A MIXED PRECISION EIGENSOLVER BASED ON THE JACOBI ALGORITHM

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

2023

Zhengbo Zhou
Department of Mathematics

Contents

Word count 12597

List of Figures

Abstract

The classic method for computing the eigendecomposition of a real symmetric matrix, the Jacobi algorithm, can be accelerated by using mixed precision arithmetic. The Jacobi algorithm is aiming to reduce the off-diagonal entries iteratively using Givens rotations. We investigate how to use the low precision to speed up this algorithm based on the approximate eigendecomposition in low precision.

We first study two different index choosing techniques, classical and cyclic-by-row, for the Jacobi algorithm. Numerical testing suggests that cyclic-by-row is more efficient. Then we discuss two different methods of orthogonalizing an almost orthogonal matrix: the QR factorization and the polar decomposition. For polar decomposition, we speed up the Newton iteration by using the one-step Schulz iteration. Based on numerical testing, using the polar decomposition approach (Newton–Schulz iteration) is not only faster but also more accurate than using the QR factorization.

A mixed precision algorithm for computing the eigendecomposition of a real symmetric matrix at double precision is provided. In doing so we compute the approximate eigenvector matrix Q_ℓ of A in single precision using `eig` and `single` in MATLAB. We then use the Newton–Schulz iteration to orthogonalize the eigenvector matrix Q_ℓ into an orthogonal matrix Q_d in double precision. Finally, we apply the cyclic-by-row Jacobi algorithm on $Q_d^T A Q_d$ and obtain the eigendecomposition of A. At this stage, we will see, from the testings, the cyclic-by-row Jacobi algorithm only need less than 10 iterations to converge by utilizing the quadratic convergence. The new mixed precision algorithm requires roughly 30% of the time used by the Jacobi algorithm on its own.

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual Property Statement

Acknowledgements

It is my pleasure to acknowledge the advices from my supervisors Prof. Nicholas J. Higham and Prof. Françoise Tisseur. I wouldn't start my research in numerical linear algebra without their guidance and this thesis cannot be completed without their unlimited patience and insightful feedback.

I would also like to thank all my friends for providing emotional support. In addition, two special thanks to Zeyu Wu, for his suggestion on coding aspects, and Anheng Xi, for the translation of several German articles from Jacobi and Schönhage.

Finally, I thank my parents, Guangyuan Zhou and Ling Li, for their unconditional love and for giving me the chance to study abroad. By standing upon the shoulders of my parents, I have seen further than them.

Chapter 1

Introduction

Modern hardware increasingly supports low precision floating-point arithmetic. Using low precision, we have opportunities to accelerate linear algebra computations. A symmetric eigenproblem is solving

$$Ax = \lambda x,$$

with $x \neq 0$ and the matrix $A \in \mathbb{R}^{n \times n}$ is symmetric. Equivalently, the problem is aiming to compute the eigendecomposition

$$A = Q\Lambda Q^T, \quad A = A^T \in \mathbb{R}^{n \times n}, \quad Q^TQ = QQ^T = I,$$

and Λ is the diagonal matrix with the eigenvalues of A along the diagonal.

Traditional ways of solving such a problem include the Jacobi algorithm, the power method and the QR algorithm [8, 2000, Section 5–7]. This thesis is concerned with the Jacobi algorithm proposed by Jacobi in 1846 which is an iterative method that reduces the off-diagonal entries at each step. In addition, we investigate how the usage of low precision arithmetic can speed up the Jacobi algorithm.

In 2000, Drmač and Veselić [5, 2000] provided a way to speed up the Jacobi algorithm on the real symmetric matrix A. Let U be the approximate eigenvector matrix of A, then applying the Jacobi algorithm on U^TAU can speed the reduction up.

In Chapter 2 we provide essential definitions and examples including norms, orthogonal matrix and the singular value decomposition. We introduce two important classes of the orthogonal matrix, the Givens rotation and the Householder transformation.

The Jacobi algorithm is introduced in Chapter 3. We describe the derivation of the Jacobi algorithm and its convergence rate. In Section 3.2 and 3.3 we study how to choose the pair of index required by the Jacobi algorithm.

The classical Jacobi algorithm, mentioned in Section 3.2, chooses the largest off-diagonal entry as the pair of index and the cyclic-by-row Jacobi algorithm, mentioned in Section 3.3, chooses the indices row by row but restricts to the superdiagonal part of the matrix. Although the classical Jacobi algorithm reduces the off-diagonal entries most optimally, the cyclic-by-row Jacobi algorithm does not require expensive off-diagonal searches. Finally, we conduct numerical tests to examine the speed and conclude that we shall stick with the faster cyclic-by-row Jacobi algorithm.

We provide a procedure to get an approximate eigenvector matrix in Chapter 4. Firstly, compute the eigendecomposition at low precision $A \approx Q_\ell D_\ell Q_\ell^T$ such that

$$\|Q_\ell^T Q_\ell - I\| \lesssim nu_\ell, \quad \|AQ_\ell - Q_\ell D_\ell\| \lesssim nu_\ell \|A\|,$$

where u_ℓ is the unit roundoff in low precision. Then we orthogonalize Q_ℓ to Q_d such that $\|Q_d^T Q_d - I\| \lesssim nu_d$ where u_d is the unit roundoff in double precision ("Unit roundoff" - defined in Section 1.2). We can use Q_d as the approximate eigenvector matrix of A and precondition A into $A_{\text{cond}} = Q_d^T A Q_d$, then the Jacobi algorithm applied on A_{cond} should converge in a few sweeps. In Section 4.2 and 4.3 we review two methods for orthogonalizing a matrix: the Householder QR factorization and the polar decomposition. Particularly, in Section 4.3, we study the Newton iteration approach and its variant Newton–Schulz iteration to compute the unitary factor of the polar decomposition. MATLAB codes are presented and after several numerical testings, among the Householder QR factorization approach, the Newton iteration approach and the Newton–Schulz iteration approach, the latter method is the most accurate and efficient one to use.

Finally, we assemble all the ingredients and produce Algorithm 9 in Chapter 5, a mixed precision Jacobi algorithm. In Section 5.2.2, we review some literature on the quadratic convergence of the cyclic-by-row Jacobi algorithm. We observe from the numerical testings that for any symmetric matrix $A \in \mathbb{R}^{n\times n}$, after preconditioning, the Jacobi algorithm should converge within 10 iterations. The numerical testings suggest the Jacobi algorithm only needs two iterations to converge and there is a 75% reduction of time using preconditioned Jacobi algorithm compare to the Jacobi algorithm on its own.

1.1 Notations

We use the Householder notations:

- Use capital letters for matrices.
- Use lower case letters for vectors.
- Use lower case Greek letters for scalars.

We use $\lambda(A)$ and $\sigma(A)$ to represent the eigenvalues and the singular values of A. We denote $\sigma_1(A)$ or σ_1 if the matrix A has been specified as the largest singular value of A.

We say that the $Ax = b$ problem is well conditioned if small changes in the data A, b always make small changes in the solution x; otherwise, it is ill-conditioned. We use $\kappa(A) = \|A\|\|A^{-1}\|$ as a measure to visualize the condition of a problem. (Norms will be introduced in Section 2.1)

We say a matrix $A \in \mathbb{R}^{n\times n}$ is almost orthogonal if $\|A^*A - I\| \lesssim nu_\ell$ where I is the $n\times n$ identity matrix. A matrix $A \in \mathbb{R}^{n\times n}$ is almost diagonal if $\|A - \operatorname{diag}(A)\|/\|A\| \lesssim nu_\ell$ and this quantity is zero if A is diagonal.

1.2 IEEE standard

The unit roundoff measures the relative error due to approximation in floating-point arithmetic. Throughout this thesis, we will only consider two precisions: single precision and double precision [14, 2002, Section 2.1]. We will denote u_ℓ as the unit roundoff at single precision and u_d as the unit roundoff at double precision. IEEE standard 754-2019 [19, 2019, Table 3.5] shown in Table 1.1 will be used. For simplicity, any matrix with subscript ℓ will be at single precision and any matrix with subscript d will be at double precision.

TABLE 1.1: *Unit roundoff defined by IEEE standard 754-2019.*

Type	Unit Roundoff
Single precision	$2^{-24} \approx$ 5.96e-8
Double precision	$2^{-53} \approx$ 1.11e-16

1.3 Reproducible Research

In this section, we will present our testing environment and system specifications. With these informations, the reader will have the ability to reproduce any results we have in this thesis.

Testing environment and system specifications:

- System OS: Windows 10.
- MATLAB version: 9.12.0.2009381 (R2022a) Update 4 [17].
- CPU: Intel(R) Core(TM) i5-9600KF CPU @ 3.70GHz.
- GPU: NVIDIA GeForce RTX 2060.
- All the testings that require the random number generator will initiate after the MATLAB command `rng(1,'twister')`[1].

[1]MATLAB built-in function to control the random number generator, see `https://www.mathworks.com/help/matlab/ref/rng.html`.

Chapter 2

Matrix Norms, Orthogonal Matrices and Singular Value Decomposition

In this chapter, we will introduce several concepts that are important for our analysis. In the first section, vector norms and matrix norms will be presented. Then we will discuss an important class of matrices, orthogonal matrices. Finally, a crucial decomposition, singular value decomposition, will be discussed.

2.1 Norms

Matrix norms are essential for matrix algorithms. For example, the matrix norms provide a way of measuring the difference between two matrices and help us to check if we have the desired solution in finite arithmetic. In this section, we will do our demonstrations in the real world but the complex case is similar. Also, we will only introduce the norm for square matrices, but these theories are all applicable to any general matrices with careful attention to matching dimensions.

2.1.1 Vector Norms

Before introducing matrix norm, a brief illustration of vector norm is necessary.

Definition 2.1 (Vector Norm)**.** A vector norm on $\mathbb{R}^n$ is a function $\|:\|\mathbb{R}^n \to \mathbb{R}$ such that it satisfies the following properties

1. $\|x\| \geq 0$ for all $x \in \mathbb{R}^n$.
2. $\|x\| = 0$ if and only if $x = 0$.
3. $\|\lambda x\| = |\lambda|\|x\|$ for all $\lambda \in \mathbb{R}$ and $x \in \mathbb{R}^n$.
4. $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in \mathbb{R}^n$.

Example 2.2. A useful class of vector norm is the p-norm

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \quad p \geq 1.$$

One particular example of the p-norm is the Euclidean norm or simply 2-norm defined by taking $p = 2$

$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2} = \sqrt{x^T x}.$$

The latter equality is obvious.

2.1.2 Matrix Norms

Let us denote $\mathbb{R}^{n \times n}$ as $n \times n$ matrices with real entries.

Definition 2.3 (Matrix Norm)**.** A matrix norm on $\mathbb{R}^{n \times n}$ is a function $\|\cdot\| : \mathbb{R}^{n \times n} \to \mathbb{R}$ satisfying the following properties

1. $\|A\| \geq 0$ for all $A \in \mathbb{R}^{n \times n}$.
2. $\|A\| = 0$ if and only if $A = 0$.
3. $\|\lambda A\| = |\lambda|\|A\|$ for all $\lambda \in \mathbb{R}$ and $A \in \mathbb{R}^{n \times n}$.
4. $\|A + B\| \leq \|A\| + \|B\|$ for all $A, B \in \mathbb{R}^{n \times n}$.

One simple matrix norm is the Frobenius norm

$$\|A\|_F = \left(\sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2} = (\mathrm{trace}(A^T A))^{1/2}. \tag{2.1}$$

We will use the Frobenius norm intensively in the next section on the Jacobi algorithm.

Another important class of matrix norms is called induced norms or subordinate norms. Given a vector norm defined in Section 2.1.1, the corresponding induced norm is defined as

$$\|A\| = \max_{\|x\|=1} \|Ax\|.$$

Example 2.4. The matrix 2-norm is induced by the vector 2-norm. From the definition of the induced norm,

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2 = \sqrt{x_\mu^T A^T A x_\mu} = \mu,$$

for some positive number μ, and x_μ is the corresponding chosen x. Since $\|x\|_2 = \sqrt{x^T x}$ for $x \in \mathbb{R}^n$, hence we have $x_\mu^T A^T A x_\mu = \mu^2$. Also, by $\|x_\mu\|_2 = x_\mu^T x_\mu = 1$, we can premultiply x_μ at both sides and we have $A^T A x_\mu = \mu^2 x_\mu$ which implies μ is the eigenvalue of $A^T A$. In case we are choosing x_μ such that μ is maximum, therefore we have the definition of the matrix 2-norm

$$\|A\|_2 = \sqrt{\lambda_{\max}(A^T A)} = \sigma_1, \tag{2.2}$$

where $\lambda_{\max}(A)$ and σ_1 denotes the largest eigenvalue and singular value of A.

Definition 2.5 (Consistent Norms)**.** A norm is consistent if it is submultiplicative

$$\|AB\| \leq \|A\|\|B\|, \tag{2.3}$$

for all A, B such that the product AB defines.

Example 2.6. The Frobenius norm and all subordinate (induced) norms are consistent. This is useful for constructing upper bounds.

2.2 Orthogonal Matrices

Recall the definition of the orthogonal matrix,

Definition 2.7 (Orthogonal Matrices)**.** A matrix $Q \in \mathbb{R}^{n\times n}$ is said to be orthogonal if

$$Q^T Q = QQ^T = I_n,$$

where I_n denotes the identity matrix of size n. If $Q \in \mathbb{C}^{n\times n}$, then we call Q unitary rather than orthogonal and we denote its conjugate transpose as Q^*.

Theorem 2.8. *The vector 2-norm is invariant under orthogonal transformations.*

Proof. Given an orthogonal matrix $Q \in \mathbb{R}^{n\times n}$ and any vector $x \in \mathbb{R}^n$, we have

$$\|Qx\|_2 = \sqrt{x^T Q^T Q x} = \sqrt{x^T x} = \|x\|_2,$$

which proves the theorem. □

Theorem 2.9. *The Frobenius norm and matrix 2-norm are the orthogonally invariant norm. Namely, the following norm equality holds for these two norms*

$$\|UAV\| = \|A\|, \quad A \in \mathbb{R}^{n\times n}, \, U \text{ and } V \text{ are orthogonal.}$$

Proof. By the definition of the Frobenius norm, we have

$$\|UAV\|_F^2 = \operatorname{trace}\big(UAVV^TA^TU^T\big) = \operatorname{trace}\big(UAA^TU^T\big). \tag{2.4}$$

By the properties $\operatorname{trace}(AB) = \operatorname{trace}(BA)$, (2.4) simplifies to

$$\|UAV\|_F^2 = \operatorname{trace}\big(A^TU^TUA\big) = \operatorname{trace}\big(A^TA\big) = \operatorname{trace}\big(AA^T\big) = \|A\|_F^2. \tag{2.5}$$

Hence we finished the proof for the Frobenius norm.

By the definition of the matrix 2-norm, we have

$$\|UAV\|_2^2 = \lambda_{\max}(V^TA^TU^TUAV) = \lambda_{\max}(V^TA^TAV).$$

Notice that if V is orthogonal, then A^TA and V^TA^TAV share the same eigenvalues. Therefore $\lambda_{\max}(V^TA^TAV) = \lambda_{\max}(A^TA)$, therefore we have

$$\|UAV\|_2^2 = \lambda_{\max}(A^TA) = \|A\|_2^2.$$

Hence we proved the theorem. □

2.2.1 Givens Rotations

Definition 2.10 (Givens rotation)**.** A Givens rotation has the form

$$G(i,k,\theta) = \begin{array}{c@{}c} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} & \begin{array}{c} \\ \\ i \\ \\ k \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & i & & k & & \end{array} & \end{array}, \tag{2.6}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ for some θ.

Given a vector $y \in \mathbb{R}^n$ and a Givens rotation $G(i,j,\theta)$, the product $G(i,j,\theta)y$ rotate y through θ radians clockwise in the (i,j) plane. Hence it is obvious that we can make either y_i or y_j become zero by manipulating θ. This idea will be utilized in Section 3.1 in order to eliminate (i,j)th entry of a symmetric matrix.

Proposition 2.11. *Givens rotations are orthogonal.*

Proof. Let $G = G(i,k,\theta)$ be a Givens rotation. By using the trigonometric identity $\cos^2(\theta) + \sin^2(\theta) = 1$, we have the following

$$G_{ik} = G([i,k],[i,k]) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \quad \rightarrow \quad G_{ik}^T G_{ik} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = G_{ik} G_{ik}^T. \tag{2.7}$$

We can calculate the following matrix products $G^T G$ and GG^T using (2.7).

$$G^T G = GG^T = I.$$

Hence G is orthogonal. □

Notice that the matrix $G(p,q,\theta)$ can be written as

$$G(p,q,\theta) = [e_1, \ldots, e_{p-1}, e_{pq}, e_{p+1}, \ldots, e_{q-1}, e_{qp}, e_{q+1}, \ldots, e_n], \tag{2.8}$$

where e_i is the ith column of the $n \times n$ identity matrix if $i \neq p, q$ and e_{pq} and e_{qp} are defined as

$$[e_{pq}]_k = \begin{cases} c & \text{if } k = p \\ -s & \text{if } k = q \\ 0 & \text{if } k \neq p, q \end{cases}, \qquad [e_{qp}]_k = \begin{cases} s & \text{if } k = q \\ c & \text{if } k = p \\ 0 & \text{if } k \neq p, q \end{cases}. \tag{2.9}$$

Proposition 2.12. *Suppose $A \in \mathbb{R}^{n\times n}$ and $G(p,q,\theta)$ is a Givens rotation. If we define B as*

$$B = G(p,q,\theta)^T A G(p,q,\theta),$$

then B and A are the same except in rows and columns p and q.

Proof. Using the notations in (2.8) and (2.9), matrix B can be written as

$$B = \begin{bmatrix} e_1^T A e_1 & \cdots & e_1^T A e_{pq} & \cdots & e_1^T A e_{qp} & \cdots & e_1^T A e_n \\ \vdots & & \vdots & & \vdots & & \vdots \\ e_{pq}^T A e_1 & \cdots & e_{pq}^T A e_{pq} & \cdots & e_{pq}^T A e_{qp} & \cdots & e_{pq}^T A e_n \\ \vdots & & \vdots & & \vdots & & \vdots \\ e_{qp}^T A e_1 & \cdots & e_{qp}^T A e_{pq} & \cdots & e_{qp}^T A e_{qp} & \cdots & e_{qp}^T A e_n \\ \vdots & & \vdots & & \vdots & & \vdots \\ e_n^T A e_1 & \cdots & e_n^T A e_{pq} & \cdots & e_n^T A e_{qp} & \cdots & e_n^T A e_n \end{bmatrix}.$$

Hence $B_{ij} = e_i^T A e_j = A_{ij}$ if $i, j \neq p, q$ and for $i, j = p$ or q, $B_{ij} \neq A_{ij}$ in general. □

2.2.2 Householder Transformations

Another important class of orthogonal matrix is the Householder transformations. Unlike the Givens rotation which replaces a specific entry into zero, Householder transformation can change multiple entries into zeros. In this section, we will first discuss how this matrix can set entries to zeros and then a MATLAB implementation will be present together with numerical testing.

Definition 2.13. A Householder transformation (synonyms are Householder matrix and Householder reflector) is an $n \times n$ matrix P of the form

$$P = I - 2\frac{vv^T}{v^T v}, \quad 0 \neq v \in \mathbb{R}^n.$$

The vector v is the Householder vector.

