Iterative Solvers

Gustav Cedergrund and Kevin Stull
Department of Applied Mathematics, University of Colorado - Boulder

Abstract:

will formulate abstract after all results are completed

1. Introduction

In the field of computational mathematics, linear systems of the form $\mathbf{Ax} = \mathbf{b}$ are ubiquitous. They remain in contexts as varied as physics, engineering, data analysis, and general optimization. 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 solving these systems is by Direct Inversion, ie $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$. The issue with this straightforward approach is that inverting a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ performs poorly as size n increases. Not only is it unstable in its accuracy, but it is also computationally expensive with a run-time of $O(n^3)$. Therefore, it is natural to search for alternative methods for solving linear systems that scale to better accommodate larger matrices.

Here is where 'Iterative Solver' methods are often implemented. These methods operate by refining an initial guess of the solution $\mathbf{x}_0$ by iterating over a sequence of successive approximations $\{\mathbf{x}_n\}$. This sequence aims to converge to the true solution over multiple such iterations such that $\{\mathbf{x}_n\} \to \mathbf{x}^*$. In principle, iterative methods require infinitely many iterations for exact convergence; however, they should give good approximations after some finite number as well. Therefore, for each type of iterative solver, we construct the iteration scheme with a halting condition based on some criteria. This criteria may be when the iteration has reached some tolerance of accuracy, when each successive iteration $\mathbf{x}_n$ 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. Specifically, we say that the method converges with order α and asymptotic error constant μ if

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

If $\alpha = 1$ and $\mu < 1$ then the iteration is said to converge linearly. If $\alpha = 2$, the iteration is quadratically convergent [1].

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 types of matrices. It was initially proposed in 1910 by Lewis Fry Richardson [2]. The technique recasts the iterative approach to a linear system problem into a root finding problem. Specifically, the root finding problem generated likens that of fixed point iteration. In this way, we repeatedly apply a function $g : \mathbb{R}^n \to \mathbb{R}^n$ with a fixed point at $\mathbf{x}^*$ (ie, $g(\mathbf{x}^*) = \mathbf{x}^*$) such that $\{g(\mathbf{x}_n)\} \to \mathbf{x}^*$. By the definition in (1) and from the Fixed Point Theorem [3], 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 [4]. The technique works by implementing an iteration scheme in relation to a Krylov subspace [5], a concept created by Alexei Krylov in 1931. 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 [6]. This is shown below in (2).

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

From this, we can generate matrix $\mathbf{K}_n$ by combining each column of $\mathcal{K}_n$ in one $n \times n$ matrix. GMRES then approximates the solution by finding a vector $\mathbf{x}_n$ in the image of the Krylov subspace matrix $\mathbf{K}_n$ with minimal residual ($\mathbf{b} - \mathbf{Ax}_n$) [4]. In this, the method becomes a minimization problem at each iteration. To find this vector $\mathbf{x}_n$, we utilize the Arnoldi method. The Arnoldi method is its own iterative technique we implement solely for the purpose of eigenvector approximation; thus, we will not cover its workings in detail here. For the complete formulation of this method, we reference the original publication by W. E. Arnoldi in 1951 [7].

This paper will first cover the mathematical derivation of both methods in section 2, and then discuss the numerical results of implementing them into practice in section 3. In sections 4 - 7, we further the talk of iterative solvers in the Independent Extension portion of the paper. Original code and lengthy proofs formulated for each section can be found in the Appendix in section 9, and will be referenced throughout the paper when needed. 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 a valuable insight to those interested in learning more about the broader field of numerical analysis and computational mathematics.

2. Mathematical Formulation for Introductory Material

We split this section to individually consider each of the two iterative methods - Richardson's Iteration and GMRES - for solving linear systems.

2.1. **Richardson's Iteration.**

In this section, we consider the mathematical formulation and numerical derivations behind the Richardson's Iteration technique. Recall from the introduction that Richardson's Iteration recasts the iterative approach to a linear system problem into a fixed point problem. First, we will investigate this recasting of the problem.

Let $\mathbf{x}^*$ represent the exact solution to the system such that $\mathbf{Ax}^* = \mathbf{b}$. Using this definition, we can rewrite the system as follows for some $\alpha \in \mathbb{R}$:

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

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

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

Therefore, 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 formalize the 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 $k + 1$, we get the following:

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

such that

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

For Richardson's Iteration to converge as a fixed point method, as discussed in the introduction, it is clear that we need to choose the correct α such that $\|(\mathbf{I} - \alpha\mathbf{A})\| < 1$. If this is the case, the error decreases at each successive step and the iterative method converges, at least, linearly.

