2024

NEW METHOD FOR COMPUTING THE EUCLIDEAN CONDITION NUMBER WITH RIM-C

Cody McCarthy
Michigan Technological University, cmccarth@mtu.edu

Recommended Citation
McCarthy, Cody, "NEW METHOD FOR COMPUTING THE EUCLIDEAN CONDITION NUMBER WITH RIM-C", Open Access Master's Report, Michigan Technological University, 2024.
https://doi.org/10.37099/mtu.dc.etdr/1785

NEW METHOD FOR COMPUTING THE EUCLIDEAN CONDITION NUMBER WITH RIM-C

By

Cody McCarthy

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Mathematical Sciences

MICHIGAN TECHNOLOGICAL UNIVERSITY

2024

This report has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Mathematical Sciences.

Department of Mathematical Sciences

Thesis Advisor:	*Dr. Jiguang Sun*
Committee Member:	*Dr. Allan Struthers*
Committee Member:	*Dr. Zhengfu Xu*
Department Chair:	*Dr. Jiguang Sun*

Contents

List of Figures

List of Tables

List of Abbreviations

LINPACK	Linear Algebra Software Package
RIM	Recursive Integral Method
RIM-C	Recursive Integral Method with Cayley Transformations

Abstract

The condition number, being critical to solving linear systems, has many important applications. Specifically for robust control analysis, the Euclidean norm has widespread use over the 1-norm and ∞-norm such as determining a control system's stability to uncertainty [1]. Much work has been done with estimating the Euclidean condition number, but current algorithms for computing said condition number, with large matrices, tend to run slow as well as requiring a large amount of computational resources. This report seeks to provide a more time efficient algorithm that utilizes MATLAB's *eigs*, *svds*, and *normest* commands as well as the recently developed RIM-C that can be ran with off-the-shelf software with commercially available hardware.

Chapter 1

Introduction

Let $\{E_k\}$ be a sequence of measurable sets in $\mathbb{R}$, and $\mu\left(\bigcup_{k=1}^{\infty} E_k\right) < \infty$. If $\inf\limits_{k\geq 1}\left(\mu\left(E_k\right)\right) = \alpha > 0$. Show that $\mu\left(\limsup\limits_{k\to\infty}\left(E_k\right)\right) \geq \alpha$.

Consider a general matrix $A \in \mathbb{C}^{n\times n}$ that takes on the map $A : \mathbb{C}^n \to \mathbb{C}^n$ under the Euclidean norm $\|\cdot\|_2$. Consider vectors $\overrightarrow{x} \in \mathbb{C}^n$ and $\overrightarrow{b} \in \mathbb{C}^n$ such that $A\overrightarrow{x} = \overrightarrow{b}$ and a small perturbation $\overrightarrow{b_\delta}$ such that the difference is relatively small, or:

$$\frac{\|\overrightarrow{b} - \overrightarrow{b_\delta}\|_2}{\|\overrightarrow{b}\|_2} \approx 0$$

If we define $\overrightarrow{x_\delta}$ such that $A\overrightarrow{x_\delta} = \overrightarrow{b_\delta}$, we additionally have:

$$\frac{\|A\overrightarrow{x} - A\overrightarrow{x_\delta}\|_2}{\|A\overrightarrow{x}\|_2} \approx 0$$

Abstractly, the relative condition of a matrix A (and all future mentions of the condition of A will be assumed relative) informs us how sensitive the solution $\overrightarrow{x}$ is to small perturbations in $\overrightarrow{b}$. In other words, how large the difference is from $\overrightarrow{x}$ to $\overrightarrow{x_\delta}$ with perturbtation to $\overrightarrow{b}$ in the form $\overrightarrow{b_\delta}$. A is called well-conditioned if the following holds:

$$\frac{\|\overrightarrow{x} - \overrightarrow{x_\delta}\|_2}{\|\overrightarrow{x}\|_2} \approx 0$$

If A is ill-conditioned, we conversely expect the quantity to be "further" from 0, or:

$$\frac{\|\overrightarrow{x} - \overrightarrow{x_\delta}\|_2}{\|\overrightarrow{x}\|_2} \gg 0$$

Knowledge of the condition of A yields information to how stable the system is. Though there exists the $1-$norm and $\infty-$norm definitions of the condition of A that have been well studied and optimized, for this report we will focus primarily on the maximal Euclidean condition number of A which will tell us the condition of A under the Euclidean norm with specific application in robust control systems. To exemplify

a possible application, knowing the condition of A can provide insight as to how to precondition the system to guarantee high accuracy in the solution $\overrightarrow{x}$ [2].

To find the Euclidean condition number of A given we are defining the perturbation to apply to the right vector $\overrightarrow{b}$, we can use the relative difference between $\overrightarrow{x}$ and $\overrightarrow{x_\delta}$ as well as inequalities to set an upper bound on the relative difference.

In finding this upper bound, consider $A\overrightarrow{x} = \overrightarrow{b}$ and note that we can rewrite $A\overrightarrow{x_\delta} = \overrightarrow{b_\delta}$ as $(A + \delta G)\,\overrightarrow{x_\delta} = \overrightarrow{b} + \delta\overrightarrow{g}$ where G is an appropriate matrix $G \in \mathbb{C}^{n\times n}$ and $\overrightarrow{g}$ is an appropriate vector $\overrightarrow{g} \in \mathbb{C}^n$. This implies $\delta = 0$ is a lack of perturbation in $\overrightarrow{b}$ so without loss of generality let $\delta > 0$. Following from Lambers' [3] construction of the ceiling on the relative difference between $\overrightarrow{x}$ and $\overrightarrow{x_\delta}$, suppose A is invertible and define A^{-1} as the inverse. Also define $r = \|\delta A^{-1}G\|_2 < 1$ for simplicity. Note that by Triangle Inequality, $\|A\overrightarrow{x}\|_2 \leq \|A\|_2\|\overrightarrow{x}\|_2$. We construct the upper bound such that:

$$
\begin{aligned}
\frac{\|\overrightarrow{x_\delta} - \overrightarrow{x}\|_2}{\|\overrightarrow{x}\|_2} &= \delta \frac{\|A^{-1}(\overrightarrow{g} - G\overrightarrow{x_\delta})\|_2}{\|\overrightarrow{x}\|_2} \\
&= \delta \frac{\|A^{-1}(\overrightarrow{g} - G\overrightarrow{x}) - A^{-1}G(\overrightarrow{x_\delta} - \overrightarrow{x})\|_2}{\|\overrightarrow{x}\|_2} \\
&\leq \delta \frac{\|A^{-1}(\overrightarrow{g} - G\overrightarrow{x})\|_2}{\|\overrightarrow{x}\|_2} + r \frac{\|\overrightarrow{x_\delta} - \overrightarrow{x}\|_2}{\|\overrightarrow{x}\|_2} \\
&\leq \frac{1}{1-r} \delta \frac{\|A^{-1}(\overrightarrow{g} - G\overrightarrow{x})\|_2}{\|\overrightarrow{x}\|_2} \\
&\leq \frac{1}{1-r} \delta \|A^{-1}\|_2 \left(\frac{\|\overrightarrow{g}\|_2}{\|\overrightarrow{x}\|_2} + \|G\|_2 \right) \\
&\leq \frac{1}{1-r} \delta \|A\|_2 \|A^{-1}\|_2 \left(\frac{\|\overrightarrow{g}\|_2}{\|\overrightarrow{b}\|_2} + \frac{\|G\|_2}{\|A\|_2} \right).
\end{aligned}
$$

This then yields:

$$
\frac{(1-r)\|\overrightarrow{x_\delta} - \overrightarrow{x}\|_2}{\delta \|\overrightarrow{x}\|_2} \left(\frac{\|\overrightarrow{g}\|_2}{\|\overrightarrow{b}\|_2} + \frac{\|G\|_2}{\|A\|_2} \right)^{-1} \leq \|A\|_2 \|A^{-1}\|_2
$$

Here we now notate the Euclidean condition number $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$ which sets a worst possible upper bound on the error a small perturbation to $\overrightarrow{b}$ can make on the solution $\overrightarrow{x}$.

It is well known that computing the inverse of a matrix is an $O(n^3)$ costly operation and so computing $\|A^{-1}\|_2$ is at least $O(n^3)$ costly. However, eigenvalues and singular values have a close relation with the norms $\|A\|_2$ and $\|A^{-1}\|_2$ as well as avoiding the

need to compute A^{-1}. Denote $\{\lambda_i\}_{i=1}^n$ as the set of eigenvalues of A, $\{\gamma_i\}_{i=1}^n$ as the set of eigenvalues of $A^H A$, and $\{\sigma_i\}_{i=1}^n$ as the set of singular values of A. Dependent on whether A is Hermitian or non-Hermitian, we can transform the problem of finding $\kappa_2(A)$ into that of finding the absolutely largest and smallest eigenvalues or singular values - even having the flexibility to choose one over the other. This stems from the fact that $\|A\|_2 = \max_i(\{\sigma_i\}_{i=1}^n)$ and $\|A^{-1}\|_2 = \frac{1}{\min_i(\{\sigma_i\}_{i=1}^n)}$. This implies a singular matrix encounters division by 0, and so it is defined that $\kappa_2(A) = \infty$ if A is singular. Depending on whether A is Hermitian or non-Hermitian, we have 2 equivalencies for $\kappa_2(A)$:

† **A is Hermitian:** If A is Hermitian, that is $A = A^H$, then we have:

$$\kappa_2(A) = \frac{\max_i(\{|\lambda_i|\}_{i=1}^n)}{\min_i(\{|\lambda_i|\}_{i=1}^n)} = \frac{|\lambda_{max}|}{|\lambda_{min}|} = \frac{\sigma_{max}}{\sigma_{min}}$$

Where the eigenvalues in the set $\{\lambda_i\}_{i=1}^n$ are the roots to the characteristic polynomial given by $\det(A - \lambda I_n)$, and I_n is the $n \times n$ identity matrix. Because A is Hermitian, $\{|\lambda_i|\}_{i=1}^n = \{\sigma_i\}_{i=1}^n$.

† **A is non-Hermitian:** If A is non-Hermitian, that is $A \neq A^H$, then we have:

$$\kappa_2(A) = \frac{\max_i(\{\sigma_i\}_{i=1}^n)}{\min_i(\{\sigma_i\}_{i=1}^n)} = \frac{\sigma_{max}}{\sigma_{min}} = \sqrt{\frac{\gamma_{max}}{\gamma_{min}}}$$

Where the eigenvalues in the set $\{\gamma_i\}_{i=1}^n$ are the roots to the characteristic

polynomial given by $\det\left(A^H A - \gamma I_n\right)$. Similar to when A is Hermitian, we have $\{\sigma_i\}_{i=1}^n = \{\sqrt{\gamma_i}\}_{i=1}^n$ in which all γ_i are guaranteed to be real and non-negative since $A^H A$ is positive semi-definite.

Different, widely used methods already exist to quickly obtain minimal and maximal singular values and eigenvalues, and these methods can be called upon and used with MATLAB's implementations *eigs* and *svds*. MATLAB also has the command *normest* which is seemingly unbeatable in obtaining $\|A\|_2$ leaving singular values and eigenvalues the task of obtaining $\|A^{-1}\|_2$.

However, in 2018 the novel method RIM-C has been introduced, and has been shown to be competitive with *eigs* with reliable accuracy. Further testing also suggests that RIM-C is more likely to produce a reliable result to more general matrices when obtaining λ_i without having to fine-tune parameters dependent on A. Since RIM-C targets eigenvalues inside a given complex contour, and the speed for smaller contours has been noticed to be faster than larger contours, this potentially makes RIM-C preferable for finding λ_{min}.

It should be noted that *normest* has been observed to produce more error in finding $\|A\|_2$ than both *eigs* and *svds*. The purpose of finding $\kappa_2(A)$ is to obtain a magnitude as the magnitude provides more information of the conditioning of A than the precise value. That is, if the condition has magnitude of 10^{14}, we care more about obtaining

a value with the same magnitude. So say the true value is $\kappa_2(A) = 2.073682E + 14$; obtaining a value of $2.000000E + 14$ is sufficient for determining how to handle the ill-conditioned matrix. So long as the approximation from any of the implemented methods is on the same magnitude as the true values for $\|A\|_2$ and $\|A^{-1}\|_2$, we should get a sufficient approximation for $\kappa_2(A)$.

The goal of this report will be to use these methods to more quickly and reliably compute a reasonable estimate of $\kappa_2(A)$ for any general matrix A by taking advantage of where each method is strongest. For future context, *normest* has been observed to be the fastest and most reliable at obtaining $\|A\|_2$ and so this method will be preferred for such a purpose. For Hermitian matrices, *eigs* has been observed to be faster with the caveat of potentially failing for finding $\|A^{-1}\|_2$ and so it is preferable to use *eigs* with other methods being used as "fail safes". *svds* seemingly fails whenever *eigs* fails as well as generally being slower for Hermitian matrices so if *eigs* fails, we will use RIM-C to find $\|A^{-1}\|_2$. For non-Hermitian matrices, *svds* tends to be faster and will be tried first before passing the computation to RIM-C. Specifically for MATLAB, determining if a matrix is Hermitian or not is negligible computationally so we can cheaply determine whether A is Hermitian without having such information prior. The MATLAB code can be ran on most computers with out of the box software such as MATLAB with memory being the leading constraint.

Chapter 2

Existing Methods

Considering *normest* performs essentially half of the work for finding $\kappa_2(A)$, it is beneficial to review the work in which *normest* is built from. Though MATLAB does not cite the references behind *normest* as was done with *eigs* and *svds*, it can be presumed it is built from algorithms found in LINPACK. This specific algorithm incorporated into LINPACK by Cline et al. [4] ultimately uses both LU and singular value decompositions to estimate $\|A\|_2$ and $\|A^{-1}\|_2$, but a discussion is provided on how QR decomposition can be used. For dense matrices with random elements, the algorithm incorporating QR decomposition is able to provide more numerical stability. However, Cline et al. make the argument that it is not common in practice to work with dense systems of linear equations with random elements. LU decomposition is able to better leverage these non-random elements to more quickly converge to κ_2's

approximation. For the sake of this report, we consider large general matrices - be they common in practice or not. Both the QR decomposition and the LU decomposition algorithms will be provided.