Householder transformation has useful properties,

1. Symmetry:
$$P^T = I^T - \frac{2}{v^Tv}(vv^T)^T = I - \frac{2}{v^Tv}vv^T = P.$$

2. Orthogonality:
$$\begin{aligned} P^TP = PP^T = PP &= \left(I - \frac{2}{v^Tv}vv^T\right)\left(I - \frac{2}{v^Tv}vv^T\right) \\ &= I - \frac{4}{v^Tv}vv^T + \frac{4}{(v^Tv)^2}vv^Tvv^T \\ &= I - \frac{4}{v^Tv}vv^T + \frac{4}{v^Tv}vv^T = I. \end{aligned}$$

Transforming a Vector

The Householder transformation can be used to zero components of a vector. Suppose we have a vector $x \in \mathbb{R}^n$ and we want

$$y = Px = \left(I - \frac{2}{v^Tv}vv^T\right)x = x - \frac{2v^Tx}{v^Tv}v \tag{2.10}$$

where $y(2:n) = 0$. Since P is orthogonal, we require $\|y\|_2 = \|x\|_2$, hence $y = \pm\|x\|_2 e_1$ where e_1 is the first column of the $n \times n$ identity matrix. We can rewrite v as

$$v = \frac{1}{\alpha}x - \frac{1}{\alpha}y, \quad \alpha = \frac{2v^Tx}{v^Tv}v,$$

and since P is independent of the scaling of v, hence for $x \neq y$, we can set $\alpha = 1$ and we can choose $v = x - y$. We can check our choice by computing Px: Firstly, we can compute v^Tx and v^Tv

$$v^Tx = (x^T - y^T)x = x^Tx - y^Tx, \quad v^Tv = (x^T - y^T)(x - y) = 2x^Tx - 2x^Ty.$$

Substitute these two components into (2.10)

$$Px = x - \frac{2v^Tx}{v^Tv}v = x - v = x - (x - y) = y.$$

The choice of the sign of y depends on the sign of the first entry of x. Suppose x is dominant by its first entry, namely x is close to a multiple of e_1, then $v = x - \text{sign}\,(x_1)\,\|x\|_2 e_1$ can have a small norm due to cancellation and this may result a large relative error in $2/(v^Tv)$. This relative error can be avoided by setting

$$v = x + \text{sign}\,(x_1)\,\|x\|_2 e_1$$

as suggested in many textbooks, such as [14, 2002, Section 19.1] and [9, 2013, Section 5.1.3]. Then we get

$$Px = y = x - v = -\operatorname{sign}(x_1)\,\|x\|_2 e_1. \tag{2.11}$$

Given $x \neq 0 \in \mathbb{R}^n$, instead of generating the Householder matrix P such that it satisfies (2.11), we should only generate the Householder vector v and the coefficient $\beta = 2/(v^T v)$. The benefit of doing so can be seen from the Householder QR factorization algorithm in section 4.2.1. Based on the previous discussion, we can conclude these into Algorithm 1.

Algorithm 1 Given $x \neq 0 \in \mathbb{R}^n$, this algorithm computes $v \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$ such that $P = I_n - \beta vv^T$ is orthogonal and $Px = -\operatorname{sign}(x_1)\,\|x\|_2 e_1$.

```
1: Form $\sigma = x(2:n)^T x(2:n),\ v = [1, x(2:n)^T]^T$
2: Calculate $\mu = \|x\|_2 = \sqrt{x_1^2 + \sigma}$
3: if $x_1 \geq 0$ then
4:     $v_1 = x_1 + \mu$
5: else
6:     $v_1 = x_1 - \mu = -\sigma/(x_1 + \mu)$
7: end if
8: $\beta = 2/(v_1^2 + \sigma)$
```

Notice that, line 6 rewrites the form of v_1 to avoid subtractive cancellation as suggested by [9, 2013, Algorithm 5.1.1]

$$v_1 = x_1 - \mu = \frac{(x_1 - \mu)(x_1 + \mu)}{(x_1 + \mu)} = \frac{x_1^2 - (x_1^2 + \sigma)}{x_1 + \mu} = \frac{-\sigma}{x_1 + \mu}.$$

Implementation and Testing

During the MATLAB implementation, we do not need to create a new vector v. It is sufficient to just overwrite the values of x since the only difference between v and x in Algorithm 1 is the first entry of these two vectors.

```matlab
function [x,b] = house(x)
n = length(x); sigma = x(2:n)'*x(2:n);
mu = sqrt(x(1)*x(1) + sigma);
if x(1) >= 0, x(1) = x(1) + mu;
else, x(1) = -sigma/(x(1) + mu); end
b = 2/(x(1)*x(1) + sigma);
end
```

To test the above function, we first generate a vector $x \in \mathbb{R}^5$ using `randn(5,1)`.

```
x' =
  -1.3499e+00,  3.0349e+00,  7.2540e-01,  -6.3055e-02,  7.1474e-01
```

Using the above function and call `[v,b] = house(x)`, we can construct P using the formula $P = I_n - \beta vv^T$. Then we examine the product Px and we have

```
(P * x)' =
  3.4748e+00              0    1.7694e-16   -9.1073e-18    1.1102e-16
```

The result satisfies our expectations.

The Householder transformation can quickly introduce zeros into a vector using orthogonal matrices and this is useful when we study the QR factorization.

2.3 Singular Value Decomposition

Theorem 2.14 (Singular value decomposition)**.** *If $A \in \mathbb{R}^{m\times n}$, $m \geq n$, then there exists two orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ such that*

$$A = U\Sigma V^T, \quad \Sigma = \operatorname{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m\times n}, \quad p = \min\{m, n\}, \tag{2.12}$$

where $\sigma_1, \dots, \sigma_p$ are all non-negative and arranged in non-ascending order. We denote (2.12) as the singular value decomposition (SVD) of A and $\sigma_1, \dots, \sigma_p$ are the singular values of A.

Proof. Proof can be done by induction. See [9, 2013, Theorem 2.4.1]. $\square$

From now, without explicit mention, A will be a square matrix of full rank (our input Q_ℓ is always square, full rank matrix). Therefore, if we have the singular value decomposition $A = U\Sigma V^T$, then $U, V \in \mathbb{R}^{n\times n}$ are unitary and $\Sigma = \operatorname{diag}(\sigma_1, \dots, \sigma_n) \in \mathbb{R}^{n\times n}$ where $\sigma_1 \geq \cdots \geq \sigma_n > 0$.

From the definition, the singular values of A can be computed via computing the eigenvalues of $A^TA = V\Lambda V^T$, where $\Lambda = \Sigma^2$. Hence, the singular values of A are the positive roots of the eigenvalues of A^TA. Based on this relationship, we have the following properties.

Corollary 2.15. *If $A \in \mathbb{R}^{n\times n}$ of full rank and normal, then $\sigma(A) = |\lambda(A)|$.*

Proof. If A is normal, then $A^TA = AA^T$. By spectral theorem, if A is normal, then $A = U\Lambda U^T$ where U is unitary and Λ is a diagonal matrix with eigenvalues of A on the diagonal. Therefore, the singular values of A are

$$\sigma(A) = \sqrt{\lambda(A^TA)} = \sqrt{\lambda(U\Lambda^2U^T)} = \sqrt{\lambda(\Lambda^2)} = \sqrt{\lambda(\Lambda)^2} = |\lambda(A)|.$$

Combine this Corollary and (2.2), we have: if A is normal, then $\|A\|_2 = \max\{|\lambda(A)|\}$. $\square$

2.4 Test Matices

Throughout this thesis, we will perform lots of MATLAB testing on our algorithms, hence we want to generate real symmetric matrices A with desired condition number and different eigenvalue distributions.

To accomplish such a goal, we use the MATLAB built-in function `gallery`[1]. If we use `gallery('randsvd',n, -kappa)` where `kappa` is a positive integer, then it will generate a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ with $\kappa(A) =$ `kappa`. Our code `my_randsvd` from Appendix B.1 is the modified version of `gallery('randsvd')` and we deliberately change some eigenvalues to negative such that the output matrix will not necessarily be positive definite. It provides two different eigenvalue distributions.

1. Mode `'geo'`, `A = my_randsvd(n, kappa, 'geo')`: The output matrix $A \in \mathbb{R}^{n\times n}$ with $\kappa(A) =$ `kappa` and the magnitude of the eigenvalues of A are geometrically distributed.

2. Mode `'ari'`, `A = my_randsvd(n, kappa, 'ari')`: The magnitude of the eigenvalues of A are arithmetically distributed.

In Chapter 5, we will test our algorithm on matrices with different distributions. For the rest of the thesis, we will use mode `'geo'` by default.

[1] A MATLAB built-in function to generate test matrices. For the full manual, you may refer to `https://www.mathworks.com/help/matlab/ref/gallery.html`.

Chapter 3

Jacobi Algorithm

The Jacobi algorithm for eigenvalues are first published by Jacobi in 1846 [16] and became widely used in the 1950s after the computer is invented. In this chapter, we will first derive the Jacobi algorithm and discuss its linear convergence. Then the rest of this chapter will focus on two different Jacobi algorithms: Classical and Cyclic-by-row. MATLAB implementation and testing will be given.

3.1 Derivation and Convergence

Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$, the idea of the Jacobi algorithm is that at the kth step, we use $A^{(k+1)} = Q_k^T A^{(k)} Q_k$ to replace $A^{(k)}$, where Q_k is orthogonal. We aim to have the property that the off-diagonal entries of $A^{(k+1)}$ are smaller than the off-diagonal entries of $A^{(k)}$. To do this, we use the Givens rotation matrix defined in section 2.2.1.

Definition 3.1 (off operator)**.** We define the quantity

$$\mathsf{off}(A) := \sqrt{\|A\|_F^2 - \sum_{i=1}^{n} a_{ii}^2} = \sqrt{\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}^2},$$

which is the Frobenius norm of the off-diagonal elements.

The aim of the Jacobi algorithm on A can be translated to reduce the quantity $\mathsf{off}(A)$. This can be done in 2 steps.

1. Choosing a pair of index (p, q). We assume that $1 \leq p < q \leq n$.

2. Overwriting A by applying $G(p,q,\theta)$ on the matrix A in the way of

$$A' = G(p,q,\theta)^T A G(p,q,\theta),$$

where θ is chosen such that $A'_{pq} = A'_{qp} = 0$.

Using the Jacobi algorithm on A, we produce a sequence of matrices $\left\{A^{(k)}\right\}_{k=0}^{\infty}$, where

$$\begin{cases} A^{(0)} = A, \\ A^{(k)} = G_{k-1}(p,q,\theta)^T A^{(k-1)} G_{k-1}(p,q,\theta), \quad \text{for } k = 1, 2, \ldots. \end{cases}$$

This set of iterations satisfy

$$\lim_{k\to\infty} \mathsf{off}(A^{(k)}) = 0. \tag{3.1}$$

In the rest of this section, we will discuss how we can achieve (3.1) using step 2 and how we can construct such a matrix $G(p,q,\theta)$ given the choice (p,q). Then in the next two sections, we present two different methods of choosing the index (p,q)

- The Classical Jacobi Algorithm.
- The Cyclic-by-row Jacobi Algorithm.

Considering the subproblem which only involves four entries of A.

$$\begin{aligned} \begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} &= \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \\ &= \begin{bmatrix} c^2 a_{pp} - cs a_{qp} - cs a_{pq} + s^2 a_{qq} & cs a_{pp} - s^2 a_{qp} + c^2 a_{pq} - cs a_{qq} \\ c^2 a_{pq} + cs a_{pp} - cs a_{qq} - s^2 a_{pq} & cs a_{qp} + s^2 a_{pp} + c^2 a_{qq} + cs a_{pq} \end{bmatrix}. \end{aligned} \tag{3.2}$$

Since A is symmetric (symmetry is preserved by orthogonal transformations), we have $a_{pq} = a_{qp}$. Hence we can make further simplification to (3.2)

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = \begin{bmatrix} c^2 a_{pp} + s^2 a_{qq} - 2cs a_{pq} & cs(a_{pp} - a_{qq}) + (c^2 - s^2) a_{pq} \\ cs(a_{pp} - a_{qq}) + (c^2 - s^2) a_{pq} & s^2 a_{pp} + c^2 a_{qq} + 2cs a_{pq} \end{bmatrix}. \tag{3.3}$$

To achieve step 2, we require $b_{pq} = b_{qp} = 0$, which means

$$cs(a_{pp} - a_{qq}) + (c^2 - s^2) a_{pq} = 0. \tag{3.4}$$

This equality is essential for us to find the desired Givens rotation and the construction will be discussed in section 3.1.1.

If we take the Frobenius norm on the first line of (3.2) and set $b_{pq} = b_{qp} = 0$, since the Frobenius norm is orthogonally invariant norm (Theorem 2.9), we have

$$b_{pp}^2 + b_{qq}^2 = a_{pp}^2 + a_{qq}^2 + 2a_{pq}^2. \tag{3.5}$$

Using the notations in proposition 2.12 and the condition $b_{pq} = b_{qp} = 0$, we have

$$\begin{aligned}
\mathsf{off}(B)^2 &= \|B\|_F^2 - \sum_{i=1}^{n} b_{ii}^2 = \|A\|_F^2 - \sum_{\substack{i=1\\ i\neq p,q}} (b_{ii}^2) - (b_{pp}^2 + b_{qq}^2) \\
&= \|A\|_F^2 - \sum_{\substack{i=1\\ i\neq p,q}}^{n} (b_{ii}^2) - a_{pp}^2 - a_{qq}^2 - 2a_{pq}^2 \quad \text{Using (3.5)} \\
&= \|A\|_F^2 - \sum_{\substack{i=1\\ i\neq p,q}}^{n} (a_{ii}^2) - a_{pp}^2 - a_{qq}^2 - 2a_{pq}^2 \quad \text{By Proposition 2.12} \\
&= \|A\|_F^2 - \sum_{i=1}^{n} (a_{ii}^2) - 2a_{pq}^2 = \mathsf{off}(A)^2 - 2a_{pq}^2.
\end{aligned} \tag{3.6}$$

Therefore, after k steps and for some choices of the index pair (p, q), we always have

$$\mathsf{off}(A^{(k+1)}) = \mathsf{off}(A^{(k)}) - 2a_{pq}^2,$$

and this reduction will never terminate until all the off-diagonal entries are zero. However, since each time we introduced a zero, the previous entry we chose will not necessarily remain zero, hence the $\mathsf{off}(A^{(k)})$ will require an infinite number of iterations to converge to zero. In Section 3.2, we proved that even if we choose the largest a_{pq} at each iteration, we still require infinite iteration for $\mathsf{off}(A^{(k)})$ converges to zero.

3.1.1 Choices of Givens rotation matrix

In this section, we provide a theory to choose c and s for constructing the Givens rotation matrix we need at each iteration. Firstly, if $a_{pq} = 0$, there is no need to do further work since $\mathsf{off}(B) = \mathsf{off}(A)$ and we can simply choose $c = 1$ and $s = 0$ which lead to an identity matrix. Otherwise, we look at (3.4)

$$cs(a_{pp} - a_{qq}) + (c^2 - s^2)a_{pq} = 0. \tag{3.6}$$

Notice that $c = \cos(\theta)$ and $s = \sin(\theta)$, we have

$$\cot(2\theta) = \frac{\cos^2(\theta) - \sin^2(\theta)}{2\cos(\theta)\sin(\theta)} = \frac{c^2 - s^2}{2cs} = \frac{a_{qq} - a_{pp}}{2a_{pq}} =: \tau.$$

By define $t = s/c$, we have transformed (3.6) into

$$t^2 + 2\tau t - 1 = 0. \tag{3.7}$$

This quadratic equation can be solved and we would like to choose the root of smaller magnitude.

$$t_{1,2} = -\tau \pm \sqrt{\tau^2 + 1}, \quad \rightarrow \quad t_{\min} = \begin{cases} -\tau + \sqrt{\tau^2+1}, & \text{if } \tau \geq 0; \\ -\tau - \sqrt{\tau^2+1}, & \text{if } \tau < 0. \end{cases}$$

Using the fact that $s^2 + c^2 = 1$ and $t_{\min} = s/c$, we can define c and s by

$$s = ct_{\min}, \quad c = \frac{1}{\sqrt{1 + t_{\min}^2}}. \tag{3.8}$$

Here we choose the smaller roots (in magnitude) of t, so that we ensure $\max(t_{\min}) = 1$ (shown in Figure 3.1), hence the angle of rotation is bounded $|\theta| \leq \pi/4$ and this choice turns out to be important for the quadratic convergence as discussed by Schönhage [20, 1961].

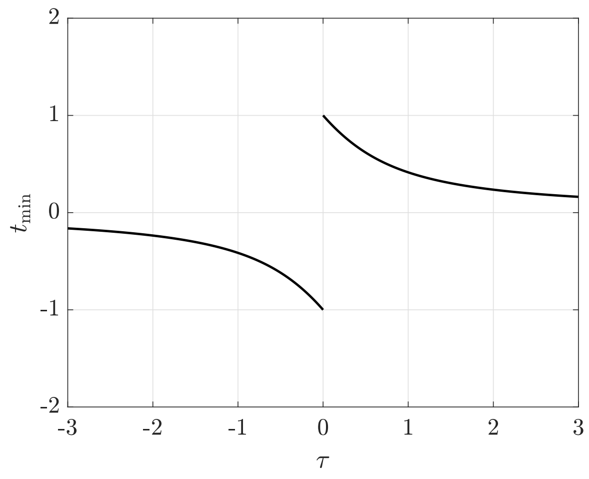

FIG. 3.1: *Behavior of the solution $t_{\min}$ of (3.7).*

3.1.2 Implementation and Testing

Section 3.1.1 can be summarized in the following algorithm.

Algorithm 2 Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$ and integers p, q such that $1 \le p < q \le n$. This algorithm computes a pair (c, s) to construct $G(p, q, \theta)$ as presented in (2.6), such that if $B = G(p, q, \theta)^T A G(p, q, \theta)$, then $b_{pq} = b_{qp} = 0$.

```
1:  if $a_{pq} = 0$ then
2:      $c = 1, s = 0$
3:  else
4:      $\tau = (a_{qq} - a_{pp})/2a_{pq}$
5:      if $\tau \ge 0$ then
6:          $t = -\tau + \sqrt{\tau^2 + 1}$
7:      else
8:          $t = -\tau - \sqrt{\tau^2 + 1}$
9:      end if
10:     $c = 1/\sqrt{1 + t^2}, s = ct$
11: end if
```

When we do the MATLAB implementation, we need to make a minor change to lines 6 and 8. In practice, we need to change these to

$$t = \frac{1}{\tau \pm \sqrt{1 + \tau^2}}, \tag{3.9}$$

where we choose the positive sign if $\tau \ge 0$ and the minus sign if $\tau < 0$. It is easy to prove these two sets of expressions are the same in exact arithmetic, however, they differ in finite precision (See Appendix A.1) and (3.9) is usually preferable. Then it is straightforward to implement the choice using MATLAB.

```matlab
function [c,s] = jacobi_pair(A,p,q)
if A(p,q) == 0
  c = 1; s = 0;
else
  tau = (A(q,q)-A(p,p))/(2*A(p,q));
  if tau >= 0
    t = 1/(tau + sqrt(1+tau*tau));
  else
    t = 1/(tau - sqrt(1+tau*tau));
  end
  c = 1/sqrt(1+t*t); s = t*c;
end
end
```

We can test this function by constructing the matrix $G(p, q, \theta)$ with (p, q) given and examine the (p, q) entry using the following routine,

```matlab
clc; clear; close all;
n = 100; format short e;
A = my_testmatrices(n);
pq = [57,64]; % choose a pair of index
[c,s] = jacobi_pair(A,pq(1),pq(2));
% construct the desire Givens rotation matrix
G = eye(n); G([pq(1),pq(2)],[pq(1),pq(2)]) = [c,s;-s,c];
```

```
B = G' * A * G;
disp('Before'); disp(A(pq(1),pq(2))); disp(A(pq(2),pq(1)));
disp('After');disp(B(pq(1),pq(2))); disp(B(pq(2),pq(1)));
```

This routine should give $b_{pq} = b_{qp} = 0$ where $p = 57$ and $q = 64$.

```
Before
  -9.4140e-01
  -9.4140e-01
After
  -2.2204e-16
  -1.1102e-16
```

Before applying the Givens rotations, the entries are -9.414×10^{-1} and after such transformation we have b_{pq} and b_{qp} small enough for computers to consider them as zeros. Hence we successfully provide a way of constructing Givens rotations that achieve step 2 as discussed earlier in this section and the remaining part of this chapter is to sophistically choose the index pair (p, q).

