上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目：Rank Revealing Algorithms and its Applications

学生姓名: 包昱嘉

学生学号: 5120719015

专　业: 数学与应用数学

指导教师: 王增琦

学院(系): 数学科学学院

Rank-reveal 类算法及其应用

摘要

随着大数据时代的来临，rank revealing QR(RRQR)分解在子集选择法(subset selection)、最小二乘(least squares problem)、完全最小二乘(total least squares problems)等秩缺失问题上有着越来越多的应用。本文对 RRQR 算法进行了系统性的研究，主要工作概括如下：

1. 系统地介绍了目前高性能、接受度广的三类 RRQR 算法，并拓展了其中一些理论性结果，独立补充了一些结果的理论性证明；
2. 基于现有方法，本文提出了新型的用于计算强 RRQR 分解的贪婪型强 RRQR 算法，提升了原算法的计算效率；
3. 设计了一系列数值算例，体现了各类算法的实际运算特点。

理论和实验结果表明，原有的强 RRQR 算法及新提出的贪婪型强 RRQR 算法都能给出一个强 RRQR 分解，而贪婪型强 RRQR 算法在计算高效性方面明显优于原算法。在求解秩缺失型问题时，RRQR 分解与传统的 SVD 分解相比，可以达到令人满意的计算精度，但时间明显占优。

关键词：rank-revealing QR 分解，秩缺失问题

RANK REVEALING ALGORITHMS AND ITS APPLICATIONS

ABSTRACT

As the era of big data is coming, rank revealing QR (RRQR) factorization has more and more applications on rank deficient problems such as subset selection, least squares problem, total least squares problem. This thesis systematically studied the RRQR algorithms. The main contributions are summarized as following:

1. This thesis presents a systematic review of three kinds of widely-used and high-performance RRQR algorithms. I extend some existing theorems and independently complete some parts of the theoretical analysis.
2. Based on the existing methods, I propose a new greedy strong RRQR algorithm for computing a strong RRQR factorization. The new algorithm greatly improves the time efficiency of the origin algorithm.
3. I design a series of numerical experiments to show the computation characteristics of different kinds of algorithms.

Theoretical analysis and numerical results show that both the origin strong RRQR algorithm and the new greedy strong RRQR algorithm can promise a strong RRQR factorization while the new algorithm is significantly faster than the origin algorithm. For rank deficient problem, RRQR factorization gives satisfactory computation accuracy while it is much more efficient than the traditional method, which involves computing the SVD.

Keywords: rank-revealing QR factorization, rank deficient problems

Contents

Chapter 1 Introduction

1.1 Overview of this Thesis

This thesis intends to be a self-contained survey for QR and RRQR factorization.

In Chapter 1, I presents the background of matrix factorization and rank revealing algorithms. SVD is the most reliable algorithm for dealing with rank deficient problems. However, it is expensive in computation. Rank revealing algorithms offer a cheaper alternative for SVD and they often work well in practice.

In Chapter 2, I discuss two important orthogonal transformations, Householder reflections and Givens rotations. These two orthogonal transformations form the basis of QR factorization. Then in Section 2.3, I compare these two orthogonal transformations and show that Householder reflection is good at zeroing out a large amount of elements simultaneously from the given vector while Givens rotation is good at zeroing out elements more selectively. This property enables fast implementation of the following RRQR algorithms.

In Chapter 3, I talk about greedy algorithms for computing an RRQR factorization. At each step, the greedy strategy tries to find the best column from the right portion and then adds it into the left portion. Before discussing the algorithms, I first formulate the problem of finding an RRQR factorization as three optimization problems in Section 3.1. Then, in Section 3.2, I present 7 greedy RRQR algorithms for solving Problem-I in a hierarchical order. I also prove the theoretical bounds that these algorithms will guarantee and show that most greedy algorithms fail to find a good permutation for Kahan matrix. After that, in Section 3.3, I focus on Problem-II. By the unification principle, all algorithms in Section 3.2 can be converted into a corresponding version for solving Problem-II. Those theoretical bounds and pessimistic examples for Problem-I can also be carried out to algorithms for Problem-II by the unification principle.

One major deficiency of these greedy algorithms is that they don't consider permuting columns which have already been selected at previous time. Based on these greedy

algorithms, Chapter 4 discusses hybrid algorithms for computing an RRQR factorization. Suppose we are given an $m \times n$ matrix $\boldsymbol{M}$ and a parameter k. Hybrid algorithm alternates between two greedy algorithms, one for Problem-I and one for Problem-II. The Problem-I algorithm permutes column k with column j, $n > j > k$, so that the new column k is the 'best' among column $k, ..., n$ of the matrix. The Problem-II algorithm permutes column k with column i, $1 < i < k$, so that the new column k is the 'worst' among column $1, ..., k$ of the matrix. Though there is no analysis on the total number of iterations that these hybrid algorithms will perform, hybrid algorithms run very fast in practice and always give a better result than the previous greedy algorithms.

However, RRQR factorization cannot guarantee a stable approximation for the right null space of the given matrix. Chapter 5 focuses on strong RRQR (SRRQR) factorization which aims to resolve this shortcoming. I first show two SRRQR algorithms for computing a strong RRQR factorization. Then I combine these two algorithms with the greedy strategy and develop two greedy strong RRQR (GSRRQR) algorithms. These GSRRQR algorithms run much faster than the origin SRRQR algorithms while still guarantee a strong RRQR factorization. After that, in subsection 5.2.4, I present a general method to reveal the numerical rank k. In the end of this chapter, I show three important techniques that enable fast implementation of hybrid, SRRQR and GSRRQR algorithms in Section 5.3 .