2.1 LINPACK Incorporation - QR Decomposition

Suppose A can be decomposed into an orthogonal matrix Q and an upper triangular matrix R such that:

$$A = QR$$

From here, we have the following equalities:

$$\|A\|_2 = \|R\|_2$$

$$\|A^{-1}\|_2 = \|R^{-1}\|_2$$

$$\kappa_2(A) = \kappa_2(R)$$

It can be stated here that computing $\|R\|_2$ is preferable to computing $\|A\|_2$. However,

$\|R^{-1}\|_2$ is still expensive. Now consider the following linear system for $\overrightarrow{x}$ and $\overrightarrow{b}$ taking on abstract meanings:

$$R\overrightarrow{x} = \overrightarrow{b}$$

Then we can seek to solve the equivalent problem:

$$\max\left(\frac{\|\overrightarrow{x}\|_2}{\|\overrightarrow{b}\|_2}\right) = \|R^{-1}\|_2$$

Consider a singular value decomposition of R such that $R = U\Sigma V^T$, and denote $\overrightarrow{u_i}$ as the ith column of U (similarly $\overrightarrow{v_i}$ as the ith column of V). We are given that the bound $\max\left(\frac{\|\overrightarrow{x}\|_2}{\|\overrightarrow{b}\|_2}\right)$ is achieved when $\overrightarrow{b} = \overrightarrow{u_n}$, or $\overrightarrow{b}$ is the last column of U. When $\kappa_2(A)$ is large, we state that there is a high probability the bound is a "good" approximation to $\|R^{-1}\|_2$.

Though this process has high probability of "good" results for κ_2 is large, we seek to refine our choice of $\overrightarrow{b}$ to exemplify this natural probability even for smaller values of κ_2. Note that the choice of $\overrightarrow{b} = \overrightarrow{u_n}$ is expensive, and so one would wish to use a cost efficient algorithm like inverse iteration. In theory, one could choose a random $\overrightarrow{b}$, solve $R\overrightarrow{x} = \overrightarrow{b}$, and use the solution as a new choice of $\overrightarrow{b}$ iteratively. What

often results, however, is the linear combination of the solved $\overrightarrow{x}$ in the first iteration has $\overrightarrow{v_n}$ as its dominant term which tends to be nearly orthogonal to $\overrightarrow{u_n}$. When this happens, the linear combination of $\overrightarrow{x}$ does not have $\overrightarrow{u_n}$ as its dominant term. In the second step when the "old" $\overrightarrow{x}$ becomes the "new" $\overrightarrow{b}$, $\overrightarrow{u_n}$ is poorly represented, and solving $R\overrightarrow{x} = \overrightarrow{b}$ will not yield a "large" solution. This results in either a poor approximation to $\|R^{-1}\|_2$, or would require many more steps.

Consider a two-step method where we solve 2 linear systems at each iteration:

$$R^T\overrightarrow{x} = \overrightarrow{b}$$

$$R\overrightarrow{y} = \overrightarrow{x}$$

Our new bound we seek to maximize then becomes $\max\left(\frac{\|\overrightarrow{y}\|_2}{\|\overrightarrow{x}\|_2}\right)$ in which now a random choice of $\overrightarrow{b}$ has a high probability to approximate $\|R^{-1}\|_2$ regardless of the size of κ_2.

2.2 LINPACK Incorporation - LU Decomposition

In practice, most dense linear systems are commonly solved with Guassian elimination; a form of pivoting matrix elements. Separating from the previous section, consider U and Q abstract. Given a matrix A, we can fully pivot the matrix with non-singular matrices multiplying such that PAQ. When $Q = I$, we say the product PA is a partial pivot of A. Given these criteria, assume we have already pivoted A appropriately and notate it A_p, and we can decompose such that, for a unit lower triangular matrix L, its row i column j^{th} elements $l_{i,j}$, and an upper triangular matrix U:

$$A_p = PAQ = LU$$

$$|l_{i,j}| \leq 1 \ \forall \ i, j$$

Similarly to how we refined the approach featuring QR decomposition, we can apply the 2-step approach such that we now consider the following systems:

$$(LU)^T \overrightarrow{x} = \overrightarrow{b} \tag{2.1}$$

$$LU\overrightarrow{y} = \overrightarrow{x}$$

Note that by pivoting, $\|A^{-1}\|_2 = \|A_p^{-1}\|_2$ so we can focus on finding the 2-norm of our inverse pivoted matrix A_p^{-1} equivalently. Again similar to the QR approach, now $\max\left(\frac{\|\overrightarrow{y}\|_2}{\|\overrightarrow{x}\|_2}\right)$ is our bound we seek. In solving 2.1, we can further decompose yielding 2 more systems to solve for each iteration:

$$U^T\overrightarrow{z} = \overrightarrow{b}$$

$$L^T\overrightarrow{x} = \overrightarrow{z}$$

Similar to the previous section, now a random choice of $\overrightarrow{b}$ should yield with high probability a good approximation to $\|A^{-1}\|_2$.

Chapter 3

MATLAB's Eigenvalue and Singular Value Solvers

Solving for $\kappa_2(A)$ via eigen and singular value solvers, there will be three methods we will utilize - two of which pre-existing in MATLAB. MATLAB's *eigs* derived from Lehoucq, Sorenson, & Yang's algorithm and improved upon by Stewart [5, 6] is used generally for Hermitian matrices as it demonstrates the best efficiency with the caveat it may fail and produce no usable results. MATLAB's *svds* derived from Larsen's bidiagonalization with partial reorthogonalization method and improved with augmented implicit restarting [7, 8] is similarly used for non-Hermitian matrices for simiiliar reasoning. Using *svds* also avoids the need to consider finding the square root of $\kappa_2\left(A^H A\right)$ as matrix multiplication can be an expensive operation.

3.1 MATLAB *eigs*

MATLAB's *eigs* function utilizes forward stable Krylov-Schur decomposition to obtain eigenvalues from A as opposed to the potentially forward unstable implicitly restarted Arnoldi algorithm. *eigs* is generally the fastest method of those used in this report. However, the function has a tendency to fail on large matrices albeit the error tends to not waste too much computational time. It is seen as generally best to attempt this method first on Hermitian matrices.

First, assume A is Hermitian, let $\overrightarrow{u}$ be abstract, and define $\overrightarrow{u_1} \in \mathbb{R}^n$ as a random vector satisfying $\|\overrightarrow{u_1}\|_2 = 1$. Generate $\overrightarrow{u_2}$, $\overrightarrow{u_3}$, ... from sequentially orthogonalizing the Krylov sequence $\overrightarrow{u_1}$, $A\overrightarrow{u_1}$, $A^2\overrightarrow{u_1}$, ... using the Lanczos method. By Lanczos, A being Hermitian implies our vectors $\overrightarrow{u_i}$ satisfy the following relation for scalars $\alpha_i, \beta_i \in \mathbb{R}$:

$$\beta_i \overrightarrow{u_{i+1}} = A_i \overrightarrow{u_i} - \alpha_i \overrightarrow{u_i} - \beta_{i-1} \overrightarrow{u_{i-1}}$$

Up to a desired order k, abstract the variables U and $\overrightarrow{b}$, and define the matrix $U_k = \begin{pmatrix} \overrightarrow{u_1} & \overrightarrow{u_2} & \dots & \overrightarrow{u_k} \end{pmatrix}$. Also define $B_k = \begin{pmatrix} \overrightarrow{b_1} & \overrightarrow{b_2} & \dots & \overrightarrow{b_k} \end{pmatrix}$ as the order k Rayleigh quotient. We define the Krylov-Schur decomposition as follows:

$$AU_k = U_k B_k + \overrightarrow{u_{k+1}}\overrightarrow{b_{k+1}}^H$$

Define $\overrightarrow{e_i}$ as the appropriately sized vector where the ith element is the only non-zero entry, and equal to 1. Because A is Hermitian, $\exists$ a tridiagonal matrix of order k, denoted by T_k, composed of α_i, β_i which satisfies:

$$AU_k = U_k T_k + \beta_k \overrightarrow{u_{k+1}}\overrightarrow{e_k}^T$$

Finishing up the Lanczos decomposition, we now have T_k as the Rayleigh quotient, $T_k = B_k$, or:

$$T_k = U_k^H A U_k$$

By Schur [9], there exists a unitary matrix Q and an upper triangular matrix S_k such that:

$$B_k = Q S_k Q^{-1}$$

Note that S_k is upper triangular and unitarily similar to B_k. We say that S_k stresses

the triangularity of B_k. Define $\hat{S}_k = \begin{pmatrix} S_k \\ \overrightarrow{b_{k+1}}^H \end{pmatrix}$ and we can write the form:

$$AU_k = U_{k+1}\hat{S}_k$$

Now we have sufficiently setup what was needed to begin the Krylov-Schur decomposition method. Each step of the method then iterates through the equality:

$$AU_k = U_kS_k + \overrightarrow{u_{k+1}}\overrightarrow{b_{k+1}}^H$$

Each step goes through two main phases; the expansion phase in which the Krylov sequence is expanded, and the contraction phase in which unwanted Ritz values are purged where the Ritz values are defined as the eigenvalues of the matrix T_k. We expand by orthogonalizing $A\overrightarrow{u_{k+1}}$ against U_{k+1}. The result is normalized to yield $\overrightarrow{u_{k+2}}$ which allows us to form S_{k+1} from the previous S_k. Let v, w, and ν be programming variables and assume U_{k+1} is initialized into U and $\hat{S}_k$ initialized into S; the following pseudocode implements the expansion phase:

In floating point arithmetic, we will need to reorthogonalize such that v is orthogonal to the column space of U.

Algorithm 1 Krylov-Shur Decomposition Expansion Phase

1: $v = AU[:, k+1]$
2: $w = U^H v$
3: $v = v - Uw$
4: $\nu = \|v\|_2$
5: $U = \begin{pmatrix} U & \frac{v}{\nu} \end{pmatrix}$
6: $\hat{S} = \begin{pmatrix} \hat{S} & w \\ 0 & \nu \end{pmatrix}$

This process can be repeated. Define the scalar η such that the pseudocode above is ran $\eta - k$ times. So for one iteration, $\eta = k+1$. This yields our expanded Krylov-Schur decomposition:

$$AU_\eta = U_\eta S_\eta + \overrightarrow{u_{\eta+1}}\overrightarrow{b_{\eta+1}}^H$$

It is now we are on the contraction phase - purging unwanted Ritz values. To do so, note that we are able to truncate the decomposition at any position. Setting up, define the block forms $U_\eta = \begin{pmatrix} U_1 & U_2 \end{pmatrix}$, $S_\eta = \begin{pmatrix} S_{1,1} & S_{1,2} \\ 0 & S_{2,2} \end{pmatrix}$, and $u_{\eta+1}\overrightarrow{b_{\eta+1}}^H = \overrightarrow{u}\begin{pmatrix} \overrightarrow{b_1}^H & \overrightarrow{b_2}^H \end{pmatrix}$. The following equality is also a Krylov-Schur decomposition:

$$AU_1 = U_1 S_{1,1} + \overrightarrow{u}\,\overrightarrow{b_1}^H$$

At this point, the approximations to the desired eigenvalues of A will be located on

the diagonal entries of $S_{1,1}$.

3.2 MATLAB *svds*

MATLAB's *svds* improves upon Larsen's bidiagonalization with partial reorthoginalizations by restarting the augmented Krylov subspaces which are obtained in the standard method. The method addresses the issue that many scientific computational problems require only a few of the largest and / or smallest singular values - this report requiring only the single largest and smallest. Since this method has been observed in all cases to be slower than *eigs* on Hermitian matrices, we only use *svds* on non-Hermitian matrices A.

To begin the method, order the singular values of A such that:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

Note this now equates $\sigma_1 = \sigma_{max}$. There will then be n associated left and right singular vectors. Denote the normalized left singular vectors $\overrightarrow{u_j} \in \mathbb{R}^m$ and the normalized right singular vectors $\overrightarrow{v_j} \in \mathbb{R}^n$ for indices $1 \leq j \leq n$. For all j, we have the following equalities:

$$A\overrightarrow{v_j} = \sigma_j \overrightarrow{u_j}$$

$$A^H \overrightarrow{u_j} = \sigma_j \overrightarrow{v_j}$$

We also have:

$$A = \sum_{j=1}^{n} \sigma_j \overrightarrow{u_j} \overrightarrow{v_j}^T$$

The matrices composed of all these singular vectors $U = [\overrightarrow{u_1}, \overrightarrow{u_2}, \cdots, \overrightarrow{u_n}]$ and $V = [\overrightarrow{v_1}, \overrightarrow{v_2}, \cdots, \overrightarrow{v_n}]$ have orthonormal columns. Per index j, $\{\sigma_j, \overrightarrow{u_j}, \overrightarrow{v_j}\}$ will be called the jth singular triplet of A. Furthermore, the 1st singular triplet is the largest singular triplet, and the nth is the smallest singular triplet.

Both Lanczos bidiagonalization and the improvement found in *svds* will compute sequences of projections of A onto well-chosen lower dimensional subspaces. Compared to the original Lanczos bidiagonalization method, *svds* introduces restarting by augmentation of Krylov subspaces determined similarly to the subspaces from Lanczos bidiagonalization.

Starting the partial Lanczos bidiagonalization, introduce a randomized vector $\overrightarrow{p_1} \in \mathbb{R}^n$

satisfying $\|\overrightarrow{p_1}\|_2 = 1$. Ideally, we want $\overrightarrow{p_1}$ to be a linear combination of the right singular vectors of the associated singular value we desire.