3.2 Classical Jacobi Algorithm

From (3.6), we proved that $\mathsf{off}(B)^2 = \mathsf{off}(A)^2 - 2a_{pq}^2$, hence it is natural to consider the choice (p, q) such that a_{pq}^2 is maximal. This choice was first proposed by Jacobi [16] in 1846 and the future literature refer to it as the *classical Jacobi algorithm*. We can summarize this choice into Algorithm 3.

Algorithm 3 (*The classical Jacobi algorithm*) Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$ and a positive tolerance tol, this algorithm overwrites A with $V^T AV$ where V is orthogonal.

```
1:  $V = I_n$, done_rot = true
2:  while done_rot do
3:      done_rot = false
4:      Choose $(p, q)$ so that $|a_{pq}| = \max_{i\neq j} |a_{ij}|$
5:      if $|a_{pq}| > tol \cdot \|A\|_2 \sqrt{|a_{pp}a_{qq}|}$ then
6:          done_rot = true
7:          Construct Givens rotation matrix $G$ using Algorithm 2
8:          Update $G^T AG \to A$
9:          Update $VG \to V$
10:     else
11:         Remove the off-diagonal entries of $A$
12:         Break
13:     end if
14: end while
```

The structure of Algorithm 3 adapts from [3, 2001, Algorithm 2.1] and the stopping criterion is adapted from [9, 2013, Section 8.5.5], [5, 2000, Theorem 1.1] and [4, 1992, Section 1].

3.2.1 Linear Convergence

Notice that since $|a_{pq}|$ is chosen to be the largest off-diagonal element, hence we have the inequality

$$\mathsf{off}(A)^2 \le N \cdot (2a_{pq}^2), \quad N = \frac{n(n-1)}{2}. \tag{3.10}$$

Using (3.6), we have

$$\begin{aligned} \mathsf{off}(B)^2 &= \mathsf{off}(A)^2 - 2a_{pq}^2 \\ &\le \mathsf{off}(A)^2 - \frac{1}{N}\mathsf{off}(A)^2 \\ &= \left(1 - \frac{1}{N}\right)\mathsf{off}(A)^2. \end{aligned} \tag{3.11}$$

Denote $A^{(k)}$ as the kth Jacobi update of $A^{(0)} = A$, then we have the iterative bound

$$\mathsf{off}(A^{(k)})^2 \le \left(1 - \frac{1}{N}\right)^k \mathsf{off}(A^{(0)})^2. \tag{3.12}$$

This implies that the classical Jacobi algorithm converges *linearly* to a diagonal matrix whose entries are the eigenvalues of A, and they may appear in any order.

Remark 3.2. Although the term $\mathsf{off}(A^{(k)})$ decrease at a rate of $(1 - 1/N)^{1/2}$, if we consider the $N = n(n-1)/2$ steps as one 'group' or 'sweep' of update, then we actually can expect *asymptotic* quadratic convergence [23, 1962] where

$$\mathsf{off}(A^{(K+1)}) \le c \cdot \mathsf{off}(A^{(K)})^2, \quad \text{for some } c,$$

where K is the number of sweeps.

3.2.2 Implementation and Testing

Before we implement the Jacobi algorithm, we need two functions (i) the function that calculates the Frobenius norm of the off-diagonal entries and (ii) the function that finds the index of the maximum entry in modulus.

(i) can be done by carefully selecting the diagonal entries and set to zero, and then calculating the Frobenius norm of the new matrix.

```
1 function offA = off(A)
2 n = length(A); % get the dimension of the matrix A
3 A(1:n+1:n*n) = 0; % set the diagonal entries to zero
4 offA = norm(A,"fro"); % calculate the norm
5 end
```

The third line of the code will eliminate the diagonal entries, then the fourth line will give us the desired output, $\mathsf{off}(A)$.

(ii) can be implemented using the following function:

```
1 function [p,q] = maxoff(A)
2 n = length(A); % dimension of matrix A
3 A(1:n+1:n*n) = 0; A = abs(A); % clear the diagonal entries
4 [val, idx1] = max(A);
5 [~, q] = max(val);
6 p = idx1(q);
7 end
```

At the fourth line, I use `[val, idx1] = max(A)` to find the index, `idx1`, of the maximum entries of each column and I store the values in `val`. Then we can apply `max()` again to find the index of the maximum entry of `val`, denoted as `q`. Finally, we locate the column number of the maximum entry by calling `idx1(q)` as shown in the sixth line.

Making use of these two functions, we can implement the classical Jacobi algorithm in MATLAB. The function `jacobi_classical` will take two inputs (i) the symmetric matrix $A^{(0)} \in \mathbb{R}^{n\times n}$ and (ii) the tolerance *tol*. I choose to output two matrices, A and V such that $\|A^{(0)}V - VA\|_2 \lesssim nu_d\|A^{(0)}\|_2$ and the number of iterations required.

Notice that if we would like to update $A^{(k)}$ by $G_k^T A^{(k)} G_k$, since $A^{(k)}, G_k \in \mathbb{R}^{n\times n}$, the cost would be $\mathcal{O}(n^3)$ flops. However, we can utilize proposition 2.12, namely, we only need to update the p and qth rows and columns of $A^{(k)}$, and this version of the update will only take $O(n)$ flops. We can see this reduction in the following code

```
1 [c,s] = jacobi_pair(A,p,q); G = [c s; -s c];
2 A([p q],:) = G'*A([p q],:);
3 A(:,[p q]) = A(:,[p q])*G;
```

From the code above, we get both the updated matrix $A^{(k+1)}$ and the orthogonal matrix G_k. Line 2 requires a matrix multiplication between $\mathbb{R}^{2\times 2} \times \mathbb{R}^{2\times n}$ which needs $\mathcal{O}(n)$ flops, and the same argument applies to line 3. Hence when updating the matrix $A^{(k)}$, we only require $\mathcal{O}(n)$ flops. Assemble all above, we have

```
1 function [V,A,counter] = jacobi_classical(A,tol)
2 counter = 0; n = length(A); V = eye(n); done_rot = true;
```

```
 3 tol1 = tol * norm(A);
 4 while done_rot
 5   if isint(counter/(n*n)), A = (A + A')/2; end % maintain symmetry
 6   done_rot = false; [p,q] = maxoff(A);
 7   if abs(A(p,q)) >  tol1 * sqrt(abs(A(p,p) * A(q,q)))
 8     counter = counter + 1; done_rot = true;
 9     [c,s] = jacobi_pair(A,p,q);
10     J = [c,s;-s,c];
11     A([p,q],:) = J'*A([p,q],:);
12     A(:,[p,q]) = A(:,[p,q]) * J;
13     V(:,[p,q]) = V(:,[p,q]) * J;
14   else
15     A = diag(diag(A)); % output diagonal matrix
16     break;
17   end
18 end
19 end
```

After several iterations, although in exact arithmetic, the matrix A should always be symmetric, in floating-point arithmetic, we need to maintain its symmetry as shown in line 5. We can then test our code by the following routine

```
clc;clear; format short e
n = 1e2; A = randn(n); A = A + A'; tol = 2^(-53);
[V,D,iter] = jacobi_classical(A,tol);
disp('norm(AV - VD)'); disp(norm(A *V - V * D))
```

Here we should expect the norm $\|AV - VD\|_2$ is about $nu_d\|A\|_2$ which is 3.0224×10^{-13} in our test.

```
norm(AV - VD)
  4.3553e-13
```

Hence the algorithm works as expected.

3.3 Cyclic-by-row Jacobi Algorithm

The classical Jacobi method requires at each step the searching of $n(n-1)$ for one maximum entry in modulus. For n is large, this can be extremely expensive. It will be better if we fixed a sequence of choice (p, q), here in cyclic order shown in the following scheme suggested by [10, 1953] and [7, 1960, Section 1.2]

$$(p_0, q_0) = (1, 2),$$
$$(p_{k+1}, q_{k+1}) = \begin{cases} (p_k, q_k + 1), & \text{if } p_k < n-1, q_k < n, \\ (p_k + 1, p_k + 2), & \text{if } p_k < n-1, q_k = n, \\ (1, 2), & \text{if } p_k = n-1, q_k = n. \end{cases}$$

We keep using (p, q) in this fashion until we meet the required tolerance and this procedure can be described in Algorithm 4.

Algorithm 4 (*The Cyclic-by-Row Jacobi algorithm*) Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$ and a positive tolerance *tol*, this algorithm overwrites A with V^TAV where V is orthogonal.

```
$V = I_n$, done_rot = true
while done_rot do
    done_rot = false
    for $p = 1, \dots, n-1$ do
        for $q = p+1, \dots, n$ do
            if $|a_{pq}| > tol \cdot \sqrt{|a_{pp}a_{qq}|}$ then
                done_rot = true
                Construct Givens rotation matrix $G$ using Algorithm 2
                Update $G^TAG \to A$
                Update $VG \to V$
            else
                Remove the off-diagonal entries of $A$
                Break
            end if
        end for
    end for
end while
```

Similarly, the structure of Algorithm 4 adapts from [3, 2001, Algorithm 2.1] and the stopping criterion is adapted from [9, 2013, Section 8.5.5], [5, 2000, Theorem 1.1] and [4, 1992, Section 1].

3.3.1 Implementation and Testing

The MATLAB code is similar to the classical Jacobi algorithm, and the only difference is the way of choosing (p, q):

```
function [V,A,iter] = jacobi_cyclic(A,tol,maxiter)
n = length(A); V = eye(n); iter = 0; done_rot = true;
while done_rot && iter < maxiter
  done_rot = false;
```

```
5      for p = 1:n-1
6        for q = p+1:n
7          if abs(A(p,q)) > tol * sqrt(abs(A(p,p)*A(q,q)))
8            done_rot = true;
9            [c,s] = jacobi_pair(A,p,q);
10           J = [c,s;-s,c];
11           A([p,q],:) = J'*A([p,q],:);
12           A(:,[p,q]) = A(:,[p,q]) * J;
13           V(:,[p,q]) = V(:,[p,q]) * J;
14         end
15       end
16     end
17     if done_rot
18       A = (A + A')/2; iter = iter + 1;
19     else
20       A = diag(diag(A));
21       return;
22     end
23   end
24   end
```

Notice that, since the cyclic-by-row Jacobi algorithm does not reduce $\mathsf{off}(A^{(k)})$ in the most optimal way, it may require much more iterations than the classical Jacobi algorithm. Hence we restrict the maximum number of iterations by `maxiter` at line 3.

Using the same routine as presented in Section 3.2.2, we have

```
norm(AV - VD)
   5.9258e-13
```

The required tolerance is $nu_d\|A\|_2 \approx 3.0260 \times 10^{-13}$ and therefore we indeed have a eigendecomposition of A at double precision.

3.4 Comparision

In the previous two sections, we discussed both the classical Jacobi algorithm and the cyclic-by-row Jacobi algorithm. However, we need to choose between them. In this section, we will examine the performance concerning both the dimension n and the condition number $\kappa(A)$. We can generate Figure 3.2 using code in Appendix B.3. Notice that, in the code, we call a function call `jacobi_eig` and this is the assembled version of the Algorithm 3 and 4 shown in Appendix B.2.

From the right figure, we can see the condition number does not affect the time too much. However, from the left figure, we observe that the downside of searching for the maximum off-diagonal entry is significant for large n. The cyclic-by-row Jacobi algorithm uses about 1/30 of the time used by the classical Jacobi algorithm for $n =$

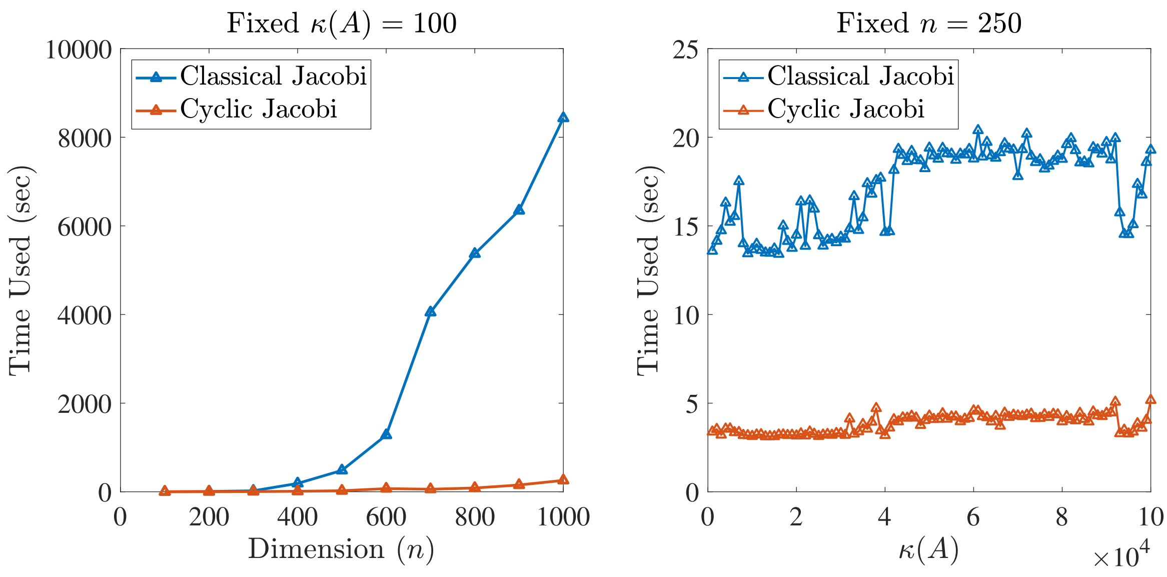

Fig. 3.2: *The time of applying both the codes* `jacobi_classical` *and* `jacobi_cyclic` *on a matrix $A \in \mathbb{R}^{n\times n}$ with respect to both dimension (n) and the condition number $\kappa(A)$. The left figure fixes the condition number $\kappa(A) = 100$ and the right figure fixes the dimension $n = 250$.*

1000. Therefore, although the classical Jacobi reduces the $\mathsf{off}(A)$ in the most optimal way, in practice, we should always stick with the cyclic-by-row Jacobi algorithm.

Chapter 4

Orthogonalization

4.1 Introduction

From Chapter 3, given a symmetric matrix $A \in \mathbb{R}^{n\times n}$, we can use the Jacobi algorithm to find an eigendecomposition $A = Q\Lambda Q^T$. However, for large n, this procedure can still be time-consuming and is not generally as fast as the symmetric QR Algorithm [9, 2013, Section 8.5]. However, the Jacobi algorithm can exploit the situation that A is almost diagonal ($\mathsf{off}(A)$ is small). In 2000, Drmač and Veselić proposed a way of speeding up this procedure using approximate eigenvectors as preconditioner [5, 2000, Section 1]. They proposed the following

1. Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$ and its approximate orthogonal eigenvector matrix Q_{app}.
2. Computing $A' = Q_{\text{app}}^T A Q_{\text{app}}$.
3. Diagonalizing A'.

Suppose Q_{app} is orthogonal in double precision and step 2 and 3 are carried out at double precision, then the eigenvalues we computed will have a small relative error and more efficient than the pure Jacobi method [5, 2000, Section 2]. In this section, we will provide a way of constructing this approximate vector matrix:

1. Compute the eigendecomposition at single precision such that

$$A = Q_\ell D_\ell Q_\ell^T, \quad \|AQ_\ell - Q_\ell D_\ell\|_2 \lesssim nu_\ell\|A\|_2, \quad \|Q_\ell^T Q_\ell - I\|_2 \lesssim nu_\ell,$$

where n is the dimension of the real symmetric matrix A and u_ℓ is the unite roundoff at single precision. This can be achieved using eig function in MATLAB via the following routine

```
[Q_low,D] = eig(single(A));
Q_low = double(Q_low);
```

2. Orthogonalize the matrix Q_ℓ to Q_d such that it is orthogonal at double precision,

$$\|Q_d^T Q_d - I\|_2 \lesssim n u_d.$$

In this chapter, we will first review two methods to orthogonalize Q_ℓ: the QR factorization and the polar decomposition. We aim to find the orthogonal (at double precision) matrix Q_d that minimizes the norm $\|Q_d - U\|_2$ where U is the exact matrix of the eigenvectors of A. However, in practice, we have no access to the matrix U, therefore, we instead find the orthogonal matrix Q_d that minimizes $\|Q_d - Q_\ell\|_2$. This problem can be transformed to: Given Q_ℓ, find the orthogonal matrix Q_d such that it satisfies

$$\min \|Q_\ell - Q_d\|_2, \quad \text{under constraint: } \|Q_d^T Q_d = I\|_2.$$

Finally, the code in MATLAB will be presented and we will access them to decide which to use in practice.

4.2 The Householder QR Factorization

The QR factorization of $A \in \mathbb{R}^{n\times n}$ is a factorization

$$A = QR$$

where $Q \in \mathbb{R}^{n\times n}$ is orthogonal and $R \in \mathbb{R}^{n\times n}$ is upper triangular. Suppose we have a QR factorization at double precision of Q_ℓ,

$$Q_\ell = Q_d R, \tag{4.1}$$

where Q_d is orthogonal at double precision, and this is already our desired preconditioner. The QR factorization can be achieved using Householder transformations in Section 2.2.2.

4.2.1 Theory of Householder QR Factorization

The idea can be illustrated using a small matrix $A \in \mathbb{R}^{4\times4}$:

$$A = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{P_1 \in \mathbb{R}^{4\times4}} \left[\begin{array}{c|ccc} \times & \times & \times & \times \\ \hline 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{array}\right] \xrightarrow{P_2 \in \mathbb{R}^{3\times3}} \left[\begin{array}{cc|cc} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ \hline 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{array}\right]$$

$$\xrightarrow{P_3 \in \mathbb{R}^{2\times2}} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = R,$$

where P_k, $k = 1, 2, 3$, are the Householder transformations. Notice that $P_2 \in \mathbb{R}^{3\times3}$ only applies to the bottom right corner of the matrix, and this can be done by embedding P_2 inside a larger 4×4 matrix via

$$\widetilde{P}_2 = \begin{bmatrix} 1 & 0 \\ 0 & P_2 \end{bmatrix}$$

Hence from the example above, we can generalize the Householder QR factorization for $A \in \mathbb{R}^{n\times n}$. It will produce a sequence of matrices $\{A_k\}_{k=1}^{n}$ where A_1 is the input A and A_n is the upper triangular matrix R. Notice, there are $k-1$ stages where the kth stage transform A_k to A_{k+1}.