Chapter 6 focuses on the application of RRQR factorization and shows their numerical performances. In Section 6.1, I compare the numerical performance of algorithm Greedy-I.1, Greedy-I.2, Greedy-I.3, QR with column pivoting, Chan, GKS, Foster, Hybrid-I, Hybrid-II, Hybrid-III, SRRQR-1, SRRQR-2, GSRRQR-1, GSRRQR-2 on 4 different kinds of matrices with different sizes. The results show that SRRQR algorithms have the best performance among all these algorithms and GSRRQR algorithms are much faster than the origin SRRQR algorithms. Then in Section 6.2, I show how to apply RRQR factorization in rank deficient least squares problem. Through theoretical analysis and numerical experiments, I show that the SVD-based solution and the RRQR-based solution are similar if the gap between σ_k and σ_{k+1} is very large. In Section 6.3, I present the application in subset selection problem. The numerical

results show that RRQR-based solution is better than the SVD-based solution though RRQR-based solution is much cheaper to compute. In Section 6.4, I show the application in matrix approximation. Theoretical analysis shows that as long as σ_k is small, the RRQR-based approximation error is also small. I further apply RRQR factorization to image compression and see that the difference between RRQR-based approximation and SVD-based approximation is negligible.

Finally, Appendix A shows all MATLAB codes for these RRQR algorithms. Appendix B shows all MATLAB codes for the numerical experiments.

1.2 Main Contributions

The major contribution of thesis is the development of these GSRRQR algorithms. Though there is no analytical bound on how much time we could save by using GSRRQR algorithm instead of the origin SRRQR algorithm, numerical results over matrices of different kinds and different sizes show that the new proposed GSRRQR algorithm is significantly faster than the origin SRRQR algoirthm.

This is the *first* survey that discusses greedy RRQR algorithms, hybrid RRQR algorithms and strong RRQR algorithms both analytically and numerically. Detailed numerical results are presented so that one can easily see the performances of different RRQR algorithms. I also show three applications of RRQR factorization. The theoretical analysis and numerical results reveal that RRQR can give approximately the same solution as SVD for rank deficient problems while it is much cheaper to compute.

In this thesis, I extended some existing theorems and results. I list them as following.

- For the analytical bound of greedy algorithms, Ipsen showed the following bound

$$\frac{\sigma_k(\boldsymbol{M})}{n\|\boldsymbol{W}^{-1}\|_2} \leq \sigma_{\min}(\boldsymbol{R}_{11}) \leq \sigma_k(\boldsymbol{M})$$

  holds for algorithm QR with column pivoting-I, Chan-I and GKS-I. Based on his proof, I further get

$$\frac{\sigma_k(\boldsymbol{M})}{\sqrt{nk}\|\boldsymbol{W}^{-1}\|_2} \leq \sigma_{\min}(\boldsymbol{R}_{11}) \leq \sigma_k(\boldsymbol{M}),$$

for algorithm Chan-I and

$$\frac{\sigma_k(\boldsymbol{M})}{\sqrt{n}\|\boldsymbol{W}^{-1}\|_2} \leq \sigma_{\min}(\boldsymbol{R}_{11}) \leq \sigma_k(\boldsymbol{M}),$$

for algorithm QR with column pivoting-I and GKS-I.

- Chan ([1], 1992) showed a brief proof of Theorem 6.3.2 for algorithm Chan. I extend it to any RRQR algorithms and show a detailed proof.
- Based on the updating formula given by ([2], Gu, 1996), in Section 5.3.1, I relax the condition of the diagonal elements and allow them to be negative. This improves the stability of the algorithm.
- I complete the second part of the proof for Theorem 6.2.1 based on Chan ([1], 1992).

I also independently proved some theorems in this thesis.

- I independently proved Theorem 3.3.1 which enable us to transfer the analytical bound of greedy algorithm for solving Problem-I to the bound of the corresponding algorithm for solving Problem-II.
- I independently proved Theorem 4.2.1 which gives the analytical bound for algorithm Hybrid-II.
- I independently showed Theorem 6.4.1, which gives an analytical bound on the approximation error of RRQR-based matrix approximation.

I fixed some errors in the previous literatures.

- In Section 5.3.3, I correct an error of the modifying formula of $\omega(\bar{\boldsymbol{A}}_k)$ given by Gu ([2], 1996).
- In the proof of Lemma 5.2.1, I correct an error in the formula of $\tilde{\boldsymbol{A}}_k^{-1}$ given by Gu ([2], 1996).
- In the derivation of the lower bound for greedy algorithms at each iteration, I correct an error in Inequality (3-9) given by Ipsen ([3], 1994).

1.3 Backgrounds

1.3.1 Matrix Factorization

Finding the numerical rank of a certain matrix is one of the most important problems. It arises from many scientific and computational problems, such as subset selection, linear least squares, rank determination and matrix approximation. It also has potential applications in image compression and features selection from big data. Matrix factorizations provide efficient methods for finding the numerical rank and have received common concerns recently.

QR factorization ([4], Francis, 1961) decomposes a matrix as $\boldsymbol{M} = \boldsymbol{QR}$, where $\boldsymbol{M}$ is a given $m \times n$ general matrix with $m \geq n$, $\boldsymbol{Q}$ is an $m \times m$ unitary matrix and $\boldsymbol{R}$ is an $m \times n$ upper triangular rectangular matrix. There are several methods for computing the QR factorization, such as the Gram-Schmidt process, Householder reflections and Givens rotations ([5], Golub et al., 2012). Since QR factorization maintains the condition number of the matrix, it provides a reliable way to solve a linear least squares problem. Another application for QR factorization is the eigenvalue problem. The QR algorithm, which performs QR factorization at each iteration, is an iterative method for computing the eigenstructure of a matrix.

LU factorization ([6], Turing, 1948) decomposes a matrix as $\boldsymbol{M} = \boldsymbol{LU}$, where $\boldsymbol{M}$ is a given square $n \times n$ general matrix, $\boldsymbol{L}$ is a lower triangular matrix of order n and $\boldsymbol{U}$ is an upper triangular matrix of order n. For computing the LU factorization, we simply apply Gaussian elimination. The algorithm requires $n^3/3$ floating point operations, while QR factorization requires $2n^3/3$ floating point operations matrix. LU decomposition is also useful in solving linear systems, but it enlarges the condition number of the origin linear system which may lead to stability issue.

