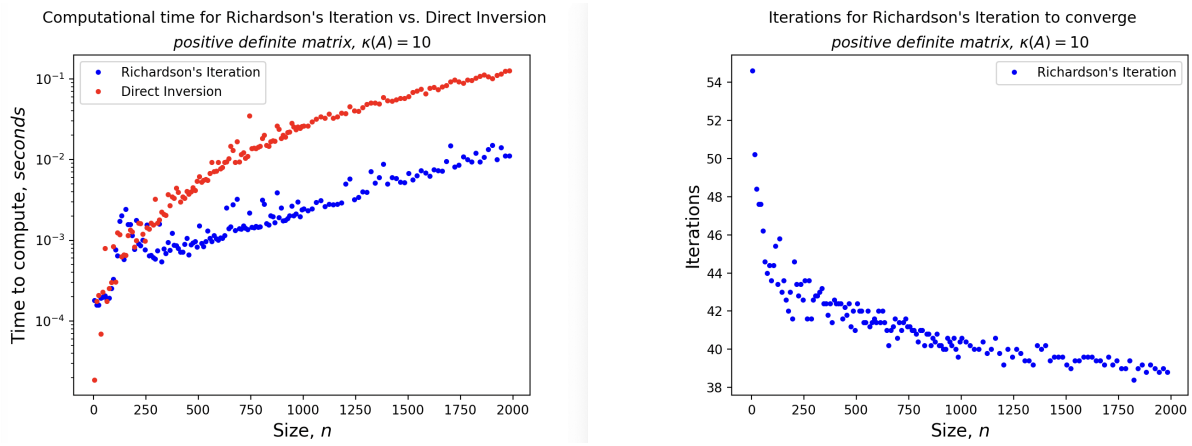
In the same order as with the Mathematical Derivation, we first cover Richardson's Iteration and then consider GMRES. The complete code for the methods implemented and matrices created for this section will be included separately in the Appendix in Section 9.

3.1. **Richardson's Iteration.**

This section continues our investigation into Richardson's Iteration with numerical experiments. For our comparisons in method performance, we will compare the iterative solver with Direct Inversion for various matrices. Each result presented below is the average of 20 simulations.

A few important choices were made in the formulation of our simulations. First, the stopping criteria designated for the iterative method is when the norm of successive iterations $\|\mathbf{x}_k\|$ changes less than 10^{-10}. This induced a very accurate approximation for the solution. If this stopping criterion was not met after 30000 iterations, we halt then as well. We also assume that the maximum and minimum eigenvalues λ_n and λ_1 are already calculated. In this, we aim to isolate the two methods such that the comparisons on linear system solving are objective. Lastly, to calculate $\mathbf{A}^{-1}$, we use the '*numpy.linalg.inv()*' method. This method is optimized explicitly for matrix inversion and performs much better than a naive algorithm. However, as it is a very commonly used method in practice, it is also used in our comparisons.

As the convergence requirement from Section 2.1 was for the matrix $\mathbf{A}$ to be PD, we first test Richardson's Iteration method on randomized positive definite matrices. For this experiment, we standardized the condition number arbitrarily as $\kappa(\mathbf{A}) = 10$. To do so, we equally space our diagonal $\mathbf{D}$ entries, which are also our eigenvalues, between $\frac{n}{10}$ and n. As $(n)/(\frac{n}{10}) = 10$ for all n, we have $\kappa(\mathbf{A}) = \lambda_n/\lambda_1 = 10$ as desired. Results are shown below in Figure 1.



(A) Time (B) Iterations

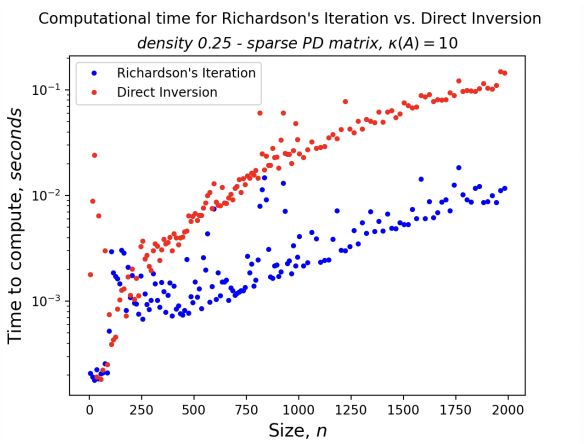
FIGURE 1. Richardson's Iteration, Positive Definite Matrices

The Time graph shows similar performance for the iterative solver and the direct-solving method for $n \leq 250$. As n increases further, however, we see Richardson's Iteration perform much better than Direct Inversion. This follows as, for Richardson's Iteration, the number of iterations is not directly correlated with n. This is made very evident in the Iteration graph. Thus, Richardson's performance

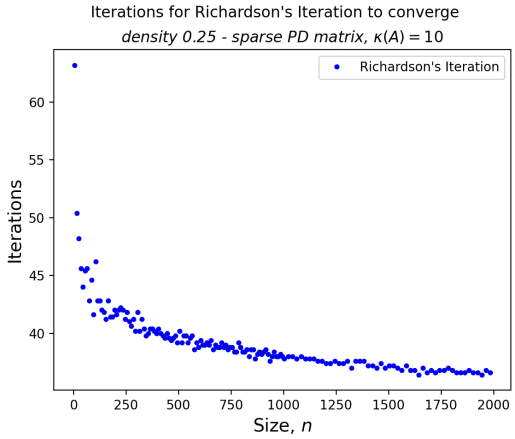
is only bounded by matrix-vector multiplication - a $O(n^2)$ operation [2]. As the introduction mentions, Direct Inversion is bounded by $O(n^3)$. Therefore, the larger the matrix, the more the iterative method outperforms the direct-solving method.

Another application of Richardson's iteration is seen with sparse matrices. *Sparse matrices* are matrices in which most elements are 0. For the non-zero entries, sparse matrices can be populated randomly or systematically according to some structure. We use the term *density* to characterize the sparsity of the matrix. A random sparse matrix with a density of 0.25 would see a random 25% of its entries filled. An example of a structured sparse matrix is a tri-diagonal matrix, in which only the three center diagonals of the matrix are filled.

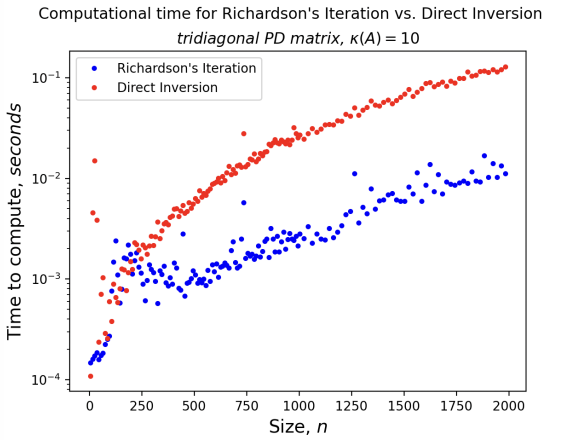
We show the performance of Richardson's Iteration on these two sparse matrix types - random with density 0.25 and structured tri-diagonally - in Figure 2. For all matrices, we retain our constant condition number of 10. To create these matrices, we made our $\mathbf{Q}$ matrix randomly or structurally sparse. We construct our eigenvalues in the same fashion as for Figure 1.



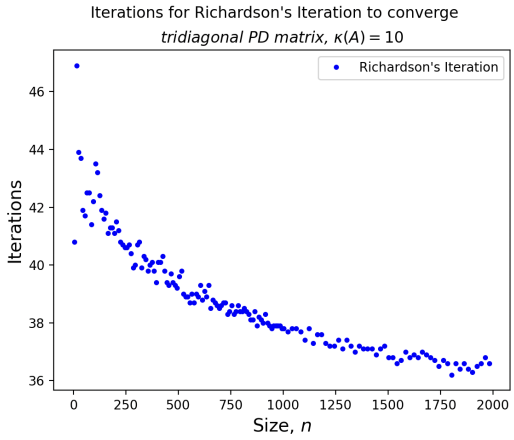
(A) Random Sparse Matrix - Time



(B) Random Sparse Matrix - Iterations



(C) Structured Sparse Matrix - Time



(D) Structured Sparse Matrix - Iterations

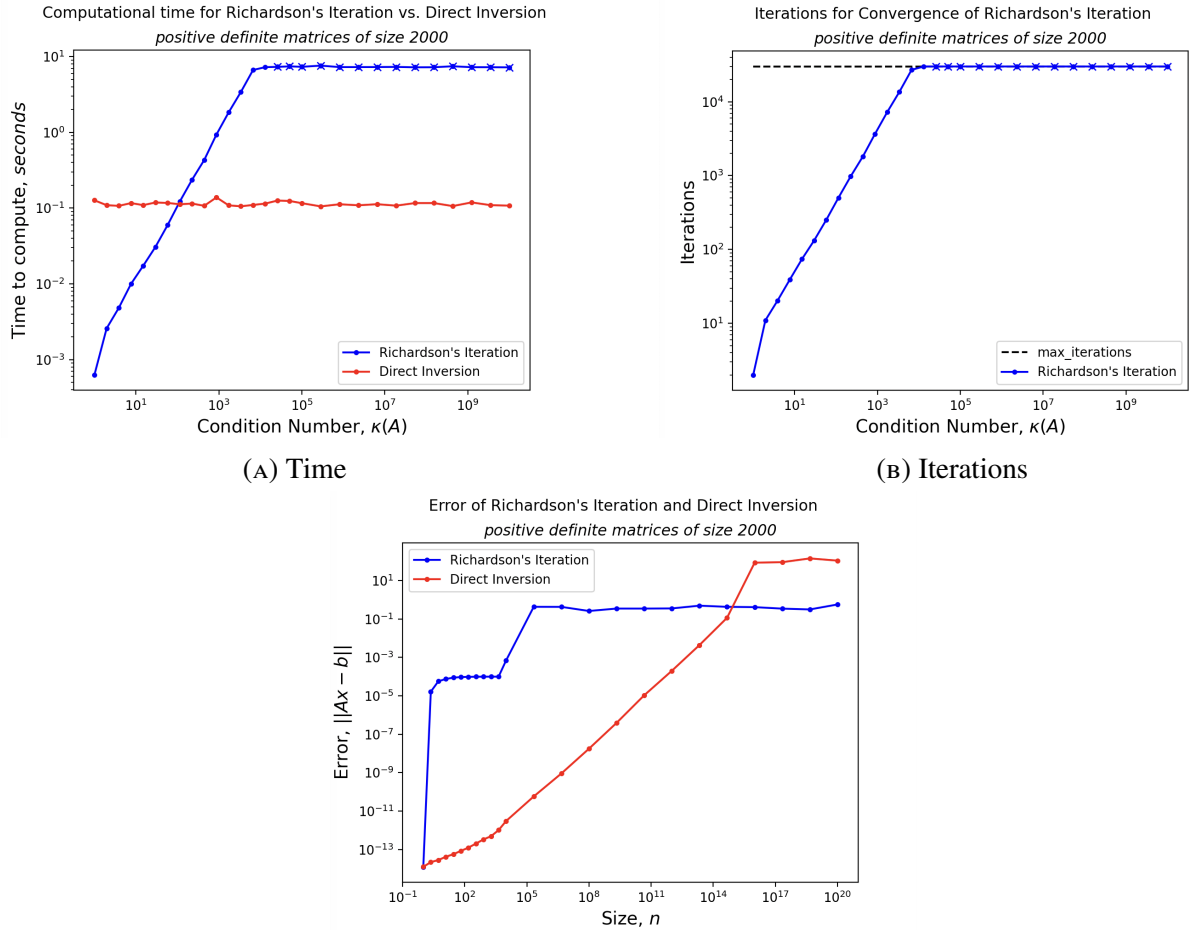
FIGURE 2. Richardson's Iteration, Sparse PD matrices

In these diagrams, we again see Richardson's Iteration performing better than Direct Inversion as size n increases. However, this difference in performance is perhaps not as much as should be expected in the case of sparse matrices. This is for several reasons.