At the kth stage of the Householder QR factorization, we can carefully partition the matrix A_k into the following form

$$A_k = \begin{bmatrix} R_{k-1} & w_k & C_k \\ 0 & x_k & B_k \end{bmatrix}, \quad R_{k-1} \in \mathbb{R}^{(k-1)\times(k-1)}, x_k \in \mathbb{R}^{n-k+1},$$

where R_{k-1} is upper triangular (achieved at the $(k-1)$th stage) and x_k is the vector that we should focused on the kth stage. Choose a Householder transformation P_k such that $P_k x_k = \operatorname{sign}(x_{k,1}) \|x_k\|_2 e_1$, where e_1 is the first column of I_{n-k+1} and $x_{k,1}$ is the first entry of the vector x_k. We then embed matrix P_k into a larger matrix $\widetilde{P}_k \in \mathbb{R}^{n\times n}$ via

$$\widetilde{P}_k = \begin{bmatrix} I_{k-1} & 0 \\ 0 & P_k \end{bmatrix}, \quad P_k \in \mathbb{R}^{(n-k+1)\times(n-k+1)}. \tag{4.2}$$

Then let $A_{k+1} = \widetilde{P}_k A_k$, we have

$$A_{k+1} = \begin{bmatrix} R_{k-1} & w_k & C_k \\ 0 & P_k x_k & P_k B_k \end{bmatrix} = \begin{bmatrix} R_{k-1} & w_k & C_k \\ 0 & \operatorname{sign}(x_{k,1}) \|x_k\|_2 e_1 & P_k B_k \end{bmatrix}, \tag{4.3}$$

which is closer to the upper triangular form. After $n-1$ steps, the matrix A_n will be upper triangular and we denote as R and we obtain the QR factorization of A

$$A_{n-1} = R = \widetilde{P}_{n-1}\widetilde{P}_{n-2}\cdots\widetilde{P}_1 A =: Q^T A.$$

Since $\widetilde{P}_k$ are composed of P_k, Householder matrices, and the identity matrix. Hence it is obvious that $\widetilde{P}_k$ are also symmetric and orthogonal. Then we can construct Q via $\widetilde{P}_1\cdots\widetilde{P}_{n-1}$. This procedure can be summarized in the Algorithm 5.

Algorithm 5 Given $A \in \mathbb{R}^{n\times n}$, this algorithm computes an orthogonal matrix Q and an upper triangular matrix R such that $A = QR$.

```
1: $Q = I_n$
2: **for** $k = 1 : n-1$ **do**
3:     $[v, \beta] = \texttt{house}(A(k:n, k))$                ▷ From Algorithm 1
4:     $A(k:n, k:n) = (I_{n-k+1} - \beta vv^T)A(k:n, k:n)$
5:     $Q(1:n, k:n) = Q(1:n, k:n)(I_{n-k+1} - \beta vv^T)$
6: **end for**
```

Line 4 is directly adapted from (4.3). Line 5 can be viewed by partitioning the orthogonal matrix Q_k into four parts and multiplying the matrix $\widetilde{P}_k$ structured as (4.2),

$$Q_{k+1} = Q_k\widetilde{P}_k = \begin{bmatrix} Q_1 & Q_2 \\ Q_3 & Q_4 \end{bmatrix}\begin{bmatrix} I_{k-1} & 0 \\ 0 & P_k \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 P_k \\ Q_3 & Q_4 P_k \end{bmatrix}. \tag{4.4}$$

Hence, at each step of the algorithm, we only update the final $n-k+1$ columns of the matrix Q_k. Also, focused on the line 4, if we construct P and do a matrix-matrix multiplication between $D_k, P_k \in \mathbb{R}^{(n-k+1)\times(n-k+1)}$ where $D_k = [x_k, B_k]$, then the cost will be about $2(n-k+1)^3$. If we utilize the components we computed from Algorithm 1, the Householder vector v and the coefficient β, we can reduce the computational cost by doing the matrix-vector products instead. We can rewrite $P_k D_k$ using v and β

$$P_k D_k = (I - \beta vv^T)D_k = D_k - \beta vv^T D_k = D_k - (\beta v)\cdot(v^T D_k).$$

By this procedure, we transform a matrix-matrix multiplication to

1. Vector-matrix multiplication: $v^T D_k$ requires $2(n-k+1)^2$ flops.

2. Scalar-vector inner product: βv requires $O(n-k+1)$ flops.

3. Vector-vector outer product: $(\beta v)\cdot(v^T D_k)$ requires $(n-k+1)^2$ flops.

4. Matrix-matrix subtraction: $D_k-(\beta v)\cdot(v^T D_k)$ requires $O(n-k+1)^2$ flops.

Therefore, the overall cost will be $4(n-k+1)^2$ flops. Compare with the matrix-matrix multiplication which requires $O((n-k+1)^3)$ flops, this approach is much more efficient.

4.2.2 Implementation and Testing

Based on these analyses in Section 4.2.1, we can implement it into MATLAB.

```
1 function [Q,A] = myqr(A)
2 [m,n] = size(A); Q = eye(n);
3 if m ~= n, error('Input should be a square matrix.'), end
4 for k = 1:n-1
5     [v,b] = house(A(k:n,k)); % compute the components of P
6     A(k:n,k:n) = A(k:n,k:n) - (b*v)*(v'*A(k:n,k:n)); % update A
7     Q(:,k:n) = Q(:,k:n) - (Q(:,k:n)*v)*(b*v'); % update Q
8 end
```

Recall from the last section, we discussed that at the kth step, updating A requires about $4(n-k+1)^2$ flops. Similarly, we can update Q based on (4.4) which only involves $(n-k+1)$ columns of Q. Therefore, line 7 requires $4(n-k+1)n$ flops at the kth step. We can then compute the theoretical overall cost,

$$\text{Cost}=\sum_{k=1}^{n-1}4(n-k+1)^2+4(n-k+1)n=O(10n^3/3).$$

To test our code, we are focused on two quantities, $\|Q^TQ-I\|_2$ and $\|QR-A\|_2$. The former evaluates whether our computed Q is orthogonal at double precision and the latter evaluates whether we have a QR factorization. In theory, we should have

$$\|Q^TQ-I\|_2\lesssim nu_d,\quad \|QR-A\|_2/\|A\|_2\lesssim nu_d.$$

We can use the following MATLAB routine, notice that we add the accuracy of the MATLAB built-in function qr as a reference.

```
clc; clear; format short e;
n = 1e2; ud = 2^(-53); A = randn(n);
[Q,R] = myqr(A); [Q1,R1] = qr(A); % qr factorization
orthg = norm(Q'*Q - eye(n)); % check orthogonality
qralg = norm(Q*R - A)/norm(A); % check if we have a qr factorization
fprintf('Orthogonal? %d \nMy QR accuracy is %d\n',orthg, qralg);
fprintf('MATLAB qr function accuracy is %d \n', norm(Q1*R1 - A)/norm(
    A));
fprintf('n*(machine precision) = %d\n', ud*n);
```

And we have the output:

```
Orthogonal? 3.567685e-15
My QR accuracy is 1.493857e-15
MATLAB qr function accuracy is 1.026458e-15
n*(machine precision) = 1.110223e-14
```

The first and second outputs are all smaller than $nu_d \approx 10^{-13}$ therefore our code works well. In addition, the second and the third outputs are the accuracy of functions `myqr` and `qr` and we can see these are about the same, therefore our code achieves similar accuracy as the MATLAB built-in function.

Moreover, I am interested in how $\|Q^TQ - I_n\|_2$ behaved as n increases. From Figure 4.1, we can see $\|Q^TQ - I_n\|_2$ does not increase too much when n increases and is well bounded by the reference line nu_d.

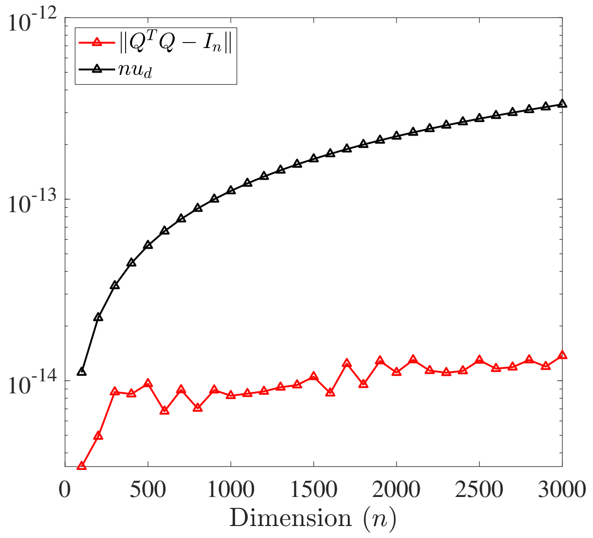

FIG. 4.1: *Behavior of* $\|Q^TQ - I_n\|_2$ *with* n *increases from* 100 *to* 3000 *with step size* 100 *and* $\kappa(A) = 100$ *using my own QR code* `myqr`*. The red line shows the computed results and the black line shows the reference line* nu_d*. The code to regenerate this graph can be found in Appendix B.4.*

By now, we have a way to orthogonalize a matrix. However, this approach does not utilize the property that our input matrix Q_ℓ is an almost orthogonal matrix. Namely, applying `myqr` to Q_ℓ has no difference from applying `myqr` to a general matrix. In the next section, we will introduce a way to orthogonalize a matrix which exploits the fact that the input is almost orthogonal.

4.3 The Polar Decomposition

In complex analysis, it is known that for any $\alpha \in \mathbb{C}$, we can write α in polar form, namely $\alpha = re^{i\theta}$. The polar decomposition is its matrix analogue. The polar decomposition can be derived from the singular value decomposition discussed in Section 2.3. Conventionally, we will present our analysis in complex but this can be restricted to real. All the norm $\|\cdot\|$ discussed in this section will be the orthogonally invariant norm.

Theorem 4.1 (Polar decomposition). *If $A \in \mathbb{C}^{n\times n}$, then there exists a unitary matrix $U \in \mathbb{C}^{n\times n}$ and a unique Hermitian positive semidefinite matrix $H \in \mathbb{C}^{n\times n}$ such that $A = UH$. We call U the unitary factor.*

Proof. By Theorem 2.14, suppose $A \in \mathbb{C}^{n\times n}$ and $\operatorname{rank}(A) = r$, then the SVD of A can be written as

$$A = P\Sigma_r V^* = \underbrace{PV^*}_{U}\underbrace{V\Sigma_r V^*}_{H} \equiv UH, \tag{4.5}$$

where $\Sigma_r = \operatorname{diag}(\sigma_1, \dots, \sigma_r, 0, \dots, 0) \in \mathbb{R}^{n\times n}$. Unitarity of U can be easily proved by its definition

$$UU^* = PV^*VP^* = I_n, \quad U^*U = VP^*PV^* = I_n.$$

H is a positive semidefinite due to its eigenvalues are the singular values of A which are all non-negative. By forming A^*A, we have $A^*A = H^*U^*UH = H^*H = H^2$, therefore H can be uniquely determined by $(A^*A)^{1/2}$ [15, 2008, Theorem 1.29]. □

Notice that, if A is full rank, then H is clearly non-singular by construction, therefore U can be uniquely determined by $U = AH^{-1}$. Similar to the QR factorization, we have a decomposition of A where one of the factors is unitary, and we expect that we can use this unitary factor as the preconditioner. The next section gives a special property which shows the unitary factor U of the polar decomposition of A is the one that minimizes the distance $\|A - U\|$.

4.3.1 Best Approximation Property

The unitary factor U of the polar decomposition of $A = UH$ satisfies the best approximation property.

Theorem 4.2 ([6, 1955, Theorem 1],[15, 2008, Theorem 8.4]). *Let $A \in \mathbb{C}^{n\times n}$ and $A = UH$ is its polar decomposition. Then*

$$\|A - U\| = \min\{\|A - Q\| \mid Q^*Q = I_n\}$$

holds for every unitarily invariant norm.

To prove Theorem 4.2, we need the following two results:

Lemma 4.3 ([18, 1960, Lemma 4]). *Let X, Y be Hermitian matrices, and denote by*

$$\xi_1 \geq \cdots \geq \xi_n, \quad \eta_1 \geq \cdots \geq \eta_n, \quad \zeta_1 \geq \cdots \geq \zeta_n$$

be the eigenvalues of X, Y and $X - Y$ respectively. Then if we denote by $(\xi - \eta)_1, \dots, (\xi - \eta)_n$ the numbers $\xi_1 - \eta_1, \dots, \xi_n - \eta_n$ arranged in non-ascending order of magnitude, then we have

$$\sum_{i=1}^{k} (\xi - \eta)_i \leq \sum_{i=1}^{k} (\zeta_i), \quad 1 \leq k \leq n.$$

Proof. See [6, 1955, Theorem 2]. □

Theorem 4.4 ([18, 1960, Theorem 5], [15, 2008, Theorem B.3]). *Let $\rho_1 \geq \cdots \geq \rho_n$ and $\sigma_1 \geq \cdots \geq \sigma_n$ be the singular values of the complex matrices A and B respectively. Then*

$$\|A - B\| \geq \|\Sigma_A - \Sigma_B\|,$$

where Σ_A and Σ_B are diagonal matrices with entries the singular values of A and B arranged in non-ascending order respectively.

Proof. We have the fact by H. Wielandt mentioned in [6, 1955, p. 113]: For any matrix M of order $n \times n$, the Hermitian matrix

$$\widetilde{M} = \begin{bmatrix} 0 & M \\ M^* & 0 \end{bmatrix}$$

of order $2n \times 2n$. The eigenvalues of $\widetilde{M}$ are precisely the singular values of M and their negatives. We denote $\tau_1 \geq \cdots \geq \tau_n$ the singular values of $A - B$ and we have

$$\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix} - \begin{bmatrix} 0 & B \\ B^* & 0 \end{bmatrix} = \begin{bmatrix} 0 & A - B \\ A^* - B^* & 0 \end{bmatrix}$$

and the eigenvalues of these three Hermitian matrices are

$$\rho_1 \geq \cdots \geq \rho_n \geq -\rho_n \geq \cdots \geq -\rho_1,$$
$$\sigma_1 \geq \cdots \geq \sigma_n \geq -\sigma_n \geq \cdots \geq -\sigma_1,$$
$$\tau_1 \geq \cdots \geq \tau_n \geq -\tau_n \geq \cdots \geq -\tau_1.$$

By Lemma 4.3, if we denote $(\rho-\sigma)_1, \dots, (\rho-\sigma)_n$ the numbers $\rho_1 - \sigma_1, \dots, \rho_n - \sigma_n$ arranged in non-ascending order of magnitude, then we have

$$\sum_{i=1}^{k} \tau_i \geq \sum_{i=1}^{k} (\rho-\sigma)_i, \quad 1 \leq k \leq n.$$

Here we focus on the proof for 2-norm, to see the proof considering all the orthogonally invariant norms, refer to [18, 1960, Theorem 1]. By Corollary 2.15

$$\begin{aligned} \|A-B\|_2 &= \|\operatorname{diag}\{\tau_1, \dots, \tau_n\}\|_2 = \tau_1 \\ &\geq (\rho-\sigma)_1 = \max_k |\rho_k - \sigma_k| \\ &= \|\operatorname{diag}\{\rho_1 - \sigma_1, \dots, \rho_n - \sigma_n\}\|_2 \\ &= \|\Sigma_A - \Sigma_B\|_2 \quad \square \end{aligned}$$

Proof of Theorem 4.2. For any unitary matrix Q, it has singular values all equal to 1. Hence by Theorem 4.4, if A has the singular value decomposition $A = P\Sigma_A V^*$, then

$$\|A - Q\| \geq \|\Sigma_A - I_n\|.$$

Remain to prove that $\|A-U\| = \|\Sigma_A - I_n\|$. From Theorem 4.1, the Hermitian part H can be written as $V\Sigma_A V^*$, hence

$$\|A-U\| = \|U(H-I_n)\| = \|H-I_n\| = \|V\Sigma_A V^* - I_n\| = \|\Sigma_A - I_n\|,$$

which completes the proof. □

This property indicates that the unitary factor of the input Q_ℓ will be the desired orthogonal matrix and then we can use it as the preconditioner. The rest of this section will focus on how to compute this factor.

4.3.2 SVD approach

The most obvious approach will be the singular value decomposition approach.

For $A \in \mathbb{C}^{n\times n}$, by Theorem 4.1, we can use the singular value decomposition to compute the polar decomposition using the following procedure:

1. Compute the singular value decomposition of $A \in \mathbb{C}^{n\times n}$ where $A = P\Sigma V^*$.
2. Form U via (4.5), $U = PV^*$.

The computational cost of computing the singular value decomposition is approximately $21n^3$ flops [9, 2013, Figure 8.6.1] with $O(n^3)$ flops to form U. This is good generally, but it does not utilize the property that, in our scenario, A is almost orthogonal ($\|A^*A - I_n\|_2$ is small). The next two sections will discuss a more sophisticated way of computing the unitary factor.

4.3.3 Newton's Method Approach

Consider the Newton iteration [15, 2008, Section 8.3]

$$\begin{aligned} X_0 &= A \in \mathbb{C}^{n\times n}, \text{nonsingular} \\ X_{k+1} &= \frac{1}{2}(X_k + X_k^{-*}), \quad k = 0, 1, \dots, \end{aligned} \tag{4.6}$$

where (X_k^{-*}) denotes the conjugate transpose of the inverse of X. The following theorem shows the iteration (4.6) converges to the unitary factor of A.

Theorem 4.5 (Newton iteration for the unitary polar factor)**.** *Suppose $A \in \mathbb{C}^{n\times n}$ is nonsingular, then the iteration (4.6) produces a sequence $\{X_k\}_k$ such that*

$$\lim_{k\to\infty} X_k = U,$$

where U is the unitary factor of the polar decomposition of A.

Proof. See [13, 1986, Section 3.2].