Singular value decomposition (SVD) factors a matrix as $\boldsymbol{M} = \boldsymbol{U\Sigma V}^T$, where $\boldsymbol{M}$ is a given $m \times n$ general matrix, $\boldsymbol{U}$ is an $m \times m$ unitary matrix, $\boldsymbol{\Sigma}$ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal and $\boldsymbol{V}$ is an $n \times n$ unitary matrix. SVD is powerful in many problems, such as null space determination, rank determination, computation of the pseudo-inverse and matrix approximation.

Usually, SVD is computed by a two-step procedure. In the first step, we transform the symmetric matrix $\boldsymbol{M}^T\boldsymbol{M}$ to a bidiagonal form by Householder reflections and this will cost $4mn^2 - 4n^3/3$ floating point operations ([7], Trefethen et al., 1997). In the second step, we compute the SVD of the bidiagonal matrix by a variant of the QR algorithm ([8], Golub et al., 1965). Computing the SVD is much more expensive than computing QR factorization and LU factorization.

1.3.2 Rank Revealing Algorithms

The SVD provides the most reliable way to find the numerical rank of a matrix. However, it is too expensive for huge scale problems in many practical applications. A promising alternative to SVD is a modified QR factorization, named as the rank revealing QR (RRQR) factorization. The problem of finding an RRQR factorization of a given matrix $\boldsymbol{M}$ consists of permuting the columns of $\boldsymbol{M}$ by a permutation matrix $\boldsymbol{\Pi}$ so that the rank deficiency of $\boldsymbol{M}$ is revealed in the QR factorization of $\boldsymbol{M\Pi}$.

To compute the RRQR factorization, one naive way is just traversing all possible permutations until we find a promising one. The complexity of the algorithm is of course combinatorial. The first practical algorithm for computing RRQR factorization is QR with column pivoting ([9], Golub, 1965; [10], Businger et al., 1965). Later on, Gragg and Stewart ([11], 1976) suggested to apply the algorithm on the inverse of the matrix. These ideas form the basis for the almost all existing RRQR algorithms.

Ten years later, Chan ([12], 1987) shows that there exist matrices, such as Kahan matrices ([13], Kahan, 1966), for which QR with column pivoting won't perform any permutations. Then he proposed an improved algorithm which involves finding the most dominant right singular vector using the power method. Chan's origin algorithm only guarantees to work for matrices with low numerical deficiency, though in practice it also works for matrices with high nullity.

In 1992, Hong and Pan [14] restricted the definition of the RRQR factorization. They proved that there exist an RRQR factorization which guarantees a specific lower bound on the smallest singular value of the upper right $k \times k$ portion of $\boldsymbol{R}$ and a specific upper bound on the largest singular value of the lower right $m-k \times n-k$ portion of $\boldsymbol{R}$,

for any $m \times n$ matrix $\boldsymbol{M}$ with numerical rank r. They also discussed the application of RRQR factorization in subset selection. Later on, Chandrasekaran and Ipsen ([3], 1994) gave a conclusion of all existing RRQR algorithms and proposed a hybrid algorithm which reach both bounds.

One shortcoming of the RRQR algorithms is that it does not guarantee a stable approximation for the right null space of $\boldsymbol{M}$. To overcome this deficiency, Gu and Eisenstat [2] further restricted the definition of RRQR factorization and defined the notion of strong RRQR factorization. They then showed an algorithm for computing such strong RRQR factorization.

Similarly as RRQR factorization, there also exists rank revealing LU (RRLU) factorization. Given a square $n \times n$ matrix $\boldsymbol{M}$, the problem of computing an RRLU factorization consists of finding two permutation matrix $\boldsymbol{\Gamma}$ and $\boldsymbol{\Pi}$ so that the LU decomposition $\boldsymbol{\Gamma M \Pi} = \boldsymbol{LU}$ reveals the nearly rank deficiency of $\boldsymbol{M}$.

Chan ([15], 1984) presented an algorithm for the case that the nearly rank deficiency of $\boldsymbol{M}$ is one. Hwang et al. ([16], 1992) extended it to the case of more than one small singular value. Later, Miranian and Gu ([17], 2003) introduced the definition of strong RRLU factorization, which is similar to the notion of strong RRQR factorization, and they showed a pivoting strategy for computing such strong RRLU factorization.

These rank revealing algorithms can be used as a reliable and efficient computational alternative to SVD for problems such as rank determination ([1], Chan and Hansen, 1992), linear dependence analysis ([18], Kane et al., 1985), ([19], Huffel and Vandewalle, 1987) and subspace tracking ([20], Bischof et al., 1990), ([21], Common and Golub, 1990).

Chapter 2 QR Factorization

Definition 2.0.1. A rectangular matrix $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ can be decomposed into a product of an orthogonal matrix $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$ and an upper triangular matrix $\boldsymbol{R} \in \mathbb{R}^{m\times n}$:

$$\boldsymbol{M} = \boldsymbol{Q}\boldsymbol{R}$$

This factorization is named as the QR factorization.

Since the condition number (measured in 2-norm) of the orthogonal matrix $\boldsymbol{Q}$ is 1, QR factorization maintains the condition number of the origin matrix and thus it plays a central role in the linear least squares problem.

The rest of this chapter is organized as follows: Section 2.1 first introduce Householder reflections, which is a kind of orthogonal transformation. Then I show the existence of the QR factorization and an practical algorithm for computing a QR factorization based on Householder reflections.

Section 2.2 describes Givens rotations, another kind of orthogonal transformation. Then an QR factorization algorithm based on Givens rotations is presented.

Section 2.3 compares these two orthogonal transformations and gives some advise on which method we should use for different matrices with certain properties.