Let l denote the partial steps we take. Abstract the variable Q and let $P_l \in \mathbb{C}^{n\times l}$ and $Q_l \in \mathbb{C}^{n\times l}$ be resulting orthonormal matrices. Let $B_l \in \mathbb{C}^{l\times l}$ be a bidiagonal matrix with main and super diagonals whose singular values are similar to A. A downfall of the basic Lanczos bidiagonalization method entails σ_{max} is more quickly approximated than σ_{min}. This will be corrected with *svds* contribution on the method. For a sufficiently small choice of l, the following decomposition exists:

$$AP_l = Q_l B_l \tag{3.1}$$

Denote $I_l \in \mathbb{C}^{l\times l}$ to be the identity matrix, $\overrightarrow{e_j} \in \mathbb{R}^n$ to be the all 0 vector save for a 1 in the jth row (axis vector), and let $\overrightarrow{r_l} \in \mathbb{R}^n$ be the residual vector. From the decomposition we also have the following:

$$A^H Q_l = P_l B_l^H + \overrightarrow{r_l}\,\overrightarrow{e_l}^T \tag{3.2}$$

$$P_l^H P_l = I_l = Q_l^H Q_l$$

$$P_l \overrightarrow{e_1} = \overrightarrow{p_1}$$

With this information, we may now introduce the Lanczos bidiagonalization method iteratively. Without further information, choose $\overrightarrow{p_1}$ to be a random unit vector. Letting α_j be the jth main diagonal entry of B_l, β_j be the jth super diagonal entry of B_l, and $\overrightarrow{q_j}$ be the jth column of Q_l, we have the following algorithm:

Algorithm 2 Lanczos Bidiagonalization Method

1: $P_1 = \overrightarrow{p_1}$
2: $\overrightarrow{q_1} = A\overrightarrow{p_1}$
3: $\alpha_1 = \|\overrightarrow{q_1}\|_2$
4: $\overrightarrow{q_1} = \frac{\overrightarrow{q_1}}{\alpha_1}$
5: $Q_1 = \overrightarrow{q_1}$
6: **for** $j = 1 : l$ **do**
7: $\quad \overrightarrow{r_j} = A^H \overrightarrow{q_j} - \alpha_j \overrightarrow{p_j}$
8: $\quad \overrightarrow{r_j} = \overrightarrow{r_j} - P_j \left(P_j^H \overrightarrow{r_j}\right)$
9: $\quad$ **if** $j < l$ **then**
10: $\quad\quad \beta_j = \|\overrightarrow{r_j}\|_2$
11: $\quad\quad \overrightarrow{p_{j+1}} = \frac{\overrightarrow{r_j}}{\beta_j}$
12: $\quad\quad P_{j+1} = [P_j, \overrightarrow{p_{j+1}}]$
13: $\quad\quad \overrightarrow{q_{j+1}} = A\overrightarrow{p_{j+1}} - \beta_j \overrightarrow{q_j}$
14: $\quad\quad \overrightarrow{q_{j+1}} = \overrightarrow{q_{j+1}} - Q_j \left(Q_j^H \overrightarrow{q_{j+1}}\right)$
15: $\quad\quad \alpha_{j+1} = \|\overrightarrow{q_{j+1}}\|_2$
16: $\quad\quad \overrightarrow{q_{j+1}} = \frac{\overrightarrow{q_{j+1}}}{\alpha_{j+1}}$
17: $\quad\quad Q_{j+1} = [Q_j, \overrightarrow{q_{j+1}}]$
18: $\quad$ **end if**
19: **end for**

As stated, this method yields better approximations of σ_{max} than σ_{min}, and these approximations get better as we use larger values of l. Noting that steps on lines (8) and (14) are reorthoginalization steps, it is stated by Baglama [7] that we often need

a full, not partial, reorthoginalization to get high accuracy.

3.2.1 Augmentation by Ritz Vectors to Obtain σ_{max}

Regarding σ_{max}, though we claim Lanczos Bidiagonalization yields good approximations, augmentation by Ritz vectors to find only σ_{max} is introduced to make *svds* less sensitive to propogated round-off error. Let $\overrightarrow{a_j}$ be the associated left singular vector of B_l, and let $\overrightarrow{b_j}$ be the associated right singular vector of B_l. We define the approximated left and right singular vectors of A respectively using singular vectors of B_l such that:

$$\overrightarrow{\tilde{u}_j} = Q_l \overrightarrow{a_j}$$

$$\overrightarrow{\tilde{v}_j} = P_l \overrightarrow{b_j}$$

Denote $\tilde{\sigma}_j$ as a singular value of B_l. Note that $\overrightarrow{\tilde{v}_j}$ is a Ritz vector of $A^H A$ associated with $(\tilde{\sigma}_j)^2$. For all j such that $1 \leq j \leq l$, we have:

$$A^H A \overrightarrow{\tilde{v}_j} - (\tilde{\sigma}_j)^2 \overrightarrow{\tilde{v}_j} = \alpha_l \overrightarrow{r_l} \overrightarrow{e_l}^H \overrightarrow{b_j}$$

Recalling decompositions 3.1 and 3.2, we can now choose the first column of P_l as a Ritz vector, and restart the computations to increase accuracy. Letting this vector be available to us, $\overrightarrow{\tilde{v}_1}$, note that the Ritz vector is orthonormal with $\overrightarrow{r_l}$. Introduce the following matrix:

$$\tilde{P}_2 = \left[\overrightarrow{\tilde{v}_1}, \overrightarrow{p_{l+1}}\right]$$

We then have the consequence that $\overrightarrow{p_{l+1}}$ is parallel to $\overrightarrow{r_l}$. Following:

$$A\tilde{P}_2 = \left[\tilde{\sigma}_1\overrightarrow{\tilde{u}_1}, A\overrightarrow{p_{l+1}}\right]$$

Orthogonalizing $A\tilde{P}_2$ against $\overrightarrow{\tilde{u}_1}$ then gives us:

$$A\overrightarrow{p_{l+1}} = \tilde{\zeta}\overrightarrow{\tilde{u}_1} + \overrightarrow{\tilde{r}_1}$$

Where $\overrightarrow{\tilde{r}_1}$ is the residual orthogonal to $\overrightarrow{\tilde{u}_1}$ and $\tilde{\zeta}$ can be easily computed such that:

$$
\begin{aligned}
\tilde{\zeta} &= \left(\overrightarrow{\tilde{u}_1}\right)^H A\overrightarrow{p_{l+1}} \\
&= \overrightarrow{p_{l+1}}^H A^H \overrightarrow{\tilde{u}_1} \\
&= \overrightarrow{p_{l+1}} \left(\tilde{\sigma}_1 \overrightarrow{\tilde{v}_1} + \overrightarrow{r_l}\,\overrightarrow{e_l}^H \overrightarrow{a_1}\right) \\
&= \beta_l \overrightarrow{e_l}^H \overrightarrow{\tilde{u}_1}
\end{aligned}
$$

Now we can construct our augmented decomposition analogous to 3.1 so that:

$$
\tilde{Q}_2 = \left[\overrightarrow{\tilde{u}_1}, \frac{\overrightarrow{\tilde{r}_1}}{\|\overrightarrow{\tilde{r}_1}\|_2}\right]
$$

$$
A\tilde{P}_2 = \tilde{Q}_2 \tilde{B}_2
$$

Where $\tilde{B}_2$ has a main diagonal and the right-most column as potentially non-zero. To get the augmented analogous decomposition to 3.2, we will need to express $A^H \tilde{Q}_2$ in terms of $\tilde{P}_2$ and $\tilde{B}_2^H$. Since the first column of $A^H \tilde{Q}_2$ is a linear combination of $\overrightarrow{\tilde{v}_1}$ and $\overrightarrow{p_{l+1}}$, we have the following:

$$A^H \overrightarrow{\tilde{u}_1} = \tilde{\sigma}_1 \overrightarrow{\tilde{v}_1} + \overrightarrow{r_l}\,\overrightarrow{e_l}^H \overrightarrow{a_1}$$

$$= \tilde{\sigma}_1 \overrightarrow{\tilde{v}_1} + \overrightarrow{p_{l+1}}\tilde{\zeta}$$

Because $A^H \frac{\overrightarrow{\tilde{r}_1}}{\|\overrightarrow{\tilde{r}_1}\|_2}$ is orthogonal to $\overrightarrow{\tilde{v}_1}$, we can express the following:

$$A^H \frac{\overrightarrow{\tilde{r}_1}}{\|\overrightarrow{\tilde{r}_1}\|_2} = \|\overrightarrow{\tilde{r}_1}\|_2 \overrightarrow{p_{l+1}} + \overrightarrow{\tilde{f}_2} \tag{3.3}$$

Where $\overrightarrow{\tilde{f}_2} \in \mathbb{C}^{n\times 1}$ is orthogonal to both $\overrightarrow{\tilde{v}_1}$ and $\overrightarrow{p_{l+1}}$, and can be computed from (3.3). This now yields the augmented decomposition analogous to (3.2):

$$A^H \tilde{Q}_2 = \tilde{P}_2 \tilde{B}_2^H + \overrightarrow{\tilde{f}_2}\,\overrightarrow{e_2}^H$$

Hence, the Lanczos bidiagonalization method can be ran on the analogous decomposition recalling $\sigma_{max} \approx \tilde{\sigma}_{max}$. Furthermore, $\tilde{\sigma}_{max}$ will be in entry $()_{1,1}$ of $\tilde{B}_2$.

3.2.2 Augmentation by Harmonic Ritz Vectors for σ_{min}

Regarding σ_{min}, we know the Lanczos bidiagonalization method often requires a full reorthoginalization to yield a high accuracy approximation. However, Kokiopoulou, Bekas, and Gallopoulos [10] observed that a faster approximation with high accuracy can be obtained when shifting by harmonic Ritz values as opposed to shifting by the standard Ritz values.

From the standard Lanczos bidiagonalization method, suppose for all indices j that α_j and β_j are non-zero - in particular B_l is non-singular. The harmonic Ritz values θ_j of $A^H A$ with respect to the partial Lanczos tridiagonalization method are the eigenvalues of the generalized problem:

$$\left(\left(B_l^H B_l\right)^2 + \alpha_l^2 \beta_l^2 \overrightarrow{e_l}\,\overrightarrow{e_l}^H\right) \overrightarrow{\hat{\omega}_j} = \theta_j B_l^H B_l \overrightarrow{\hat{\omega}_j} \tag{3.4}$$

Where $\overrightarrow{\hat{\omega}_j} \in \mathbb{C}^{l\times 1}$ is non-zero (further details found from Morgan or Paige, Parlett, and van der Vorst). By defining $\overrightarrow{\omega_j} = B_l \overrightarrow{\hat{\omega}_j}$, we can avoid computing $B_l^H B_l$ by choosing $\overrightarrow{\omega_j}$ to be piecewise orthonormal and formulating:

$$\left(B_l B_l^H + \beta_l^2 \overrightarrow{e_l}\,\overrightarrow{e_l}^H\right) \overrightarrow{\omega_j} = \theta_j \overrightarrow{\omega_j} \tag{3.5}$$

Define $P_{l+1} = \left[P_l, \frac{\overrightarrow{r_l}}{\beta_l}\right]$ and let $B_{l,l+1}$ be defined by $A^H Q_l = P_{l+1} B_{l,l+1}^H$. We also have the following:

$$B_{l,l+1} B_{l,l+1}^H = B_l B_l^H + \beta_l^2 \overrightarrow{e_l} \overrightarrow{e_l}^H$$

Now introduce the notation $\{\hat{\sigma}_j, \overrightarrow{\hat{a}_j}, \overrightarrow{\hat{b}_j}\}$ as the singular triplets (singular value, left and right associated singular vectors respectively) of $B_{l,l+1}$ for $1 \leq j \leq l$. Now we order the singular values so that:

$$0 < \hat{\sigma}_1 \leq \hat{\sigma}_2 \leq \ldots \leq \hat{\sigma}_l \tag{3.6}$$

Considering we are interested in only the smallest singular value of A, we have the following:

$$B_{l,l+1} \overrightarrow{\hat{b}_1} = \overrightarrow{\hat{a}_1} \left[\hat{\sigma}_1\right] \tag{3.7}$$

$$B_{l,l+1}^H \overrightarrow{\hat{a}_1} = \overrightarrow{\hat{b}_1} \left[\hat{\sigma}_1\right] \tag{3.8}$$

Referring now to (3.7) and (3.8) as the partial singular value decompositions of $B_{l,l+1}$, it follows also from (3.6) that $\hat{\sigma}_1^2$ and $\overrightarrow{\hat{a}_1}$ form an eigenpair of (3.5). We also have $\hat{\sigma}_1^2$ and $B_l^{-1}\overrightarrow{\hat{a}_1}$ as an eigenpair of (3.4). This implies the two mentioned eigenpairs can be determined from (3.7) and (3.8) in which Baglama and Reichel claim for a sufficiently small l are computationally negligible to obtain.