Richardson's Iteration converges independently of size, as shown in Figure 1. Thus, each iteration of Richardson's hinges entirely upon matrix-vector multiplication. Sparse matrices typically have $O(n)$ non-zero entries. By extension, this means that matrix-vector multiplication becomes an $O(n)$ operation [12]. Therefore, theoretically, the solution of solving a sparse linear system with Richardson's becomes a strictly $O(n)$ operation. Practically, however, the required pre-processing of sparse matrices mixed with NumPy's already remarkably optimized matrix multiplication eliminates the potential performance gain from the sparsity. Thus, these graphs show similar performance for sparse matrices to the fully populated matrices in Figure 1.

As for comparisons of convergence with sparsity type, there is little to notice. Richardson's Iteration seems to converge well regardless of sparsity type, given that the condition number is 10.

However, this note raises a question regarding the exact relationship between convergence and condition number. To investigate further, we restrict matrix size to $n = 2000$ and include simulations for varying condition numbers. We condition our PD matrices by evenly spacing eigenvalues between $2000 \cdot (1 - \frac{c-1}{c+1})$ and $2000 \cdot (1 + \frac{c-1}{c+1})$, where c is our desired condition number. We include these comparisons in Figure 3. Those iterations that did not converge are marked with an 'x.'



(A) Time

(B) Iterations

(C) Error (extended x-axis)

FIGURE 3. Richardson's Iteration, PD matrices

In these graphs, we see some fascinating behavior. First, in the Iteration graph, we see the required

iterations for convergence for Richardson increase linearly. This reflects the linear convergence dependent on condition number; as $\kappa(\mathbf{A})$ increases, we expect the method to take more iterations to reach the same accuracy. At $\kappa(\mathbf{A}) \sim 10^4$, the iteration stops converging as the max iteration halting condition is reached. The same phenomenon is mimicked with the Time graph in time to compute. Also noteworthy is that at $\kappa(\mathbf{A}) = 1$, Richardson converges in 1 iteration with immense accuracy. This is reasonable as we anticipate quadratic convergence when all eigenvalues are identical.

Comparing the performance of Richardson's Iteration with Matrix Inversion, the direct-solving method performs better for $\kappa(\mathbf{A}) \geq 100$. We also see that the computational time for Direct Inversion is completely constant. This follows from the method run-time being solely dependent on matrix size, a constant factor in these simulations. However, this constant run-time does not mean the method is immune to error. In the Error graph, both Richardson's Iteration and Direct Inversion produce inaccurate results when the condition number is high enough.

From these experiments, Richardson's Iteration is worthwhile given that the maximum and minimum eigenvalues are known, the matrix is Positive Definite, and the condition number is relatively low (less than 100). We now move on to the results for GMRES, which has less stringent conditions and is more widely used in practice.

3.2. **GMRES Iteration.**

In this section, we explore the numerical results obtained from various experiments with GMRES. To test the iterative solver in a variety of situations, six different matrix scenarios were formulated:

1) A matrix with 2000 distinct well separated eigenvalues.

2) A full rank matrix with 1 distinct eigenvalue.

3) A full rank matrix with 3 distinct eigenvalues.

4) A full rank matrix where all eigenvalues are in a ball of radius $1 \cdot 10^{-5}$ centered at 1.

5) A matrix whose condition number is larger than $1 \cdot 10^{20}$.

6) A matrix who has one zero eigenvalue with $\mathbf{b} \in \text{Range}(\mathbf{A})$.

Three different linear solving methods were compared using this set of metrics. Those methods were:

1) GMRES_ours - the GMRES algorithm we created

2) GMRES_scipy - the GMRES algorithm in the SciPy library

3) direct inverse: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$

For each experiment, the time to converge and the quality of the returned solution are reported. The norm of the residual quantifies the quality of the returned solution, that is, $\|\mathbf{r}\| = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|$. The convergence behavior of Direct Inversion is not commented on as the method is not an iterative solver. It is also worth noting that there is a distinction in the halting conditions of GMRES_ours and GMRES_scipy. While GMRES_ours uses the norm of the residual as a stopping criterion, GMRES_scipy uses the relative error of successive residuals [13]. This does not appear to significantly affect the solution's absolute error, so the methods are compared directly without further consideration.

We start the numerical investigation with scenario 1; results are given below in Table 1.
In Table 1, we see that no iterative solver could converge on a suitable solution. This shows that a full-rank, randomly generated matrix will cause issues for any technique without pre-conditioning, that is, some pre-processing of the matrix. Furthermore, GMRES_scipy has a faster run-time than

TABLE 1. Comparison of Algorithms: Matrix Scenario 1

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	False	False	-
Time (sec)	7339	132	0.28
Ax – b	4.8×10^{-1}	6.71×10^{-2}	9.93×10^{-5}

GMRES_ours, and Direct Inversion has a very quick run-time. As we continue our experiments, this general pattern will continue; specifically, the optimized GMRES_scipy will converge quicker than GMRES_ours, and Direct Inversion will always have a fast run-time with varying degrees of success.

We continue now onto the results for matrix scenarios 2 and 3.

TABLE 2. Comparison of Algorithms: Matrix Scenario 2

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	True	True	-
Time (sec)	0.037	0.001	0.29
Ax – b	7.82×10^{-10}	1.10×10^{-12}	3.19×10^{-14}

TABLE 3. Comparison of Algorithms: Matrix Scenario 3

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	True	True	-
Time (sec)	0.01	0.002	0.30
Ax – b	1.255×10^{-13}	3.03×10^{-13}	5.5×10^{-14}

In Table 2 and 3, it seems to be the case that all three techniques very quickly find a solution of high accuracy. Moreover, the time to converge for both GMRES methods is substantially faster than for Direct Inversion. This implies that GMRES excels in an environment with fewer eigenvalues relative to the dimension of the matrix.

TABLE 4. Comparison of Algorithms: Matrix Scenario 4

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	True	True	-
Time (sec)	2394	0.001	0.26
Ax – b	3.77×10^{-12}	9.43×10^{-5}	7.82×10^{-14}

This implication can be extended when considering the results of Table 4, where behavior similar to Table 2 and Table 3 for GMRES_scipy can be seen. Here, a tight grouping of the eigenvalues near 1 provides ideal conditions for method convergence in both time and accuracy. The clear caveat is GMRES_ours, which sees a very slow run-time; however, once converged, the method elicits the most accurate solution.

In the results of Table 5, we see both versions of GMRES struggle with a very large condition number. Still, both methods converge to a more accurate solution than Direct Inversion. This indicates

TABLE 5. Comparison of Algorithms: Matrix Scenario 5

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	False	False	-
Time (sec)	6863	132	0.30
$\mathbf{Ax - b}$	1.47×10^1	1.90×10^1	1.44×10^3

TABLE 6. Comparison of Algorithms: Matrix Scenario 6

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	True	True	-
Time (sec)	26	47	0.29
$\mathbf{Ax - b}$	5.5×10^{-16}	1.17×10^{-2}	6.13×10^4

that finding the solution of linear systems with ill-conditioned matrices will always be challenging, no matter the method used.

Table 6 shows the results for the last matrix scenario. For this matrix, there is a single eigenvalue equal to zero, but **b** is in Range(**A**). This was achieved by setting **b** equal a random column of **A**. All three algorithms tackle this challenge with varying degrees of efficiency. Interestingly, in this case, GMRES_scipy converges worse than GMRES_ours. Also interesting is that Direct Inversion fails to reach an accurate solution.

Summarizing these results, it is clear that a matrix with fewer eigenvalues allows GMRES to converge more quickly. Similarly, matrices with eigenvalues of high density will also see fast convergence. However, ill-conditioned matrices or matrices with a large spectrum of eigenvalues should be avoided. In general, it would appear that for ideal systems - such as in matrix scenarios 2, 3, and 4 - GMRES_scipy and GMRES_ours are both fast and lead to good approximations. With more difficult systems, GMRES takes longer to compute a result, but that result is often of a higher quality than the direct-solving method. In our investigations, there was no case where the algorithm could not get reasonably close to a solution.

With the insights gleaned from this initial investigation in the introductory material, we move into the independent extension portion of our paper.

4. Introduction to Independent Extension

As we transition to the independent extension portion of our project, we also transition the problem to which we apply iterative solvers. Before, we were considering solutions to linear systems of the form $\mathbf{Ax} = \mathbf{b}$. Next, we will discuss the use of iterative solvers in solving the eigenvalue problem of the form $\mathbf{Av} = \lambda\mathbf{v}$. Here, λ is the eigenvalue constant, and $\mathbf{v}$ is a normalized eigenvector.

Finding the eigenvalues and eigenvectors of a matrix, like solving a linear system, is one of the most important problems in numerical analysis. In a general sense, eigenvalues and eigenvectors characterize important properties of a matrix, linear transformation, or mathematical model. They find applications in diverse fields, including physics, market analysis, and image processing [1].

Contrary to the case for linear systems, however, no scalable direct solving algorithm exists for eigenvalue computation [2]. Given an example as trivial as a 4×4 matrix, finding the eigenvalues directly would require determining the roots of a quartic equation - a tremendously complex problem.

Thus, iterative solvers for finding eigenvalues are indispensable.

To this end, we will introduce and compare two iteration schemes for the eigenvalue problem: the Power Method and the QR Iteration. We derive these methods and their extensions in Section 5. In Section 6, we apply these methods to real matrices and draw conclusions on their potentials and limitations. Additionally, we will discuss in detail some of the real-world applications of applying iterative solvers to the eigenvalue problem in Section 7.

5. Mathematical Formulation for Independent Extension

5.1. **Power Method.**

In this section, we consider the mathematical formulation for the Power Method. We also extend this algorithm slightly to derive more powerful iteration schemes.