2.1 Householder QR Factorization

2.1.1 Householder Reflections

Definition 2.1.1. Let $\boldsymbol{v} \in \mathbb{R}^m$ be a nonzero vector. An $m \times m$ matrix $\boldsymbol{H}$ of the form

$$\boldsymbol{H} = \boldsymbol{I} - \beta \boldsymbol{v}\boldsymbol{v}^T, \quad \beta = \frac{2}{\boldsymbol{v}^T\boldsymbol{v}}$$

is called a Householder reflection.

The name 'reflection' comes from the fact that if we left multiply a vector $\boldsymbol{x}$ by $\boldsymbol{H}$,

then $\boldsymbol{x}$ is reflected in the hyperplane $\text{span}\{\boldsymbol{v}\}^{\perp}$. The vector $\boldsymbol{v}$ is called the Householder vector.

Lemma 2.1.1. *Householder matrices are symmetric and orthogonal.*

Proof. Since $\boldsymbol{H} = \boldsymbol{I} - \beta \boldsymbol{v}\boldsymbol{v}^T$, we have

$$\boldsymbol{H}^T = (\boldsymbol{I} - \beta \boldsymbol{v}\boldsymbol{v}^T)^T = \boldsymbol{I} - \beta \boldsymbol{v}\boldsymbol{v}^T = \boldsymbol{H}.$$

This shows the symmetry property. By the definition of β, we have

$$\begin{aligned} \boldsymbol{H}\boldsymbol{H}^T = \boldsymbol{H}\boldsymbol{H} &= (\boldsymbol{I} - \beta \boldsymbol{v}\boldsymbol{v}^T)(\boldsymbol{I} - \beta \boldsymbol{v}\boldsymbol{v}^T) \\ &= \boldsymbol{I} - 2\beta \boldsymbol{v}\boldsymbol{v}^T + \beta^2 \boldsymbol{v}\boldsymbol{v}^T \boldsymbol{v}\boldsymbol{v}^T \\ &= \boldsymbol{I} - 2\beta \boldsymbol{v}\boldsymbol{v}^T + (\beta \boldsymbol{v}^T \boldsymbol{v})\beta \boldsymbol{v}\boldsymbol{v}^T \\ &= \boldsymbol{I}, \end{aligned}$$

which shows the orthogonality of a Householder matrix. □

Householder reflections can be used to zero out selected entries of a given vector. Suppose we are given a nonzero vector $\boldsymbol{x} \in \mathbb{R}^m$ and we want vector

$$\boldsymbol{H}\boldsymbol{x} = \left(\boldsymbol{I} - \frac{2\boldsymbol{v}\boldsymbol{v}^T}{\boldsymbol{v}^T \boldsymbol{v}} \right) \boldsymbol{x} = \boldsymbol{x} - \frac{2\boldsymbol{v}^T \boldsymbol{x}}{\boldsymbol{v}^T \boldsymbol{v}} \boldsymbol{v} \tag{2-1}$$

to be a multiple of $\boldsymbol{e}_1 = [1, 0, ..., 0]^T$. Since equation (2-1) implies $\boldsymbol{v} \in \text{span}\{\boldsymbol{x}, \boldsymbol{e}_1\}$, we assume $\boldsymbol{v} = \boldsymbol{x} + \alpha \boldsymbol{e}_1$. It gives

$$\boldsymbol{v}^T \boldsymbol{x} = \boldsymbol{x}^T \boldsymbol{x} + \alpha \boldsymbol{x}_1 \quad \text{and} \quad \boldsymbol{v}^T \boldsymbol{v} = \boldsymbol{x}^T \boldsymbol{x} + 2\alpha \boldsymbol{x}_1 + \alpha^2$$

Plug this back into Equation (2-1), we obtain

$$\begin{aligned}\boldsymbol{H}\boldsymbol{x} &= \boldsymbol{x} - \frac{2\boldsymbol{v}^T\boldsymbol{x}}{\boldsymbol{v}^T\boldsymbol{v}}(\boldsymbol{x}+\alpha\boldsymbol{e}_1) \\ &= \left(1 - 2\frac{\boldsymbol{x}^T\boldsymbol{x}+\alpha\boldsymbol{x}_1}{\boldsymbol{x}^T\boldsymbol{x}+2\alpha\boldsymbol{x}_1+\alpha^2}\right)\boldsymbol{x} - 2\alpha\frac{\boldsymbol{v}^T\boldsymbol{x}}{\boldsymbol{v}^T\boldsymbol{v}}\boldsymbol{e}_1 \\ &= \left(\frac{\alpha^2 - \|\boldsymbol{x}\|_2^2}{\boldsymbol{x}^T\boldsymbol{x}+2\alpha\boldsymbol{x}_1+\alpha^2}\right)\boldsymbol{x} - 2\alpha\frac{\boldsymbol{v}^T\boldsymbol{x}}{\boldsymbol{v}^T\boldsymbol{v}}\boldsymbol{e}_1\end{aligned}$$

Since we want $\boldsymbol{H}\boldsymbol{x}$ to be a multiple of $\boldsymbol{e}_1$, the coefficient of $\boldsymbol{x}$ should be set to zero, which means $\alpha = \pm\|\boldsymbol{x}\|_2$. Then we have $\boldsymbol{v} = \boldsymbol{x} \pm \|\boldsymbol{x}\|_2\boldsymbol{e}_1$ and

$$\boldsymbol{H}\boldsymbol{x} = -2\alpha\frac{\boldsymbol{v}^T\boldsymbol{x}}{\boldsymbol{v}^T\boldsymbol{v}}\boldsymbol{e}_1 = \mp\|\boldsymbol{x}\|_2\boldsymbol{e}_1.$$

Though we have the freedom to choose the sign of α, underflow issues would arise if it is not chosen correctly. Let $\boldsymbol{x}_1$ denote the first element of vector $\boldsymbol{x}$. To avoid underflow issue, in practical computation, we set

$$\begin{aligned}\boldsymbol{v} &= \boldsymbol{x} + \|\boldsymbol{x}\|_2\boldsymbol{e}_1 \qquad \text{if } \boldsymbol{x}_1 \geq 0, \\ \boldsymbol{v} &= \boldsymbol{x} - \|\boldsymbol{x}\|_2\boldsymbol{e}_1 \qquad \text{if } \boldsymbol{x}_1 < 0.\end{aligned}$$

Algorithm 2.1 shows the pseudocode of finding the Householder vector that zeroes out all but one entries of the given vector.