The family harmonic Ritz vectors of $A^H A$ associated with each θ_j is then given by:

$$\overrightarrow{h_j} = P_l \overrightarrow{\hat{\omega}_j}$$

It was observed by Morgan the Zeng that the residual errors associated with each harmonic Ritz pair θ_j and $\overrightarrow{h_j}$ are parallel. This result becomes central to the augmentation used in the method and so we obtain the result:

$$
\begin{aligned}
A^H A\overrightarrow{h_j} - \theta_j \overrightarrow{h_j} &= A^H A P_l \overrightarrow{\hat{\omega}_j} - \theta_j P_l \overrightarrow{\hat{\omega}_j} \\
&= \left(P_l B_l^H B_l + \alpha_l \overrightarrow{r_l}\,\overrightarrow{e_l}^H\right) \overrightarrow{\hat{\omega}_j} - \theta_j P_l \overrightarrow{\hat{\omega}_j} \\
&= P_l \left(B_l^H B_l - \theta_j I_l\right) \overrightarrow{\hat{\omega}_j} + \alpha_l \overrightarrow{r_l}\,\overrightarrow{e_l}^H \overrightarrow{\hat{\omega}_j} \\
&= P_l B_l^{-1} \left(B_l B_l^H - \theta_j I_l\right) \overrightarrow{\omega_j} + \overrightarrow{r_l}\,\overrightarrow{e_l}^H \overrightarrow{\omega_j} \\
&= P_l B_l^{-1} \left(-\beta_j^2 \overrightarrow{e_l}\,\overrightarrow{e_l}^H \overrightarrow{\omega_j}\right) + \overrightarrow{r_l}\,\overrightarrow{e_l}^H \overrightarrow{\omega_j} \\
&= \overrightarrow{e_l}^H \overrightarrow{\omega_j} \left(\overrightarrow{r_l} - \beta_l^2 P_l B_l^{-1} \overrightarrow{e_l}\right)
\end{aligned}
$$

We now have our scaled residual vector:

$$
\overrightarrow{\hat{r}_l} = \overrightarrow{p_{l+1}} - \beta_l P_l B_l^{-1} \overrightarrow{e_l}
$$

And finally we can form our relation analogous to 3.2 such that:

$$
\left[\overrightarrow{h_1}\hat{\sigma}_1, \overrightarrow{\hat{r}_l}\right] = P_{l+1} \begin{pmatrix} B_l^{-1} \overrightarrow{\hat{a}_1} \left[\hat{\sigma}_1\right] & \beta_l B_l^{-1} \overrightarrow{e_l} \\ 0 & 1 \end{pmatrix}
$$

To get to the proper form, introduce the following QR-factorization:

$$\begin{pmatrix} B_l^{-1}\overrightarrow{\hat{a}_1}\left[\hat{\sigma}_1\right] & \beta_l B_l^{-1}\overrightarrow{e_l} \\ 0 & 1 \end{pmatrix} = \hat{Q}_2\hat{R}_2$$

Defining $\hat{P}_2 = P_{l+1}\hat{Q}_2$, we obtain our proper analogous form:

$$\begin{aligned} A\hat{P}_2 &= \left[AP_l, A\overrightarrow{p_{l+1}}\right]\hat{Q}_2 \\ &= \left[Q_l B_l, A\overrightarrow{p_{l+1}}\right]\begin{pmatrix} B_l^{-1}\overrightarrow{\hat{a}_1}\left[\hat{\sigma}_1\right] & \beta_l B_l^{-1}\overrightarrow{e_l} \\ 0 & 1 \end{pmatrix}\hat{R}_2^{-1} \\ &= \left[Q_l\overrightarrow{\hat{a}_1}\left[\hat{\sigma}_1\right], -\beta_l\overrightarrow{q_l} + A\overrightarrow{p_{l+1}}\right]\hat{R}_2^{-1} \end{aligned}$$

Using the above form in Lanczos bidiagonalization method is then possible, and will yield an approximation to σ_{min} of A.

Chapter 4

RIM-C and Proposing *newEuclidCond*

4.1 RIM-C

RIM-C is a novel eigenvalue solver that arose from the transmission eigenvalue problem. RIM-C is valued in this report for its reliability as well as the method of targeting a contour to find eigenvalues within. This method has been observed to require more time the larger the spectral contour making it seemingly ideal for eigenvalues near the origin - specifically λ_{min}. Consider the generalized eigenvalue problem:

$$A\overrightarrow{x} = \lambda B\overrightarrow{x}$$

Though we find the eigenvalues of A such that $B = I_n$ for this report, keep B sufficiently abstract for the sake of the algebra. Assuming $A : \mathbb{C}^n \to \mathbb{C}^n$ is bounded on a complex Hilbert space, the resolvent set of A is given as:

$$\rho(A) = \{z \in \mathbb{C} : \exists \ (A - zB)^{-1}\}$$

If $z \in \rho\,(A)$, then the resolvent is given as:

$$R_z\,(A) = (A - zB)^{-1} \implies \tau\,(A) = \mathbb{C} \setminus \rho\,(A)$$

Where $\tau\,(A)$ is the spectrum of A. Assuming that A has only point spectrum and each eigenspace associated with any $\lambda_k \in \tau\,(A)$ is finite-dimensional, we can form the spectral projection given as, for a closed spectral contour Γ:

$$P_\Gamma\,(A) = \frac{1}{2\pi i}\int_\Gamma R_z\,(A)\,dz$$

For a random vector $\overrightarrow{f}$ satisfying $\|\overrightarrow{f}\|_2 = 1$, $P_\Gamma\,(A)\,\overrightarrow{f}$ is named the indicator,

and indicates there is at least one $\lambda_k \in \Gamma$ if $\|P_\Gamma(A)\overrightarrow{f}\|_2 \neq 0$. Applying a simple quadrature rule, let $\{\omega_k\}_{k=1}^W$ be the set of quadrature points and $\{\overrightarrow{x_k}\}_{k=1}^W$ be the solutions to the linear systems $(A - z_k B)\overrightarrow{x_k} = \overrightarrow{f}$ for $\{z_k\}_{k=1}^W \in \rho(A)$ - we can approximate the indicator such that:

$$P_\Gamma(A)\overrightarrow{f} \approx \frac{1}{2\pi i}\sum_{k=1}^{W}\omega_k\overrightarrow{x_k} \implies \mathcal{P}_\Gamma(A) = \|P_\Gamma(A)\overrightarrow{f}\|_2$$

Lastly, we face the problem of deciding when the indicator is zero. With a constant tolerance $T > 0$, if we do not scale our indicator then we could miss eigenvalues even though $\mathcal{P}_\Gamma(A) < T$ is true. To circumvent this, we project onto the normalization of $P_\Gamma(A)\overrightarrow{f}$ such that we obtain our scaled indicator:

$$\delta_\Gamma = \|P_\Gamma\left(\frac{P_\Gamma(A)\overrightarrow{f}}{\mathcal{P}_\Gamma(A)}\right)\|_2 \tag{4.1}$$

Thus we can iterate. Suppose Γ is a simple spectral circle of radius r and let T be the tolerance. If $\delta_\Gamma < T$, we claim there are no eigenvalues in Γ and we expand our search region such that for $\Gamma \subset \dot{\Gamma}$, we test $\mathcal{P}_{\dot{\Gamma}}(A)$. Conversely, we can shrink our search radius. Let $\epsilon > 0$ be real and a minimum radius. Denoting $\hat{r} > \epsilon$ as the radius of $\hat{\Gamma} \subset \Gamma$, we check $\mathcal{P}_{\hat{\Gamma}}(A)$ with decreasing $\hat{r}$ until our radius equals ϵ in which case we have an approximate for the observed λ_k with absolute error at most $\epsilon\sqrt{2}$, and

claim $\lambda_k \in \Gamma$.

Now that we have outlined the standard RIM method, we notice most of the computational cost comes from obtaining δ_Γ since it effectively doubles the linear systems we must solve - once for the indicator, and again to scale the indicator. Since δ_Γ plays a major role in not missing eigenvalues, we circumvent the large number of linear systems we need to solve by introducing Cayley transformation as well as carrying out the Arnoldi process.

We begin with Cayley transformation to take advantage of the parametarized linear systems with the same structure as those required for RIM. Consider the family of linear systems for $z \in \mathbb{C}$:

$$(A - zB)\,\overrightarrow{x} = \overrightarrow{f} \tag{4.2}$$

If B^{-1} exists, we can left multiply B^{-1} to both sides such that we obtain:

$$\left(B^{-1}A - zI_n\right)\overrightarrow{x} = B^{-1}\overrightarrow{f}$$

Given a matrix M, vector $\overrightarrow{b}$, and a scalar $m \in \mathbb{N}$, the Krylov subspace is defined as:

$$K_m\left(M,\,\overrightarrow{b}\right) = span\left(\{M^{k-1}\overrightarrow{b}\}_{k=1}^{m}\right)$$

For $\alpha, \beta \in \mathbb{C}$, by shift-invariance we also have:

$$K_m\left(\alpha M + \beta I_n,\,\overrightarrow{b}\right) = K_m\left(M,\,\overrightarrow{b}\right)$$

This implies the Krylov subspace of $B^{-1}A - zI_n$ is the same as $B^{-1}A$ which alleviates dependence on z. Though Cayley transformation is able to fix the case where B^{-1} does not exist, Arnoldi's method is able to exploit a result of Cayley transformation hence why we perform the transformation regardless that $B = I_n$. To continue, suppose $\psi \neq z$ is not a generalized eigenvalue. Left multiply $(A - \psi B)^{-1}$ to 4.2 to obtain:

$$\begin{aligned}
(A - \psi B)^{-1}\overrightarrow{f} &= (A - \psi B)^{-1}(A - zB)\overrightarrow{x} \\
&= (A - \psi B)^{-1}(A - \psi B + (\psi - z)B)\overrightarrow{x} \\
&= \left(I_n + (\psi - z)(A - \psi B)^{-1}B\right)\overrightarrow{x}
\end{aligned}$$

Let $M = (A - \psi B)^{-1}B$ and $\overrightarrow{b} = (A - \psi B)^{-1}\overrightarrow{f}$. Then 4.2 can be treated as:

$$(I_n + (\psi - z)\, M)\, \overrightarrow{x} = \overrightarrow{b} \tag{4.3}$$

Because the Krylov subspace exhibits shift invariance, $K_m\left(I_n + (\psi - z)\, M,\, \overrightarrow{b}\right) = K_m\left(M,\, \overrightarrow{b}\right)$. This result is exploitable. Consider the orthogonal projection method for $M\overrightarrow{x} = \overrightarrow{b}$. Take an initial guess of $\overrightarrow{x_0} = \overrightarrow{0}$; we seek an approximate solution $\overrightarrow{x_m}$ in $K_m\left(M,\, \overrightarrow{b}\right)$ by imposing the following Galerkin condition:

$$\left(\overrightarrow{b} - M\overrightarrow{x_m}\right) \perp K_m\left(M,\, \overrightarrow{b}\right) \tag{4.4}$$

Consider Arnoldi's full orthogonalization method [11]:

Algorithm 3 Arnoldi's Full Orthogonalization Method

```
Generate v_1^→ satisfying ||v_1^→||_2 = 1
for j = 1 : m do
    for i = 1 : j do
        h_{i,j} = ⟨M v_j^→, v_i^→⟩
    end for
    w_j^→ = M v_j^→ − Σ_{i=1}^{j} h_{i,j} v_i^→
    h_{j+1,j} = ||v_j^→||_2
    if h_{j+1,j} = 0 then
        Stop algorithm
    end if
    v_{j+1}^→ = w_j^→ / h_{j+1,j}
end for
```

Let $V_m = \begin{pmatrix} \overrightarrow{v_1} & \dots & \overrightarrow{v_m} \end{pmatrix}$ be an orthogonal matrix, and H_m be $m \times m$ Hessenberg matrix with entries enumerated by the above algorithm. Proposition 6.5 from Saad

[11] yields for us:

$$MV_m = V_m H_m + \overrightarrow{v_m} h_{m+1,m} \overrightarrow{e_m}^T$$

4.4 is constrained by:

$$span\left(col\left(V_m\right)\right) = K_m\left(M,\ \overrightarrow{b}\right)$$

By letting $\overrightarrow{x_m} = V_m \overrightarrow{y}$, 4.4 becomes:

$$V_m^T \overrightarrow{b} - V_m^T M V_m \overrightarrow{y} = \overrightarrow{0}$$

Shown by Saad [12], $V_m^T M V_m = H_m$ holds implying:

$$H_m \overrightarrow{y} = V_m^T \overrightarrow{b}$$

By construction, $\overrightarrow{v_1} = \frac{\overrightarrow{b}}{\| \overrightarrow{b} \|_2}$. Let $\beta = \| \overrightarrow{b} \|_2$. Then we can write:

$$\overrightarrow{y} = \beta H_m^{-1} \overrightarrow{e_1}$$

This lets us rewrite the residual of the approximate solution $\overrightarrow{x_m}$ as:

$$\| \overrightarrow{b} - M\overrightarrow{x_m} \|_2 = h_{m+1,m} |\overrightarrow{e_m}^T \overrightarrow{y}| \tag{4.5}$$

By shift invariance, we further have that:

$$(I_n + (\psi - z)\, M)\, V_m = V_m\, (I_m + (\psi - z)\, H_m) + (\psi - z)\, \overrightarrow{v_{m+1}} h_{m+1,m} \overrightarrow{e_m}^T \tag{4.6}$$

We can impose a Galerkin condition similar to 4.4 to obtain:

$$V_m^T \overrightarrow{b} - V_m^T\, (I_n + (\psi - z)\, M)\, V_m\, \overrightarrow{y} = 0$$

$$\implies (I_m + (\psi - z)\, H_m)\, \overrightarrow{y} = \beta \overrightarrow{e_1} \tag{4.7}$$

From 4.5 we can then obtain:

$$\| \overrightarrow{b} - (I_n + (\psi - z)\, M)\, \overrightarrow{x_m} \|_2 = (\psi - z)\, h_{m+1,m} |\overrightarrow{e_m}^T \overrightarrow{y}| \tag{4.8}$$

Here, M is a $n \times n$ matrix and H_m is the $m \times m$ upper Hessenberg matrix such that $m \ll n$. H_m and V_m constructed by Arnoldi's process can be used to solve 4.7 for different values of z in which the residual is given by 4.8. Since the residual is relatively easy to compute, the additional cost associated with monitoring the residual is small.

Now we can discuss how Arnoldi's process is used with RIM to produce RIM-C. Recall how we are approximating the indicator with the a parameterized system of 4.2 with $\overrightarrow{x_k}$ being a solution per k in $(A - z_k B)\, \overrightarrow{x_k} = \overrightarrow{f}$. We are able to solve across quadrature points z_k by choosing a proper shift - a non-generalized eigenvalue ψ. Define $M = (A - \psi B)^{-1} B$ and $\overrightarrow{b} = (A - \psi B)^{-1} \overrightarrow{f}$. From 4.3, we have:

$$(I_n + (\psi - z_k)\, M)\, \overrightarrow{x_k} = \overrightarrow{b}$$

Further from 4.6 and 4.7 we have:

$$\overrightarrow{y_k} = \beta \left(I_m + (\psi - z_k) H_m \right)^{-1} \overrightarrow{e_1} \tag{4.9}$$

$$\overrightarrow{x_k} \approx V_m \overrightarrow{y_k} \tag{4.10}$$

$$P \overrightarrow{f} \approx \frac{1}{2\pi i} \sum w_k V_m \overrightarrow{y_k} \tag{4.11}$$

Hence the Krylov subspace for M can be used to solve for $\overrightarrow{x_k}$ associated with quadrature points z_k which are near the shift ψ. Because we are reliant on z_k being near ψ, different Krylov subspaces may be used for differing z_k to evaluate $P_\Gamma(A) \overrightarrow{f}$.