Let the singular value decomposition of A be $P\Sigma Q^*$, where P and Q are unitary. Then by Theorem 4.1, we have $A = UH$ where $U = PQ^*$ and $H = Q\Sigma Q^*$. The trick of the proof is to define a new sequence D_k where

$$D_k = P^* X_k Q. \tag{4.7}$$

From iteration (4.6), we have

$$\begin{aligned} D_0 &= P^* X_0 Q = P^* A Q = \Sigma, \\ D_{k+1} &= P^* X_{k+1} Q = P^* \frac{1}{2}(X_k + X_k^{-*})Q \\ &= \frac{1}{2}(P^* X_k Q + P^* X_k^{-*} Q). \end{aligned}$$

Notice that $P^*X_kQ = D_k$ and $P^*X_k^{-*}Q = (P^*X_kQ)^{-*}$, hence the iteration becomes

$$D_0 = \Sigma, \quad D_{k+1} = \frac{1}{2}(D_k + D_k^{-*}). \tag{4.8}$$

Since D_0 is diagonal with positive diagonal entries (since A is full rank), hence the iteration (4.8) is well-defined and will produce a sequence of diagonal matrices $\{D_k\}$ and we can write $D_k = \mathrm{diag}(d_i^{(k)})$, where $d_i^{(k)} > 0$. Using this notation, we can rewrite the iteration (4.8) elementwise:

$$d_i^{(0)} = \sigma_i, \quad d_i^{(k+1)} = \frac{1}{2}\left(d_i^{(k)} + \frac{1}{d_i^{(k)}}\right)$$

where $i = 1, \dots, n$. By an argument discussed in [12, 1964, p. 84], we can write

$$d_i^{(k+1)} - 1 = \frac{1}{2d_i^{(k)}}\left({d_i^{(k)}}^2 - 2d_i^{(k)} + 1\right) = \frac{1}{2d_i^{(k)}}\left(d_i^{(k)} - 1\right)^2 \tag{4.9}$$

This implies that the diagonal entries of D_k converge to 1 quadratically as $k \to \infty$. Since the argument holds for any $i = 1, \dots, n$, we conclude that $\lim_{k\to\infty} D_k = I_n$ and use the definition of D_k we have

$$\lim_{k\to\infty} X_k = PD_kQ^* = PQ^* = U,$$

as $k \to \infty$, where U is the unitary factor of the polar decomposition of A. □

The Newton iteration for the unitary factor of a square matrix A can also be derived by applying the Newton's method to the equation $X^*X = I_n$. Consider a function $F(X) = X^*X - I_n$, then X is unitary if X is a zero of the function $F(X)$.

Suppose Y is an approximate solution to the equation $F(\widetilde{X}) = 0$, we can write $\widetilde{X} = Y + E$ and substitute $\widetilde{X}$ into the equation $F(\widetilde{X}) = 0$

$$(Y + E)^*(Y + E) - I_n = Y^*Y + Y^*E + E^*Y + E^*E - I_n = 0.$$

Dropping the second order term, we get Newton iteration $Y^*Y + Y^*E + E^*Y - I_n = 0$ and this is the Sylvester equation for E [15, 2008, Problem 8.18]. We aim to solve E and update Y by $Y + E$, which gives the Newton iteration at the kth step:

$$\begin{cases} X_k^*X_k + X_k^*E_k + E_k^*X_k - I_n = 0, \\ X_{k+1} = X_k + E_k. \end{cases} \tag{4.10}$$

Assume that $X_k^* E_k$ is Hermitian, namely $X_k^* E_k = E_k^* X_k$, then we can further simplify the iteration by rewritten (4.10) as $X_k^* E_k = (I_n - X_k^* X_k)/2$ and the expression we got for $X_k^* E_k$ is indeed Hermitian, hence our assumption is valid. Therefore, we have a explicit expression for E_k, $E_k = (X_k^{-*} - X_k)/2$, and the iteration (4.10) becomes

$$X_{k+1} = X_k + \frac{1}{2}(X_k^{-*} - X_k) = \frac{1}{2}(X_k^{-*} + X_k)$$

choosing $X_0 = A$, we have our desired Newton iteration (4.6).

Convergence of Newton Iteration

The convergence of the iteration (4.6) can be seen from the proof of Theorem 4.5.

Theorem 4.6 (Convergence of the iteration (4.6), [15, 2008, Theorem 8.12])**.** *Let $A \in \mathbb{C}^{n\times n}$ be non-singular. Then the Newton iterates X_k in (4.6) converges quadratically to the unitary polar factor U of A with*

$$\|X_{k+1} - U\| \le \frac{1}{2}\|X_k^{-1}\|\|X_k - U\|^2,$$

where $\|\cdot\|$ is any unitarily invariant norm.

Proof. Given $A \in \mathbb{C}^{n\times n}$, the singular value decomposition of A is $P\Sigma Q^*$ and the unitary factor is $U = PQ^*$. From (4.9), we have

$$d_i^{(k+1)} - 1 = \frac{1}{2d_i^{(k)}}\left(d_i^{(k)} - 1\right)^2.$$

This is true for all $1 \le i \le n$, therefore we can rewrite this in matrix form

$$\begin{bmatrix} d_1^{(k+1)} - 1 & & \\ & \ddots & \\ & & d_n^{(k+1)} - 1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1/d_1^{(k)} & & \\ & \ddots & \\ & & 1/d_n^{(k)} \end{bmatrix}\begin{bmatrix} d_1^{(k)} - 1 & & \\ & \ddots & \\ & & d_n^{(k)} - 1 \end{bmatrix}^2 \tag{4.11}$$

Recall $D_k = \mathrm{diag}(d_i^{(k)})$, then (4.11) is same as $D_{k+1} - I = D_k^{-1}(D_k - I)^2/2$. Therefore we can conclude that D_k converges to the identity I quadratically and therefore X_k converges to the unitary factor U quadratically.

From $D_{k+1} - I = D_k^{-1}(D_k - I)^2/2$, by taking the norm on both sides we have

$$\|D_{k+1} - I\| = \frac{1}{2}\|D_k^{-1}(D_k - I)^2\| \le \frac{1}{2}\|D_k^{-1}\|\|D_k - I\|^2 \tag{4.12}$$

The final inequality comes from the fact that every unitarily invariant norm is subordinate [2, 1997, Proposition IV 2.4]. Also, the unitary invariance implies the following three equalities,

- $\|D_{k+1} - I\| = \|PD_{k+1}Q^* - PQ^*\| = \|X_{k+1} - U\|$.
- $\|D_k^{-1}\| = \|QD_k^{-1}P^*\| = \|X_k^{-1}\|$.
- Using the same argument as 1, we have $\|D_k - I\| = \|X_k - U\|$.

As a result, (4.12) can be rewritten as

$$\|X_{k+1} - U\| \leq \frac{1}{2}\|X_k^{-1}\|\|X_k - U\|^2.$$

□

We can test the iteration (4.6) by the following code

```
A = my_testmatrices(10,100); X = A;
for i = 1:10
    Y = inv(X);
    X = (Y' + X)/2;
end
[P,~,V] = svd(A); U = P*V';
disp(norm(X-U));
```

With only 10 iterations, the distance between our computed polar factor and the theoretical polar factor (formed by singular value decomposition) is as small as the unit roundoff. However, the disadvantage of this iteration is that it requires the explicit form of the inverse of a matrix at each step, therefore we can introduce another method using one step of the Schulz iteration [1, 1965] which will remove the requirement of the matrix inverse.

4.3.4 Newton–Schulz Iteration Approach

Given $A \in \mathbb{C}^{n\times n}$, an iterative method for computing the inverse A^{-1}, proposed by Schulz [21, 1933, p. 58], is defined by

$$X_{k+1} = X_k(2I - AX_k), \quad k = 0, 1, 2, \dots, \tag{4.13}$$

where X_0 is an approximation to A^{-1}. Based on the (4.6), we can use a one-step Schulz iteration to approximate the inverse of X_k^* which can be written as $Y_1 = Y_0(2I - X_k^* Y_0)$, where Y_0 is an approximation to X_k^{-*}. In our situation, the input X_0 is an almost unitary matrix ($\|X_0^*X_0 - I\|_2$ is small) and the output is U which is unitary. Hence the sequence generated by the Newton iteration, $\{X_k\}$, should always be almost unitary, therefore X_k should be a good approximate to $(X_k^*)^{-1}$. The idea of Newton–Schulz iteration can be formulated by the Algorithm 6.

Algorithm 6 Given $A \in \mathbb{C}^{n\times n}$ such that $\|A^*A - I\|_2 \approx u_\ell$, the algorithm computes the unitary factor U of A defined by Theorem 4.1.

```
1: $X_0 = A$
2: for $k = 0, 1, 2, \dots$ do
3:     $X_k^{-*} = X_k(2I - X_k^*X_k)$                      ▷ One-step Schulz iteration
4:     $X_{k+1} = \frac{1}{2}\left(X_k^{-*} + X_k\right)$    ▷ Newton iteration
5: end for
```

Notice that, the one-step Schulz iteration can be embedded into the Newton iteration by substituting the X_k^{-*} from line 3 into line 4 and we have

$$X_{k+1} = \frac{1}{2}\left(X_k(2I - X_k^*X_k) + X_k\right) = \frac{1}{2}X_k\left(3I - X_k^*X_k\right), \quad X_0 = A, \tag{4.14}$$

and this is known as the Newton–Schulz iteration [15, 2008, Section 8.3].

Convergence of Newton–Schulz Iteration

We would like to know, the iteration (4.14) converges to 1 under which condition. Using the similar approach as proof of Theorem 4.5, by writing $D_k = P^*X_kQ$ where P and Q are the unitary parts of the singular value decomposition of $A = P\Sigma Q^*$, we can consider the iteration (4.14) as the scalar form:

$$d_i^{(k+1)} = \frac{1}{2}d_i^{(k)}\left(3 - {d_i^{(k)}}^2\right), \quad d_i^{(0)} = \sigma_i, \quad \text{for } k = 0, 1, \dots, \tag{4.15}$$

where σ_i is the ith singular value of A. To see the convergence of (4.14), we first find the interval of convergence of (4.15).

Lemma 4.7. *If $\sigma_i \in (0, \sqrt{3})$, then the iteration (4.15) converges quadratically to 1.*

Proof. If $\sigma_i = 1$, then the iteration converges immediately. To simplify the notation, we rewrite the iteration (4.15) as

$$d_{k+1} = f(d_k) = \frac{1}{2}d_k(3 - d_k^2), \quad k = 0, 1, \dots. \tag{4.16}$$

Case 1: If $d_k \in (0,1)$, then $(3-d_k^2)/2 > 1$ which gives $d_k(3-d_k^2)/2 > d_k$. Differentiate $f(d_k)$ once we have $f'(d_k) = -d_k^2 + (3-d_k^2)/2$ and this will always be positive for $d_k \in (0,1)$. Therefore, if we start with $d_k \in (0,1)$, then $d_k < f(d_k) = d_{k+1} < f(1) = 1$, namely, start from $d_0 \in (0,1)$, then the iteration (4.16) will converges to 1.

Case 2: If $d_k \in (1,\sqrt{3})$, then by similar approach we have the following

$$f(1) = 1, \quad f(\sqrt{3}) = 0, \quad f'(d_k) < 1 \quad \text{for } d_k \in (1,\sqrt{3}).$$

Therefore, the function f will monotonically decreasing from 1 to 0 within the interval $(1,\sqrt{3})$ and $\sup_{d_k\in(1,\sqrt{3})} f(d_k) = 1$. Hence, if $d_k \in (0,\sqrt{3})$, then $f(d_k)$ will belongs to $(0,1)$, then the iteration (4.16) converges to 1 as discussed in case 1. The properties of the function $f(d_k)$ in both case 1 and case 2 can be seen from the Figure 4.2 which shows that for $d_k \in (0,1)$, the function f monotonically increasing from 0 to 1 and for $d_k \in (1,\sqrt{3})$, the function f monotonically decreasing from 1 to 0.

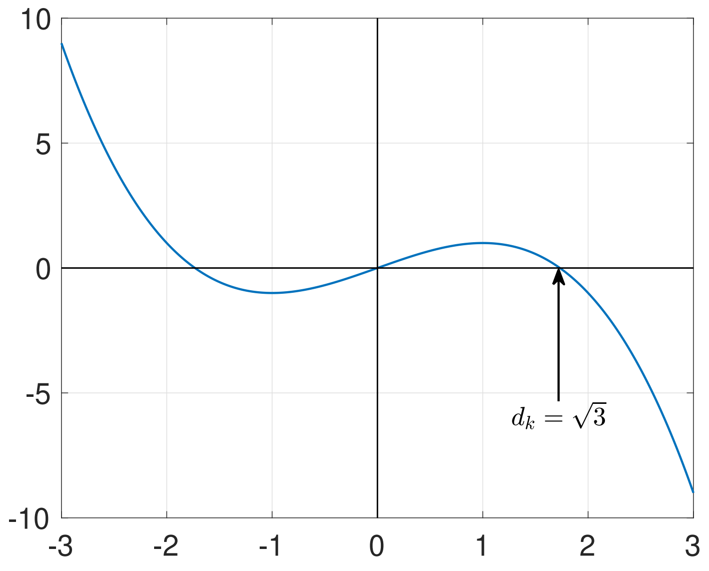

FIG. 4.2: *The plot of $f(d_k)$ defined in (4.16) over $[-3,3]$. Three zeros of $f(d_k)$ are $d_k = -\sqrt{3}, 0$ and $\sqrt{3}$.*

Other parts of the real line may still converges to 1 but the interval of converges are too short to consider and control (for example, if we start with $d_0 = 2.2$, then $d_1 = f(d_0) = -2.0240$ and $d_2 = 1.1097 \in (0,\sqrt{3})$ which will converges to 1).

Remain to show that the convergence is of order 2. By consider the difference $d_{k+1} - 1$ (similar approach as (4.9)), we have

$$d_{k+1} - 1 = -\frac{1}{2}(d_k - 1)^2(d_k + 2)$$

which implies quadratic convergence. □

From the quadratic convergence of the scalar Newton–Schulz iteration, we have the following theorem.

Theorem 4.8 (Convergence of Newton–Schulz iteration (4.14))**.** *For $A \in \mathbb{C}^{n\times n}$ of rank n which has the polar decomposition $A = UH$, if $\sigma_i(A) \in (0, \sqrt{3})$ or equivalently $\|A\|_2 < \sqrt{3}$, then the Newton–Schulz iteration*

$$X_{k+1} = \frac{1}{2}X_k(3I - X_k^* X_k), \quad X_0 = A,$$

converges to the unitary factor U quadratically as $k \to \infty$ and

$$\|X_{k+1} - U\|_2 \leq \frac{1}{2}\|X_k + 2U\|_2 \|X_k - U\|_2^2. \tag{4.17}$$

Proof. The result is obvious follows the proof of Theorem 4.5 and 4.6 and Lemma 4.7. □

Implementation and Testing

The Newton–Schulz iteration can be utilized on the almost orthogonal matrix Q_ℓ as discussed in Section 4.1 since the eigenvalues of Q_ℓ are all near 1 which definitely satisfies the convergence condition for the Newton–Schulz iteration. For $A \in \mathbb{C}^{n\times n}$, iteration (4.14) can be formalized by Algorithm 7 with restrictions on the singular values of A based on Theorem 4.8.

Algorithm 7 Given a matrix $A \in \mathbb{C}^{n\times n}$ with $\sigma(A) \subset (0, \sqrt{3})$, this algorithm computes the unitary factor U of the polar decomposition of $A = UH$ using the Newton–Schulz iteration (4.14).

```
1: $X_0 = A$
2: for $k = 0, 1, \dots$ do
3:     Compute $X_k^* X_k$
4:     if $\|X_k^* X_k - I\|_F \lesssim n u_d$ then
5:         Break, output $X_k$
6:     end if
7:     $X_{k+1} = \frac{1}{2}X_k(3I - X_k^* X_k)$
8: end for
```

Notice that, in line 3, we use the Frobenius norm to examine the orthogonality instead of the 2-norm, this is due to the Frobenius norm being easier and cheaper to compute. We can further simplify the algorithm based on the following analysis.

From now on, the explanation will focus on the 2-norm, but it can be generalized to all unitarily invariant norms. Suppose Q_ℓ has a singular value decomposition $Q_\ell = P\varSigma Q^*$, then we can rewrite $\|Q_\ell^* Q_\ell - I\|_2$ as

$$\|Q_\ell^* Q_\ell - I\|_2 = \|Q\varSigma^* P^* P\varSigma Q^* - I\|_2 = \|\varSigma^2 - I\|_2 \approx n u_\ell.$$

Since $\varSigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_n)$ where σ_i are sorted in non-ascending order. Using Corollary 2.15 and the fact that the singular values of the almost unitary matrix are around one, we have $\|\varSigma^2 - I\|_2 \approx \sigma_1^2 - 1 \approx n \cdot 10^{-8}$, therefore

$$\|\varSigma - I\|_2 \approx \sigma_1 - 1 \approx \sqrt{1 + n \cdot 10^{-8}} - 1.$$

From relation (4.17) and the fact that $\|Q_\ell - U\|_2 = \|\varSigma - I\|_2$, using the notations in (4.14), we have

$$\|X_1 - U\|_2 \leq \|X_0 - U\|_2^2 = \|Q_\ell - U\|_2^2 = \|\varSigma - I\|_2^2 \approx (\sqrt{1 + n \cdot 10^{-8}} - 1)^2.$$

Similarly, after the second iteration, we have

$$\|X_2 - U\|_2 \leq \|X_1 - U\|_2^2 \approx (\sqrt{1 + n \cdot 10^{-8}} - 1)^4.$$

We can compare this quantity with the desired tolerance $n u_d$ and we can produce Figure 4.3 using Appendix B.5.

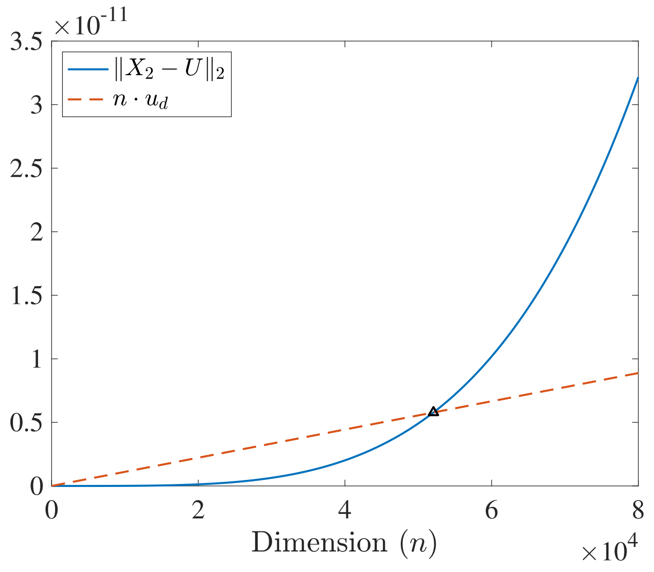

FIG. 4.3: *The figure shows, after two iterations, the difference between the matrix X_2 (from Algorithm 7) and the unitary factor U with respect to the dimension. The black marked point is the intersection at $n = 52071$.*

From Figure 4.3, we observe that for n less than 52000, Algorithm 7 will converge in 2 steps. If the dimension exceeds 52400, by the same analysis, for any $Q_\ell \in \mathbb{R}^{n\times n}$, if $n < 2172548$, then the Newton–Schulz iteration will converge in three steps. In practice, MATLAB is not able to generate such a big matrix. Therefore, we can remove the terminate condition in Algorithm 7 as shown in Algorithm 8,

Algorithm 8 (Practical algorithm for Q_ℓ) Given a matrix $Q_\ell \in \mathbb{C}^{n\times n}$ such that $\|Q_\ell^* Q_\ell - I\|_2 \approx n\cdot 10^{-8}$ and $n < 2\times 10^6$, this algorithm computes the unitary factor U of the polar decomposition of $Q_\ell = UH$ using the Newton–Schulz iteration (4.14).

```
1: $X_0 = Q_\ell$
2: if $n \geq 52400$ then
3:     $\alpha = 2$
4: else
5:     $\alpha = 1$
6: end if
7: for $k = 0 : \alpha$ do
8:     $X_{k+1} = \frac{1}{2} X_k (3I - X_k^* X_k)$
9: end for
10: Output $X_{\alpha+1}$
```

Implementation of Algorithm 8 in MATLAB gives

```
function A = newton_schulz(A)
[~,n] = size(A);
if n >= 52000, iter = 3;
else, iter = 2; end
for k = 1:iter
  A = 0.5*A*(3*eye(n) - A'*A);
end
```

For each iteration, there are only 2 matrix-matrix multiplications which cost $4n^3$ flops and hence the total cost when applying `newton_schulz` to $A \in \mathbb{C}^{n\times n}$ will be about $8n^3 + O(n^2)$ flops. Compare to the singular value decomposition approach discussed in Section 4.3.2 which costs about $21n^3$, the Newton–Schulz iteration is usually the preferred method when the input is almost unitary.