Algorithm 2.1 Householder Reflection

Input: $\boldsymbol{x} \in \mathbb{R}^m$
Output: $\boldsymbol{v} \in \mathbb{R}^m$ with $\boldsymbol{v}_1 = 1$ such that $(\boldsymbol{I} - \beta\boldsymbol{v}\boldsymbol{v}^T)\boldsymbol{x} = \pm\|\boldsymbol{x}\|_2\boldsymbol{e}_1$.
  procedure HOUSE($\boldsymbol{x}$)
    if $\boldsymbol{x}_1 \geq 0$ **then**
      $\boldsymbol{v} = \boldsymbol{x} + \|\boldsymbol{x}\|_2\boldsymbol{e}_1$
    else
      $\boldsymbol{v} = \boldsymbol{x} - \|\boldsymbol{x}\|_2\boldsymbol{e}_1$
    end if
  end procedure

2.1.2 Algorithm Householder QR

Theorem 2.1.1. *If $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, then there exists an orthogonal matrix $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$ and an upper triangular matrix $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ so that $\boldsymbol{M} = \boldsymbol{QR}$.*

Proof. We prove this by induction.

At the base case $n = 1$, $\boldsymbol{M}$ degenerates to a column vector. Let $\boldsymbol{Q}$ be a Householder matrix so that $\boldsymbol{R} = \boldsymbol{QM}$ has value zero at all but the first entries. Since Householder matrices are orthogonal and symmetric, $\boldsymbol{M} = \boldsymbol{QR}$ is a QR factorization of $\boldsymbol{M}$.

For $n > 1$, partition $\boldsymbol{M}$ as $\boldsymbol{M} = \begin{bmatrix}\boldsymbol{M}_1 & \boldsymbol{v}\end{bmatrix}$, where $\boldsymbol{v}$ denotes the last column $\boldsymbol{M}$. By inductive hypothesis, there exists a QR factorization $\boldsymbol{M}_1 = \boldsymbol{Q}_1\boldsymbol{R}_1$. Let $\boldsymbol{w} = \boldsymbol{Q}_1\boldsymbol{v}$ and we have

$$\boldsymbol{Q}_1\boldsymbol{M} = \begin{bmatrix}\boldsymbol{R}_1 & \boldsymbol{w}\end{bmatrix}.$$

Denote $\boldsymbol{w}_a$ as a subvector of $\boldsymbol{w}$ formed by its first $n-1$ entries and $\boldsymbol{w}_b$ as a subvector of $\boldsymbol{w}$ formed by its last $m - n + 1$ entries. By the base case, we can QR factor $\boldsymbol{w}_b$, which gives $\boldsymbol{w}_b = \boldsymbol{Q}_2\boldsymbol{R}_2$. Define

$$\boldsymbol{Q} = \boldsymbol{Q}_1\begin{bmatrix}\boldsymbol{I}_{n-1} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{Q}_2\end{bmatrix}.$$

Then we have

$$\boldsymbol{M} = \boldsymbol{Q}\left[\begin{array}{c|c} & \boldsymbol{w}_a \\ \boldsymbol{R}_1 & \\ & \boldsymbol{R_2}\end{array}\right]$$

being a QR factorization of $\boldsymbol{M}$. □

The proof of Theorem 2.1.1 also gives a Householder-based algorithm for computing the QR factorization. I illustrate the procedure in detail for a 5×4 matrix $\boldsymbol{M}$. At

the first step, we find a Householder matrix $\boldsymbol{H}_1$ such that

$$\boldsymbol{H}_1 M = \begin{bmatrix} * & * & * & * \\ 0 & \boldsymbol{*} & * & * \\ 0 & \boldsymbol{*} & * & * \\ 0 & \boldsymbol{*} & * & * \\ 0 & \boldsymbol{*} & * & * \end{bmatrix},$$

$\boldsymbol{H}_1$ zeroes out all but the first entries from the first column. Now we focus on these bold entries. Let $\tilde{\boldsymbol{H}}_2$ be a Householder matrix such that

$$\tilde{\boldsymbol{H}}_2 \begin{bmatrix} \boldsymbol{*} \\ \boldsymbol{*} \\ \boldsymbol{*} \\ \boldsymbol{*} \end{bmatrix} = \begin{bmatrix} * \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

and define $\boldsymbol{H_2}$ by

$$\boldsymbol{H_2} = \begin{bmatrix} \boldsymbol{I}_1 & 0 \\ 0 & \tilde{\boldsymbol{H}}_2 \end{bmatrix}.$$

Matrix $\boldsymbol{H_2}$ is still a Householder matrix and it gives

$$\boldsymbol{H_2 H_1 M} = \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix}.$$

Repeat this procedure. For an $m \times n$ matrix $\boldsymbol{M}$, after n steps we will get an upper triangular matrix $\boldsymbol{R} = \boldsymbol{H}_n \boldsymbol{H}_{n-1} \cdots \boldsymbol{H}_1 \boldsymbol{M}$. Since Householder matrices are symmetry and orthogonal, we obtain $\boldsymbol{M} = \boldsymbol{QR}$ by setting $\boldsymbol{Q} = \boldsymbol{H}_1 \cdots \boldsymbol{H}_n$. The pseudocode of Householder QR factorization is presented in Algorithm 2.2.