Assume now that we require that $\mathbf{A}$ be a matrix that is either Positive Definite (PD), or Positive Semi-Definite (PSD). For a PD matrix, all eigenvalues are positive, and for a PSD matrix, all eigenvalues are greater than or equal to 0. For both types, we also guarantee that the matrix is symmetric such that $\mathbf{A} = \mathbf{A}^\mathsf{T}$ [8], in which case $\mathbf{I} - \alpha\mathbf{A} = (\mathbf{I} - \alpha\mathbf{A})^\mathsf{T}$ by extension.

Thus, as $\mathbf{I} - \alpha\mathbf{A}$ is symmetric, we are able to say that $\|(\mathbf{I} - \alpha\mathbf{A})\| = \rho(\mathbf{I} - \alpha\mathbf{A})$ where ρ indicates the spectral radius of the matrix [8]. Thus, for convergence by (5), it is enough to require that the spectral radius of $\mathbf{I} - \alpha\mathbf{A}$ is less than 1. That is, $\forall i \in \{1, 2, \ldots, n\}$, $|\lambda_i| < 1$ where each λ_i is an eigenvalue of $\mathbf{I} - \alpha\mathbf{A}$.

Now, we continue formulation with a couple of 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, we can define the eigenvalues as

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

By extension, we can thus define the eigenvalues of $\mathbf{I} - \alpha\mathbf{A}$, assuming an $\alpha \geq 0$, as

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

which means that

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

Notice now that if we let

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

we get that $\alpha \geq 0$ as assumed previous, Also, if we now plug this α back into (6), with our current definitions of λ_1, we get that:

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

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

As the $\rho(\mathbf{I} - \alpha\mathbf{A})$ is not strictly less than 1, we currently cannot guarentee that Richardson's Iteration converges. However, if we require instead in (8) that $\mathbf{A}$ is exclusively PD such that $\lambda_1 > 0$, then the equality in (9) becomes strict as 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) such that our fixed point iteration conditions are satisfied and Richardson's iteration converges.

As for the rate of convergence of the method, we can see from (5) and (8) that Richarson's Iteration converges linearly with a constant μ dependent on the ratio of λ_n/λ_1. This larger the ratio, the slower the convergence; if $\lambda_1 = \lambda_n$, then the method converges quadratically.

Important to note is that this convergence criteria based on the 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's algorithm below:

Algorithm 1 - Richardson's Iteration

$\mathbf{x_0} \leftarrow \mathbf{0}$
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.**

In this section, we consider the mathematical formulation of GMRES - a Krylov subspace method.

In general, the convergence of Krylov subspace methods is an open question. Naively, it should not be surprising that searching along the span of linear combinations of $\mathbf{A}$ and $\mathbf{b}$ should be useful in yielding a solution. Since it is a bit ambitious to expect an exact formulation of $\mathbf{x}$ to appear from this approach, a least squares solution for $\mathbf{x}$ can be found instead.

Let $\mathbf{x}_0$ be an inital guess for the solution and residual vector $\mathbf{r}$ be equal to $\mathbf{r} = \mathbf{A}\mathbf{x}_0 - \mathbf{b}$. Now, we can form the Krylov subspace of $\mathbf{A}$ and $\mathbf{r}$ to be searched by GMRES 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 factorization of the matrix representing the Krylov search space. One can apply Arnoldi iteration, defined via a recurrence relationship of the form

$$\mathbf{A}\mathbf{Q}_n = \mathbf{Q}_{n_1}\mathbf{H}_n. \tag{12}$$

Here, $\mathbf{Q}$ is the orthonormal basis representing our Krylov search space and $\mathbf{H}$ is a Hessenberg matrix. With some work, one can arrive at the relationship

$$\|\mathbf{r}_n\| = \|\beta\mathbf{e}_1 - \mathbf{H}_n\mathbf{y}_n\| \tag{13}$$

In this equation, $\mathbf{e}_1$ is the first canonical vector (i.e. $e_1 = [1, 0, 0, 0, ..., 0]$) and β is a constant equal to the norm of the initial residual $\mathbf{r}_0$. Putting it all together, this gives a linear system in the form of $\mathbf{A}\mathbf{x} = \mathbf{b}$ to be minimized as a least squares problem.

$$\mathbf{r}_n = \mathbf{H}_n\mathbf{y}_n - \beta\mathbf{e}_1 \tag{14}$$

This can be realized with the following algorithm:

Algorithm 2 - GMRES

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

When introducing the psuedo-code for GMRES, special care must be given to the dimensions of the matrices used to arrive at the residual minimizing vector $\mathbf{y}$. As the matrix $\mathbf{H} \in \mathbb{R}^{(k+1)\times k}$ and $\mathbf{Q} \in \mathbb{R}^{(n)\times(k+1)}$, the solution to (13) can never be exact, unless $n = k$.