Recall 4.1 how we projected the indicator onto a normalization of itself to scale the indicator. This created two linear systems with different right hand sides - the different representations being $\overrightarrow{f}$ and $\frac{P_\Gamma(A)}{\mathcal{P}_\Gamma(A)} \overrightarrow{f}$. To solve these linear systems for one shift ψ, we are constructing two Krylov subspaces in which a new approach to a more efficient indicator is proposed.

The idea behind this new indicator is to assume Γ is a square contour, and apply different sets of trapezoidal quadrature points that take advantage of Arnoldi's

exploitation of the Cayley transformation result 4.3. Because a circle can be circumscribed inside a square, we can use this quadrature for said circumscribed circle by how we choose to subdivide the initial Γ. It is stated that this method works well in practice without introducing extra "eigenvalues" [13].

Define $P_\Gamma(A)\overrightarrow{f}|_N$ as the approximation of $P_\Gamma(A)$ with N quadrature points. In section 4.6.5 of the work of Davis et al. [14], it is given that trapezoidal quadratures of a periodic function converges exponentially. That is for a constant C dependent on $\overrightarrow{f}$:

$$\|P_\Gamma(A)\overrightarrow{f} - \left(P_\Gamma(A)\overrightarrow{f}|_N\right)\|_2 = O\left(e^{-CN}\right)$$

Since this is an approximation, $P_\Gamma(A)\overrightarrow{f}|_N \neq 0$ still indicates that $\exists$ at least one $\lambda \in \Gamma$, and conversely $P_\Gamma(A)\overrightarrow{f}|_N \approx 0$ indicates all $\lambda \notin \Gamma$. For a sufficiently large N_0, we further have that:

$$\frac{\|P_\Gamma(A)\overrightarrow{f}|_{2N_0}\|_2}{\|P_\Gamma(A)\overrightarrow{f}|_{N_0}\|_2} = \begin{cases} \frac{\|P_\Gamma(A)\overrightarrow{f}\|_2 + O\left(e^{-C2N}\right)}{\|P_\Gamma(A)\overrightarrow{f}\|_2 + O(e^{-CN})} & \exists\ \lambda \in \Gamma \\ \frac{O\left(e^{-C2N}\right)}{O(e^{-CN})} = O\left(e^{-CN}\right) & \lambda \notin \Gamma \end{cases}$$

Hence this is our new indicator, which we will denote $\bar{\delta}_\Gamma = \frac{\|P_\Gamma(A)\overrightarrow{f}|_{2N_0}\|_2}{\|P_\Gamma(A)\overrightarrow{f}|_{N_0}\|_2}$. Based on

numerical examples presented in the work of Huang et al. [13], $\bar{\delta}_\Gamma > 0.2$ is said to indicate $\exists$ at least one $\lambda \in \Gamma$ where $\delta_0 = 0.2 \approx O\left(e^{-CN}\right)$ was chosen from experimentation, and it is stated that truly computing $O\left(e^{-CN}\right)$ in practice is difficult. The notation δ_0 will be useful when presenting the algorithm. Because of 4.9, 4.10, and 4.11, computing $\bar{\delta}_\Gamma$ is not expensive.

Now that we have outlined RIM and its algorithm, and the theory behind the improvements Cayley transformation and Arnoldi's method both offer to RIM, we can now outline the algorithm for RIM-C. Given Γ, we position our first shift ψ at Γ's center. Then $K_m\left(M, \overrightarrow{b}\right)$ is constructed and stored in memory. For an initial quadrature point z, the algorithm first attempts to solve the linear system given by 4.2. If the residual is greater than a given tolerance ϵ, another Krylov subspace with a new shift is constructed, stored, and then used to solve the linear system. In the next step, RIM-C constructs multiple Krylov subspaces for shifts corresponding to all differing quadrature points z_k, and uses them to solve each linear system dependent on z_k. Again because of 4.9, 4.10, and 4.11, we speed up solving this family of n linear systems for all z_k by solving a reduced family of m linear systems for most z_k. These improvements upon RIM is what yields RIM-C. The full algorithm for RIM-C can be found in the work of Huang et al. [13].

4.2 Proposing *newEuclidCond*

It is here that we can propose the algorithm for *newEuclidCond*. Since the goal is for use on any general matrix with no priori information, the only input required is a matrix $A \in \mathbb{C}^{m \times n}$, and the output being $\kappa_2(A)$. Consider the algorithm below:

Algorithm 4 *newEuclidCond*

```
if m ≠ n then
    κ2(A) = ∞
    End algorithm
end if

if A^H = A then                                   ▷ Find ‖A⁺‖2
    try
        e_min = |eigs(A, 1,' smallestabs')|
    catch ERR
        if ERR is singular matrix then
            e_min = 0
        else
            e_min = RIMCSearch(A)
        end if
    end try
else
    e_min = svds(A, 1,' smallest')
end if

if e_min = 0 then
    κ2(A) = ∞
    End algorithm
end if

if A^H = A then                                   ▷ Find ‖A‖2
    e_max = |eigs(A, 1,' largestabs')|
    if e_max = NaN then
        e_max = svds(A, 1,' largest)
    end if
else
    e_max = svds(A, 1,' largest')
end if

κ2(A) = e_max / e_min
```

Chapter 5

Results and Conclusion

In this section, we test the proposed *newEuclidCond* algorithm compared to commonly incorporated eigenvalue and singular value solvers for large linear systems - namely MATLAB's *eigs* and *svds*. As the purpose is to more reliably and efficiently compute κ_2 for any general matrix, the approach used will assume we have no priori information. It should also be noted that RIM-C is purely an eigenvalue solver - eigenvectors are not computed and so *eigs* is already disadvantaged having to spend more computational time to return a vector we do not need. Regardless, the following results can give comparable information on the abilities of each algorithm to converge to a result as well as the speed at which they perform. All tests are performed on the M2 Macbook Air with 8GB memory. Though not discussed, note that MATLAB's *cond* for the Euclidean norm has no estimate - it is exact, and *condest* is strictly for

the 1-norm.

5.1 Example 1: nemeth15

The *nemeth15* matrix, sourced from SuiteSparse Matrix Collection, is a 9506×9506 real, Hermitian matrix resulting from the 15th Newton-Schultz iteration. The matrix is said to have a Euclidean condition number $\kappa_2 = 1.116283E+02$ and is 0.5973638% dense. Below are the results of how each algorithm performed:

Table 5.1
κ_2 estimate of *nemeth15*

	time	κ_2 estimate
eigs	1.996167E±00	NaN
svds	1.874850E±00	NaN
cond	3.164562E+01	1.116283E+02
newEuclidCond	2.103470E±00	1.116183E+02

Here we see that *eigs* and *svds* took less time of the same magnitude just to output NaN. Upon closer inspection, *eigs* and *svds* both failed to converge to λ_{max} and σ_{max} respectively leading to NaN results. Though not faster, this is a case where *newEuclidCond* demonstrates its ability to better obtain an answer with no priori information where the additional time from *eigs* failing is relatively $\approx 21\%$ for *nemeth15* with relative estimate error $\approx 0.009\%$ from the provided value of κ_2.

5.2 Example 2: ramage02

The *ramage02* matrix, sourced from SuiteSparse Matrix Collection, is a 16830×16830 real, Hermitian matrix resulting from a finite element scheme applied to Navier Stokes and Continuity equations used in computational fluid dynamics. The matrix is said to have a Euclidean condition number $\kappa_2 = 2.705462E + 112$ and is 1.011955% dense. Below are the results of how each algorithm performed:

Table 5.2
κ_2 estimate of *ramage02*

	time	κ_2 estimate
eigs	2.036057E+01	Inf
svds	3.005157E+01	Inf
cond	1.821085E+02	7.031472E+286
newEuclidCond	2.147360E+01	Inf

Considering the *ramage02* matrix is real and has no infinite elements, we know that *eigs*, *svds*, and *newEuclidCond* are all returning that the matrix is singular. Specifically, *eigs* states *ramage02* is singular when attempting to find the smallest absolute λ, and does not return any value. Both *svds* and *newEuclidCond* returned $\lambda_{min} \approx 0$. Even *cond*'s estimate differs from the previous claim of κ_2's true value. This problem can likely be circumvented by adjusting the minimum tolerances in each algorithm accordingly to obtain non-zero λ_{min} and σ_{min} values, but again this report works under the assumption we have no priori information on A.

5.3 Example 3: Trefethen_20000

The *Trefethen_20000* matrix, sourced from SuiteSparse Matrix Collection, is a 20000× 20000 Hermitian matrix with prime elements listed along the main diagonal and 1's scattered following the construction from Problem 7 of the Hundred-dollar, Hundred-digit Challenge Problems, SIAM News vol 35 no. 1. It is stated that *Trefethen_20000* has a Euclidean condition number of $2.005593E+05$ with 0.138617% density. Table 4.3 displays the results of each algorithm:

Table 5.3
κ_2 estimate of *Trefethen_20000*

	time	κ_2 estimate
eigs	1.212770E+01	NaN
svds	1.608059E+02	NaN
cond	3.276611E+02	2.005593E+05
newEuclidCond	5.083751E+02	2.005593E+05

Here we have another instance where *eigs* and *svds* produce NaN results. Again, λ_{max} and σ_{max} are what causes *eigs* and *svds* respectively to return said results. And even though *newEuclidCond* obtained a result, this is an instance where the algorithm is terribly slow even compared to MATLAB's *cond*. This is due to how RIM-C requires a contour input. Note that aside from "startup time", RIM-C's time mostly correlates to the time required to run; using *newEuclidCond* to find κ_2 of a matrix with an absolutely large λ_{max} will result in longer runtime unless the algorithm to increase

the contour to find λ_{max} is itself optimized.

5.4 Example 4: mycielskian15

The *mycielskian15* matrix, sourced from SuiteSparse Matrix Collection, is a 24575×24575 Hermitian binary matrix representative of an undirected graph. The general set of *mycielskian* matrices share the characteristics of a known minimum of colors required to color the vertices without same-colored neighbors, and being triangle free. It is stated that *mycielskian15* has 1.839799% density while we are given no information of the expected κ_2. Below is how each algorithm performed:

Table 5.4
κ_2 estimate of *mycielskian15*

	time	κ_2 estimate
eigs	2.628998E+01	4.996015E+05
svds	3.932115E+02	4.996015E+05
cond	7.594096E+02	4.996015E+05
newEuclidCond	2.863092E+01	4.996015E+05

In this case, all algorithms obtained the same κ_2 to 7 significant figures and so we only wish to observe the time each algorithm took in which we see *eigs* was the fastest, and *newEuclidCond* being comparable. Since *newEuclidCond* attempts to run *eigs* first on Hermitian matrices to obtain λ_{min} before falling back to other algorithms, and attempting *eigs* or power iteration to obtain λ_{max}, it makes sense to assume that *newEuclidCond* in this case is running the same operations as *eigs* with extra checks to ensure this is the right decision. With *svds* and *cond* being on the next magnitude

in terms of time, there is little discussion to be had other than these algorithms are not preferable.

5.5 Example 5: case39

The *case39* matrix, sourced from SuiteSparse Matrix Collection, is a 40216×40216 real, Hermitian matrix used in an electrical power system. More importantly, this matrix is used to benchmark the algorithm presented by Quanyuan et al. to solve 2 linear systems sequentially with 2 right vectors. Again, we are not given κ_2 from the source. Note that. With *case39* having density $9.892869E-06\%$, below is how each algorithm performed:

Table 5.5
κ_2 estimate of *case39*

	time	κ_2 estimate
eigs	5.807436E±00	1.228018E+14
svds	6.850321E+02	Inf
cond	1.738608E-01	NaN
newEuclidCond	5.520929E±00	1.228018E+14

5.6 Example 6: raefsky3

The *raefsky3* matrix, sourced from SuiteSparse Matrix Collection, is a 21200×21200 real, non-Hermitian matrix used in computational fluid dynamics - though with no further information on the usage. The matrix is said to have a Euclidean condition number $\kappa_2 = 1.980662E + 11$ and is 0.331250% dense. Below is how each algorithm performed:

Table 5.6
κ_2 estimate of *raefsky3*

	time	κ_2 estimate
eigs	9.093270E-01	1.980627E+11
svds	9.997307E±00	Inf
cond	3.984497E+02	1.980627E+11
newEuclidCond	1.008368E+01	Inf

Here the results are seemingly surprising. Covered in Chapter 1, to use *eigs* on a non-Hermitian matrix, we need to input $A^H A$ which first requires computing the matrix product.

5.7 Example 7: human_gene2

The *human_gene2* matrix, sourced from SuiteSparse Matrix Collection, is a 14340×14340 real, Hermitian matrix representing a human gene regulatory network. The matrix is said to have a Euclidean condition number $\kappa_2 = 3.833767E + 05$ and is $8.786605E \pm 00\%$ dense. Below is how each algorithm performed:

Table 5.7
κ_2 estimate of *human_gene2*

	time	κ_2 estimate
eigs	6.535021E+01	3.833767E+05
svds	4.769057E+01	3.833767E+05
cond	1.172534E+02	3.833767E+05
newEuclidCond	6.373140E+01	3.833767E+05

Here we can see that all algorithms got the correct value of κ_2 to seven significant figures. Though *svds* computed κ_2 the fastest, *newEuclidCond* was able to come in second place with a time comparable to *eigs*.

5.8 Example 8: windtunnel_evap3d

The *windtunnel_evap3d* matrix, sourced from SuiteSparse Matrix Collection, is a 40816×40816 real, non-Hermitian matrix containing data from a simulation of soil-water evaporating from a water-filled sand box to the air via a pipe. The condition number is not given from the source, but the matrix is found to be $4.825955E-02\%$ dense. Below is how each algorithm performed:

Table 5.8
κ_2 estimate of *windtunnel_evap3d*

	time	κ_2 estimate
eigs	5.130059E+02	8.335535E+13
svds	6.160788E+02	Inf
cond	1.490372E-01	NaN
newEuclidCond	6.100461E+02	Inf

Here, *eigs* yielded errors hinting at the poor condition of the matrix, but was the only algorithm to yield a numeric answer while being faster than *svds* and *newEuclidCond*. Following from the previous matrix, this places *newEuclidCond* in second again as it was roughly six seconds faster than *svds*.