Algorithm 2.2 Householder QR Factorization, (Explanation Use)

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$ and upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ such that $\boldsymbol{M} = \boldsymbol{QR}$.
procedure HOUSEHOLDERQR($\boldsymbol{M}$)
$\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{R} = \boldsymbol{M}$
for $i = 1, 2, ..., n-1, n$ **do**
Use Algorithm 2.1 to find $\tilde{\boldsymbol{H}}_i$;
Concatenate $\boldsymbol{I}_{i-1}$ with $\tilde{\boldsymbol{H}}_i$ to get $\boldsymbol{H}_i$;
$\boldsymbol{R} = \boldsymbol{H}_i\boldsymbol{R}$;
$\boldsymbol{Q} = \boldsymbol{QH}_i$;
end for
end procedure

There are some important implementation techniques that remain to be mentioned here. In practice, the pseudocode presented in Algorithm 2.2 is rarely used since it does not exploit the property of a Householder matrix.

Algorithm 2.3 Householder QR Factorization (Practical Use)

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$ and upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ such that $\boldsymbol{M} = \boldsymbol{QR}$.
procedure HOUSEHOLDERQR($\boldsymbol{M}$)
$\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{R} = \boldsymbol{M}$
for $i = 1, 2, ..., n-1, n$ **do**
Use Algorithm 2.1 to find $\boldsymbol{v}$;
$\beta = 2/(\boldsymbol{v}'\boldsymbol{v})$;
$\bar{\boldsymbol{R}} = \bar{\boldsymbol{R}} - \beta\boldsymbol{v}(\boldsymbol{v}^T\bar{\boldsymbol{R}})$, where $\bar{\boldsymbol{R}}$ is submatrix of $\boldsymbol{R}$ with its last $m-i+1$ rows;
$\tilde{\boldsymbol{Q}} = \tilde{\boldsymbol{Q}} - \beta(\tilde{\boldsymbol{Q}}\boldsymbol{v})\boldsymbol{v}^T$, where $\tilde{\boldsymbol{Q}}$ is submatrix of $\boldsymbol{Q}$ with its last $m-i+1$ columns;
end for
end procedure

Typically, we store the Householder vector instead of the whole Householder matrix not only due to space efficiency but also because of time complexity. Note that matrix multiplication between 2 matrices of size m will cost $\mathcal{O}(m^3)$ operations. However, if we multiply an order m matrix by $\boldsymbol{I} - \beta\boldsymbol{vv}^T$, the time complexity will be reduced to $\mathcal{O}(m^2)$. Also notice that we don't need to update the whole matrix $\boldsymbol{R}$ and $\boldsymbol{Q}$ at each step. At the ith step, the first $i-1$ rows of $\boldsymbol{R}$ and the first $i-1$ columns of $\boldsymbol{Q}$ won't change.

Algorithm 2.3 shows the pseudocode using these techniques. For a $m \times n$ matrix $\boldsymbol{M}$, the leading order of the operations is $2n^2(m - n/3)$.

2.2 Givens QR Factorization

2.2.1 Givens Rotations

Definition 2.2.1. Givens rotations of rank-2 are matrix of the form

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

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

Theorem 2.2.1. *Givens rotations are orthogonal.*

Proof. Multiply $\boldsymbol{G}(i,k,\theta)$ by its transpose. Then all the off-diagonal entries are zero. All the diagonal elements, except the ith and jth, are 1. The ith and kth diagonal entries have value $s^2 + c^2 = \sin^2(\theta) + \cos^2(\theta) = 1$. So Givens rotations are orthogonal. □

The name 'rotation' comes from the fact that multiplication by $\boldsymbol{G}(i,k,\theta)^T$ amounts to a counterclockwise rotation of angle θ in the (i,k) coordinate plane. To show this, suppose we are given a nonzero vector $\boldsymbol{x} \in \mathbb{R}^m$. Then we have $\boldsymbol{y} = \boldsymbol{G}(i,k,\theta)^T\boldsymbol{x}$ with

$$\boldsymbol{y}_j = \begin{cases} c\boldsymbol{x}_i - s\boldsymbol{x}_k, & j = i, \\ s\boldsymbol{x}_i + c\boldsymbol{x}_k, & j = k, \\ \boldsymbol{x}_j, & j \neq i,k. \end{cases}$$

If we want to zero out $\boldsymbol{y}_k$, we just need to set

$$c = \frac{\boldsymbol{x}_i}{\sqrt{\boldsymbol{x}_i^2 + \boldsymbol{x}_k^2}}, \quad s = \frac{-\boldsymbol{x}_k}{\sqrt{\boldsymbol{x}_i^2 + \boldsymbol{x}_k^2}}.$$

In practice, we do not compute c, s by the above formula directly. Instead we use Algorithm 2.4 for computation stability.

Algorithm 2.4 Givens Rotation

Input: scalars a and b
Output: $c = \cos(\theta)$ and $s = \sin(\theta)$ such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

procedure GIVENS(a, b)
  if $b = 0$ **then**
    $c = 1; s = 0$
  else if $|b| > |a|$ **then**
    $\tau = -a/b; s = 1/\sqrt{1+\tau^2}; c = s\tau;$
  else
    $\tau = -b/a; c = 1/\sqrt{1+\tau^2}; s = c\tau;$
  end if
end procedure

2.2.2 Algorithm Givens QR

We can also apply Givens rotations to get the desired QR factorization. It is illustrated in the following diagram for a 4×3 matrix:

$$\boldsymbol{M} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ 0 & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * \\ * & * & * \\ 0 & * & * \\ 0 & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \\ 0 & * & * \end{bmatrix}$$
$$\longrightarrow \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \end{bmatrix} = \boldsymbol{R}$$

In the diagram, at each step, the Givens rotation denoted as $\boldsymbol{G}_i$ is defined by those 2 bold entries. We again zeroes out elements column by column. But at each column, we need to perform several times of Givens rotations such that all elements below the diagonal are eliminated. At the end, we define $\boldsymbol{Q} = \boldsymbol{G}_1 \cdots \boldsymbol{G}_T$, where T is the total number of rotations. Then we get $\boldsymbol{R} = \boldsymbol{Q}^T \boldsymbol{M}$.