Additionally, the algorithm has two termination conditions: either a maximum number of iterations is reached or $\|\mathbf{r}_0\| <$ tol is achieved. Currently, the algorithm does not use $k > n$; instead, once $k = n$ is found without minimizing the residual within the specified tolerance, $\mathbf{x}_k$ is returned and used as an updated guess in a new GMRES function call. This technique, called restarted GMRES [9], limits memory requirements while allowing for faster convergence of the algorithm. The number of times this restarting can occur is dictated by an epochs parameter. We define an epoch as the number of times the GMRES function is called with an updated initial guess for $\mathbf{x_0}$. The default behavior is expanded on in the numerical results section 3.2.

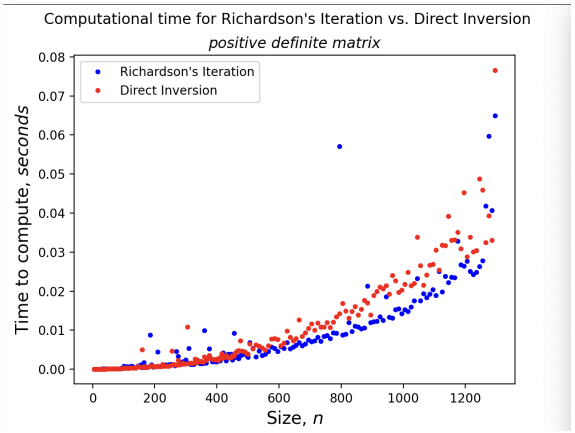
3. Numerical Work for Introductory Material

3.1. **Richardson's Iteration.**

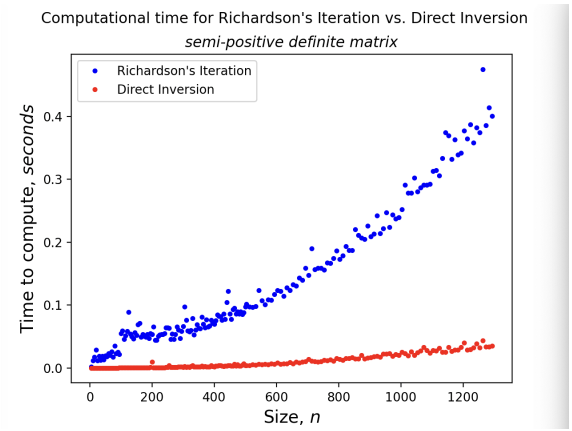
In this section, we continue our investigation into Richardson's Iteration by delving into numerical results. For our comparisons in method performance, we will compare the iteration technique with solving by Direct Inversion for various types of matrices. For all simulations, we assume that the maximum 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. Also, important to note is 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 would; however, as it is the method most commonly used in practice, we also use this method for our comparisons. The code for the methods used for this section will be included separately in the Appendix in section 9.

Recalling our assumption from section 2.1 for the matrix $\mathbf{A}$ to be either PD or PSD, we first test the method on randomized matrices of both types. Results can be seen in Figure 1.

Clear to see in these graphs is the difference of Richardson's Iteration convergence depending on matrix type. These results seem to confirm our findings in section 2.1 that we require matrix $\mathbf{A}$ to be Positive Definite. For these Positive Definite matrices, we see that generally Richardson's converges



(A) Positive Definite

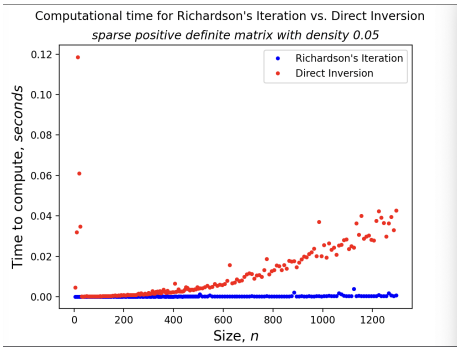


(B) Positive Semi-Definite

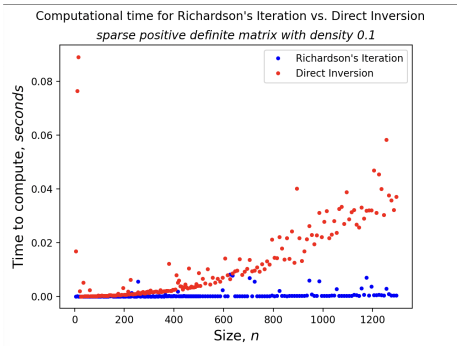
FIGURE 1. Richardson's Iteration, Random Matrices

quicker than direct inversion, with a difference that increases as size n increases.