The Power Method starts with the understanding that the eigenvectors of a full-rank $n \times n$ matrix $\mathbf{A}$ provide an orthonormal basis for the vector space $\mathbb{R}^n$. This is to say, for any vector $\mathbf{x}_0 \in \mathbb{R}^n$, $\exists c_1, c_2, \ldots, c_n \in \mathbb{R}$ such that

$$\mathbf{x}_0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_1 + \cdots + c_n\mathbf{v}_n \tag{15}$$

where $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ are the non-zero eigenvectors of $\mathbf{A}$.

Using this property, and that $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$ for $i = 1, 2, \ldots, n$, left multiplying (15) by matrix $\mathbf{A}$ gives the following:

$$\begin{aligned}\mathbf{A}\mathbf{x}_0 &= \mathbf{A}(c_1\mathbf{v}_1 + c_2\mathbf{v}_1 + \cdots + c_n\mathbf{v}_n)\\ &= c_1\mathbf{A}\mathbf{v}_1 + c_2\mathbf{A}\mathbf{v}_1 + \cdots + c_n\mathbf{A}\mathbf{v}_n\\ &= c_1\lambda_1\mathbf{v}_1 + c_2\lambda_2\mathbf{v}_2 + \cdots + c_n\lambda_n\mathbf{v}_n\end{aligned}$$

Likewise, continuing to left-multiply $\mathbf{x}_0$ by $\mathbf{A}$ 'k' times gives

$$\begin{aligned}\mathbf{A}^k\mathbf{x}_0 &= c_1\lambda_1^k\mathbf{v}_1 + c_2\lambda_2^k\mathbf{v}_1 + \cdots + c_{n-1}\lambda_{n-1}^k\mathbf{v}_{n-1} + c_n\lambda_n^k\mathbf{v}_n\\ &= \lambda_n^k\left[c_1\left(\frac{\lambda_1}{\lambda_n}\right)^k\mathbf{v}_1 + c_2\left(\frac{\lambda_2}{\lambda_n}\right)^k\mathbf{v}_2 + \cdots + c_{n-1}\left(\frac{\lambda_{n-1}}{\lambda_n}\right)^k\mathbf{v}_{n-1} + c_n\mathbf{v}_n\right].\end{aligned} \tag{16}$$

Assume that $|\lambda_1| \leq |\lambda_2| \leq \cdots \leq |\lambda_{n-1}| < |\lambda_n|$. In this case, each $\frac{\lambda_i}{\lambda_n}$ term in (16) for $i \in \{1, \ldots, n-1\}$ approaches 0 as k increases. Thus, for sufficiently high k,

$$\mathbf{A}^k\mathbf{x}_0 = c_n \cdot \lambda_n^k\mathbf{v}_\mathbf{n}.$$

Normalized, this gives

$$\frac{\mathbf{A}^k\mathbf{x}_0}{\left\|\mathbf{A}^k\mathbf{x}_0\right\|} = \frac{c_n \cdot \lambda_n^k\mathbf{v}_\mathbf{n}}{\left\|c_n \cdot \lambda_n^k\mathbf{v}_\mathbf{n}\right\|} = \mathbf{v}_n \tag{17}$$

such that the normalized version of $\mathbf{A}^k\mathbf{x}_0$ converges to the dominant eigenvalue's eigenvector. Using this, we formalize the Power Method iteration scheme by defining $\mathbf{x}_k = \frac{\mathbf{A}^k\mathbf{x}_0}{\left\|\mathbf{A}^k\mathbf{x}_0\right\|}$.

The final step is to use the derived eigenvector to determine the eigenvalue. This involves implementing the Rayleigh-Ritz Quotient. Namely, for a matrix $\mathbf{A}$ and eigenvector $\mathbf{v}$,

$$(\mathbf{v})^\mathsf{T}\mathbf{A}\mathbf{v} = (\mathbf{v})^\mathsf{T}\lambda\mathbf{v} = \lambda(\mathbf{v})^\mathsf{T}\mathbf{v}.$$

Therefore,

$$\lambda = \frac{(\mathbf{v})^\mathsf{T}\mathbf{A}\mathbf{v}}{(\mathbf{v})^\mathsf{T}\mathbf{v}}. \tag{18}$$

We combine the iteration scheme from (17) and the calculation from (18) below in Algorithm 3.

Algorithm 3 - Power Method

$\mathbf{x_0} \leftarrow$ arbitrary vector in $\mathbb{R}^n$
for $k = 1, 2, \ldots, n$ **do**
  $\mathbf{b} \leftarrow \mathbf{A}\mathbf{x}_{k-1}$
  $\mathbf{x_k} \leftarrow \frac{1}{\|\mathbf{b}\|} \cdot \mathbf{b}$
end for
$\lambda \leftarrow (\mathbf{x}_n)^\top \mathbf{A}(\mathbf{x}_n)$

Inspecting the rate of convergence, one can intuitively see that it is the $\left(\frac{\lambda_{n-1}}{\lambda_n}\right)^k$ term in (16) that converges to 0 slowest. Therefore, the method is linearly convergent with an asymptotic error constant $\mu = \frac{\lambda_{n-1}}{\lambda_n}$ as the error at each successive iteration is scaled by this factor. The more separated the dominant eigenvalue is from the rest of the eigenvalues, the faster the Power Method converges.

We can also extend this naive version of the Power Method. First, recognize that the eigenvalues of $\mathbf{A}^{-1}$ are simply the reciprocal of the eigenvalues of $\mathbf{A}$. Thus, running the power method on $\mathbf{A}^{-1}$ converges to the reciprocal of the least dominant eigenvalue - that is, $1/\lambda_1$ - given that $|\lambda_1|$ is distinct. We call this iteration scheme the Inverse Power Method, and its rate of convergence depends on the ratio of $(1/\lambda_2)/(1/\lambda_1) = \lambda_1/\lambda_2$.

Next, we introduce the concept of shifts. For a matrix $\mathbf{A}$ and a constant c

$$(\mathbf{A} - c\mathbf{I})\mathbf{v} = \mathbf{A}\mathbf{v} - c\mathbf{v} = \lambda\mathbf{v} - c\mathbf{v} = (\lambda - c)\mathbf{v}$$

which indicates if λ is an eigenvalue of $\mathbf{A}$, $\lambda - c$ is an eigenvalue of matrix $\mathbf{A} - c\mathbf{I}$. Also, the smallest eigenvalue of $\mathbf{A} - c\mathbf{I}$ is λ^* where

$$\lambda^* = \min_{i=1,\ldots,n} |\lambda_i - c|. \tag{19}$$

Therefore, by (19), applying the Inverse Power Method to $\mathbf{A} - c\mathbf{I}$ converges to the eigenvector corresponding to the eigenvalue closest to c. In this, we formulate a new technique where at each iteration k, a shift of λ_k - the eigenvalue as given by (18) of approximated eigenvector $\mathbf{x}_k$ - is applied. This algorithm, the Rayleigh-Ritz Iterative Method, is listed in Algorithm 4.

Algorithm 4 - Rayleigh-Ritz Method

$\mathbf{x_0} \leftarrow$ arbitrary normal vector in $\mathbb{R}^n$
$\lambda_0 \leftarrow (\mathbf{x}_0)^\top \mathbf{A}(\mathbf{x}_0)$
for $k = 1, 2, \ldots, n$ **do**
  $\mathbf{b} \leftarrow (\mathbf{A} - \lambda_{k-1}\mathbf{I})^{-1}\mathbf{x}_{k-1}$
  $\mathbf{x_k} \leftarrow \frac{1}{\|\mathbf{b}\|} \cdot \mathbf{b}$
  $\lambda_k \leftarrow (\mathbf{x}_k)^\top \mathbf{A}(\mathbf{x}_k)$
end for

This iteration scheme is certainly more expensive than the other methods, requiring solving a linear system at each iteration. However, its rate of convergence is significantly improved compared to the Power Method. This is because the method effectively chooses $\mathbf{x}_k$ at each iteration that minimizes $\left\|\mathbf{A}\mathbf{v}_n - \frac{(\mathbf{x}_k)^\top \mathbf{A}\mathbf{x}_k}{(\mathbf{x}_k)^\top \mathbf{x}_k}\mathbf{v}_n\right\|$, an expression which equals 0 when $\mathbf{x}_k = \mathbf{v}_n$ by (18). As it turns out, this convergence is cubic [3].

For all of these techniques, there are three main problems. First, $\mathbf{x}_0$ must be non-zero in each eigenvector direction such that $c_i \neq 0$ for any i. Also, there could be two dominant eigenvalues such that $\lambda_{n-1} = \lambda_n$, or $\lambda_1 = \lambda_2$ for the Inverse Power Method. The first issue is incredibly rare for random matrices, but the second is more important to consider. In this case, $\mathbf{x}_k$ would converge to a vector in the eigen-space of $\mathbf{v}_{n-1}$ and $\mathbf{v}_n$. The third problem of the Power Method is, of course, that it only converges to one eigenvalue. To address this limitation, we now introduce the QR Iteration.

5.2. **QR Iteration.**

In this section, we derive the QR Iteration. The QR Iteration expands the usability of the Power Method to find all eigen-pairs of a matrix $\mathbf{A}$.

In Section 5.1, the Power Method was derived by multiplying matrix $\mathbf{A}$ k times by a random initial vector $\mathbf{x}_0$. This is re-written below in (20):

$$\mathbf{A}^k\mathbf{x}_0 = \lambda_n^k \left[c_1 \left(\frac{\lambda_1}{\lambda_n}\right)^k \mathbf{v}_1 + c_2 \left(\frac{\lambda_2}{\lambda_n}\right)^k \mathbf{v}_2 + \cdots + c_{n-1} \left(\frac{\lambda_{n-1}}{\lambda_n}\right)^k \mathbf{v}_{n-1} + c_n\mathbf{v}_n \right]. \tag{20}$$

One can expand this theory for some $\mathbf{X}_0 \in \mathbb{R}^{n\times p}$ with $p \leq n$ linearly independent columns $\mathbf{x}_1^{(0)}, \mathbf{x}_2^{(0)}, \ldots, \mathbf{x}_p^{(0)}$. Define the same iteration scheme as with the Power Method such that

$$\mathbf{X}_{k+1} = \mathbf{A}\mathbf{X}_k.$$

Writing out the ith column of $\mathbf{X}_k$ as done in (20), factoring out λ_{n-p+1}^k instead of λ_n^k, gives

$$\mathbf{x}_i^{(k)} = \lambda_{n-p+1}^k \left[\sum_{j=0}^{n-p} c_{i,j} \left(\frac{\lambda_j}{\lambda_{n-p+1}}\right)^k \mathbf{v}_j + \sum_{j=n-p+1}^{n} c_{i,j} \left(\frac{\lambda_j}{\lambda_{n-p+1}}\right)^k \mathbf{v}_j \right]. \tag{21}$$

As $k \to \infty$, the first summation in (21) will decay to 0 assuming that $|\lambda_{n-p}| < |\lambda_{n-p+1}|$. In this way, it becomes evident that the columns of $\mathbf{X}_k$ will converge to a basis of eigenvectors $\mathbf{v}_{n-p+1}, \ldots, \mathbf{v}_n$.