5.9 Example 9: bayer01

The *bayer01* matrix, sourced from SuiteSparse Matrix Collection, is a 57735×57735 real, non-Hermitian matrix with no given information to its usage; the source also does not provide an estimate for κ_2. The matrix can be found to be $8.252827E-03\%$ dense. Below is how each algorithm performed:

Table 5.9
κ_2 estimate of *bayer01*

	time	κ_2 estimate
eigs	1.452240E+00	Inf
svds	3.493545E-01	Inf
cond	5.583635E-02	NaN
newEuclidCond	3.134055E-01	Inf

We can see that aside from *cond* not having enough memory to complete the computation, *newEuclidCond* found the unanimous estimation $\kappa_2 = \text{Inf}$ in the shortest amount of time with *svds* requiring about 0.03 more seconds, and *eigs* at least able to output a usable answer. Unfortunately, not a single algorithm tested was able to obtain κ_2 numerically and resorting to assuming $\sigma_{min} = 0$.

5.10 Example 10: c-67

The *c-67* matrix, sourced from SuiteSparse Matrix Collection, is a 57975×57975 real, Hermitian matrix with no given information to its usage nor κ_2 approximation. The matrix is found to be $1.577546E-02$ dense. Below is how each algorithm performed:

Table 5.10
κ_2 estimate of *c-67*

	time	κ_2 estimate
eigs	4.405881E+00	1.931770E+13
svds	4.751642E+02	Inf
cond	9.391576E-02	NaN
newEuclidCond	4.599063E+00	1.931770E+13

Here, *newEuclidCond* is competative with *eigs* being roughly two tenths of a second slower while obtaining the same approximation. Meanwhile, *svds* took roughly one hundred times longer while yielding Inf results.

5.11 Example 11: crankseg_2

The *crankseg_2* matrix, sourced from SuiteSparse Matrix Collection, is a 63838×63838 real, Hermitian matrix with no given information to its usage nor κ_2 approximation. The matrix is found to be $3.471865E-01\%$ dense. Below is how each algorithm performed:

Table 5.11
κ_2 estimate of *crankseg_2*

	time	κ_2 estimate
eigs	3.015859E+00	4.823690E+07
svds	5.334748E+01	4.823690E+07
cond	6.861389E-03	NaN
newEuclidCond	3.244273E+00	4.823690E+07

Here, the three approximation algorithms correctly determined κ_2 with *eigs* being the fastest by two tenths of a second compared to *newEuclidCond*.

5.12 Example 12: ACTIVSg70K

The *ACTIVSg70K* matrix, sourced from SuiteSparse Matrix Collection, is a 69999×69999 real, Hermitian matrix which holds data representing a synthetic electric grid which is statistically and functionally similar to that of a real electric grid while containing no confidential critical energy infrastructure information. The source does not provide an approximation for κ_2, but the matrix can be found to be $4.870078E-03\%$ dense. Below is how each algorithm performed:

Table 5.12
κ_2 estimate of *ACTIVSg70K*

	time	κ_2 estimate
eigs	1.502308E-01	2.169271E+08
svds	3.128693E-01	2.169271E+08
cond	1.965708E-03	NaN
newEuclidCond	1.329327E-01	2.169271E+08

Here, each algorithm was able to obtain the same κ_2 to seven significant figures where *newEuclidCond* is shown to be the fastest; we disregard the NaN results of *cond*. However, *newEuclidCond* is faster than *eigs* by the small margin of less than two hundredths of a second.

5.13 Example 13: consph

The *consph* matrix, sourced from SuiteSparse Matrix Collection, is a 83334×83334 real, Hermitian matrix which was used to benchmark matrix multiplication on graphics hardware in 2008 by NVIDIA. The source does not provide an approximation for κ_2, but yields enough information to find that the matrix is $8.654953E-02\%$ dense. Below is how each algorithm performed:

Table 5.13
κ_2 estimate of *consph*

	time	κ_2 estimate
eigs	9.015638E±00	9.398337E+06
svds	3.609578E+02	9.398337E+06
cond	7.924231E-02	NaN
newEuclidCond	8.728722E±00	9.398337E+06

To seven significant figures, all algorithms but *cond* found the same estimate of κ_2. We also observe that *newEuclidCond* was again the fastest algorithm, outperforming *eigs* by roughly three tenths of a second. *svds* however took well over three hundred seconds while *eigs* and *newEuclidCond* both stayed under ten seconds.

5.14 Example 14: rajat16

The *rajat16* matrix, sourced from SuiteSparse Matrix Collection, is a 94294×94294 real, non-Hermitian matrix with no information given as to its usage nor an estimation of κ_2. The matrix can be found to be $5.362128E-03\%$ dense. Below is how each algorithm performed:

Table 5.14
κ_2 estimate of *rajat16*

	time	κ_2 estimate
eigs	1.042170E+03	3.757711E+12
svds	6.289650E+02	Inf
cond	1.234742E-01	NaN
newEuclidCond	6.367650E+02	Inf

Here, *eigs* was the only algorithm to return a numeric approximation while taking the most time to run. *svds* and *newEuclidCond* each took less time to run comparatively, but both returned Inf results indicating the assumption of $\sigma_{min} = 0$ was utilized.

5.15 Example 15: Goodwin_095

The *Goodwin_095* matrix, sourced from SuiteSparse Matrix Collection, is s 100037×100037 real, non-Hermitian matrix. The data represents a finite element discretization of the Navier-Stokes (and similar transport equations) on geometries provided by Ralph Goodwin. The source does not provide an approximation to κ_2 but it can be found that the matrix is $3.223680E-02\%$ dense. Below is how each algorithm performed:

Table 5.15
κ_2 estimate of *Goodwin_095*

	time	κ_2 estimate
eigs	4.185872E+00	2.265450E+07
svds	9.049208E+00	2.265451E+07
cond	2.468913E-02	NaN
newEuclidCond	9.052039E+00	2.265451E+07

Here we see that *svds* and *newEuclid* performed near identical with a time difference in the thousandths of seconds yet the same κ_2 to seven significant figures. However, *eigs* was able to agree with the other approximations to six of seven significant figures while performing a little under five seconds faster.

5.16 Conclusion

First note that *cond* was never meant to be a competing algorithm but rather a way to confidently ensure κ_2. With *cond* having given NaN results nine times, we will leave this algorithm out of the discussion.

In terms of speed, *newEuclidCond* performed faster than both *eigs* and *svds* in four of the tested matrices (*case39*, *bayer01*, *ACTIVSg70K*, and *consph*). Of the eleven remaining matrices, *newEuclidCond* was able to be faster than either *eigs* or *svds* seven times (*ramage02*, *mycielskian15*, *human_gene2*, *windtunnel_evap3d*, *c-67*, *crankseg_2*, *rajat16*). This means in eleven matrices, *newEuclidCond* was able to secure at least second place with respect to only time - even including cases where the other two algorithms simply failed.

Trivially numerical outputs are valid; define an output of Inf as valid, and only consider matrices where the three algorithms being observed yield valid results (*ramage02*, *mycielskian15*, *case39*, *raefsky3*, *human_gene2*, *windtunnel_evap3d*, *bayer01*, *c-67*, *crankseg_2*, *ACTIVSg70K*, *consph*, *rajat16*, *Goodwin_095*). If we sum each algorithm's time across all these matrices, we end up with the following table:

Table 5.16
Total Time Across Valid Outputs

	time
eigs	1.696119E+03
svds	3.310208E+03
newEuclidCond	1.402322E+03

The table above implies that *newEuclidCond* would be able to find κ_2 of the thirteen previously listed matrices faster than both *eigs* and *svds* where all three algorithms would run to completion (IE yield a valid result).

With reliability being a less straightforward goal, a quick discussion for each of the fifteen matrices will be provided:

1. *nemeth15*: *newEuclidCond* was the only non-exact algorithm to approximate κ_2. The approximation is correct up to four significant figures.

2. *ramage02*: None of the algorithms obtained the correct κ_2 to even one significant figure. It will be stated that $\kappa_2 = 2.705462E + 112$ represents a very poorly conditioned matrix. Depending on the application, Inf may be sufficient.

3. *Trefethen_20000*: Again *newEuclidCond* is the only non-exact algorithm to approximate κ_2. The approximation is correct to seven significant figures.

4. *mycielskian15*: Though the source did not provide κ_2, all four algorithms agree on the same value to seven significant figures.

5. *case39*: *eigs* and *newEuclidCond* are the only two algorithms to output a numeric approximation, and agree. *svds* yielding Inf may suffice dependent on application.

6. *raefsky3*: *newEuclidCond* failed to find a numeric approximation while *eigs* succeeded. *svds* and *newEuclidCond* agree on Inf.

7. *human_gene2*: All four algorithms agree on the value of κ_2 to seven significant figures.

8. *windtunnel_evap3d*: *eigs* was the only algorithm to obtain κ_2 numerically, and again *svds* agrees with *newEuclidCond* on Inf.

9. *bayer01*: All three non-exact algorithms agree on an output of Inf where we cannot use the source nor *cond* to confirm $\sigma_{min} = 0$.

10. *c-67*: *eigs* and *newEuclidCond* agree on the same numeric value to seven significant figures where *svds* yields Inf.

11. *crankseg_2*: All three non-exact algorithms agree on κ_2 numerically.

12. *ACTIVSg70K*: All three non-exact algorithms agree on κ_2 numerically.

13. *consph*: All three non-exact algorithms agree on κ_2 numerically.

14. *rajat16*: *eigs* was the only algorithm to obtain κ_2 numerically, and again *svds* agrees with *newEuclidCond* on Inf.

15. *Goodwin_095*: *svds* and *newEuclidCond* agree on κ_2 to seven significant figures. All three non-exact algorithms agree on κ_2 to six significant figures.

If we define reliable as all algorithms agreeing on κ_2 numerically or *newEuclidCond* agreeing with the source (or with simply *cond* if the source does not provide such information), both to most of the significant figures, we say *newEuclidCond* is reliable in eight of the tested matrices. Holding *eigs* to this standard, it is reliable in seven of the matrices, and likewise *svds* is reliable in six of the tested matrices. It will be left as a note that this is a rather strict definition of reliability, and more testing would have to be done to deny false negatives (IE *c-67* counts *eigs* and *newEuclidCond* as unreliable since κ_2 wasn't given and *svds* didn't numerically agree with both *eigs* and *newEuclidCond*; also potential cases where $\sigma_{min} = 0$ may be true implying $\kappa_2 = \text{Inf}$ is reliable), or potentially false positives.

Overall, *newEuclidCond* has been shown to be faster with select matrices, and faster in bulk testing (when excluding NaN results). *newEuclidCond* also has been shown to be competitively reliable against *eigs*. And though *svds* performed better on select matrices, its overall performance is poor compared to the other non-exact algorithms.

References

[1] Braatz, R. D.; Morari, M. *Minimizing the Euclidean Condition Number*, Vol. 32; 1994.

[2] Ye, Q. *Numerical Linear Algebra with Applications* **2020**, *27*(4), e2315.

[3] Lambers, J.; Sumner, A. *Explorations in Numerical Analysis;* WORLD SCIENTIFIC, 2018.

[4] Cline, A. K.; Moler, C. B.; Stewart, G. W.; Wilkinson, J. H. *An estimate for the condition number of a matrix*, Vol. 16; SIAM, 1979.

[5] Lehoucq, R. B.; Sorensen, D. C.; Yang, C. *ARPACK users' guide*, Vol. 6 of *Software, Environments, and Tools;* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[6] Stewart, G. W. *Addendum to "A Krylov–Schur Algorithm for Large Eigenproblems"*, Vol. 24; 2002.

[7] Baglama, J.; Reichel, L. *Augmented implicitly restarted Lanczos bidiagonalization methods*, Vol. 27; 2005.

[8] Larsen, R. *Lanczos Bidiagonalization With Partial Reorthogonalization*, Vol. 27; 1999.

[9] Schott, James, R. pages 175–178.

[10] Kokiopoulou, E.; Bekas, C.; Gallopoulos, E. *Applied Numerical Mathematics* **2004**, *49*(1), 39–61.

[11] Saad, Y. *Numerical Methods for Large Eigenvalue Problems: Revised Edition*, Classics in Applied Mathematics; Society for Industrial and Applied Mathematics, 2011.

[12] Saad, Y. *Iterative Methods for Sparse Linear Systems*, Other Titles in Applied Mathematics; Society for Industrial and Applied Mathematics, second ed., 2003.

[13] Huang, R.; Sun, J.; Yang, C. *Recursive integral method with Cayley transformation*, Vol. 25; 2018.

[14] Davis, P.; Rabinowitz, P.; Rheinbolt, W. *Methods of Numerical Integration*, Computer Science and Applied Mathematics; Elsevier Science, 2014.

[15] Huang, R.; Struthers, A. A.; Sun, J.; Zhang, R. *Recursive integral method for transmission eigenvalues*, Vol. 327; Elsevier BV, 2016.

[16] Huang, R.; Sun, J.; Yang, C. July **2018**, *25*(6).