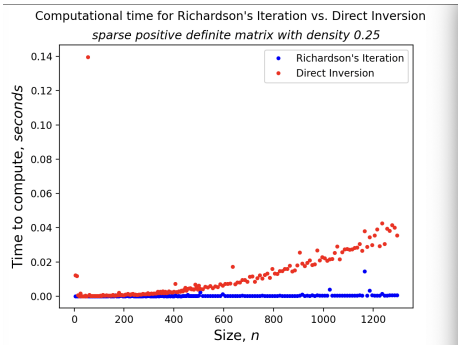
The real power of Richardson's iteration, however, lies in its application to Sparse matrices. Sparse matrices are matrices in which most of the elements are 0. We use the term 'density' to characterize just how sparse the sparse matrix is. For example, a sparse matrix with density 0.1 would see only 10% of its entries filled, with 90% being left blank and equal to 0. We show the performance of



(A) Density 0.05



(B) Density 0.1



(C) Density 0.25

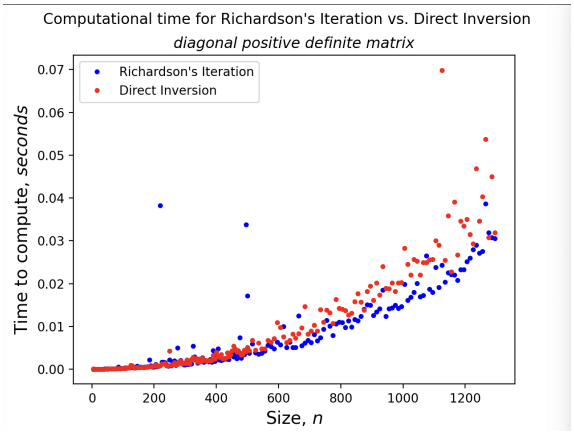
FIGURE 2. Richardson's Iteration, Sparse PD matrices

Richardson's Iteration on Sparse matrices with three different densities in Figure 2.

In these diagrams we see how well Richardson's Iteration performs for sparse matrices compared to Direct Inversion. As for comparisons of convergence with different densities, there is not much to notice. It seems that as long as the matrix itself is sparse enough, rapid convergence will follow.

The reason for this is actually quite interesting as it is not necessarily attached to the rate of convergence definition explored previously. Sparse matrices typically have $O(n)$ non-zero entries. By extension, this means that matrix-vector multiplication becomes an $O(n)$ operation [10]. Therefore, as Richardson's Iteration is entirely built on matrix-vector multiplication, each iteration simply takes less time than for non-sparse matrices. Direct Inversion methods, however, still are constricted to the same $O(n^3)$ approach. Thus, we converge to the solution in much less time than when compared to Direct Inversion.

Another type of matrix - Diagonal matrices - are by definition sparse as all non-diagonal elements are zero; however, their introduction of a pattern to this sparsity makes the inverse very easy to compute. In fact, all that is required is taking the reciprocal of every diagonal element. We still include the comparison of a normal diagonal matrix in Figure 3 to show the performance of Richardson's.



(A) Diagonal

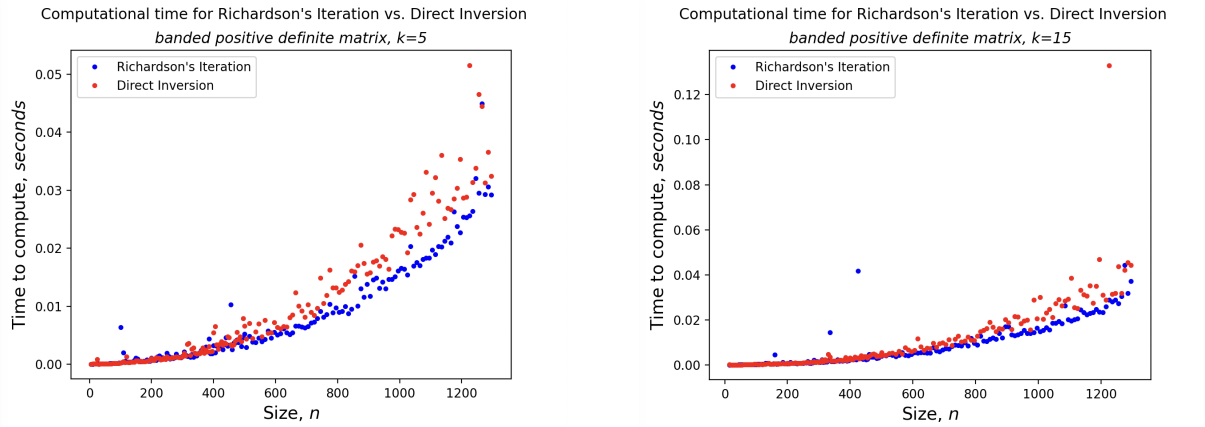
FIGURE 3. Richardson's Iteration, Diagonal PD matrices

In this, we can see that even with the immense optimization of Direct Inversion with a diagonal matrix, Richardson's Iteration is still comparable in convergence with a diagonal matrix being sparse. We can extend this to show that if this diagonal is made wider, such as in the case of the Banded matrix, that the same general pattern occurs as both methods become slightly less optimized. In Figure 4, we include the convergence for such a matrix, where integer k represents how many diagonals above and below the main diagonal are filled.