However, by Power Method intuition, each initially linearly independent column of $\mathbf{X}_k$ will get very close to $\mathbf{v}_n$. In this manner, the basis becomes very close to linear dependence. Thus, at each step, it is necessary to re-enforce linear independence of the columns of $\mathbf{X}_k$. Here, QR-factorization is implemented to reduce the columns of $\mathbf{X}_k$ into an orthonormal basis. Accordingly, at a sufficiently large iteration k, the columns of $\mathbf{X}_k$ become an orthonormal basis for the p eigenvectors of the p largest eigenvalues of $\mathbf{A}$. In fact, this orthonormal basis is the eigenvectors themselves [14].

One can use a matrix-sized version of the Rayleigh-Ritz Quotient to find the eigenvalues from these eigenvectors. This is shown below in (22):

$$\mathbf{X}_k^\mathsf{T}\mathbf{A}\mathbf{X}_k = \Lambda = \operatorname{diag}(\lambda_{n-p+1}, \lambda_{n-p+1}, \ldots, \lambda_n). \tag{22}$$

Setting $p = n$, this method finds all n eigen-pairs of $\mathbf{A}$, assuming that $|\lambda_1| < |\lambda_2| < \cdots < |\lambda_n|$. The formalized method is included below in Algorithm 5.

This derived algorithm is called Simultaneous Iteration and can be proven to be equivalent to the more famous "Pure QR" algorithm [14]. The Pure QR iteration scheme operates as follows:

$$\mathbf{Q}_k\mathbf{R}_k = \mathbf{X}_{k-1} \tag{23}$$

$$\mathbf{X}_k = \mathbf{R}_k\mathbf{Q}_k \tag{24}$$

Solving for $\mathbf{R}_k$ in (23) gives us $(\mathbf{Q}_k)^\mathsf{T}\mathbf{X}_{k-1}$. Plugging this into (24) gives

$$\mathbf{X}_k = (\mathbf{Q}_k)^\mathsf{T}\mathbf{X}_{k-1}\mathbf{Q}_k.$$

Algorithm 5 - Simple QR Iteration (Simultaneous Iteration)

$\mathbf{X}_0 \leftarrow \mathbf{I}$ // n linearly independent columns
for $k = 1, 2, \ldots, n$ **do**
  $\mathbf{X}_k \leftarrow \mathbf{A}\mathbf{X}_{k-1}$
  $\mathbf{Q}_k\mathbf{R}_k \leftarrow \mathbf{X}_k$ // perform QR factorization
  $\mathbf{X}_k \leftarrow \mathbf{Q}_k$
end for
$\Lambda \leftarrow (\mathbf{X}_n)^\mathsf{T}\mathbf{A}(\mathbf{X}_n)$

This indicates that $\mathbf{X}_k$ is *similar* to $\mathbf{X}_{k-1}$. This is to say they have the same eigenvalues.

Interestingly enough, this Pure QR Iteration can be seen as the inverse of the Simultaneous Iteration [14]. Therefore, it is possible to shift this iteration similarly to what was done with the Inverse Power Iteration. Indeed, for some μ, define the shifted QR iteration scheme as

$$\mathbf{Q}_k\mathbf{R}_k = \mathbf{X}_{k-1} - \mu\mathbf{I} \tag{25}$$

$$\mathbf{X}_k = \mathbf{R}_k\mathbf{Q}_k + \mu\mathbf{I} \tag{26}$$

Solving for $\mathbf{R}_k$ in (25) gives us $(\mathbf{Q}_k)^\mathsf{T}(\mathbf{X}_{k-1} - \mu\mathbf{I})$ such that (26) can be written as

$$\begin{aligned}
\mathbf{X}_k &= (\mathbf{Q}_k)^\mathsf{T}(\mathbf{X}_{k-1} - \mu\mathbf{I})\mathbf{Q}_k + \mu\mathbf{I} \\
&= (\mathbf{Q}_k)^\mathsf{T}\mathbf{X}_{k-1}\mathbf{Q}_k - (\mathbf{Q}_k)^\mathsf{T}\mu\mathbf{I}\mathbf{Q}_k + \mu\mathbf{I} \\
&= (\mathbf{Q}_k)^\mathsf{T}\mathbf{X}_{k-1}\mathbf{Q}_k - \mu\mathbf{I} + \mu\mathbf{I} \\
&= (\mathbf{Q}_k)^\mathsf{T}\mathbf{X}_{k-1}\mathbf{Q}_k
\end{aligned}$$

This indicates that, even with the added shift, $\mathbf{X}_k$ is similar to $\mathbf{X}_{k-1}$. Thus, the eigenvalues that $\mathbf{X}_k$ converge to during the shifted iteration are the same as those of $\mathbf{A}$.

Also, similarly to the Inverse Power Method with a shift, the QR Iteration with a shift converges to the eigenvalue closest to the shift μ. The challenging part of the shifted QR Iteration now becomes choosing which shift μ to pick. For this choice, we offer a Lazy Shift and a Complex Shift.

Our Lazy Shift picks $\mu = a_{n,n}$: the entry of the last row and column of matrix $\mathbf{A}$. In this, the method aims to converge to the "bottom-right" eigenvalue quickly without much added complexity. However, an unavoidable consequence of this naive shift is that the iteration can *stall* in its convergence; this is to say, the iteration stops converging to the desired solution. To handle this, we only implement the naive μ as a shift until the bottom right entry has converged. At this point, we remove the original shift and proceed according to the pure QR method with a shift of 0.

Our Complex Shift is more involved. Here, we set $\mu_k = x^{(k)}_{n,n}$: the entry of the last row and column of k-th iteration matrix $\mathbf{X}_k$. Then, when this entry converges to the desired eigenvalue, we *deflate* the matrix; this involves removing the last row and column of $\mathbf{X}_k$. After this, the Complex Shift iteration is run recursively on the smaller $n - 1 \times n - 1$ sized matrix. In this way, we continue up the diagonal until all eigenvalues are found. Although this involves iterating through each diagonal element, the shift allows for finding each eigenvalue very quickly. We include the complete method below in Algorithm 6.

The discussion of the QR Iteration is continued with numerical simulations in Section 6.2.

Algorithm 6 - Complex Shift QR Iteration

$\mathbf{X}_0 \leftarrow \mathbf{A}$
$k = 0$
for $m = n, n-1, \ldots, 1$ **do**
  while $x_{m,m}^{(k)}$ not converged **do** // while eigenvalue not found
    $k = k + 1$
    $\mu_k = x_{m,m}^{(k)}$ // shift equal to last row and column entry of $\mathbf{X}_k$
    $\mathbf{Q}_k\mathbf{R}_k \leftarrow \mathbf{X}_{k-1} - \mu\mathbf{I}$ // perform QR factorization with shift
    $\mathbf{X}_k \leftarrow \mathbf{R}_k\mathbf{Q}_k + \mu\mathbf{I}$
  end while
  $\Lambda_m = x_{m,m}^{(k)}$ // set bottom right entry as m-th eigenvalue
  $\mathbf{X}_k \leftarrow \mathbf{X}_k[: m-1, : m-1]$ // deflate matrix by removing last column and row
end for

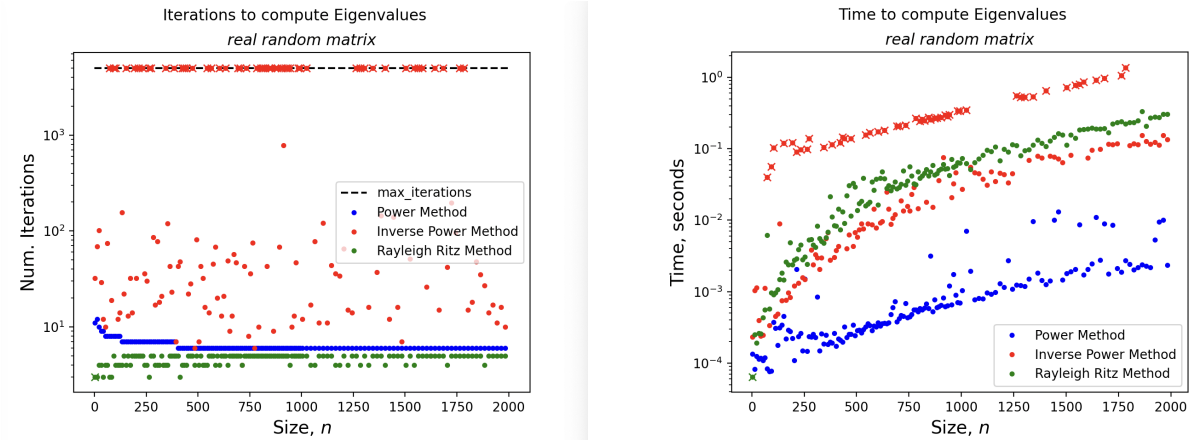
6. Numerical Work for Independent Extension

In this section, we continue our investigation into the eigenvalue problem with the iterative techniques formulated in Section 5. All results presented (except those of Figure 4 to emphasize iteration divergence) are the averages after 20 simulations. The complete code for the methods implemented and matrices created for this section will be included separately in the Appendix in Section 9.

6.1. **Power Method.**

As explained in Section 4, no direct-solving method exists for eigenvalue computation. Instead, we compare the formulated methods to each other in iterations and time for convergence. We plot with an 'x' the iterations that did not converge to an accurate solution.

First, we test the methods on random matrices and include results in Figure 4. We create the matrices using the *numpy.random.rand()* method.



(A) Iterations (B) Time

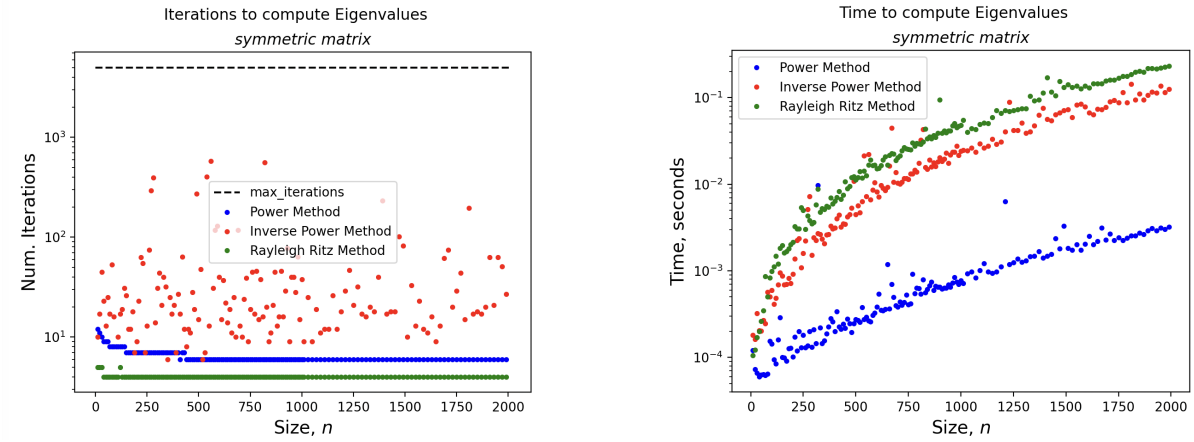
Figure 4. Power Methods, Random Matrices

In the Iteration graph, one generally sees that the Rayleigh-Ritz and Power Method converge quickest. The difference in iterations between the two is not significant; Rayleigh-Ritz converges in around 1-3

iterations, and the Power Method converges in 5-10. Thus, when we direct our attention to the time graph, we see the Power Method converging much faster than Rayleigh-Ritz. This follows from the quick run-time of each step in the Power Method, which only requires matrix multiplication. This operation is not nearly as expensive as the system-solving required each iteration of the Rayleigh-Ritz Method.