We will perform the following test: Given a matrix $A \in \mathbb{R}^{n\times n}$ with fixed condition number, here $\kappa(A) = 100$, we compute its eigendecomposition at single precision $A = QDQ^*$. Then we use this Q as our input of `newton_schulz` and see how the quantity $\|Q_d^* Q_d - I\|_2$, where Q_d is the output of the function, changes as the dimension n changes. Using the code from Appendix B.6, we produce Figure 4.4.

From Figure 4.4, the quantity $\|Q_d^* Q_d - I\|_2$ is well bounded by the tolerance nu_d and the result is consistent with our expectations and analyses.

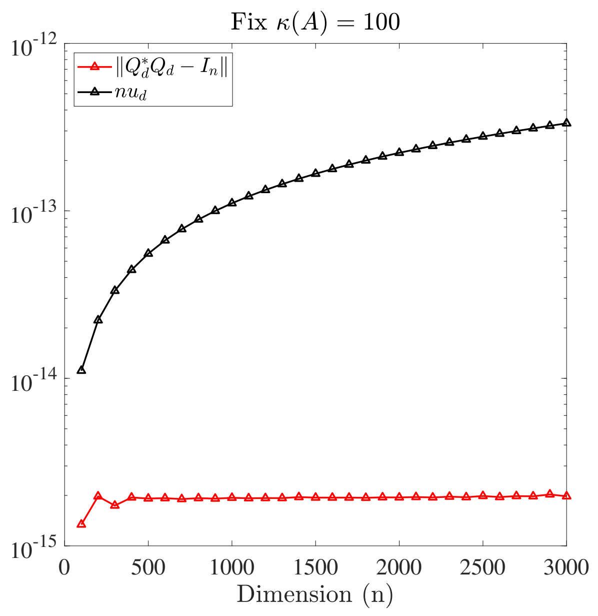

FIG. 4.4: *This figure shows how the quantity $\|Q_d^* Q_d - I\|_2$ increases as the dimension of the input matrix Q increases while we fix $\kappa(A)$. The red line shows the quantity $\|Q_d^* Q_d - I\|_2$ after orthogonalization using Newton–iteration and the black line is the reference line nu_d.*

4.4 Numerical Comparison

In this section, we will examine the performance and accuracy of both the QR factorization and the polar decomposition using the Newton–Schulz iteration. Theoretically, for our almost orthogonal matrix Q_ℓ, the QR factorization requires about $10/3n^3$ flops and the Newton–Schulz iteration method requires about $8n^3$ flops. However, MATLAB computes matrix-matrix products faster, hence Newton–Schulz iteration has the potential to be faster than the QR factorization. We set up our numerical comparison via the following

1. Generate the symmetric matrix $A \in \mathbb{R}^{n\times n}$ and its approximate eigendecomposition $A = Q_\ell D_\ell Q_\ell^*$. Here, we fixed $\kappa(A) = 100$. Even though no matter how we change the condition number of A, the eigenvector matrix at single precision, Q_ℓ, will always have condition number near 1, recall this is also the reason why we can always apply the Newton–Schulz iteration to the matrix Q_ℓ.

2. For different n, applying `myqr` and `newton_schulz` to Q_ℓ and output Q_d which has been numerically proved that it is orthogonal at double precision, then we examine these functions via

   (a) Accuracy: $\|Q^* Q - I\|_2$.

(b) Performance : time used measured by `tic`,`toc`.

4.4.1 Accuracy

The accuracy is measured by $\|Q_d^* Q_d - I\|_2$ where Q_d is the output of `myqr` and `newton_schulz`. From previous testings, both QR factorization and Newton–Schulz iteration are accurate enough to generate an orthogonal matrix at double precision. Therefore, here we are comparing the accuracy between these two methods shown in Figure 4.5. In addition, we add the accuracy of the MATLAB built-in function `qr()` served as a reference line.

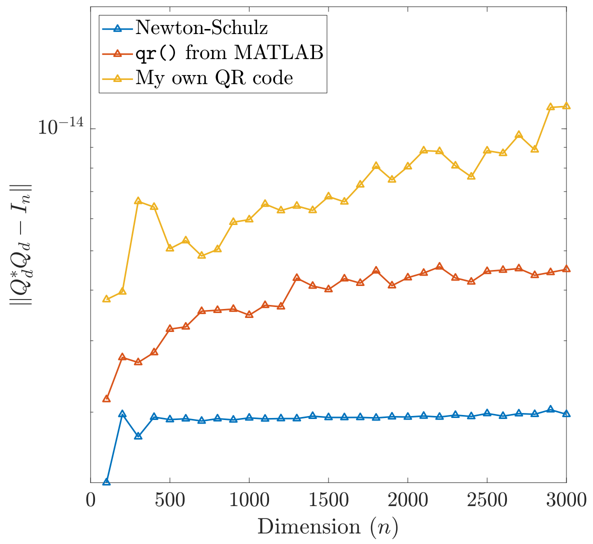

FIG. 4.5: *Behavior of $\|Q_d^* Q_d - I_n\|_2$ changes as the dimension n increases using both the QR factorization and the Newton Schulz iteration. The blue line shows the results using the Newton–Schulz iteration, the red line shows the results using the MATLAB built-in* `qr()` *function and the black line shows the results using the my own QR factorization code* `myqr()` *from Section 4.2.2. The code to regenerate this graph can be found in Appendix B.7.*

From the figure, we can see that the Newton–Schulz iteration is generally more accurate than the QR factorization method.

4.4.2 Performance

The performance can be measured by the MATLAB built-in function `tic` and `toc`. Using similar code from Section 4.4.1, we are able to compare the time used by `myqr` and `newton_schulz` and produce Figure 4.6.

From the figure, the Newton–Schulz iteration is generally faster than the QR factorization approach.

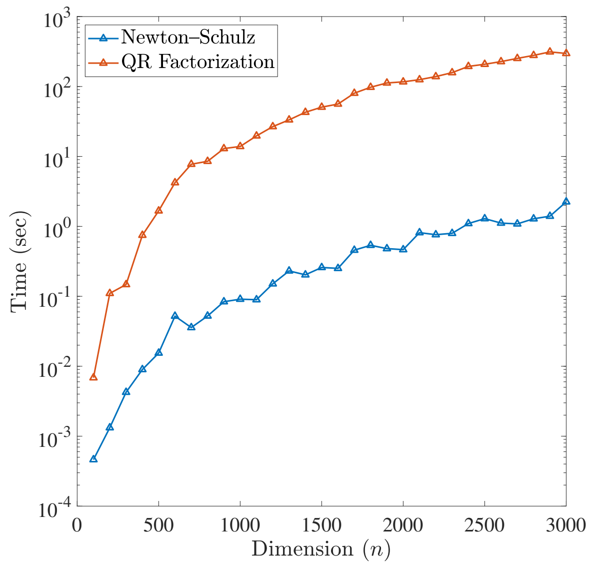

FIG. 4.6: *The time of applying the QR factorization and the Newton–Schulz iteration to the same almost orthogonal matrix Q. The red line shows the results using the Newton–Schulz iteration and the black line shows the results using the QR factorization. The code to regenerate this graph can be found in Appendix B.8.*

Consequently, based on the accuracy and performance, we shall stick with the Newton–Schulz iteration approach as the method to orthogonalize the matrix Q_ℓ.

Chapter 5

Mixed Precision Jacobi Algorithm

In this chapter, a mixed precision algorithm will be presented with intensive testings on its convergence, speed and accuracy. As mentioned in the Section 2.4, we will perform the testings for both mode `'geo'` and mode `'ari'`.

5.1 Algorithm

As discussed from [5, 2000], we can improve the Jacobi algorithm by using a preconditioner. From the discussions in Chapter 3 and 4, we assemble them all and collapse into the following algorithm,

Algorithm 9 (*Mixed precision Jacobi algorithm*) Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$ and a positive tolerance *tol*, this algorithm computes the eigendecomposition of A using the cyclic-by-row Jacobi algorithm with preconditioning.

```
1: Compute the eigendecomposition in single precision, $A = Q_\ell D_\ell Q_\ell^T$.
2: Orthogonalize $Q_\ell$ by using the Newton–Schulz iteration as shown in the Algorithm 8, and we have the output $Q_d$.
3: Precondition $A$ via $A_{\text{cond}} = Q_d^T A Q_d$ and apply the cyclic-by-row Jacobi algorithm on $A_{\text{cond}}$ to get the orthogonal matrix $V$ and the diagonal matrix $\Lambda$ such that $V\Lambda V^T = A_{\text{cond}}$.
4: Output the diagonal matrix $\Lambda$ and the orthogonal matrix $Q = Q_d V$ such that $A = Q\Lambda Q^T$.
```

5.1.1 Implementation and Testing

It is straightforward to implement Algorithm 9 by utilizing the existing codes. MATLAB code can be found in Appendix B.9.

We would like to first access its accuracy concerning both the dimension and the condition number. Using code in Appendix B.10 we can produce Figure 5.1.

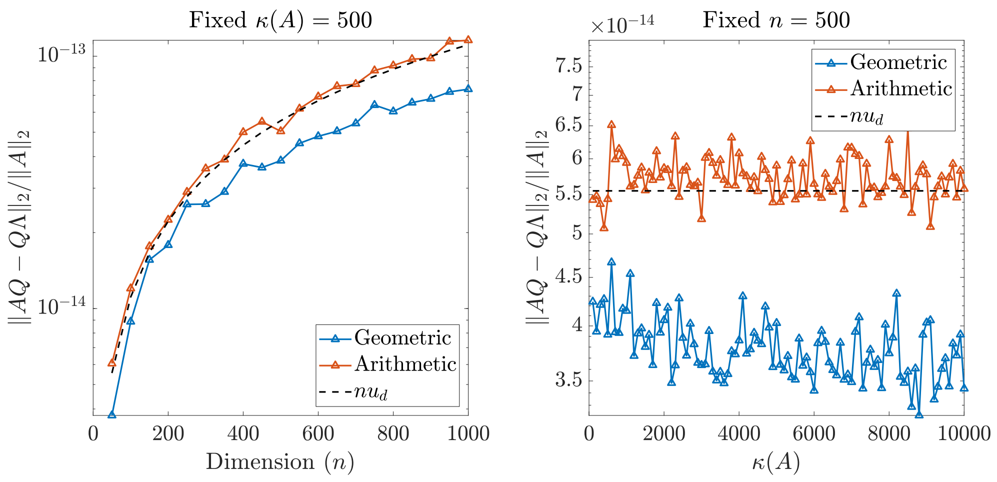

FIG. 5.1: *Behavior of the relative error $\|AQ - Q\Lambda\|_2/\|A\|_2$ with the dimension n and the condition number $\kappa(A)$ increases. The matrices for the testings on the blue line have geometrically distributed singular values and those on the red line have arithmetically distributed singular values. The black line is the reference tolerance nu_d.*

As shown from the figure, the Algorithm 9 gives more accurate results for the matrices with geometrically distributed singular values. But regardless of the distributions, the computed Q and Λ satisfies the condition

$$\|AQ - Q\Lambda\|_2 \lesssim nu_d\|A\|_2,$$

and therefore this algorithm can be used to find the eigendecomposition of a real symmetric matrix in practice. The remaining of this chapter is to find out two things

1. How many sweeps of Jacobi will be necessary for convergence for a preconditioned real symmetric matrix A_{cond} as described in Algorithm 9?

2. How much time can be saved by using preconditioning?

5.2 Convergence and Speed

In this section, we will analyze how many sweeps will be sufficient for the cyclic-by-row Jacobi algorithm converges for a preconditioned symmetric matrix A_{cond}.

5.2.1 Preconditioning

Using the notations from Algorithm 9, Q_d is the nearest orthogonal matrix compare to the approximate eigenvector matrix Q_ℓ, hence A_{cond} is expected to be almost diagonal. Consequently, we expect this eigenvalue problem should be able to be solved more quickly by the Jacobi algorithm [5, 2000, Sec. 1, Corollary 3.2]. Quantitatively, after preconditioning, the magnitude of the off-diagonal entries of A_{cond} should be no more than $nu_\ell\|A\|_2$, which is equivalently saying $\mathsf{off}(A_{\text{cond}}) \lesssim n^2(n-1)u_\ell\|A\|_2$. In practice, this bound is sharper than the theoretical bound and we can have

$$\mathsf{off}(A_{\text{cond}}) \lesssim nu_\ell\|A\|_2. \tag{5.1}$$

This upper bound can be numerically tested by the following procedure: First, generate a symmetric matrix $A \in \mathbb{R}^{n\times n}$ with controlled dimension and condition number, then we perform steps 1,2 and 3 in the Algorithm 9 to see how much $\mathsf{off}(A)$ is reduced based on this precondition technique. Using the code in Appendix B.11, we generate Figure 5.2.

From the top-left, top-right and bottom-left parts of the Figure 5.2, we see that for $n = 50, 250$ and 500, the quantity $\mathsf{off}(A_{\text{cond}})/\|A\|_2$ is always smaller or similar to nu_ℓ regardless of its singular values distribution. Notice that, as n changes from 50 to 500, the perturbation of the quantity $\mathsf{off}(A_{\text{cond}})/\|A\|_2$ becomes more and more negligible compare to nu_ℓ. Together with the bottom-right figure, as the size of A passes 100, the quantity $\mathsf{off}(A_{\text{cond}})/\|A\|_2$ is always well bounded by nu_ℓ. Based on these observations, we verify the idea of preconditioning and prove numerically that after preconditioning A by $Q_d^T A Q_d$, the off-diagonal entries are reduced significantly and this property can be captured by the Jacobi algorithm.

5.2.2 Quadratic Convergence

Based on the preconditioning process, in this section, we will discuss the convergence of the cyclic Jacobi algorithm applied to a preconditioned real symmetric matrix. Then we will deliver some numerical testings concerning different dimensions and condition numbers. Finally, we will test the improvements in speed by using the preconditioners.

Recall that the number of iterations in one sweep is equal to half of the number of

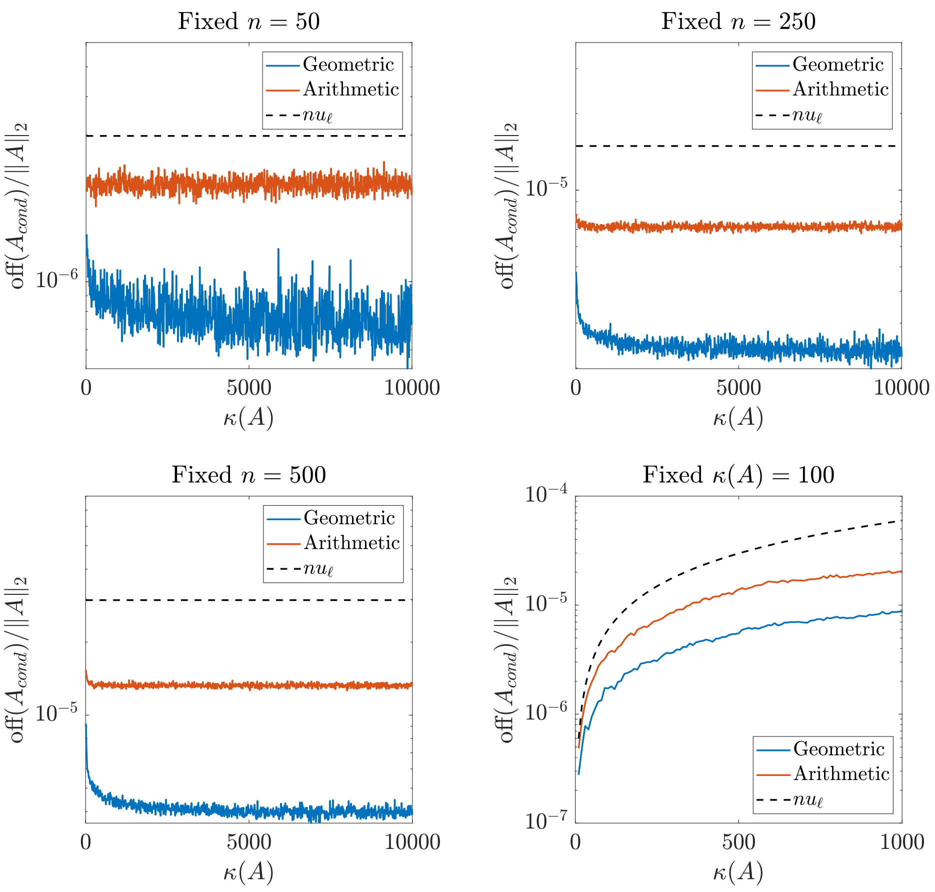

FIG. 5.2: *Top-left, Top-right, Bottom-left: These figures show the* $\mathsf{off}(A_{\text{cond}})/\|A\|_2$ *for* $A \in \mathbb{R}^{n\times n}$ *with different condition number while fixing* $n = 50, 250$ *and* 500 *respectively and varying* $\kappa(A)$ *from* 10 *to* 10^3. *Bottom-right: This figure shows the* $\mathsf{off}(A_{\text{cond}})/\|A\|_2$ *for* $A \in \mathbb{R}^{n\times n}$ *with different* n *while fixing* $\kappa(A) = 100$ *and varying* n *from* 10 *to* 1000. *The blue and red line represent the testings on the real symmetric matrices with geometrically and arithmetically distributed singular values respectively.*

off-diagonal entries,

$$\text{one sweep of } A = \frac{n(n-1)}{2}, \quad A \in \mathbb{R}^{n\times n}.$$

From the previous derivation, the Jacobi algorithm is proved to converge linearly. However, an improvement by Schönhage in 1961 [20, 1961] stated that the Jacobi algorithm for symmetric matrices with distinct eigenvalues is converging quadratically. Let us denote $A^{(k)} \in \mathbb{R}^{n\times n}$ be the matrix produced by the kth iteration of the Jacobi algorithm and therefore $A^{(0)} \in \mathbb{R}^{n\times n}$ is the original input matrix. Denote $\lambda_1, \dots, \lambda_n$ be the eigenvalues of $A^{(0)}$ and suppose we have

$$|\lambda_i - \lambda_j| < 2\delta, \quad i \neq j. \tag{5.2}$$

Suppose we reach the stage that the $\mathsf{off}(A^{(k)}) < \delta/8$ and the angles of rotation generate by each iteration are smaller than $\pi/4$, as controlled in Section 3.1.1, Schönhage's result said

$$\mathsf{off}(A^{(N+k)}) \le \frac{n\sqrt{(n-2)/2}}{\delta}\mathsf{off}(A^{(k)})^2, \quad N = \frac{n(n-1)}{2},$$

which implies quadratic convergence. Later in 1962, Wilkinson provided a sharper bound [23, 1962, Section 3]. Under the same condition as described by Schönhage, Wilkinson proposed

$$\mathsf{off}(A^{(N+k)}) \le \mathsf{off}(A^{(k)})^2/\delta.$$

In 1966, van Kempen proved the quadratic convergence without the assumption of distinct eigenvalues. In his paper [22, 1966, Section 2], instead of define δ as in (5.2), he denoted δ as

$$\min_{\lambda_i \ne \lambda_j} |\lambda_i - \lambda_j| \ge 2\delta. \tag{5.3}$$

Suppose after k iterations, $\mathsf{off}(A^{(k)}) < \delta/8$, then

$$\mathsf{off}(A^{(k+N)}) \le \frac{\sqrt{\frac{17}{9}}\mathsf{off}(A^{(k)})^2}{\delta}$$

Note that although the above bound is correct, the proof in [22, 1966] is not correct. This mistake was unveiled by Hari who proposed a sharper bound [11, 1991, Section 2].