Thus, in many cases, we can conclude Richardson's Iteration to be worthwhile given that the maximum and minimum eigenvalues are computed and the matrix is Positive Definite. If in addition the matrix is sparse, then the iterative method is strongly preferred to direct solving techniques.

3.2. **GMRES Iteration.**

In this section, we explore of some of the numerical results obtained from various experiments with



(A) Banded, k=5

(B) Banded, k=15

FIGURE 4. Richardson's Iteration, Banded matrices

the GMRES algorithm. In order to test GMRES in a variety of different situations, five different scenarios were investigated:

1) A matrix with 2,000 distinct well separated eigenvalues.

2) A full rank matrix with 3 distinct eigenvalues

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

4) A matrix whose condition number is larger than $1 * 10^{20}$.

5) A matrix who has one zero eigenvalue with b in the range(A).

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

1) GMRES_ours

2) GMRES_scipy

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

Because of the large variability in convergence times between algorithms and matrix conditions, the time taken is binned into three categories; these are: fast run-times (t < 1 second), medium run-times (t < 15 minutes), and slow run-times (t > 1 hour). Finally, the resulting solution returned by each algorithm's quality (absolute error) can be quantified by taking the norm of the residual that is: $\|\mathbf{r}\| = \|\mathbf{Ax} - \mathbf{b}\|$, which is reported as well. Consider scenario 1, given below in Table 1. Notice in Table 1 we see that no algorithm was able to converge on a suitable solution. This shows

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	False	False	False
Speed	slow	med	fast
$\mathbf{Ax} - \mathbf{b}$	1.28×10^{-1}	8.16×10^{2}	9.93×10^{-3}

TABLE 1. Comparison of Algorithms: Matrix 1

that a dense, randomly generated matrix will cause issues for any technique without some form of pre-conditioning. Further, a characteristic pattern will begin to emerge in which GMRES_ours will

have a slow run time, the GMRES_scipy will have a medium run time, and the direct matrix inversion will have a fast run time. Going forward, the matrix inversion will always have a fast run time with varying degrees of success. Next, attention should be directed to the absolute error column. Here we can see that in some situations, an algorithm will converge to the tolerance provided at the input; however, the absolute error can vary depending on the situation. This highlights a key difference between GMRES_ours and GMRES_scipy methods. The former uses absolute convergence error and the latter relative convergence error. This likely accounts for the significant disparity in average run-times between the two algorithms.

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	True	True	True
Speed	fast	fast	fast
$\mathbf{Ax - b}$	1.76×10^{-13}	8.17×10^{2}	5.5×10^{-14}

TABLE 2. Comparison of Algorithms: Matrix 2

Moving on to Table 2, it seems to be the case that all three techniques quickly converge on a valid solution. The only disparity being the larger absolute error of GMRES_scipy. This implies that iterative methods excel in an environments with smaller amount of eigenvalues relative to the dimension of the matrix. This intuition can be extended when considering the results of Table 3, where behavior similar to Table 2 can seen. The tight grouping of the eigenvalues near 1 provides ideal conditions for the convergence of iterative methods.

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	True	False	True
Speed	slow	med	fast
$\mathbf{Ax - b}$	1.76×10^{-12}	8.33×10^{2}	2.89×10^{-11}

TABLE 3. Comparison of Algorithms: Matrix 3

The main curiosity to be found in Table 3 is the fact that GMRES_ours did converge while GMRES_scipy did not. In Table 2, it appeared that a few unique eigenvalues was an ideal case. Thus, one might expect a spectrum of eigenvalues grouped tightly around a single point to do well. However, it appears that if the eigenvalues are all different, it can still cause enough problems to hinder the convergence of the iterative techniques.

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	False	False	False
Speed	slow	med	fast
$\mathbf{Ax - b}$	1.51×10^{1}	2.05×10^{62}	7.06×10^{3}

TABLE 4. Comparison of Algorithms: Matrix 4

Another interesting example of GMRES_ours potentially outperforming GMRES_scipy can be observed in Table 4. While neither algorithm converged, the absolute error produced by GMRES_ours was the lowest of the three. Also, it seems that GMRES_scipy was divergent as the value of its absolute error was 2.05×10^{62}. This is likely due to a difference in how often the minimizing

vector $\mathbf{y}_k$ is calculated. This trade-off between the speed of the algorithm and the accuracy of its result is paradigmatic in our investigation of these techniques. It would appear that for easy cases, GMRES_scipy and GMRES_ours are both fast and lead to good approximations. In hard cases, our algorithm takes a longer time to compute a result, but that result is often of a higher quality than the imported method. In our investigations, there was not a case where the algorithm could not get reasonably close to a solution. While both algorithms did use resetting GMRES, the internal parameter of scipy's function was set to a default value of 20 while ours was set to $n = 2000$ by default. It is possible that raising this parameter to match ours could lead to a similar results. The authors of GMRES_scipy mention such a possibility in the documentation of their code.