Regarding the Inverse Power Method, there is more to be said. In around half of the matrices, we see the method not converging. When investigating this, the non-converging matrices are those in which the smallest eigenvalues are complex. Here lies a limitation of the Power Method and its extensions: they only converge to real eigenvalues. As real matrices can have complex eigenvalues, this is not a possibility we can ignore.

Running the methods on symmetric matrices, however, results in convergence for all iterations as all eigenvalues are real [1]. We construct the symmetric random matrices by mirroring the upper triangular portion of a *numpy.random.rand()* matrix onto its lower triangular part. Simulation results are seen in Figure 5.



(A) Iterations

(B) Time

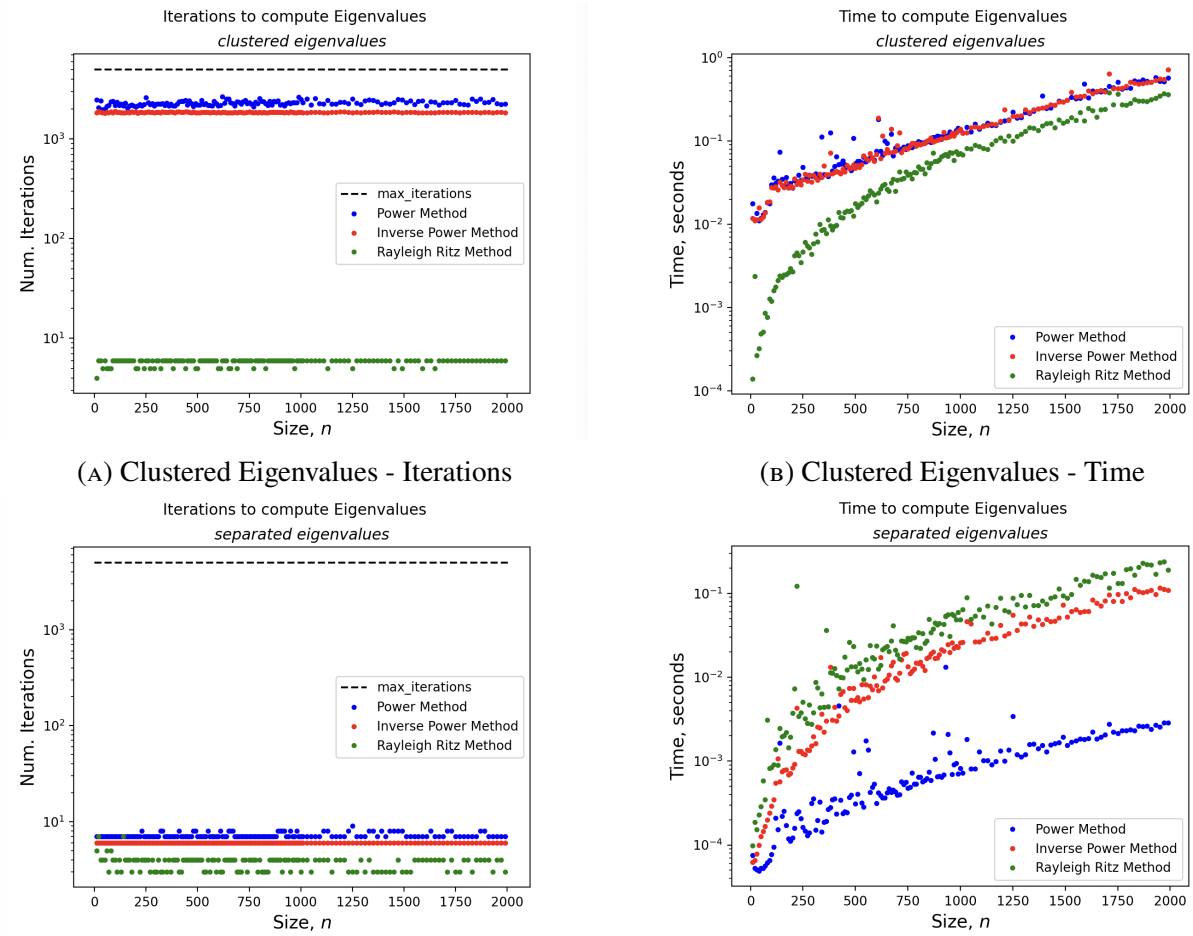
FIGURE 5. Power Methods, Symmetric Random Matrices

In these simulations, Rayleigh-Ritz again converges in slightly fewer iterations than the Power Method. However, as both methods converge in less than ten iterations, it is hard to see the theoretical "much faster" cubic convergence of Rayleigh-Ritz examined in Section 5.1.

We still see slower convergence for the Inverse Iteration than for the Power and Rayleigh-Ritz Methods. The reason for this is fairly intricate and deals with the stochastic theory of eigenvalues. For instance, Wigner's semicircle law measures the distribution of eigenvalues in symmetric random matrices [15]. The law notes that the dominant eigenvalues of these matrices tend to spread and avoid clumping. However, the phenomenon works oppositely for the smallest eigenvalues; thus, the eigenvalues closer to zero are much denser in proximity. Accordingly, methods that attempt to converge to the largest eigenvalue, such as the Power and Rayleigh-Ritz Methods, converge well as the ratio of λ_{n-1}/λ_n is relatively small. Conversely, the ratio of λ_1/λ_2 is relatively large such that Inverse Iteration converges slower.

This insight presented a question on how the distribution of eigenvalues affected the convergence of the methods. Thus, we formulate two test matrices: one with tight-clustered eigenvalues and one with more well-separated eigenvalues. The clustered eigenvalue matrix had one eigenvalue at 99, one at 101, and the rest at 100. The separated eigenvalue matrix had one eigenvalue at 1, one at

2500, and the rest at 50. For these matrices, the asymptotic error constants λ_1/λ_2 and λ_{n-1}/λ_n are practically equal such that comparisons between methods can be made objectively. We show the results in Figure 6.



(A) Clustered Eigenvalues - Iterations

(B) Clustered Eigenvalues - Time

(C) Separated Eigenvalues - Iterations

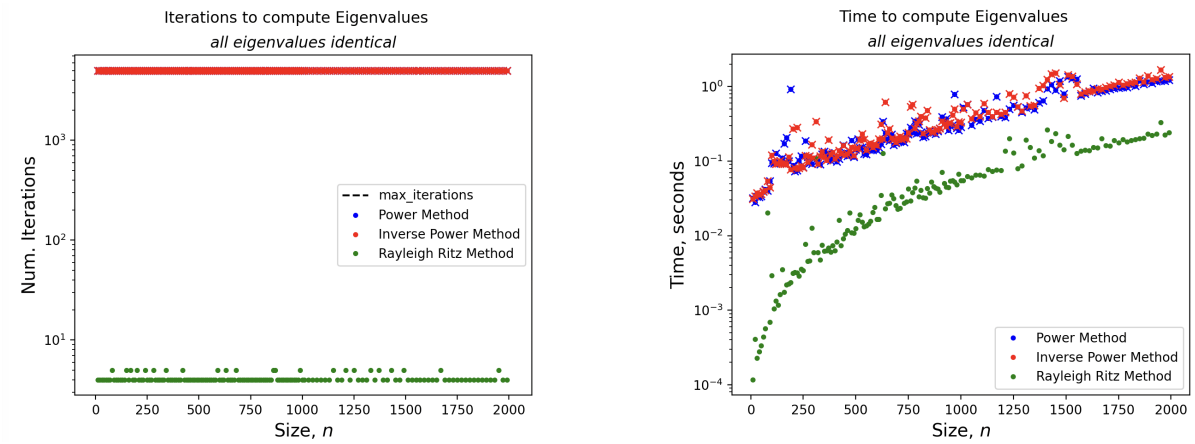
(D) Separated Eigenvalues - Time

FIGURE 6. Richardson's Iteration, Eigenvalue Distribution on PD Matrices

These matrices see very different performance than those of the figures shown previously. For the Clustered Eigenvalue matrix, the Power and Inverse Power Methods converge similarly only after many iterations. This follows from the error constant being close to one (~ 0.99); thus, the error takes many iterations to converge to zero. On the other hand, Rayleigh-Ritz Iteration utilizes shifts to bypass this slow rate of convergence. This results in Rayleigh-Ritz converging in under ten iterations for all n while the Power and Inverse Power Methods converge only after 1000+ iterations.

For the Separated Eigenvalue Matrix, we see all methods converging in under ten iterations. The Rayleigh-Ritz Method still converges in the least iterations, but the difference is negligible compared to the Clustered matrix. In the time graph, we see a repeat of the phenomenon looked at earlier: the Power Method converges much faster with more iterations. This emphasizes again the inexpensive computational cost of each iteration in the Power Method.

The last matrix type to test is the theoretical 'fail-case,' in which all eigenvalues are identical in magnitude. We build this matrix such that $|\lambda_i| = n$ for all i. Results are seen in Figure 7.



(A) Iterations (B) Time

FIGURE 7. Power Methods, Equivalent Eigenvalues

For these matrices, we see that the Power and Inverse Power Methods do not converge. The distinction of the most and least dominant eigenvalues are conditions of the Power Method and Inverse Power Method, respectively. The divergence of these methods follows. Notably, however, is that Rayleigh-Ritz Iteration can bypass this requirement due to its added shift.

To summarize the findings from all matrix types, the methods presented generally converge relatively quickly and accurately to real eigenvalues. Moreover, due to its simplicity in each iteration, the Power Method is generally the most time-effective method for finding the maximum eigenvalue of a matrix. However, if the eigenvalues are tightly clustered or equivalent, Rayleigh-Ritz converges quicker due to fewer total iterations for convergence. We now continue to results of the QR Iteration.

6.2. **QR Iteration.**

This section discusses the numerical work of applying the QR Iterations developed in 5.2 to various matrices. Before doing so, however, we discuss a few notes on the results presented.

Like with the Power Method, we compare the formulated methods to each other in the number of iterations and elapsed time. The maximum allotted number of iterations for each method was 20000. Additionally, we compare the time our QR Iterations took with the optimized algorithm implemented in the *numpy.linalg.eig()* method.