Similarly as in Section 2.1.2, due to time efficiency, we do not formed the $m \times m$ Givens rotation matrix explicitly. Note, when we apply a Givens rotation. Only two rows of $\boldsymbol{R}$ and two columns of $\boldsymbol{Q}$ will change. This leads to a more efficient implementation and the pseudocode is presented in Algorithm 2.5. For an $m \times n$ matrix $\boldsymbol{M}$, the leading order of operations of Algorithm 2.2 is $3n^2(m - n/3)$.

Algorithm 2.5 Givens QR Factorization

Input: $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ with $m \geq n$.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m \times m}$ and upper triangular $\boldsymbol{R} \in \mathbb{R}^{m \times n}$ such that $\boldsymbol{M} = \boldsymbol{Q}\boldsymbol{R}$.

procedure GIVENSQR(a, b)
  $\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{R} = \boldsymbol{M}$
  for $i = 1, 2, ..., n-1, n$ **do**
    for $j = m, m-1, ..., i+1$ **do**
      if $\boldsymbol{M}_{j,i} \neq 0$ **then**
        Let $a = \boldsymbol{M}_{j-1,i}, b = \boldsymbol{M}_{j,i}$;
        Use Algorithm 2.4 to get c, s;
        Let $\boldsymbol{G} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$;
        $\boldsymbol{R}(j-1:j, i:n) = \boldsymbol{G}^T \boldsymbol{R}(j-1:j, i:n)$,
        where $\boldsymbol{R}(j-1:j, i:n)$ is the submatrix formed by the $j-1, j$th rows and $i, ..., n$th columns of $\boldsymbol{R}$;
        $\boldsymbol{Q}(1:m, j-1:j) = \boldsymbol{Q}(1:m, j-1:j)\boldsymbol{G}$,
        where $\boldsymbol{Q}(1:m, j-1:j)$ is the submatrix formed by the $1, ..., m$th rows and $j-1, j$th columns of $\boldsymbol{Q}$;
      end if
    end for
  end for
end procedure

2.3 Comparison between Householder Reflections and Gives Rotations

Householder QR requires $2n^2(m-n/3)$ operations while Givens QR requires $3n^2(m-n/3)$. It seems that Householder QR is always our best choice since it is efficient in zeroing out a large amount of entries of a given vector. However, in some cases that we know some certain property of the given matrix, Givens QR may be much more efficient than Householder QR because it can zero out entries selectively. Here I illustrate it on a 5×5 Hessenberg matrix.

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{*} & * & * & * & * \\ \boldsymbol{*} & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * & * \\ 0 & \boldsymbol{*} & * & * & * \\ 0 & \boldsymbol{*} & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & \boldsymbol{*} & * & * \\ 0 & 0 & \boldsymbol{*} & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & \boldsymbol{*} & * \\ 0 & 0 & 0 & \boldsymbol{*} & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix} = \boldsymbol{R}$$

We can see actually Givens QR only needs to perform 4 times of Givens rotation for this matrix. It can be shown that Givens QR only requires $3n^2$ operations for an $n \times n$ Hessenberg matrix.

Suppose now we run Householder QR. At the first column, we only need to compute the two norm of the first two elements. However, after dealing with the first column, the Hessenberg property is ruined.

Chapter 3 Greedy Rank Revealing QR Factorization

This chapter is organized as follows: Section 3.1 introduces some related background and formulates the problem of finding a rank revealing QR factorization. Section 3.2 presents 7 RRQR algorithms for solving Problem-I and then shows the theoretical bounds they guarantee. A pessimistic matrix for which some algorithms won't work is also shown at the end. Section 3.3 presents the unification principle, by which all algorithms in Section 3.2 can be converted into a corresponding version for solving Problem-II. Those theoretical bounds for Problem-I can also be carried out by the unification principle.

3.1 Background

Given a matrix $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$. Organize the singular values $\sigma_i(\boldsymbol{M})$ as following,

$$\sigma_1(\boldsymbol{M}) \geq \sigma_2(\boldsymbol{M}) \geq \cdots \geq \sigma_n(\boldsymbol{M}).$$

Definition 3.1.1 (Numerical rank)**.** Given two tolerance $\delta, \epsilon > 0$, the numerical rank k of matrix $\boldsymbol{M}$ is defined as the largest integer $1 \leq k \leq n$ that satisfies

$$\sigma_{k+1}(\boldsymbol{M}) > \delta\sigma_k(\boldsymbol{M}) \quad \text{and} \quad \sigma_{k+1}(M) > \epsilon\sigma_1(\boldsymbol{M}),$$

simultaneously.

The definition of rank revealing QR (RRQR) factorization and does not make use of the exact definition of the numerical rank, but it is useful to keep in mind that $\sigma_k(\boldsymbol{M})$ and $\sigma_{k+1}(\boldsymbol{M})$ are well-separated and $\sigma_{k+1}(\boldsymbol{M})$ is sufficiently small.

Definition 3.1.2 (RRQR factorization)**.** Given a matrix $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and an integer k. Let $\boldsymbol{M\Pi} = \boldsymbol{QR}$ be the QR factorization of $\boldsymbol{M}$ with its columns permuted

according to the permutation matrix $\boldsymbol{\Pi}$. Partition $\boldsymbol{R}$ as

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix},$$

where $\boldsymbol{R}_{11} \in \mathbb{R}^{k\times k}$ is an upper triangular matrix. RRQR factorization aims to choose $\boldsymbol{\Pi}$ such that

$$\sigma_{\min}(\boldsymbol{R}_{11}) \approx \sigma_k(\boldsymbol{M}) \quad \text{or} \quad \sigma_{\max}(\boldsymbol{R}_{22}) \approx \sigma_{k+1}(\boldsymbol{M})$$

or both holds simultaneously.