Algorithm	GMRES_ours	GMRES_scipy	numpy.linalg.inv
Converged	True	True	False
Speed	med	fast	fast
$\mathbf{Ax - b}$	0.33×10^{-11}	5.81×10^{-16}	3.02×10^{-5}

TABLE 5. Comparison of Algorithms: Matrix 5

The last case investigated is found in Table 5, this is the interesting case where there is a single eigenvalue equal to zero, but $\mathbf{b}$ is in Range($\mathbf{A}$). This was achieved in a very naive method, $\mathbf{b}$ was simply set equal to the second column of $\mathbf{A}$. As one would expect, all three algorithms tackle this challenge with ease quite efficiently. It is interesting to see that this is the only case where GMRES_scipy had a lower absolute error than GMRES_ours. That level of accuracy suggests that their solution is correct to machine precision as one would expect when the solution to the system is just the second canonical vector. In this somewhat contrived case, direct inversion fails to achieve the desired precision. Considering the fact that the algorithm always runs in less than a second (fast), it is possible that using higher memory floating point numbers for the calculations could give it a better chance of out-performing the GMRES based algorithms in regards to problems that require more computational time.

Taking the previous insights into consideration, certain criteria can be inferred when designing a suitable pre-conditioner for the GMRES algorithm. It is clear that GMRES performs better when exposed to sparse matrices, as opposed to dense matrices, as it is an iterative technique. Further, a matrix with fewer eigenvalues seems to allow it to converge more quickly. What should be avoided are ill-conditioned matrices or matrices with a large spectrum of eigenvalues. With the insights gleaned from this initial investigation, we move into independent extension portion of our study.

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 a linear system 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 a constant and $\mathbf{v}$ is an normalized vector.

Finding the eigenvalues of a matrix, like solving a linear system, is an intrinsic challenge for a variety of domains and is one of the most important problems in numerical analysis. In a general sense, eigenvalues characterize important properties of a matrix, linear transformation, or mathematical model. Fields ranging from physics & geology to market analysis to image processing all utilize eigenvalues for some purpose [8].

Contrary to the case for Linear Systems, however, there exists no scale-able direct solving algorithm for eigenvalue computation. 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 in itself. Thus, iterative solvers are indispensable for larger, more general matrices.

To this end, we will introduce and compare two iteration schemes for the eigenvalue problem: the Power Method and the QR algorithm. In section 5, we derive these methods and their extensions. In section 6, we apply these methods onto 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

As referenced in the introduction for the independent extension, we will consider two iterative methods for solving eigenvalue problems: the Power Method and the QR algorithm. Below, we formulate each.

5.1. **Power Method.**

In this section, we will consider the mathematical formulation for the power method. We will also extend the power method slightly to derive more powerful iteration schemes.

5.1.1. *Derivation.* The Power Method starts with the understanding the the eigenvectors for an $n \times n$ matrix $\mathbf{A}$ provide an orthonormal basis for the vector space $\mathbb{R}^n$. This is to say that 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 $\mathbf{A}$.

Now let's see what happens if we left-multiply (15) by matrix $\mathbf{A}$, utilizing the property that $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$, with λ_i being the corresponding eigenvalue of eigenvector $\mathbf{v}_i$.

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

Notice that if we continue to left-multiply $\mathbf{x}_0$ by $\mathbf{A}$ 'k' times, we get that

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

If we assume that $|\lambda_1| \leq |\lambda_2| \leq \cdots \leq |\lambda_{n-1}| < |\lambda_n|$, then each $\frac{\lambda_i}{\lambda_n}$ term in (16) for $i \in \{1, \ldots, n-1\}$ goes to 0 as k increases. Thus, for sufficiently high k, we say that

$$\mathbf{A}^k\mathbf{x}_0 = c_n \cdot \lambda_n^k\mathbf{v_n}$$

which we can normalize to get

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

such that we converge to the dominant eigenvalue's eigenvector. Using this, we formalize the Power Method iteration scheme by defining $\mathbf{x}_k$ as the normalized version of $\mathbf{A}^k\mathbf{x}_0$ at the k-th iteration.