We include many of the results below in tables. For brevity, in these tables, we abbreviate Simultaneous Iteration as SI, Lazy Shifted QR as LS, and Complex Shifted QR as CS. Also, in these tables, an italicized quantity indicates that all simulations did not converge; the italicized quantity itself represents the average of both the converging and diverging simulations. A '$-$' indicates that none of the simulations conducted converged to an accurate solution within the allotted iterations.

First, we test the methods on non-symmetric random matrices created by the *numpy.random.rand()* method. We include results in Table 7 below.

This table shows complete failure for all the formulated methods when $n > 2$. This is because the eigenvalues for random matrices are not all real. In fact, after investigating the results, we see each iterative method converging to all real eigenvalues of the given matrix while failing to discover the complex ones. Thus, like with the previous Power Methods, a limitation of our QR Iterations is the

TABLE 7. QR Iteration: Random Matrix

Size	SI Iter.	SI Time	LS Iter.	LS Time	CS Iter.	CS Time	eig() time
2	11.55	2.465e-04	28.9	4.605e-04	3.85	7.365e-05	2.640e-05
5	*17022.15*	*3.07e-01*	*17001.6*	*2.494e-01*	*15805.8*	*2.471e-01*	1.195e-04
10	-	-	-	-	-	-	3.644e-04

inability to converge to non-real eigenvalues.

We try instead the methods on symmetric random matrices in Table 8. We create these matrices in the same fashion as in Section 6.1 for Figure 5.

TABLE 8. QR Iteration: Symmetric Random Matrix

Size	SI Iter.	SI Time (s)	LS Iter.	LS Time (s)	CS Iter.	CS Time (s)	eig() time (s)
10	415.65	8.446e-3	120.85	2.441e-3	17.6	3.61e-4	1.55e-04
25	3549.7	0.11252	1116.35	0.034519	45.4	1.174e-3	1.73e-4
50	9805.4	0.827592	4815.2	0.333424	88.85	3.993e-3	5.87e-4
100	*17667.2*	*11.558*	*10969.8*	*3.789*	174.2	0.0365	3.51e-3
250	-	-	*18701.4*	*56.866*	395.2	0.531	0.0316
500	-	-	-	-	778.4	4.575	0.130
1000	-	-	-	-	1510.667	51.19	0.843

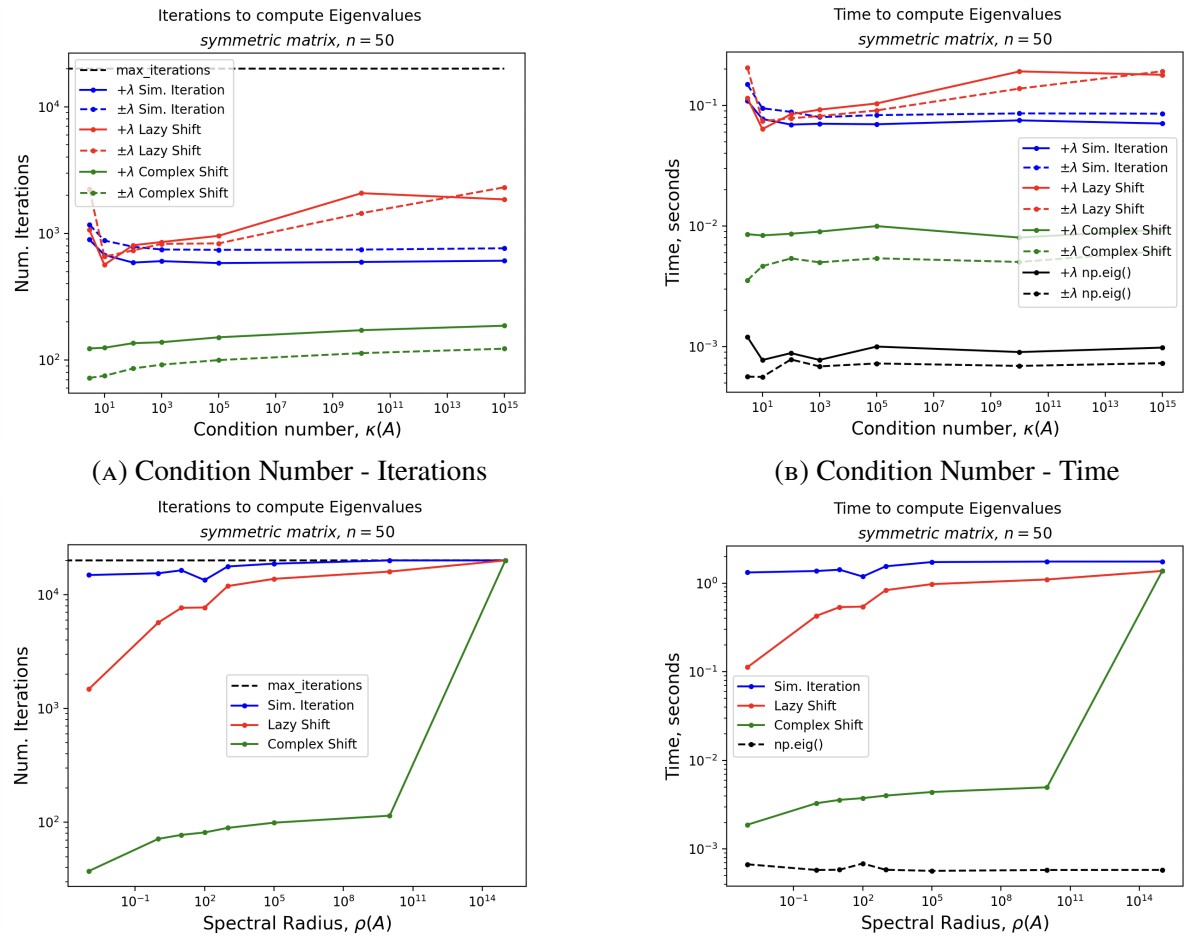
As symmetric matrices are guaranteed to have real eigenvalues, the QR Iterations can perform much better than before. However, the variance in performance between methods is quite remarkable. We see only partial convergence in 100×100 sized matrices for the Simultaneous Iteration. For the Lazy Shifted QR Method, we see partial convergence up to 250×250 sized matrices. Notably, the Complex Shift QR iteration converges for all simulations up through 1000×1000 sized matrices. Still, even this method is dominated by the numpy's optimized eig() method; eig() converges in a quick 0.8 seconds compared to the Complex Shift's 51.1 seconds for $n = 1000$ sized matrices.

All methods are also shown to experience a near-exponential increase of iterations and time required for convergence as the matrix size doubles. Even the optimized eig() method is susceptible to this phenomenon. This reflects the utilization of QR factorization at each iteration for all methods, which is a highly costly procedure of computational complexity $O(n^3)$ [1].

We now investigate the QR Iteration applied to other types of matrices. To do so, we fix the analyzed matrices as symmetric matrices of size 50×50, controlling other factors instead. This size was chosen as all methods converge accurately in Table 8 on 50×50 matrices.

We started by running simulations with the isolation of condition number and spectral radius. We create matrices for the condition number tests similar to what was done for Figure 3 in Section 3.1. However, on this base matrix, we split the test in two; namely, we run all methods separately on matrices with all positive eigenvalues and matrices with a mix of positive and negative eigenvalues. We do this to determine if the sign of the eigenvalue affects convergence. Recalling definitions from Section 2.1, the spectral radius $\rho(\mathbf{A})$ is the magnitude of the largest eigenvalue. Thus, we create the spectral radius matrices by randomly selecting 50 eigenvalues in a ball of radius r centered at 0. Simulation results are shown in Figure 8.

The condition number graphs show similar convergence patterns between methods as in earlier



(A) Condition Number - Iterations

(B) Condition Number - Time

(C) Spectral Radius - Iterations

(D) Spectral Radius - Time

FIGURE 8. QR Methods, Symmetric 50×50 Matrices

tables. There seems to be slightly improved performance for the iterative technique with better-conditioned matrices, but not unequivocally. There does not seem to be a pervasive pattern for all methods regarding eigenvalue signs either. The eig() method and Complex Shift Iteration perform better for the $\pm\lambda$ case, but only slightly. This indicates that the QR Iteration converges independently of matrix conditioning.

However, we see a general pattern for the spectral radius graphs. Namely, a higher spectral radius makes the QR Iteration perform worse. We see very poor performance for all our methods for spectral radii above order 10^{10}. We even see our first failure of Complex Shifted QR at a spectral radius of order 10^{15}. This indicates that the versions of the QR Iteration we developed perform better with more clustered eigenvalues than with very well-separated ones.

The last test matrix was the 'fail-case' based on the requirement derived in Section 5.2. Specifically, we tested the methods on matrices with duplicate eigenvalues. To do so, we construct matrices with one or three distinct eigenvalues. We also design a case where the eigenvalues are equal in magnitude but can vary in sign. We show results in Table 9.

In these simulations, we notice all methods converge very quickly for one and three distinct eigenvalues, given that they are all positive. However, if we relax the condition to allow for opposite signs, we see the Simultaneous Iteration and Lazy Shifted QR Iteration fail completely. This is because

TABLE 9. QR Iteration: Symmetric 50×50 Matrices w/ Distinct Eigenvalues

Type	SI Iter.	SI Time	LS Iter.	LS Time	CS Iter.	CS Time	eig() time
1 distinct ($+\lambda$)	2	2.717e-4	2	2.519e-4	1	1.596e-4	2.86e-4
3 distinct ($+\lambda$)	43.25	5.81367e-3	36.85	4.70212e-3	7.3	9.32e-4	3.6832e-4
1 distinct ($\pm\lambda$)	20000	1.733	20000	1.378	6.05	5.214e-04	2.934e-04
3 distinct ($\pm\lambda$)	20000	1.733	20000	1.376	14.85	9.878e-04	4.018e-04

the Simultaneous Iteration and the Lazy Shifted QR operate by updating several vectors simultaneously. When two eigenvalues have the same magnitude with different signs, the corresponding eigenvectors can point in opposite directions. Thus, the iteration cannot converge to either eigenvector and instead stalls in the middle. On the other hand, the Complex Shifted iteration only converges to one eigenvalue at a time; therefore, it has no issue converging for any case of duplicated eigenvalues.

In this section, we have shown how the extensions of the simple QR method can drastically improve performance, as in the case of the Complex Shift iteration. Still, there are many limitations to this more complicated method. Namely, it fails to converge in the case of complex eigenvalues and performs poorly for matrices with high spectral radii.

As it turns out, this "complex" method is still straightforward compared to other existing iterative techniques for eigenvalue approximation. However, it is intrinsic to the formulation of those other methods. An example of this can be seen in the Implicitly Restarted Arnoldi Iteration, or just Arnoldi Iteration, as mentioned in the Introduction and Section 2.2. Abstractly, it works by building an upper Hessenberg matrix, $\mathbf{H}_m$, that represents $\mathbf{A}$ orthogonally projected onto the Krylov subspace $\mathcal{K}_m$. A property of this Krylov $\mathbf{H}_m$ matrix is that its eigenvalues usually approximate those of $\mathbf{A}$. Thus, the Arnoldi concludes by applying a very optimized QR Iteration onto $\mathbf{H}_m$ [16]. Interestingly enough, this is the algorithm used in the *numpy.linalg.eig()* method we have utilized in our comparisons [17].