Back to our situation, as described in Section 5.2.1, after preconditioning, $\mathsf{off}(A) \lesssim nu_\ell \|A\|_2$. Assuming our eigenproblem is well conditioned and the minimum distance between the distinct eigenvalues is large enough such that

$$\mathsf{off}(A) \lesssim 0.1 < \delta/8$$

where δ is defined by (5.3). Then we can expect quadratic convergence and the number of iterations should not exceed five since it will only take 4 sweeps for the quantity $\mathsf{off}(A)$ reduces from 0.1 to 10^{-16} ($0.1^4 = 10^{-16}$).

5.2.3 Numerical Testing

By referring to the function `jacobi_precondi` in Appendix B.9, we can output the number of iterations required. Therefore, we can investigate the number of sweeps required using the routine from Appendix B.12. The procedure is exactly the same as the testing in Section 5.1.1 and we can produce Figure 5.3.

Suppose our testing matrices are not ill-conditioned, then for the matrices with arithmetically distributed singular values, it usually requires two iterations for the

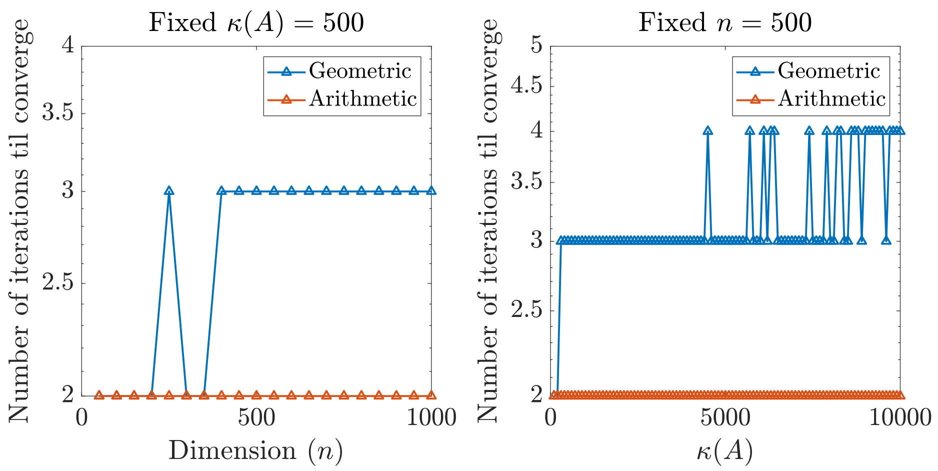

FIG. 5.3: *Behavior of the number of iterations with the dimension and the condition number of the input matrix vary. Matrices in the left and right figures fix its condition number and dimension respectively. The blue and red triangular dots represent the number of iterations for each generated matrices with geometrically and arithmetically distributed singular values respectively.*

cyclic-by-row Jacobi algorithm to converge since the eigenvalue are relatively far apart and this can result in quadratic convergence as discussed in Section 5.2.2. However, for the matrices with geometrically distributed singular values, the algorithm can take four iterations to converge. By construction, the geometrically distributed singular values has smaller δ in (5.3) and this can result in slower convergence since it will require more sweep for $\mathsf{off}(A)$ to satisfies $\mathsf{off}(A) < \delta/8$. We can exaggerate this observation by generating a matrix with very close eigenvalues.

```
close all; clear; clc; rng(1,'twister');
A_geo = my_randsvd(1e3,1e16,'geo');
A_ari = my_randsvd(1e3,1e16,'ari');
```

The matrix `A_geo` has $\min_{\lambda_i \neq \lambda_j} |\lambda_i - \lambda_j| \approx 10^{-18}$, whereas the matrix `A_ari` has $\min_{\lambda_i \neq \lambda_j} |\lambda_i - \lambda_j| \approx 10^{-3}$. Clearly, the Algorithm 9 can hardly attained the quadratic rate of convergence on `A_geo`.

After applying Algorithm 9 on both `A_geo` and `A_ari`, the former testing requires 25 sweeps to converge and the latter one only need three sweeps. Therefore, the Jacobi algorithm performs better when the distance between distinct eigenvalues are large since the algorithm can take advantage of quadratic convergence.

The final testing is to see how much time the mixed precision algorithm can save, we can use the methodology in Section 4.4.2 to produce Figure 5.4. For large n, the benefit of using preconditioner is significant for both distributions of the singular values.

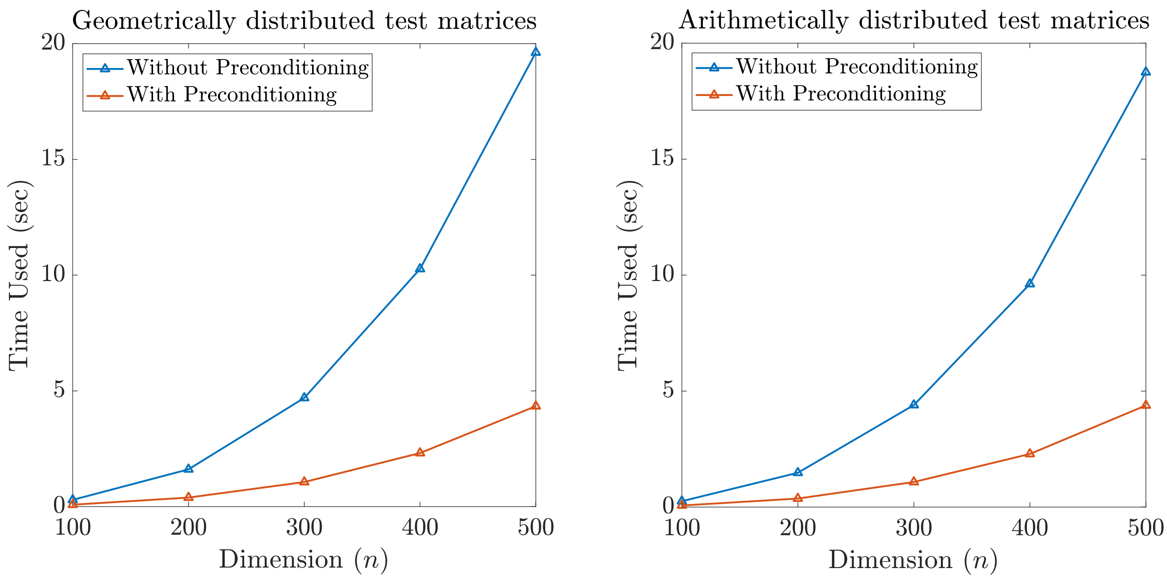

FIG. 5.4: *Time used by the cyclic-by-row Jacobi algorithm with and without preconditioning on a real symmetric matrix A with respect to the dimension n. Here we fixed the condition number $\kappa(A) = 100$. The left and right figures show the testings on the real symmetric matrices with geometrically and arithmetically distributed singular values respectively. The code to regenerate this figure can be found in Appendix B.13.*

Also, besides using the figure to visualize the difference in time, we can calculate how much the algorithm is improved for each n via

$$\text{Improvement} = \frac{t_{\text{normal}} - t_{\text{precondition}}}{t_{\text{normal}}},$$

where t_{normal} and $t_{\text{precondition}}$ are the time used by the cyclic-by-row Jacobi algorithm without and with preconditioning.

```
>> format short
>> (t_normal_g - t_precond_g)./t_normal_g % geometrically distributed
   SVs
ans =
   0.7150    0.7574    0.7739    0.7747    0.7789
>> (t_normal_a - t_precond_a)./t_normal_a % Arithmetically
   distributed SVs
ans =
   0.7294    0.7513    0.7543    0.7619    0.7663
```

The entries are corresponding to $n = 100, 200, \ldots, 500$. Based on the data, we can conclude that by using the preconditioning technique stated in Algorithm 9, we saved roughly 75% of time.

Chapter 6

Conclusion and Further Work

We studied the classical Jacobi algorithm and the cyclic-by-row Jacobi algorithm and concluded that the cyclic-by-row Jacobi algorithm is much more efficient than the other one. Then, after performing implementation and testing on the QR factorization and the Newton–Schulz iteration, in order to orthogonalize an almost orthogonal matrix, we shall always use the Newton–Schulz iteration. Finally, we followed the preconditioning technique in [5, 2000] and successfully propose the mixed precision algorithm 9 for the real symmetric eigenproblem. By utilizing the `eig` function at single precision and preconditioning, we can save roughly 75% of the time compared to applying the Jacobi algorithm alone.

Further works of this thesis include:

1. Rounding error analysis hasn't been conducted.

2. There exist more ways to orthogonalize an almost orthogonal matrix. For example, the Cholesky QR factorization.

3. Only one low precision level has been studied. These theories can apply to half precision as well. However, it requires more work on exploring a half precision version of `eig` function in order to become faster than our Algorithm 9 which uses MATLAB built-in single precision.

4. There are other ways that can be used to speed up the Jacobi algorithm. For example, the threshold Jacobi algorithm [24, 1965, Section 5.12] and the block Jacobi procedure [9, 2013, Section 8.5.6].

Bibliography

[1] Adi Ben-Israel. An iterative method for computing the generalized inverse of an arbitrary matrix. *Mathematics of Computation*, 19(91):452–455, 1965. (Cited on page 48.)

[2] Rajendra Bhatia. *Matrix Analysis*. Springer-Verlag, New York, 1997. xi+347 pp. ISBN 978-1-4612-6857-4. (Cited on page 48.)

[3] Philip I. Davies, Nicholas J. Higham, and Françoise Tisseur. Analysis of the Cholesky method with iterative refinement for solving the symmetric definite generalized eigenproblem. *SIAM Journal on Matrix Analysis and Applications*, 23(2):472–493, 2001. (Cited on pages 30 and 33.)

[4] James Demmel and Krešimir Veselić. Jacobi's method is more accurate than QR. *SIAM Journal on Matrix Analysis and Applications*, 13(4):1204–1245, 1992. (Cited on pages 30 and 33.)

[5] Zlatko Drmač and Krešimir Veselić. Approximate eigenvectors as preconditioner. *Linear Algebra and its Applications*, 309(1):191–215, 2000. (Cited on pages 10, 30, 33, 36, 57, 59, and 64.)

[6] Ky Fan and A. J. Hoffman. Some metric inequalities in the space of matrices. *Proceedings of the American Mathematical Society*, 6(1):111–116, 1955. (Cited on page 43.)

[7] George E. Forsythe and Peter Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Transactions of the American Mathematical Society*, 94(1):1–23, 1960. (Cited on page 32.)

[8] Gene H. Golub and Henk A. van der Vorst. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1):209–239, 2001. (Cited on page 10.)

[9] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins studies in the mathematical sciences. 4th edition, Johns Hopkins University Press, Baltimore, MD, USA, 2013. ISBN 978-1-4214-0794-4. (Cited on pages 21, 22, 30, 33, 36, 45, and 64.)

[10] Robert T. Gregory. Computing eigenvalues and eigenvectors of a symmetric matrix on the ILLIAC. *Mathematics of Computation*, 7(44):215–220, 1953. (Cited on page 32.)

[11] Vjeran Hari. On sharp quadratic convergence bounds for the serial Jacobi methods. *Numerische Mathematik*, 60(1):375–406, 1991. (Cited on page 61.)

[12] Peter Henrici. *Elements of Numerical Analysis*. Wiley, New York, USA, 1964. ISBN 978-0471372417. (Cited on page 46.)

[13] Nicholas J. Higham. Computing the polar decomposition—with applications. *SIAM Journal on Scientific and Statistical Computing*, 7(4):1160–1174, 1986. (Cited on page 45.)

[14] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, January 2002. xxx+680 pp. ISBN 0-89871-521-0. (Cited on pages 12 and 21.)

[15] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. xx+425 pp. ISBN 978-0-898716-46-7. (Cited on pages 42, 43, 45, 46, 47, and 49.)

[16] Carl G. J. Jacobi. Über ein leichtes verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Journal für die reine und angewandte Mathematik*, 1846(30):51–94, 1846. (Cited on pages 24 and 29.)

[17] MATLAB. version 9.12.0.2009381 (r2022a), 2022. (Cited on page 13.)

[18] Leon Mirsky. Symmetric gauge functions and unitarily invariant norms. *The Quarterly Journal of Mathematics*, 11(1):50–59, 1960. (Cited on pages 43 and 44.)

[19] ———. IEEE standard for floating–point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754–2008)*, pages 1–84, 2019. (Cited on page 12.)

[20] Arnold Schönhage. Zur konvergenz des Jacobi-verfahrens. *Numerische Mathematik*, 3(1):374–380, 1961. (Cited on pages 27 and 60.)

[21] Günther Schulz. Iterative Berechung der reziproken Matrix. *Zeitschrift für Angewandte Mathematik und Mechanik*, 13(1):57–59, 1933. (Cited on page 48.)

[22] H. P. M. van Kempen. On the quadratic convergence of the special cyclic Jacobi method. *Numerische Mathematik*, 9(1):19–22, 1966. (Cited on page 61.)

[23] James H. Wilkinson. Note on the quadratic convergence of the cyclic Jacobi process. *Numerische Mathematik*, 4(1):296–300, 1962. (Cited on pages 30 and 61.)

[24] James H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, UK, 1965. xviii+662 pp. ISBN 978-0198534181. (Cited on page 64.)

Appendix A

Supplementary Explanation

A.1 Algorithm 2

We have two different expressions for computing t as shown in Table A.1.

Table A.1: *Two different approaches for computing t in Algorithm 2.*

	Approach 1	Approach 2
$\tau \geq 0$	$t = -\tau + \sqrt{\tau^2 + 1}$	$t = \dfrac{1}{\tau + \sqrt{1 + \tau^2}}$
$\tau < 0$	$t = -\tau - \sqrt{\tau^2 + 1}$	$t = \dfrac{1}{\tau - \sqrt{1 + \tau^2}}$

These two expressions are the same by simple calculation,

$$t = \frac{1}{\tau + \sqrt{1 + \tau^2}} = \frac{\tau - \sqrt{1 + \tau^2}}{\left(\tau + \sqrt{1 + \tau^2}\right)\left(\tau - \sqrt{1 + \tau^2}\right)} = -\tau + \sqrt{1 + \tau^2}. \tag{A.1}$$

However, when we implement these into MATLAB, approach 1 will output $c = 1, s = 0$ for $a_{pq} \approx 10^{-8}$, and approach 2's output gives $s \neq 0$ even for $a_{pq} \approx 10^{-14}$. Namely, by using approach 1, the computed output is less accurate than the output from approach 2.

Appendix B

Supplementary Code

B.1 Testing Matrices

By using the following code, we can generate a random real symmetric matrix with predefined condition number.

```
function A = my_randsvd(n, kappa, mode)
classname = 'double';
if isempty(mode)
  mode = 'Geo';
end
switch mode % Set up vector sigma of singular values.
  case {'Geometric','geometric','Geo','geo'}
    % geometrically distributed singular values
    factor = kappa^(-1/(n-1));
    sigma = factor.^(0:n-1);
  case {'Arithmatic', 'arithmatic', 'Ari', 'ari'}
    % arithmetically distributed singular values
    sigma = ones(n,1) - (0:n-1)'/(n-1)*(1-1/kappa);
  otherwise
    error(message('MATLAB:randsvd:invalidMode'));
end
sigma = cast(sigma,classname); signs = sign(rand(1,n-2));
sigma(2:n-1) = sigma(2:n-1).* signs;
Q = qmult(n,1,classname);
A = Q*diag(sigma)*Q';
A = (A + A')/2;
```

B.2 Full Jacobi Algorithm Code

```
function [V,D,counter] = jacobi_eig(Aini,tol,type,maxiter)
%JACOBI_EIG   Jacobi Eigenvalue Algorithm
%   [A,V,counter] = JACOBI_EIG(Aini,tol,type,maxiter) computes the
  eigen-
%   decomposition of Aini, Aini = VDV', where V is orthogonal.
```

```
%Input:
%   Aini -------- Input matrix
%   tol --------- Desired tolerance, usually 1.1e-16 (double
   precision)
%   type -------- Type of Jacobi algorithm, either
%                     * 'classical': For classical Jacobi algorithm
%                     * 'cyclic': For cyclic-by-row Jacobi algorithm
%   maxiter ----- Maximum number of sweep, specially for cyclic-by-
   row
%                Jacobi algorithm. If undetermined, the maximum sweep
   will
%                be set as 10 sweeps.
%Output:
%   V ------------ Orthogonal matrix
%   D ------------ Resulting matrix after apply Jacobi algorithm.
%   counter ------ Number of iteration
if nargin <= 2, error('Not enough input arguments.');
elseif nargin >= 5, error('Too many input arguments.');
else
  % select method, 'classical' or 'cyclic'
  switch type
    case {'classical','Classical'}
      method = 1;
    case {'cyclic','Cyclic'}
      method = 2;
    otherwise
      error("The Jacobi algorithm avaliable are " + ...
        "'classical' and 'cyclic'")
  end
  switch method
    case 1
      [V,D,counter] = jacobi_classical(Aini,tol);
    case 2
      switch nargin
        case 4
          maxiteration = maxiter;
          [V,D,counter] = jacobi_cyclic(Aini,tol,maxiteration);
        case 3
          maxiteration = 10;
          [V,D,counter] = jacobi_cyclic(Aini,tol,maxiteration);
        otherwise
          error('Not enough input arguments.');
      end
  end
end
end

% classical Jacobi algorithm
function [V,A,counter] = jacobi_classical(A,tol)
counter = 0; n = length(A); V = eye(n); done_rot = true;
tol1 = tol * norm(A);
while done_rot
  if isint(counter/(n*n)), A = (A + A')/2; end
  done_rot = false; [p,q] = maxoff(A);
  if abs(A(p,q)) >  tol1 * sqrt(abs(A(p,p) * A(q,q)))
    counter = counter + 1; done_rot = true;
    [c,s] = jacobi_pair(A,p,q);
    J = [c,s;-s,c];
    A([p,q],:) = J'*A([p,q],:);
    A(:,[p,q]) = A(:,[p,q]) * J;
    V(:,[p,q]) = V(:,[p,q]) * J;
  else
    A = diag(diag(A));
    break;
  end
end
end

% Cyclic-by-row Jacobi algorithm
function [V,A,iter] = jacobi_cyclic(A,tol,maxiter)
n = length(A); V = eye(n); iter = 0; done_rot = true;
while done_rot && iter < maxiter
  done_rot = false;
  for p = 1:n-1
    for q = p+1:n
      if abs(A(p,q)) > tol * sqrt(abs(A(p,p)*A(q,q)))
        done_rot = true;
        [c,s] = jacobi_pair(A,p,q);
        J = [c,s;-s,c];
        A([p,q],:) = J'*A([p,q],:);
        A(:,[p,q]) = A(:,[p,q]) * J;
        V(:,[p,q]) = V(:,[p,q]) * J;
      end
    end
  end
  if done_rot
    A = (A + A')/2; iter = iter + 1;
  else
    A = diag(diag(A));
    return;
  end
%   fprintf('off(A) = %d, sweep = %d\n', off(A), iter);
end
end

% compute the Frobenius norm of the off-diagonal entries of the input
    matrix
function num = off(A)
n = length(A); A(1:n+1:n*n) = 0;
num = norm(A,'fro');
end

% find the index of maximum off-diagonal entry of the input matrix A
function [p,q] = maxoff(A)
n = length(A); % dimension of matrix A
A(1:n+1:n*n) = 0; A = abs(A); % clear the diagonal entries
[val, idx1] = max(A);
[~, q] = max(val); p = idx1(q);
end

% calculate the Givens rotation matrix's entries (c,s).
function [c,s] = jacobi_pair(A,p,q)
if A(p,q) == 0
  c = 1; s = 0;
else
  tau = (A(q,q)-A(p,p))/(2*A(p,q));
  if tau >= 0
    t = 1/(tau + sqrt(1+tau*tau));
  else
    t = 1/(tau - sqrt(1+tau*tau));
  end
  c = 1/sqrt(1+t*t);
  s = t*c;
end
end
```