The final step now is to use the eigenvector derived after a sufficient number of iterations to determine the eigenvalue. To do this, we use the Rayleigh Quotient. For a matrix $\mathbf{A}$ and eigenvector $\mathbf{v}$, knowing that $\mathbf{Av} = \lambda\mathbf{v}$

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

such that

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

And as for sufficiently high iteration, n, the vector $\mathbf{x}_k$ converges to a normalized eigenvector, our final iteration schema can be summarized by combining (17) and (18) in the following algorithm:

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
$\lambda \leftarrow (\mathbf{x}_n)^\mathsf{T}\mathbf{A}(\mathbf{x}_n)$

Inspecting the rate of convergence, we 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 as the error is scaled by a factor of at most $\frac{\lambda_{n-1}}{\lambda_n}$ at each iteration. The more separated the dominant eigenvalue is from the rest of the eigenvalues, the faster the convergence.

5.1.2. *Extensions.* We can extend this naive version of the Power Method in two ways.

First we can recognize that the eigenvalues of $\mathbf{A}^{-1}$ are simply the reciprocal of the eigenvalues of $\mathbf{A}$. Thus, if we run the power method on $\mathbf{A}^{-1}$, we converge to the reciprocal of the least dominant eigenvalue, that is, $1/\lambda_1$ (given that $|\lambda_1|$ is distinct from $|\lambda_2|$). We call this iteration scheme the Inverse Power Method.

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

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

which indicates if λ is an eigenvalue of $\mathbf{A}$, $\lambda - c$ is an eigenvalue of matrix $\mathbf{A} - c\mathbf{I}$. Also, we can see that 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), if we apply the Inverse Power Method to $\mathbf{A} - c\mathbf{I}$, we converge to the eigenvector corresponding to the eigenvalue closest to c.

In this, we can formulate a new technique where at each iteration, we apply a shift of λ_k, the current approximated eigenvalue of the approximated eigenvector $\mathbf{x}_k$. We formalize this algorithm, called Rayleigh Quotient Iteration, below

While this is more expensive as it requires solving a system of equations at each iteration, the rate of convergence of this method is significantly improved compared to the initial power method. As such, it requires less total iterations. The reason for this is because we are effectively choosing $\mathbf{x}_k$

Algorithm 4 - Rayleigh Quotient Method

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

at each iteration that minimizes $\left\|\mathbf{A}\mathbf{v}_n - \frac{(\mathbf{x}_k)^\mathsf{T}\mathbf{A}\mathbf{x}_k}{(\mathbf{x}_k)^\mathsf{T}\mathbf{x}_k}\mathbf{v}_n\right\|$ - an expression which we know equals 0 when $\mathbf{x}_k = \mathbf{v}_n$ by (18). As it turns out, this convergence is cubic [1].

5.1.3. *Drawbacks.* There are two problems with this technique. For one, $\mathbf{x}_0$ must be non-zero in each direction such that $c_i \neq 0$ for any i. Also, there could be two dominant eigenvalues such that $\lambda_{n-1} = \lambda_n$. The first issue is incredibly rare for random matrices, but the second is more important to consider. In this case, our $\mathbf{x}_k$ would converge to a vector in the eigenspace of $\mathbf{v}_{n-1}$ and $\mathbf{v}_n$. Other than those, the main limitation of the power method is, of course, that it only converges to one eigenvalue. Thus, we introduce now the QR algorithm for finding eigenvalues.

5.2. **QR Algorithm.**
In this section we formulate the QR algorithm. The QR algorithm expands the usability of the Power Method to find all eigen-pairs of a matrix $\mathbf{A}$.
This derived algorithm is called simultaneous iteration and is one of the identical versions of the QR algorithm. We also discuss performance of the method in section 6.2.

6. Numerical Work for Independent Extension

6.1. **Power Method.**
In this section, we continue our investigation into the Power Method technique by delving into numerical results of it and its extensions. As explained in section 4, there exists no direct solving method for eigenvalue computation. Instead, we compare each method to each other, in number of iterations and in elapsed time. The code for the methods used for this section will be included separately in the Appendix in section 9.

6.2. **QR Algorithm.** In this section, we discuss numerical results of applying the QR algorithm to various matrices. The code for the methods used for this section will be included separately in the Appendix in section 9.

7. Discussion of Independent Extension

The methods explored in the independent extension are really quite simple compared to other existing iterative techniques; still, they are intrinsic to the formulation of those other methods. An example of this can be seen in the Arnoldi iteration - as was mentioned in the Introduction - for eigenvalue approximation. 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 $\mathbf{H}_m$ is that its eigenvalues usually

approximate those of $\mathbf{A}$. Thus, the Arnoldi concludes by simply applying a slightly optimized QR algorithm on $\mathbf{H}_m$ to approximate the m largest eigenvalues of $\mathbf{A}$; applying the method on a Hessenberg matrix rather than $\mathbf{A}$ directly allows for much faster convergence. Interestingly enough, this is actually the method Python's numpy library utilizes in its '*numpy.linalg.eig*' method.
plan to include connection to real world eigenvalue iterative techniques here, like for Google's Page Rank Algorithm