We continue the talk of eigenvalue iterative techniques with their "real-world" applications in Section 7 below.

7. Discussion of Independent Extension

In 1998, a start-up called Google was formed based on their algorithm PageRank [18]. This algorithm could find relevant search results of a web query exceptionally quickly. This algorithm was also, at its root, an eigenvalue iterative solver. We explain the workings of this iteration below.

Given the sheer scale of the Internet, checking all text and hyperlinks on every web page for every search is completely infeasible. Thus, some more manageable data structure representative of each web page's information must be derived. Let S be a connected graph that is such a representation. In this graph, each node represents a web page, and each edge between node i and node j represents a hyperlink in i that directs to j. From S, PageRank constructs an adjacency matrix M; in this matrix, $m_{i,j} = 1$ if node i has a connection to node j, and $m_{i,j} = 0$ if it does not. As it turns out, the eigenvector corresponding to the maximum eigenvalue of M effectively ranks relevant web pages in S for a given search. This vector is called the *PageRank vector*, and with this, an endlessly expanding list of web pages - of various relevance - can be generated to suit the user's needs.

Therefore, when a search is made, the PageRank algorithm works to find the largest eigenvector of M. However, this matrix can grow to be incredibly large, considering how many web pages it represents. Thus, an iterative technique is used to approximate the eigenvector quickly. The iterative

solver that PageRank utilizes is the Power Method.

While PageRank is no longer used by Google today, this ingenious algorithm positioned the company as the premier search engine at the beginning of the Internet's global proliferation. The marriage of applied mathematics and big data would set the stage for the next revolution spearheaded by Google, neural networks, which Google currently uses for its browsers [19].

Still, PageRank is not the only real-world use-case of eigenvalue iterative solvers. Due to the eigenvector's ability to reveal the long-term behavior of a matrix, they are also indispensable in aerospace stability analysis [20].

Aerospace can loosely be interpreted in terms of fluid flow dynamics. Therefore, the general idea for stability analysis is to describe and simulate an aircraft as a time-invariant system - or matrix. Increasing the granularity of this matrix is the chief way to increase the robustness of the simulation. More particles may lead to better accuracy and capturing of emergent non-linear phenomena, and an aircraft that has been tested more robustly saves development time and improves pilot safety.

After this matrix is generated, the principal step in determining the airplane's stability is the calculation of the eigenvectors. The computation of these vectors for such a granular matrix is expensive; thus, engineers must always balance system size with calculation speed. To this end, however, eigen-based iterative solvers allow the engineers to use more data and generate better results, maxing out the scale of the simulations. At companies like Boeing, these theoretical aerospace engineers use QR Iteration to determine these eigenvectors.

The problems of quickly searching an impossibly large matrix and modeling the interactions of a nearly infinite number of particle interactions may not initially seem to have much in common. The only real connection is the use of a matrix to simulate a large system. However, as we have seen, these systems' eigenvectors give essential information that can be leveraged to understand their general behavior. At the heart of both applications is the need for a fast, robust approximation to the eigenvalue problem, which both the Power Method and the QR Iteration can readily determine.

8. Conclusion

In this paper, we have considered iterative solvers for approximating matrix problems. Our exploration began by examining the application of Richardson's Iteration and GMRES to linear systems of equations. In this, we saw Richardson's Iteration converge efficiently in the case of well-conditioned positive definite matrices. For GMRES, we showed quick convergence for more general matrices, given that they were not ill-conditioned nor had an extremely large spectrum of eigenvalues.

Next, we moved beyond these linear systems to the eigenvalue problem, encompassing iterative techniques such as the Power Method and the QR Iteration. For these methods, we saw convergence only to real eigenvalues, a property guaranteed for symmetric matrices. There were also additional requirements for convergence, such as the uniqueness of the desired eigenvalue. For the Power Method, this meant the uniqueness of the dominant eigenvalue. For the QR Iteration, this meant the uniqueness of all eigenvalues. For both methods, however, we saw how an added shift could eliminate this requirement and drastically improve the rate of convergence.

Last, we examined eigenvalue solvers in industry, such as Google's PageRank. In this, we witnessed the universal influence of these iterative methods in shaping algorithms fundamental to modern technologies. Moreover, we have seen how an eigenvector acts as an abstract representation of a system; it captures essential characteristics without requiring individual element scrutinizing.

Therefore, we also saw how iterative solvers for isolating these eigenvectors prove instrumental in aerospace engineering with fluid dynamics calculations.

Throughout the development of this paper and its results, the crucial importance of iterative solvers has become increasingly pronounced. As we navigate the intricate landscape presented by the evolving world of big data, these techniques are poised to remain integral in developing and advancing computational methodologies. This accentuates the role of iterative solvers in not only enhancing the efficiency of numerical solutions, but also in providing a foundational framework for future breakthroughs in the many applications of scientific computation.

References

[1] Gilbert Strang. *Introduction to Linear Algebra, Fifth Edition*. Wellesley-Cambridge Press, 2016.

[2] Richard L. Burden and J Douglas Faires. *Numerical Analysis (10th ed.)*. Cengage Learning, 2016.

[3] Michelle Schatzman. *Numerical Analysis: A Mathematical Introduction (1st ed.)*. Oxford University Press, 2002.

[4] Lewis Fry Richardson and Richard Tetley Glazebrook. IX. the approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philosophical Transactions of the Royal Society of London*, 210(459):307–357, 1997. Publisher: Royal Society.

[5] Vittorino Pata. *Fixed Point Theorems and Applications*. Springer Cham, 2019.

[6] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. and Stat. Comput.*, 7(3):856–869, 1986. Publisher: Society for Industrial and Applied Mathematics.

[7] A. N. Krylov. On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined. *Izvestiya Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk*, 7(4):491–539, 1931.

[8] Jörg Liesen and Zdenek Strakos. *Krylov Subspace Methods: Principles and Analysis*. OUP Oxford, 2013.

[9] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951.

[10] Zhong-Zhi Bai. Motivations and realizations of krylov subspace methods for large sparse linear systems. *Journal of Computational and Applied Mathematics*, 283:71–78, 2015.

[11] Mark Embree. The tortoise and the hare restart gmres. *SIAM review*, 45(2):259–266, 2003.

[12] Wolfgang Hackbusch. *Iterative solution of large sparse systems of equations (2nd ed.)*. Springer, 2016.

[13] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[14] David S. Watkins. Understanding the qr algorithm. *SIAM Review*, 24(4):427–440, 1982.

[15] L Arnold. On wigner's semicircle law for the eigenvalues of random matrices. *Z. Wahrscheinlichkeitstheorie verw Gebiete*, 19:191–198, 1971.

[16] Danny C. Sorensen. *Implicitly restarted Arnoldi/Lanczos methods for large scale Eigenvalue calculations*. Springer, 1997.

[17] S.J. van der Walt C.R. Harris, K.J. Millman. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[18] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[19] Justin Burr. 9 ways we use ai in our products. *Google*, 2023.

[20] Giovanni Mengali. Role of eigenvectors in aircraft dynamics optimization. *Journal of guidance, control, and dynamics*, 26(2):340–346, 2003.

9. Appendix

In this section we include supplementary material, as referred to in main body of report. For *all* of the code, visit our GitHub repositiory at this link - here.

9.1. **Matrix Generation Code.**

Below is the code used for matrix generation for all parts of this paper. We distinguish matrices by iterative method type for easy access.

```python
# overarching method to generate matrices with specific eigenvalues
def generate_matrix_with_eigenvalues(n, eigenvalues):
    Q, _ = np.linalg.qr(np.random.rand(n, n))
    D = np.diag(eigenvalues)
    A = (Q @ D) @ np.linalg.inv(Q)
    return A

# Richardson's Iteration (positive definite) matrices
def createMatrix_Richardson(n, sparse=False, tri=False, cond=False):

    # sparse matrix, random or structured
    if sparse:
        eig = np.linspace(n/10, n, n) # condition number 10
        np.random.shuffle(eig)
        D = np.diag(eig)
        if tri: # create random tridiagonal matrix
            k = 2
            k1, k2 = np.random.random((k, n)), np.arange(0, k, 1)
            sp = scipy.sparse.diags(k1, k2)
            A = sp.toarray()
        else: # sparse matrix, density 0.25
            A = scipy.sparse.random(n, n, 0.25)
            A = A.A
        A = np.triu(A + D)
        A = A + np.transpose(A) - np.diag(np.diag(A))

    # explicit condition number
    elif cond:
        # define b+t s.t. condition number will be c_num after linearly spacing
        ↪  eigenvalues b/w b and t
        b = n * (1 - ((c_num - 1) / (1 + c_num)))
        t = n * (1 + (c_num - 1) / (1 + c_num))
        eig = np.linspace(b, t, n)
        np.random.shuffle(eig)
        A = generate_matrix_with_eigenvalues(n, eig)

    # random positive definite matrix with condition number 10
    else:
        eig = np.linspace(n/10, n, n) # condition number 10
        np.random.shuffle(eig)
        A = generate_matrix_with_eigenvalues(n, eig)

    # calculate alpha and define b
    eig_vals = np.sort(np.linalg.eigh(A)[0])
    eig_val1,eig_val2 = eig_vals[0].real, eig_vals[-1].real
    alpha = 2 / (eig_val1 + eig_val2)
    b = np.random.rand(n, 1)
    return A, alpha, b

# GMRES matrices
def build_mat(n, opt, density = 0.1):

    # 2000 distinct well separated eigenvalues
```

```
    if opt == 1:
        eig = np.linspace(n, 1e-5, n)
        sign = np.random.choice([-1, 1], size=n)
        eig = np.multiply(eig, sign)

    # full rank with 1 distinct eigenvalue
    if opt == 2:
        nums = [10]
        eig = np.random.choice(nums, size=n)

    # full rank with 3 distinct eigenvalues
    if opt == 3:
        nums = [10, 20, 30]
        eig = np.random.choice(nums, size=n)

    # full rank with all eigenvalues in ball of radius 1e-5 centered at 1
    if opt == 4:
        r = 1e-5
        eig = np.random.uniform(1 - r, 1 + r, size=n - 1)
        edge = np.random.choice([1, -1]) * r
        eig = np.append(eig, 1 + edge)

    # ill-conditioned ie k(A) > 1E20
    if opt == 5:
        eig = np.linspace(1e-10, 1e11, n)
        sign = np.random.choice([-1, 1], size=n)
        eig = np.multiply(eig, sign)

    # one zero eigenvalue and b in range(A)
    if opt == 6:
        eig = np.linspace(0, n - 1, n)
        sign = np.random.choice([-1, 1], size=n)
        eig = np.multiply(eig, sign)
        A = generate_matrix_with_eigenvalues(n, eig)
        b = np.reshape(A[:, np.random.randint(0, n)], (n, 1))
        return A, b

    # Create a matrix A with the chosen eigenvalues
    A = generate_matrix_with_eigenvalues(n, eig)
    b = np.random.rand(n, 1)

    return A, b

# Power Method matrices
def createMatrix_PowerMethod(n, sym=False, clustered=False, separated=False,
↪   same=False):

    # clustered eigenvalues
    if clustered:
        eig = np.array([99] + [100] * (n - 2) + [101])
        sign = np.random.choice([-1, 1], size=n)
        eig = np.multiply(eig, sign)
        A = generate_matrix_with_eigenvalues(n, eig)

    # separated eigenvalues
    elif separated:
        eig = np.array([1] + [50] * (n - 2) + [2500])
        sign = np.random.choice([-1, 1], size=n)
        eig = np.multiply(eig, sign)
        A = generate_matrix_with_eigenvalues(n, eig)

    # all identical eigenvalues
    elif same:
        eig = np.array([n] * (n))
        sign = np.random.choice([-1, 1], size=n)
```

```
        eig = np.multiply(eig, sign)
        A = generate_matrix_with_eigenvalues(n, eig)

    # random symmetric matrix
    elif sym:
        A = np.triu(np.random.random((n, n)))
        A = A + np.transpose(A) - np.diag(np.diag(A))

    # random matrix
    else:
        A = np.random.rand(n, n)

    return A

# QR Iteration matrices
def createMatrix_QR(n, neg=False, sym=False, eq_spaced=False, spectral=False,
↪  distinct=False, nums=[1, 1, 1]):
    sign = np.ones(n)

    # if both positive and negative eigenvalues
    if neg:
        sign = np.random.choice([-1, 1], size=n)

    # equally spaced from a to b, used for condition number
    if eq_spaced:
        eig = np.linspace(nums[0], nums[1], n)
        eig = np.multiply(eig, sign)
        A = generate_matrix_with_eigenvalues(n, eig)

    # 1 or 3 distinct eigenvalues
    elif distinct:
        if nums[1] == 0:
            eig = np.multiply(nums[0], np.ones(n))
        else:
            eig = np.random.choice(nums, size=n)
        eig = np.multiply(eig, sign)
        A = generate_matrix_with_eigenvalues(n, eig)

    # spectral radius r at x
    elif spectral:
        r, x = nums[0], nums[1]
        eig = np.random.uniform(x - r, x + r, size=n - 1)
        rad_explicit = np.random.choice([1, -1]) * r
        eig = np.append(eig, x + rad_explicit)
        A = generate_matrix_with_eigenvalues(n, eig)

    # random symmetric matrix
    elif sym:
        A = np.triu(np.random.random((n, n)))
        A = A + np.transpose(A) - np.diag(np.diag(A))

    # random matrix
    else:
        A = np.random.rand(n, n)

    return A
```