Appendix A

Code 1: *newEuclidCond*

The code for the proposed algorithm *newEuclidCond* is provided as below:

A.1 newEuclidCond.m

```
function C = newEuclidCond(A)

    % Constants
    global MAT_A;
    global MAT_SIZE;
    MAT_A = A;
    clear A; % Free up memory - don't double store
    MAT_SIZE = size(MAT_A);
    E_MIN_TOL = 10^(-13);
```

```
IS_HERMITIAN = false;

% Ensure we don't attempt to find singular matrix ↩
   condition number
if MAT_SIZE(1,1) ~= MAT_SIZE(1,2)
    C = Inf;
    fprintf('\nInputted matrix is non-square and ↩
       therefore singular!\n')
    return;
end

% Ensure A is sparse
if not(issparse(MAT_A))
    MAT_A = sparse(MAT_A);
end

% Determine Hermitian
IS_HERMITIAN = ishermitian(MAT_A);

% Find |\lambda _{min}|: A is Hermitian
if IS_HERMITIAN

    % Try using eigs since it is fastest; resort to ↩
       RIM-C if fails
    try
        E_min = abs(eigs(MAT_A,1,'smallestabs'));
    catch ERR
        if strcmp(ERR.identifier,'MATLAB:eigs:↩
           SingularA') % If error thrown is singular ↩
           matrix
            E_min = 0;
        else
            fprintf('\nUnidentified error from "eigs↩
               "! Using RIM-C:\n')
```

```
            E_min = RIMCSearch();
        end
    end

    % Determine if |\lambda _{min}| = 0
    if E_min <= E_MIN_TOL
        C = Inf;
        fprintf('\nInputted matrix is detected to be↩
            singular!\n')
        return;
    end

% A is non-Hermitian: Find \sigma _{min}
else
    E_min = svds(MAT_A,1,'smallest');

    % Determine if \sigma _{min} = 0
    if E_min <= E_MIN_TOL
        C = Inf;
        fprintf('\nERROR: MATLAB "svds" returned  a ↩
           "0" singular value!\n')
        return;
    end
end

% Find |\lambda _{max}|
if IS_HERMITIAN
    warning('off')
    E_max = abs(eigs(MAT_A,1,'largestabs'));
    warning('on')
    if isnan(E_max) % 'eigs' failed to find eig
        %envalue
        E_max = powerIter();
    end
```

```
    % A is asymmetric: Find \sigma _{max}
    else
        E_max = svds(MAT_A,1,'largest');
    end

    % Get final result
    C = E_max / E_min;

end

%% Standard Power Iteration
function curr_E = powerIter()

    global MAT_A;
    global MAT_SIZE;
    mat_size = MAT_SIZE(1,1);
    prev_eVec = rand(mat_size,1) + (1i .* rand(mat_size↩
       ,1));
    curr_eVec = prev_eVec;
    prev_E = 1;
    curr_E = 0;
    iterations = 100;

    while abs(prev_E - curr_E) >= 10^(-6)
        prev_E = curr_E;
        for i=1:1:iterations
            curr_eVec = (MAT_A*curr_eVec) ./ (norm(MAT_A↩
               *curr_eVec));
        end
        curr_E = abs(mean((MAT_A*curr_eVec)./curr_eVec))↩
           ;

        if isnan(curr_E) % If NaN result, reset and try ↩
           again with less iterations
```

```
            if iterations == 1
                curr_E = prev_E;
                return;
            else
                iterations = ceil(iterations / 2);
                curr_eVec = prev_eVec;
                curr_E = prev_E;
                prev_E = curr_E + 1;
            end
        else
            prev_eVec = curr_eVec;
        end

    end

end

%% RIM-C Search Algorithm
function E = RIMCSearch()

    global MAT_B;
    global MAT_SIZE;
    MAT_B = speye(MAT_SIZE);
    s = [-(1/sqrt(8));-(1/sqrt(8));(1/sqrt(8));(1/sqrt↩
       (8))]; % Initial search region with area 1/2
    E = [];
    vecSize = size(E);
    while vecSize(1) == 0
        E = RIMCv0(s);
        s = 2 .* s;
        vecSize = size(E);
    end
    E = min(abs(E));
```

```
end

%% RIM-C Code
function E = RIMCv0(s)
% The code computes generalized eigenvalues of
%          Ax = lambda Bx
% in a rectangle s on the complex plane.
% Input:
%        A    -- N x N matrix {GLOBAL}
%        R    -- N x N matrix {GLOBAL}
%        s    -- 4 x 1 vector [xmin ymin xmax ymax]
% Output: l
%        lambda -- generalized eigenvalues in s
% Jiguang Sun, 05/09/2017, jiguangs@mtu.edu
% Please report bugs to jiguangs@mtu.edu
% Copyright (c) 2017, Jiguang Sun, all rights reserved.
% THIS SOFTWARE IS PROVIDED "AS IS".
% Redistribution and use in source and binary forms, ↩
   with or without
% modification, for academic purpose only are permitted.
% References:
% 1. J.Sun and A.Zhou, Finite Element Methods for ↩
   Eigenvalue Problems, CRC Press, 2016.
% 2. R.Huang, A.Struthers, J.Sun and R.Zhang,
%    Recursive integral method for transmission ↩
   eigenvalues,
%    Journal of Computational Physics, Vol. 327, ↩
   830-840, 2016.
% 3. R Huang, J Sun, C Yang,
%    Recursive Integral Method with Cayley ↩
   Transformation
%     - arXiv preprint arXiv:1705.01646, 2017 - arxiv.↩
   org
```

```
l1=s(1); r1=s(2); l2=s(3); r2=s(4);
tol = 1.0e-12;
size_Krylov_space=50;
delta_len = min(min((l2-l1)/50,(r2-r1)/50),0.04);


global f_test;
global MAT_A;
N = length(MAT_A);
f_test=rand(N,1)+2*exp(1i*rand(N,1));
Square_point=zeros(10000,5);
Selected_point=zeros(10000,5);
final_point=zeros(10000,1);
tol_robust=0.1*tol;

global p;
global p_test;
p↩
   =[0,1/2,1/4,3/4,1/8,5/8,3/8,7/8,1/16,9/16,5/16,13/16,3/16,11/1

p_test=[0,1/2,1/4,3/4,1/8,5/8,3/8,7/8];

f_test=f_test/norm(f_test);
len_l=l2-l1; len_r=r2-r1;
size_length=ceil(len_l/delta_len);
size_width=ceil(len_r/delta_len);

global k;
global number_reduce_sys;
global V_h;
global D_h;
global V_h_inv_v;
global H_para;
global shift_phi;
```

```
k=size_Krylov_space;
V_h=zeros(k,k,100);
D_h=zeros(k,100);
V_h_inv_v=zeros(k,100);
H_para=zeros(1,100);
shift_phi=zeros(1,100);
number_reduce_sys=0;

for i=1:size_length
    for j=1:size_width
        Square_point((j-1)*size_length+i,1)=l1+(i-1)↩
           *delta_len;
        Square_point((j-1)*size_length+i,2)=r1+(j-1)↩
           *delta_len;
        Square_point((j-1)*size_length+i,3)=l1+i*↩
           delta_len;
        Square_point((j-1)*size_length+i,4)=r1+j*↩
           delta_len;
    end
end
Square_number=size_length*size_width;

for i=1:Square_number
    ind=[i-1,i-size_length,i-size_length+1,i-↩
       size_length-1];
    ind=ind(ind>0);
    if isempty(ind)==1 || number_reduce_sys == 0
        shift=(Square_point(i,1)+Square_point(i,3))↩
           /2+1i*(Square_point(i,2)+Square_point(i,4)↩
           )/2;
        arnoldi(shift);
        Square_point(i,5)=number_reduce_sys;
    else
```

```
        flag=check_ind(Square_point(ind,5),↩
           Square_point(i,1:4),tol_robust);
        if (flag~=0)
            Square_point(i,5)=flag;
        else
            shift=(Square_point(i,1)+Square_point(i↩
               ,3))/2+1i*(Square_point(i,2)+↩
               Square_point(i,4))/2;
            arnoldi(shift);
            Square_point(i,5)=number_reduce_sys;
        end
    end
end

tol_iter=ceil(log2(delta_len/tol_robust));

for i=1:tol_iter
    Selected_number=0;
    for j=1:Square_number
        if (Check_square(Square_point(j,:))==1)
            Selected_number= Selected_number+1;
            Selected_point(Selected_number,:)=↩
               Square_point(j,:);
        end
    end
    Square_number=Selected_number*4;
    for j=1:Selected_number
        l1=Selected_point(j,1);
        r1=Selected_point(j,2);
        l2=Selected_point(j,3);
        r2=Selected_point(j,4);
        index=Selected_point(j,5);
```

```
            Square_point((j-1)*4+1,1:5)=[l1,r1,(l1+l2)↩
               /2,(r1+r2)/2,index];
            Square_point((j-1)*4+2,1:5)=[(l1+l2)/2,r1,l2↩
               ,(r1+r2)/2,index];
            Square_point((j-1)*4+3,1:5)=[(l1+l2)/2,(r1+↩
               r2)/2,l2,r2,index];
            Square_point((j-1)*4+4,1:5)=[l1,(r1+r2)/2,(↩
               l1+l2)/2,r2,index];
        end
    end

    Num_Eigenvalue=0;
    for i=1:Selected_number
        if (Selected_point(i,5)~=0)
            tmp=Selected_point(i,1:4);
            num=1;
            for j=1:Selected_number
                if (norm(Selected_point(i,1:4)-↩
                   Selected_point(j,1:4))<16*tol)
                    num=num+1;
                    tmp=tmp+Selected_point(j,1:4);
                    Selected_point(j,5)=0;
                end
            end
            Num_Eigenvalue=Num_Eigenvalue+1;
            tmp=tmp./num;
            final_point(Num_Eigenvalue)=(tmp(1)+tmp(3))↩
               /2+1i*(tmp(2)+tmp(4))/2;
        end
    end

    E=final_point(1:Num_Eigenvalue);
end
```

```
function flag = Check_square(point)
    flag=0;
    center=(point(1)+point(3))/2+1i*(point(2)+point(4))↩
       /2;
    radius=sqrt((real(center)-point(1))^2+(imag(center)-↩
       point(2))^2);
    ind=point(5);

    global p;
    global k;
    global D_h;
    global V_h_inv_v;
    global shift_phi;

    Para_p=center+radius*exp(1i*2*pi*p);
    iter=[0,2,4,8,16];
    w=[1/2,1/4,1/8,1/16];
    res=zeros(1,4);
    u=rand(k,1);
    level = 3;
    for i=1:level
        for j=iter(i)+1:iter(i+1)
            tmp_v=u'*((V_h_inv_v(:,ind)./(ones(k,1)+(↩
               shift_phi(ind)-Para_p(j))*D_h(:,ind))));
            res(i)=res(i)+w(i)*(Para_p(j)-center)*tmp_v;
        end
        res(i+1)=res(i)*1/2;
    end

    if (abs(norm(res(level-1))/norm(res(level)))<15)
        flag=1;
    end
end
```

```
function ind = check_ind(index,point,tol)

    global p_test;
    global k;
    global V_h;
    global D_h;
    global V_h_inv_v;
    global H_para;
    global shift_phi;

    center=(point(1)+point(3))/2+1i*(point(2)+point(4))↩
       /2;
    radius=sqrt((real(center)-point(1))^2+(imag(center)-↩
       point(2))^2);
    Para_p=center+radius*exp(1i*2*pi*p_test);

    ind=0;
    for i=1:length(index)
        flag=1;
        for j=1:length(Para_p)
            if abs(H_para(index(i))*V_h(k,:,index(i))*(↩
               V_h_inv_v(:,index(i))./((ones(k,1)+(↩
               shift_phi(index(i))-Para_p(j))*D_h(:,index↩
               (i))))))>tol
                flag=0;
            end
        end
        if flag==1
            ind=index(i);
            break;
        end
    end
end
```

```
function arnoldi(phi)

    global f_test;
    global k;
    global number_reduce_sys;
    global V_h;
    global D_h;
    global V_h_inv_v;
    global H_para;
    global shift_phi;
    global MAT_A;
    global MAT_B;

    I=sparse(k,1);
    I(1)=1;
    number_reduce_sys=number_reduce_sys+1;
    [L,U,P,Q]=lu(MAT_A-phi*MAT_B);
    b=Q*(U\(L\(P*f_test)));
    V1=zeros(length(MAT_A),k+1);
    H1=zeros(k+2,k+1);
    V1(:,1)=b/norm(b);
    for i=1:k
        v1=Q*(U\(L\(P*(MAT_B*V1(:,i)))));
        for j=1:i
            H1(j,i)=V1(:,j)'*v1;
            v1=v1-H1(j,i)*V1(:,j);
        end
        H1(i+1,i)=norm(v1);
        V1(:,i+1)=v1/H1(i+1,i);
    end
    [V,D]=eig(H1(1:k,1:k));
    V_h(:,:,number_reduce_sys)=V;
    D_h(:,number_reduce_sys)=diag(D);
    V_h_inv_v(:,number_reduce_sys)=V\I;
```

```
    shift_phi(number_reduce_sys)=phi;
    H_para(number_reduce_sys)=H1(k+1,k);
end
```

Appendix B

Code 2: Test Each Method's Euclidean Norm Approximations

Each of the methods mentioned throughout this paper - RIM-C, *eigs*, *svds*, and *normest* - have been tested on a collection of matrices from public repositories such as SuiteSparse Matrix Collection to test each method's efficacy and efficiency. The information from these tests was used to determine the algorithm for *newEuclidCond*.