8. Conclusion

will create section after completion of independent extension

References

[1] Michelle Schatzman. *Numerical Analysis: A Mathematical Introduction (1st ed.)*. Oxford University Press, 2002.
[2] 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. 210(459):307–357. Publisher: Royal Society.
[3] Vittorino Pata. *Fixed Point Theorems and Applications*. Springer Cham.
[4] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. 7(3):856–869. Publisher: Society for Industrial and Applied Mathematics.
[5] 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.
[6] Jörg Liesen and Zdenek Strakos. *Krylov Subspace Methods: Principles and Analysis*. OUP Oxford.
[7] 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.
[8] Gilbert Strang. *Introduction to Linear Algebra, Fifth Edition*. Wellesley-Cambridge Press, 2016.
[9] Mark Embree. The tortoise and the hare restart gmres. *SIAM review*, 45(2):259–266, 2003.
[10] Wolfgang Hackbusch. *Iterative solution of large sparse systems of equations (2nd ed.)*. Springer, 2016.

9. Appendix

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

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

Code for testing Richardson's Iteration vs Direct Inversion:

```
# define size 'n'
# generate A matrix and b vector of size 'n' based on specific conditions
# define alpha according to formula in section 2.1
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(np.matmul(IalphA, x0), alphb)
    if np.linalg.norm(x1 - x0) < tol:
        print("converged after", i, "iterations")
        rich_sol = x1
        break
    x0 = x1
    if i == (Nmax-1): print("no solution found")
rt1 = time.time()
time_richardson = rt1 - rt0

# Direct Inversion
it0 = time.time()
ex_sol = np.matmul(np.linalg.inv(A), b)
it1 = time.time()
time_inversion = it1 - it0

# check that rich_sol and ex_sol converge to same vector
# compare time_richardson and time_inversion
```

9.2. **Power Method Testing Code.**
Code for testing Power Method, Inverse Power Method, and Rayleigh Quotient:

```
# define size 'n'
A = np.random.rand(n, n) # generate random A matrix
v0 = np.random.rand(n, 1) # generate random initial vector

tol = 1e-10 # some tolerance
Nmax = 10000 # maximum # of iterations


# Power Method
pt0 = time.time()
for i in range(Nmax):
    v1 = np.matmul(A, v0)
    v1 = np.multiply(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 = np.dot(np.matmul(v1t, A), v1)[0][0]
pt1 = time.time()
time_power = pt1 - pt0

# Inverse Power Method
it0 = time.time()
A = np.linalg.inv(A)
for i in range(Nmax):
```

```
    v1 = np.matmul(A, v0)
    v1 = np.multiply(1 / np.linalg.norm(v1), v1)
    if np.linalg.norm(abs(v1) - abs(v0)) < tol:
        iter_inverse = i + 1
        break
    v0 = v1
v1t = np.transpose(v1)
ev = np.dot(np.matmul(v1t, A), v1)[0][0]
inverse_eigenval = 1 / ev
it1 = time.time()
time_inverse = it1 - it0

# Rayleigh Quotient Method
rt0 = time.time()
e0 = np.dot(np.matmul(np.transpose(v0), A), v0) / np.dot(np.transpose(v0),
↪  v0))[0][0]
for i in range(Nmax):
    v1 = np.linalg.solve(A - np.multiply(e0, np.eye(n)), v0)
    v1 = np.multiply(1 / np.linalg.norm(v1), v1)
    e1 = np.dot(np.matmul(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
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
```

9.3. **GMRES Code.**

```
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. **Generate Matrix Code.**

```
def build_mat(n, opt, density = 0.1):
    R = scipy.sparse.random(n, n, density)
    R = R.toarray()
    b = np.random.rand(n, 1)
    Q, _ = qr(R, mode='economic')
    if opt == 0:
        diagonal_entries = np.linspace(n, 1e-10, n)
    if opt == 1:
        diagonal_entries = []
        for i in range(n):
            diagonal_entries.append(np.random.randint(1, 4))
    if opt == 2:
        diagonal_entries = np.linspace(-1E-5, 1E-5, n)
    if opt == 3:
        diagonal_entries = []
        for i in range(n):
            diagonal_entries.append(np.random.randint(3))
    if opt == 4:
        R = np.random.rand(n, n)
        Q, _ = qr(R, mode='economic')
        diagonal_entries = np.linspace(0, 1E-5, n)
        D = np.diag(diagonal_entries)
        A = np.dot(Q.T, np.dot(D, Q))
        b = A[:,1]
        return A, b
    D = np.diag(diagonal_entries)
    A = np.dot(Q.T, np.dot(D, Q))
    return A, b
```

9.5. **QR Algorithm Testing Code.**