9.2. **Richardson's Iteration Testing Code.**

Code for testing Richardson's Iteration vs Direct Inversion:

```
n = # desired matrix size

# generate matrix and vector, pass in parameters according to desired type
```

```
A, alpha, b = createMatrix_Richardson(n)
tol = 1e-10 # some tolerance
Nmax = 5000 # maximum # of iterations

IalphA = np.eye(n) - np.multiply(alpha, A)
alphb = np.multiply(alpha, b)

# Richardson's Iteration
rt0 = time.time()
for i in range(1, Nmax):
    x1 = np.add(IalphA @ x0, alphb)
    if np.linalg.norm(x1 - x0) < tol:
        rich_iter, rich_sol = i, x1
        break
    x0 = x1
    if i == (Nmax-1): print("no solution found")
rt1 = time.time()
time_richardson = rt1 - rt0

# Direct Inversion
it0 = time.time()
inv_sol = np.linalg.inv(A) @ b
it1 = time.time()
time_inversion = it1 - it0

err_rich = np.linalg.norm(A @ rich_sol - b)
err_rich = np.linalg.norm(A @ inv_sol - b)

# compare error in err_rich and err_inv
# compare time_richardson and time_inversion
# examine rich_iter
```

9.3. **GMRES Code.**

Code for testing GMRES:

```
def gmres(A, b, x0, k, tol=1E-10):
    n = np.shape(A)[0]
    x0 = np.reshape(x0, [len(x0),])
    b = np.reshape(b, [len(b),])
    H = np.zeros([k+1, k])
    V = np.zeros([n,k+1])
    e1 = np.zeros(k+1)
    e1[0] = 1
    r_init = b - A @ x0
    Beta = np.linalg.norm(r_init)
    V[:,0] = r_init / Beta
    for j in range(1, k+1):
        for i in range(1, j+1):
            H[i-1,j-1] = (A @ V[:,j-1]) @ V[:,i-1]
        vhat = A @ V[:,j-1]
        for i in range(1, j+1):
            vhat -= H[i-1,j-1] * V[:,i-1]
        H[j,j-1] = np.linalg.norm(vhat)
        V[:, j] = vhat / H[j, j-1]
        y, _, _, _ = np.linalg.lstsq(H[:,:j], Beta*e1, rcond=None)
        x = x0 + V[:,:j] @ y
        if (np.linalg.norm(A@x - b.transpose()) < tol):
            return x, True, j
    return x, False, k
```

9.4. **Power Method Testing Code.**

Code for testing Power Method, Inverse Power Method, and Rayleigh-Ritz:

```
# define methods:
powerMethod(A, n, tol, Nmax):
    for i in range(Nmax):
        v1 = A @ v0
        v1 = (1 / np.linalg.norm(v1)) * v1
        if np.linalg.norm(abs(v1) - abs(v0)) < tol:
            iter_power = i + 1
            break
        v0 = v1
    v1t = np.transpose(v1)
    power_eigenval = ((v1t @ A) @ v1)[0][0]
    return power_eigenval, iter_power

rayleighRitz(A, n, tol, Nmax):
    v0t = np.transpose(v0)
    e0 = ((v0t @ A) @ v0)[0][0] / (v0t @ v0))[0][0]
    I = np.eye(n)
    for i in range(Nmax):
        v1 = np.linalg.solve(A - np.multiply(e0, I), v0)
        v1 = (1 / np.linalg.norm(v1)) * v1
        e1 = ((np.transpose(v1) @ A) @ v1)[0][0]
        if abs(e1) - abs(e0) < tol:
            iter_rayleigh = i + 1
            break
        e0, v0 = e1, v1
    rayleigh_eigenval = e1
    return rayleigh_eigenval, iter_rayleigh

n = # desired matrix size

# generate matrix, pass in parameters according to desired type
A = createMatrix_PowerMethod(n)
v0 = np.random.rand(n, 1) # generate random initial vector
tol = 1e-10 # some tolerance
Nmax = 5000 # maximum # of iterations

# Power Method
pt0 = time.time()
power_eigenval, iter_power = powerMethod(A, n, tol, Nmax)
pt1 = time.time()
time_power = pt1 - pt0

# Inverse Power Method
it0 = time.time()
A = np.linalg.inv(A)
ev, iter_inverse = powerMethod(A, n, tol, Nmax)
inverse_eigenval = 1 / ev
it1 = time.time()
time_inverse = it1 - it0

# Rayleigh-Ritz Method
rt0 = time.time()
rayleigh_eigenval, iter_rayleigh = rayleighRitz(A, n, tol, Nmax)
rt1 = time.time()
time_rayleigh = rt1 - rt0

# check that power_eigenval, inverse_eigenval, rayleigh_eigenval match
# compare iter_power, iter_inverse, and iter_rayleigh
# compare time_power, time_inverse, and time_rayleigh
```

9.5. **QR Iteration Testing Code.**

Code for testing QR Iteration and its extensions.

```python
# define methods:
def simultaneousQR(A, n, max_iter=20000, tol=1e-10):
    X0, e1 = np.eye(n), np.zeros(n)
    count = max_iter
    for i in range(max_iter):
        e0 = e1
        X1 = A @ X0
        Qk, Rk = np.linalg.qr(X1)
        X0 = Qk
        e1 = np.diag(X0)
        if np.all(np.abs(e1 - e0) < tol):
            count = i + 1
            break
    eigen = np.diag((np.transpose(X0) @ A) @ X0)
    return eigen, count

def lazyShiftQR(A, n, max_iter=20000, tol=1e-10):
    X0 = A.copy()
    count, m = 0, n - 1

    # bottom right
    shift = X0[-1, -1]
    shifted_mat = shift * np.eye(n)
    X0 -= shifted_mat
    while abs(X0[m, m - 1]) > tol:
        count += 1
        Q, R = np.linalg.qr(X0)
        X1 = R @ Q
        if count >= max_iter:
            return np.diag(X1), max_iter
        X0 = X1
    X0 += shifted_mat

    # rest of matrix
    e1 = np.zeros(n)
    for i in range(count, max_iter):
        count = i + 1
        e0 = e1
        Q, R = np.linalg.qr(X0)
        X1 = R @ Q
        e1 = np.diag(X1)
        if np.all(np.abs(e1 - e0) < tol):
            break
        X0 = X1
    return e1, count

def complex_shiftedQR(A, n, max_iter=20000, tol=1e-10):
    X0 = A.copy()
    count, eigs = 0, np.zeros(n)
    for m in range(n - 1, 0, -1):
        I = np.eye(m + 1)
        while abs(X0[m, m - 1]) > tol:
            count += 1
            shift_mat = X0[m, m] * I
            Q, R = np.linalg.qr(X0 - shift_mat)
            X1 = (R @ Q) + shift_mat
            if count >= max_iter:
                return np.diag(X1), max_iter
            X0 = X1
        eigs[m] = X0[m, m]
        X0 = X0[:m, :m]
    eigs[0] = X0[0, 0]
    return e, count

n = # desired matrix size
```

```
66
67  # generate matrix, pass in parameters according to desired type
68  A = createMatrix_QR(n)
69
70  # result vectors
71  iterations = np.zeros(3)
72  times = np.zeros(4)
73  error = np.zeros(3)
74
75  # numpy.linalg.eig() method
76  t0 = time.time()
77  e_real = np.sort(np.linalg.eig(A)[0])
78  times[3] = time.time() - t0
79
80  # simultaneous iteration testing
81  t0 = time.time()
82  e_sim, iterations[0] = simultaneousQR(A, n, max_iter=20000)
83  times[0] = time.time() - t0
84  error[0] = np.linalg.norm(np.sort(e_sim) - e_real)
85
86  # lazy shift testing
87  t0 = time.time()
88  e_lazy, iterations[1] = lazyShiftQR(A, n, max_iter=20000)
89  times[1] = time.time() - t0
90  error[1] = np.linalg.norm(np.sort(e_lazy) - e_real)
91
92  # complex shift testing
93  t0 = time.time()
94  e_comp, iterations[2] = complex_shiftedQR(A, n, max_iter=20000)
95  times[2] = time.time() - t0
96  error[2] = np.linalg.norm(np.sort(e_comp) - e_real)
97
98  # make sure error is low enough to indicate convergence of each method
99  # compare iterations for convergence
100 # compare time for convergence
```