B.3 Code for Figure 3.2

Figure 3.2 can be regenerated using the following routine

```
clc; clear; close all; format short e;
condi1 = 100; tol = 2^(-53);
rng(1,'twister');
for n = 1e2:1e2:1e3
  A = my_randsvd(n,condi1,'geo');
  fprintf('iteration of n %d/%d\n', n/1e2, 1e3/1e2);
  tic
  [V1,D1,iter1] = jacobi_eig(A,tol,'Classical');
  t_classical_n(n/1e2) = toc;
  tic
  [V2,D2,iter2] = jacobi_eig(A,tol,'Cyclic',20);
  t_cyclic_n(n/1e2) = toc;
end
nplot = 1e2:1e2:1e3;
n = 2.5e2;
rng(1,'twister');
for condi = 1e3:1e3:1e5
  A = my_randsvd(n,condi,'geo');
  fprintf('iteration of cond %d/%d...',condi/1e3,1e5/1e3);
  tic
  [V1,D1,iter1] = jacobi_eig(A,tol,'Classical');
  t_classical_condi(condi/1e3) = toc;
  tic
  [V2,D2,iter2] = jacobi_eig(A,tol,'Cyclic',20);
  t_cyclic_condi(condi/1e3) = toc;
  fprintf('complete \n');
end
condiplot = 1e3:1e3:1e5;
close all;
subplot(1,2,1); hold off;
plot(nplot,t_classical_n,'-^','LineWidth',2);
hold on;
plot(nplot,t_cyclic_n,'-^','LineWidth',2);
xlabel('Dimension $(n)$');
ylabel('Time Used (sec)');
title('Fixed $\kappa(A) = 100$');
legend('Classical Jacobi','Cyclic Jacobi','Location','northwest','
    FontSize',20);
axis square
subplot(1,2,2); hold off;
```

```
plot(condiplot,t_classical_condi,'-^','LineWidth',1.5);
hold on;
plot(condiplot,t_cyclic_condi,'-^','LineWidth',1.5);
xlabel('$\kappa(A)$');
ylabel('Time Used (sec)');
title('Fixed $n = 250$');
legend('Classical Jacobi','Cyclic Jacobi','Location','northwest','
   FontSize',20);
axis square; axis([0,1e5,0,25])
```

B.4 Code for Figure 4.1

Figure 4.1 can be regenerated using the following routine

```
clc; clear; close all; format short e;
ud = 2^(-53); n = 1e2:1e2:3e3; condi = 100; rng(1,'twister');
for i = 1:length(n)
  fprintf('iteration %d/%d\n', i, length(n));
  A = gallery('randsvd', n(i), -condi, 3);
  [Q,~] = myqr(A);
  orthog(i) = norm(Q'*Q - eye(n(i)));
end
semilogy(n,orthog,'-^r',LineWidth=1.5);
hold on; semilogy(n, n.*ud, '-^k',LineWidth=1.5);
legend("$\|Q^TQ - I_n\|$", '$nu_d$','Location','northwest');
xlabel('Dimension $(n)$')
```

B.5 Code for Figure 4.3

Figure 4.3 can be regenerated using the following routine

```
clc; clear; close all;
n = 10:10:8e4;
sq = (sqrt(1+n .* eps(single(1/2))) - 1).^4;
sq_ref = n .* eps(1/2);
plot(n,sq); hold on;
plot(n,sq_ref,'--');
plot(52400,5.81757e-12,'^k')
legend('$\|X_2 - U\|_2$','$n\cdot u_d$','Location','northwest');
xlabel('Dimension $(n)$')
```

B.6 Code for Figure 4.4

Figure 4.4 can be regenerated using the following routine

```
clc; clear; close all; format short e; rng(1,'twister');
ud = 2^(-53);
n = 1e2:1e2:3e3; condi = 1e2;
```

```
for i = 1:length(n)
  fprintf('Start iteration %d/%d,\n',i,length(n));
  A = my_randsvd( n(i), condi, 'geo');
  [Q,~] = eig(single(A)); Q = double(Q);
  Q1 = newton_schulz(Q);
  orthog(i) = norm(Q1'*Q1 - eye(n(i)));
end
semilogy(n,orthog,'-^r',LineWidth=1.5);
hold on;
semilogy(n, n.*ud, '-^k',LineWidth=1.5);
legend("$\|Q_d^*Q_d - I_n\|$",'$nu_d$','Location','northwest');
xlabel('Dimension (n)'); title('Fix $\kappa(A) = 100$'); axis square
```

B.7 Code for Figure 4.5

Figure 4.5 can be regenerated using the following routine

```
clc; clear; close all; format short e; rng(1,'twister');
ud = 2^(-53); n = 1e2:1e2:3e3; condi = 1e2;
for i = 1:length(n)
  fprintf('iteration %d/%d\n',i,length(n));
  A = my_randsvd(n(i),condi,'geo');
  [Q,~] = eig(single(A)); Q = double(Q);
  Q1 = newton_schulz(Q);
  [Q2,~] = qr(Q);
  [Q3,~] = myqr(Q);
  orthogns(i) = norm(Q1'*Q1 - eye(n(i)));
  orthogqr(i) = norm(Q2'*Q2 - eye(n(i)));
  orthogmyqr(i) = norm(Q3'*Q3 - eye(n(i)));
end
semilogy(n,orthogns,'-^',LineWidth=1.5); hold on;
semilogy(n,orthogqr,'-^',LineWidth=1.5);
semilogy(n,orthogmyqr,'-^',LineWidth=1.5);
legend("Newton-Schulz", ...
  "\texttt{qr()} from MATLAB", ...
  "My own QR code",...
  'interpreter','latex', ...
  'Location','northwest', ...
  'FontSize',20);
xlabel('Dimension $(n)$');
ylabel('$\|Q_d^*Q_d - I_n\|$')
axis([0,3000,0,2e-14])
axis square
```

B.8 Code for Figure 4.6

Figure 4.6 can be regenerated using the following routine

```
clc; clear; close all; format short e;
ud = 2^(-53); n = 1e2:1e2:3e3; condi = 100;
time_ns = zeros(length(n),1); time_qr = time_ns;
```

```
rng(1,'twister'); % RNG
for i = 1:length(n)
  fprintf('iteration %d/%d\n',i,length(n));
  A = my_randsvd(n(i),condi,'geo');
  [Q,~] = eig(single(A)); Q = double(Q);
  tic; [Q1] = newton_schulz(Q); time_ns(i) = toc;
  tic; [Q2,~] = myqr(Q); time_qr(i) = toc;
end
semilogy(n,time_ns,'-^',LineWidth=1.5); hold on;
semilogy(n,time_qr,'-^',LineWidth=1.5);
legend('Newton--Schulz','QR Factorization','Location','northwest','
   interpreter','latex','FontSize',20);
xlabel('Dimension $(n)$')
ylabel('Time (sec)')
axis square
```

B.9 Mixed Precision Jacobi Algorithm Code

```
function [Q,D,iter] = jacobi_precondi(A,tol,maxiter)
%JACOBI_PRECONDI(A,tol,maxiter) compute the eigendecomposition of A
   using cyclic-by-row Jacobi algorithm with preconditioning.
%% Preliminaries
switch nargin
  case 2, max_it = 10; case 3, max_it = maxiter;
  otherwise, error('Please check the number of inputs');
end
[n,m] = size(A);
if m ~= n, error('Input matrix should be square matrix'), end
if issymmetric(A) == 0, error('Input matrix should be symmetric');
   end
%% Eigendecomposition at single precision
[Q_low,~] = eig(single(A)); Q_low = double(Q_low);
%% Newton--Schulz iteration to orthogonalize
Q_d = newton_schulz(Q_low);
%% Cyclic-by-row Jacobi algorithm on preconditioned A
A_cond = Q_d' * A * Q_d;
[V,D,iter] = jacobi_eig(A_cond,tol,'Cyclic',max_it);
%% Output
Q = Q_d * V;
```

B.10 Code for Figure 5.1

Figure 5.1 can be regenerated using the following routine

```
clc; clear; format short e;
n = 50:50:1e3; condi = 500; nplot = n;
rng(1,'twister');
for i = 1:length(n)
  fprintf('iteration %d/%d...',i,length(n))
  A_geo = my_randsvd(n(i),condi,'geo');
  A_ari = my_randsvd(n(i),condi,'ari');
  [V1,D1,iter1] = jacobi_precondi(A_geo,eps(1/2));
  [V2,D2,iter2] = jacobi_precondi(A_ari,eps(1/2));
  accuracy_n_geo(i) = norm(A_geo * V1 - V1 * D1);
  accuracy_n_ari(i) = norm(A_ari * V2 - V2 * D2);
  reference_n(i) = n(i) * norm(A_geo) * eps(1/2);
  fprintf('complete\n');
end
%%
subplot(1,2,1); hold off;
semilogy(nplot,accuracy_n_geo,'-^'); hold on;
semilogy(nplot,accuracy_n_ari,'-^');
semilogy(nplot,nplot.*eps(1/2),'--k');
xlabel('Dimension $(n)$');
ylabel('$\|AQ - Q\Lambda\|_2/\|A\|_2$');
legend('Geometric','Arithmetic','$nu_d\|A\|_2$','Location','southeast
    ','FontSize',20);
title('Fixed $\kappa(A) = 500$')
axis square
%%
condiplot = 1e2:1e2:1e4; n = 5e2;
for i = 1:length(condiplot)
  fprintf('iteration %d/%d...',i,length(condiplot))
  A_geo = my_randsvd(n,condiplot(i),'geo');
  A_ari = my_randsvd(n,condiplot(i),'ari');
  [V1,D1,iter1] = jacobi_precondi(A_geo,eps(1/2));
  [V2,D2,iter2] = jacobi_precondi(A_ari,eps(1/2));
  accuracy_condi_geo(i) = norm(A_geo * V1 - V1 * D1);
  accuracy_condi_ari(i) = norm(A_ari * V2 - V2 * D2);
  reference_n(i) = n * norm(A_geo) * eps(1/2);
  fprintf('complete\n');
end
%%
close all;
subplot(1,2,1); hold off;
semilogy(nplot,accuracy_n_geo,'-^'); hold on;
semilogy(nplot,accuracy_n_ari,'-^');
semilogy(nplot,nplot.*eps(1/2),'--k');
xlabel('Dimension $(n)$');
ylabel('$\|AQ - Q\Lambda\|_2/\|A\|_2$');
legend('Geometric','Arithmetic','$nu_d$','Location','southeast','
    FontSize',20);
title('Fixed $\kappa(A) = 500$')
axis square
subplot(1,2,2); hold off;
semilogy(condiplot,accuracy_condi_geo,'-^'); hold on;
semilogy(condiplot,accuracy_condi_ari,'-^');
semilogy(condiplot,reference_n,'--k');
xlabel('$\kappa(A)$');
ylabel('$\|AQ - Q\Lambda\|_2/\|A\|_2$');
legend('Geometric','Arithmetic','$nu_d$','Location','northeast','
    FontSize',20);
title('Fixed $n = 500$')
axis square; axis([0,1e4,0,8e-14]);
```

B.11 Code for Figure 5.2

```matlab
clc; clear all; close all; figure(1);
n = [50,250,500];
for i = 1:3
  subplot(2,2,i)
  testing_1(n(i));
  if i == 1
    axis([0,1e4,0,6e-6]);
    title('Fixed $n=50$');
  elseif i == 2
    axis([0,1e4,0,3.8e-5]);
    title('Fixed $n = 250$');
  elseif i == 3
    axis([0,1e4,0,8e-5]);
    title('Fixed $n = 500$');
  end
  pause(0.1);
end
subplot(2,2,4); hold off
condi = 1e2; n1 = 10:10:1e3; offA = zeros(1,length(n1)); rng(1,'
   twister');
for i = 1:length(n1)
  fprintf('start %d/%d\n',i,length(n1));
  A_geo = my_randsvd(n1(i),condi,'geo');
  A_ari = my_randsvd(n1(i),condi,'ari');
  [Q1,~] = eig(single(A_geo)); Q1 = double(Q1);
  Q1_d = newton_schulz(Q1);
  offA_geo(i) = off(Q1_d' * A_geo * Q1_d)/norm(A_geo);
  [Q2,~] = eig(single(A_ari)); Q2 = double(Q2);
  Q2_d = newton_schulz(Q2);
  offA_ari(i) = off(Q2_d' * A_ari * Q2_d)/norm(A_ari);
end
semilogy(n1,offA_geo); hold on;
semilogy(n1,offA_ari);
semilogy(n1,n1 .* double(eps(single(1/2))),'--k');
legend('Geometric','Arithmetic','$nu_\ell$','Location','southeast');
xlabel('$\kappa(A)$'); ylabel('off$(A_{cond})/\|A\|_2$','Interpreter'
   ,'latex');
set(gca,'FontSize',20);
axis square;
title('Fixed $\kappa(A) = 100$')
function testing_1(n)
condi = 10:10:1e4; offA_geo = zeros(1,length(condi)); offA_ari =
   offA_geo;
rng(1,'twister');
for i = 1:length(condi)
  fprintf('start %d/%d\n',i,length(condi));
  A_geo = my_randsvd(n,condi(i),'geo');
  [Q1,~] = eig(single(A_geo)); Q1 = double(Q1);
  Q1_d = newton_schulz(Q1);
  offA_geo(i) = off(Q1_d' * A_geo * Q1_d)/norm(A_geo);
  A_ari = my_randsvd(n,condi(i),'ari');
  [Q2,~] = eig(single(A_ari)); Q2 = double(Q2);
  Q2_d = newton_schulz(Q2);
  offA_ari(i) = off(Q2_d' * A_ari * Q2_d)/norm(A_ari);
end
```

```
semilogy(condi,offA_geo,'LineWidth',1.5); hold on;
semilogy(condi,offA_ari);
semilogy(condi,n * double(eps(single(1/2))) * ones(1,length(condi)),'
    --k','LineWidth',1.5);
legend('Geometric','Arithmetic','$nu_\ell$');
xlabel('$\kappa(A)$'); ylabel('off$(A_{cond})/\|A\|_2$','Interpreter'
    ,'latex');
set(gca,'FontSize',20);
axis square;
end
```

B.12 Code for Figure 5.3

Figure 5.3 can be regenerated using the following routine

```
clear; clc; rng(1,'twister');
n = 50:50:1e3; condi = 500; nplot = n;
for i = 1:length(n)
  A_geo = my_randsvd(n(i),condi,'geo');
  [V1,D1,iter1] = jacobi_precondi(A_geo,eps(1/2));
  iteration_n_geo(i) = iter1;
  A_ari = my_randsvd(n(i),condi,'ari');
  [V2,D2,iter2] = jacobi_precondi(A_ari,eps(1/2));
  iteration_n_ari(i) = iter2;
  fprintf('iteration %d/%d complete\n',i,length(n));
end
condiplot = 1e2:1e2:1e4; n = 5e2;
for i = 1:length(condiplot)
  A_geo = my_randsvd(n,condiplot(i),'geo');
  [V1,D1,iter1] = jacobi_precondi(A_geo,eps(1/2));
  iteration_condi_geo(i) = iter1;
  A_ari = my_randsvd(n,condiplot(i),'ari');
  [V2,D2,iter2] = jacobi_precondi(A_ari,eps(1/2));
  iteration_condi_ari(i) = iter2;
  fprintf('iteration %d/%d complete\n',i,length(condiplot));
end
close all
subplot(1,2,1);
semilogy(nplot,iteration_n_geo,'-^'); hold on;
semilogy(nplot,iteration_n_ari,'-^');
xlabel('Dimension $(n)$');
ylabel('Number of iterations til converge');
title('Fixed $\kappa(A) = 500$')
legend('Geometric','Arithmetic','Location','northeast');
axis square; axis([0,1000,0,4]);
subplot(1,2,2);
semilogy(condiplot,iteration_condi_geo,'-^'); hold on;
semilogy(condiplot,iteration_condi_ari,'-^');
legend('Geometric','Arithmetic','Location','northeast');
xlabel('$\kappa(A)$');
ylabel('Number of iterations til converge');
title('Fixed $n = 500$')
axis square; axis([0,1e4,0,5]);
```

B.13 Code for Figure 5.4

Figure 5.4 can be regenerated using the following routine

```
clc; clear; close all; rng(1,'twister');
n = 1e2:1e2:5e2; condi = 500; tol = 2^(-53);
for i = 1:length(n)
  Ag = my_randsvd(n(i),condi,'geo');
  Aa = my_randsvd(n(i),condi,'ari');
  tic; [V1,D1,iter1] = jacobi_precondi(Ag,tol);
  t_precond_g(i) = toc;
  tic; [V2,D2,iter2] = jacobi_eig(Ag,tol,'Cyclic');
  t_normal_g(i) = toc;
  tic; [Va1,Da1,iter1] = jacobi_precondi(Aa,tol);
  t_precond_a(i) = toc;
  tic; [Va2,Da2,iter2] = jacobi_eig(Aa,tol,'Cyclic');
  t_normal_a(i) = toc;
  fprintf('Iteration %d/%d complete\n',i,length(n));
end
subplot(1,2,1)
plot(n,t_normal_g,'-^');
hold on;
plot(n,t_precond_g,'-^');
xlabel('Dimension $(n)$')
ylabel('Time Used (sec)')
legend('Without Preconditioning', 'With Preconditioning','Location','
    northwest');
title('Geometrically distributed test matrices')
axis square
subplot(1,2,2)
plot(n,t_normal_a,'-^');
hold on;
plot(n,t_precond_a,'-^');
xlabel('Dimension $(n)$')
ylabel('Time Used (sec)')
legend('Without Preconditioning', 'With Preconditioning','Location','
    northwest');
title('Arithmetically distributed test matrices')
axis square
```