Provided below is the code used to test each method to determine how *newEuclidCond* should best be formulated:

B.1 valueTimings.m

```
% Clean workspace to clear memory
close all;
clear all;
clc;

% Inform user of format
fprintf("Format: Method - time for ||A|| & value - time ↩
   for ||A^{+}|| & value if applicable\n\n")

% Vector of matrices to test; negligable to memory
mats = [
    "nemeth15.mat"
    "ramage02.mat"
    "Trefethen_20000.mat"
    "mycielskian15.mat"
    "case39.mat"
    "raefsky3.mat"
    ];

% Initialize time & estimate variables to be averaged
currTime = 0;
currEst = 0;

% Iterate through each matrix testing each method
for i=1:length(mats)
    %% Setup
    % Load matrix, store, & clear uneeded data
    load(mats(i));
    mat = Problem.A;
```

```
clear Problem;

% Tell user which matrix is being ran
fprintf("Matrix %s results:\n\n", mats(i))

%% normest
% sigma_{max}
tic
warning('off')
currEst = normest(mat) + normest(mat) + normest(mat)↩
   ;
warning('on')
currTime = toc;
fprintf("normest time: %.6E\nnormest approximation ↩
   sigma_{max}: %.6E\n\n", currTime/3, currEst/3)
% Reset
currTime = 0;
currEst = 0;

%% eigs
% sigma_{max}
tic
warning('off')
currEst = eigsHelper(mat, 1) + eigsHelper(mat, 1) + ↩
   eigsHelper(mat, 1);
warning('on')
currTime = toc;
fprintf("eigs time: %.6E\neigs approximation sigma_{↩
   max}: %.6E\n", currTime/3, currEst/3)
% Reset
currTime = 0;
currEst = 0;
% sigma_{min}
tic
```

```
warning('off')
currEst = eigsHelper(mat, 2) + eigsHelper(mat, 2) + ↩
   eigsHelper(mat, 2);
warning('on')
currTime = toc;
fprintf("eigs time: %.6E\neigs approximation sigma_{↩
   min}: %.6E\n\n", currTime/3, currEst/3)
% Reset
currTime = 0;
currEst = 0;

%% svds
% sigma_{max}
tic
warning('off')
currEst = svds(mat, 1, 'largest') + svds(mat, 1, '↩
   largest') + svds(mat, 1, 'largest');
warning('on')
currTime = toc;
fprintf("svds time: %.6E\nsvds approximation sigma_{↩
   max}: %.6E\n", currTime/3, currEst/3)
% Reset
currTime = 0;
currEst = 0;
% sigma_{min}
tic
warning('off')
currEst = svds(mat, 1, 'smallest') + svds(mat, 1, '↩
   smallest') + svds(mat, 1, 'smallest');
warning('on')
currTime = toc;
fprintf("svds time: %.6E\nsvds approximation sigma_{↩
   min}: %.6E\n\n", currTime/3, currEst/3)
% Reset
```

```
    currTime = 0;
    currEst = 0;

    %% RIM-C
    tic
    warning('off')
    currEst = RIMCSearch(mat) + RIMCSearch(mat) + ↩
       RIMCSearch(mat);
    warning('on')
    currTime = toc;
    fprintf("RIM-C time: %.6E\nRIM-C approximation ↩
       sigma_{min}: %.6E\n\n\n", currTime/3, currEst/3)
    % Reset
    currTime = 0;
    currEst = 0;
end

%% Helper Functions
function eigVal = eigsHelper(inMat, target)
matSize = size(inMat);

% Obtain desired eigenvalue
if target == 1 % If wanting sigma_{max}
    if ishermitian(inMat) % Eigen values are singular ↩
       values
        eigVal = abs(eigs(inMat, 1, 'largestabs'));
        return;
    else % Eigen values are squared singular values
        if matSize(1) >= matSize(2) % Tall skinny matrix
            eigVal = sqrt(eigs(inMat'*inMat, 1, '↩
               largestabs'));
            return;
        else
            % Short fat matrix
```

```
            eigVal = sqrt(eigs(inMat*inMat', 1, '↩
               largestabs'));
            return;
        end
    end
else % If wanting sigma_{min}
    if ishermitian(inMat)
        % Get min eigenvalue; possibly 0
        try
            eigVal = abs(eigs(inMat, 1, 'smallestabs'));
        catch ERR
            if strcmp(ERR.identifier,'MATLAB:eigs:↩
               SingularA') % If error thrown is singular ↩
               matrix
                eigVal = 0;
                return;
            else % Unidentified and unhandleable error; ↩
               NaN
                eigVal = NaN;
                return;
            end
        end
    else
        if matSize(1) >= matSize(2) % Tall skinny matrix
            % Get min eigenvalue; possibly 0
            try
                eigVal = sqrt(eigs(inMat'*inMat, 1, '↩
                   smallestabs'));
            catch ERR
                if strcmp(ERR.identifier,'MATLAB:eigs:↩
                   SingularA') % If error thrown is ↩
                   singular matrix
                    eigVal = 0;
                    return;
```

```
                else % Unidentified and unhandleable ↩
                    error; NaN
                     eigVal = NaN;
                     return;
                end
            end
        else % Short fat matrix
            % Get min eigenvalue; possibly 0
            try
                eigVal = sqrt(eigs(inMat*inMat', 1, '↩
                    smallestabs'));
            catch ERR
                if strcmp(ERR.identifier,'MATLAB:eigs:↩
                    SingularA') % If error thrown is ↩
                    singular matrix
                     eigVal = 0;
                     return;
                else % Unidentified and unhandleable ↩
                    error; NaN
                     eigVal = NaN;
                     return;
                end
            end
        end
    end
end
end

% RIM-C Code here & all below
% Find minimal eigenvalue
function E = RIMCSearch(inMat)

    global MAT_A;
    global MAT_B;
```

```
    mat_size = size(inMat);
    matHermitian = ishermitian(inMat);

    if not(matHermitian)
        if mat_size(1) >= mat_size(2)
            MAT_A = inMat'*inMat;
        else
            MAT_A = inMat*inMat';
        end
    else
        MAT_A = inMat;
    end


    mat_size = size(inMat);
    MAT_B = speye(mat_size);
    s = [-(1/sqrt(8));-(1/sqrt(8));(1/sqrt(8));(1/sqrt↩
       (8))]; % Initial search region with area 1/2
    E = [];
    vecSize = size(E);
    while vecSize(1) == 0
        E = RIMCv0(s);
        s = 2 .* s;
        vecSize = size(E);
    end
    clear global;
    E = min(abs(E));

    if not(matHermitian)
        E = sqrt(E);
    end
end

function E = RIMCv0(s)
```

```
% The code computes generalized eigenvalues of
%         Ax = lambda Bx
% in a rectangle s on the complex plane.
% Input:
%        A    -- N x N matrix {GLOBAL}
%        R    -- N x N matrix {GLOBAL}
%        s    -- 4 x 1 vector [xmin ymin xmax ymax]
% Output: l
%        lambda -- generalized eigenvalues in s
% Jiguang Sun, 05/09/2017, jiguangs@mtu.edu
% Please report bugs to jiguangs@mtu.edu
% Copyright (c) 2017, Jiguang Sun, all rights reserved.
% THIS SOFTWARE IS PROVIDED "AS IS".
% Redistribution and use in source and binary forms, ↩
  with or without
% modification, for academic purpose only are permitted.
% References:
% 1. J.Sun and A.Zhou, Finite Element Methods for ↩
  Eigenvalue Problems, CRC Press, 2016.
% 2. R.Huang, A.Struthers, J.Sun and R.Zhang,
%    Recursive integral method for transmission ↩
  eigenvalues,
%    Journal of Computational Physics, Vol. 327, ↩
  830-840, 2016.
% 3. R Huang, J Sun, C Yang,
%    Recursive Integral Method with Cayley ↩
  Transformation
%     - arXiv preprint arXiv:1705.01646, 2017 - arxiv.↩
  org

    l1=s(1); r1=s(2); l2=s(3); r2=s(4);
    tol = 1.0e-12;
    size_Krylov_space=50;
    delta_len = min(min((l2-l1)/50,(r2-r1)/50),0.04);
```

```
global f_test;
global MAT_A;
N = length(MAT_A);
f_test=rand(N,1)+2*exp(1i*rand(N,1));
Square_point=zeros(10000,5);
Selected_point=zeros(10000,5);
final_point=zeros(10000,1);
tol_robust=0.1*tol;

global p;
global p_test;
p↩
   =[0,1/2,1/4,3/4,1/8,5/8,3/8,7/8,1/16,9/16,5/16,13/16,3/16,11/16,7/1

p_test=[0,1/2,1/4,3/4,1/8,5/8,3/8,7/8];

f_test=f_test/norm(f_test);
len_l=l2-l1; len_r=r2-r1;
size_length=ceil(len_l/delta_len);
size_width=ceil(len_r/delta_len);

global k;
global number_reduce_sys;
global V_h;
global D_h;
global V_h_inv_v;
global H_para;
global shift_phi;

k=size_Krylov_space;
V_h=zeros(k,k,100);
D_h=zeros(k,100);
```

```
V_h_inv_v=zeros(k,100);
H_para=zeros(1,100);
shift_phi=zeros(1,100);
number_reduce_sys=0;

for i=1:size_length
    for j=1:size_width
        Square_point((j-1)*size_length+i,1)=l1+(i-1)↩
           *delta_len;
        Square_point((j-1)*size_length+i,2)=r1+(j-1)↩
           *delta_len;
        Square_point((j-1)*size_length+i,3)=l1+i*↩
           delta_len;
        Square_point((j-1)*size_length+i,4)=r1+j*↩
           delta_len;
    end
end
Square_number=size_length*size_width;

for i=1:Square_number
    ind=[i-1,i-size_length,i-size_length+1,i-↩
       size_length-1];
    ind=ind(ind>0);
    if isempty(ind)==1 || number_reduce_sys == 0
        shift=(Square_point(i,1)+Square_point(i,3))↩
           /2+1i*(Square_point(i,2)+Square_point(i,4)↩
           )/2;
        arnoldi(shift);
        Square_point(i,5)=number_reduce_sys;
    else
        flag=check_ind(Square_point(ind,5),↩
           Square_point(i,1:4),tol_robust);
        if (flag~=0)
            Square_point(i,5)=flag;
```

```
        else
            shift=(Square_point(i,1)+Square_point(i↩
               ,3))/2+1i*(Square_point(i,2)+↩
               Square_point(i,4))/2;
            arnoldi(shift);
            Square_point(i,5)=number_reduce_sys;
        end
    end
end

tol_iter=ceil(log2(delta_len/tol_robust));

for i=1:tol_iter
    Selected_number=0;
    for j=1:Square_number
        if (Check_square(Square_point(j,:))==1)
            Selected_number= Selected_number+1;
            Selected_point(Selected_number,:)=↩
               Square_point(j,:);
        end
    end
    Square_number=Selected_number*4;
    for j=1:Selected_number
        l1=Selected_point(j,1);
        r1=Selected_point(j,2);
        l2=Selected_point(j,3);
        r2=Selected_point(j,4);
        index=Selected_point(j,5);

        Square_point((j-1)*4+1,1:5)=[l1,r1,(l1+l2)↩
           /2,(r1+r2)/2,index];
        Square_point((j-1)*4+2,1:5)=[(l1+l2)/2,r1,l2↩
           ,(r1+r2)/2,index];
```

```
            Square_point((j-1)*4+3,1:5)=[(l1+l2)/2,(r1+↩
               r2)/2,l2,r2,index];
            Square_point((j-1)*4+4,1:5)=[l1,(r1+r2)/2,(↩
               l1+l2)/2,r2,index];
        end
    end

    Num_Eigenvalue=0;
    for i=1:Selected_number
        if (Selected_point(i,5)~=0)
            tmp=Selected_point(i,1:4);
            num=1;
            for j=1:Selected_number
                if (norm(Selected_point(i,1:4)-↩
                   Selected_point(j,1:4))<16*tol)
                    num=num+1;
                    tmp=tmp+Selected_point(j,1:4);
                    Selected_point(j,5)=0;
                end
            end
            Num_Eigenvalue=Num_Eigenvalue+1;
            tmp=tmp./num;
            final_point(Num_Eigenvalue)=(tmp(1)+tmp(3))↩
               /2+1i*(tmp(2)+tmp(4))/2;
        end
    end

    E=final_point(1:Num_Eigenvalue);
end

function flag = Check_square(point)
    flag=0;
    center=(point(1)+point(3))/2+1i*(point(2)+point(4))↩
       /2;
```

```
    radius=sqrt((real(center)-point(1))^2+(imag(center)-↩
       point(2))^2);
    ind=point(5);

    global p;
    global k;
    global D_h;
    global V_h_inv_v;
    global shift_phi;

    Para_p=center+radius*exp(1i*2*pi*p);
    iter=[0,2,4,8,16];
    w=[1/2,1/4,1/8,1/16];
    res=zeros(1,4);
    u=rand(k,1);
    level = 3;
    for i=1:level
        for j=iter(i)+1:iter(i+1)
            tmp_v=u'*((V_h_inv_v(:,ind)./(ones(k,1)+(↩
               shift_phi(ind)-Para_p(j))*D_h(:,ind))));
            res(i)=res(i)+w(i)*(Para_p(j)-center)*tmp_v;
        end
        res(i+1)=res(i)*1/2;
    end

    if (abs(norm(res(level-1))/norm(res(level)))<15)
        flag=1;
    end
end

function ind = check_ind(index,point,tol)

    global p_test;
    global k;
```

```
    global V_h;
    global D_h;
    global V_h_inv_v;
    global H_para;
    global shift_phi;

    center=(point(1)+point(3))/2+1i*(point(2)+point(4))↩
       /2;
    radius=sqrt((real(center)-point(1))^2+(imag(center)-↩
       point(2))^2);
    Para_p=center+radius*exp(1i*2*pi*p_test);

    ind=0;
    for i=1:length(index)
        flag=1;
        for j=1:length(Para_p)
            if abs(H_para(index(i))*V_h(k,:,index(i))*(↩
               V_h_inv_v(:,index(i))./((ones(k,1)+(↩
               shift_phi(index(i))-Para_p(j))*D_h(:,index↩
               (i))))))>tol
                flag=0;
            end
        end
        if flag==1
            ind=index(i);
            break;
        end
    end
end

function arnoldi(phi)

    global f_test;
    global k;
```

```
    global number_reduce_sys;
    global V_h;
    global D_h;
    global V_h_inv_v;
    global H_para;
    global shift_phi;
    global MAT_A;
    global MAT_B;

    I=sparse(k,1);
    I(1)=1;
    number_reduce_sys=number_reduce_sys+1;
    [L,U,P,Q]=lu(MAT_A-phi*MAT_B);
    b=Q*(U\(L\(P*f_test)));
    V1=zeros(length(MAT_A),k+1);
    H1=zeros(k+2,k+1);
    V1(:,1)=b/norm(b);
    for i=1:k
        v1=Q*(U\(L\(P*(MAT_B*V1(:,i)))));
        for j=1:i
            H1(j,i)=V1(:,j)'*v1;
            v1=v1-H1(j,i)*V1(:,j);
        end
        H1(i+1,i)=norm(v1);
        V1(:,i+1)=v1/H1(i+1,i);
    end
    [V,D]=eig(H1(1:k,1:k));
    V_h(:,:,number_reduce_sys)=V;
    D_h(:,number_reduce_sys)=diag(D);
    V_h_inv_v(:,number_reduce_sys)=V\I;
    shift_phi(number_reduce_sys)=phi;
    H_para(number_reduce_sys)=H1(k+1,k);
end
```