In this chapter, we assume k is pre-given. In Section 5.2.4, we discuss the strategy for determining k. Typically, k is chosen such that $\sigma_k(\boldsymbol{M})$ and $\sigma_{k+1}(\boldsymbol{M})$ are well separated and $\sigma_{k+1}(\boldsymbol{M})$ is sufficiently small. Though the origin definition of RRQR factorization also requires $\boldsymbol{R}_{22}$ to be an upper triangular matrix, this is not necessary if we succeed in achieving $\sigma_{\max}(\boldsymbol{R}_{22}) \approx \sigma_{k+1}(\boldsymbol{M})$ because the matrix $\boldsymbol{R}_{22}$ is negligible compared with $\boldsymbol{R}_{11}$.

An interesting question is, is there any connection between $\sigma_{\min}(\boldsymbol{R}_{11}) \approx \sigma_k(\boldsymbol{M})$ and $\sigma_{\max}(\boldsymbol{R}_{22}) \approx \sigma_{k+1}(\boldsymbol{M})$? In practical, there is. Chan ([12], 1987) presented an algorithm for solving $\sigma_{\max}(\boldsymbol{R}_{22}) \approx \sigma_{k+1}(\boldsymbol{M})$. He reported that in practice, this algorithm also gives $\sigma_{\min}(\boldsymbol{R}_{11}) \approx \sigma_k(\boldsymbol{M})$. But there is no theoretical proof for that connection till now. In this thesis, these three targets as three independent problems.

Lemma 3.1.1 (Interlacing properties of the singular values)**.** *For any permutation* $\boldsymbol{\Pi}$*, we have*

$$\sigma_{\min}(\boldsymbol{R}_{11}) \le \sigma_k(\boldsymbol{M}) \quad \textit{and} \quad \sigma_{\max}(\boldsymbol{R}_{22}) \ge \sigma_{k+1}(\boldsymbol{M}).$$

Proof. Apply Corollary 8.6.3 in ([5], Golub, 2012) to $\boldsymbol{R}^T$. □

By the interlacing properties, I can formulate the RRQR problem as three optimiza-

tion problems:

$$\begin{aligned}
&\text{Problem-I:} && \max_{\boldsymbol{\Pi}} \sigma_{\min}(\boldsymbol{R}_{11}) \\
&\text{Problem-II:} && \min_{\boldsymbol{\Pi}} \sigma_{\max}(\boldsymbol{R}_{22}) \\
&\text{Problem-III:} && \max_{\boldsymbol{\Pi}} \sigma_{\min}(\boldsymbol{R}_{11}) \ \text{ and } \ \min_{\boldsymbol{\Pi}} \sigma_{\max}(\boldsymbol{R}_{22})
\end{aligned}$$

To find the optimal permutation, we can always traverse all possible permutations $\boldsymbol{\Pi}$. However, the time complexity is always combinatorial and such method does not exploit any properties of the given matrix. Actually, we just want to find $\boldsymbol{\Pi}$ that guarantees

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{p_1(k,n)} \quad \text{or} \quad \sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M}) p_2(k,n),$$

where $p_1(k,n)$, $p_2(k,n)$ are functions bounded by low-degree polynomials in k and n.

Lemma 3.1.2. *Suppose $\boldsymbol{M}\boldsymbol{\Pi} = \boldsymbol{Q}\boldsymbol{R}$ is a QR factorization of $\boldsymbol{M}$ with columns permuted according to $\boldsymbol{\Pi}$. If $\boldsymbol{R}\tilde{\boldsymbol{\Pi}} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{R}}$ is an RRQR factorization of $\boldsymbol{R}$, then*

$$\boldsymbol{M}(\boldsymbol{\Pi}\tilde{\boldsymbol{\Pi}}) = (\boldsymbol{Q}\tilde{\boldsymbol{Q}})\tilde{\boldsymbol{R}}$$

is an RRQR factorization of $\boldsymbol{M}$.

Proof. The proof is carried out by the fact that the singular values of $\boldsymbol{R}$ and $\boldsymbol{M}$ are the same. □

By Lemma 3.1.2, we can iteratively update the triangular matrix $\boldsymbol{R}$ instead of working on the origin matrix $\boldsymbol{M}$.

Now, for the ease of notation, during the algorithm's procedure, I partition $\boldsymbol{R}$ as

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{C} \end{bmatrix},$$

where $\boldsymbol{A}$ is a $k \times k$ upper triangular matrix. $\boldsymbol{R}_{11}$, $\boldsymbol{R}_{12}$, $\boldsymbol{R}_{22}$ defined in Definition 3.1.2 is used to denote the submatrix of the final result.

3.2 Greedy Algorithms for Problem Type-I

3.2.1 Algorithm Greedy-I.1

Let $\boldsymbol{R}^{(l)}$ be the $m \times n$ matrix $\boldsymbol{R}$ at step l. Partition it as

$$\boldsymbol{R}^{(l)} = \begin{array}{c} \\ l \\ m-l \end{array} \begin{array}{c} \begin{array}{cc} l & n-l \end{array} \\ \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{C} \end{bmatrix} \end{array},$$

where $\boldsymbol{A}$ is an upper triangular matrices. Denote the columns of $\boldsymbol{B}$ and $\boldsymbol{C}$ by $\boldsymbol{b}_i, \boldsymbol{c}_i$ for $1 \leq i \leq n-l$. First, I present the pseudocode of Greedy-I.1 in Algorithm 3.1.

Algorithm 3.1 RRQR Greedy-I.1

Input: $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ with $m \geq n$ and k.

Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m \times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m \times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n \times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

procedure GREEDYI.1($\boldsymbol{M}$,k)

$\boldsymbol{R}^{(0)} = \boldsymbol{M}$;

$\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{\Pi} = \boldsymbol{I}$;

for $l = 0, 1, ..., k-1$ **do**

Find j such that $\max_{1 \leq i \leq n-l} \sigma_{\min} \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \boldsymbol{c}_i \end{bmatrix} = \sigma_{\min} \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_j \\ \boldsymbol{0} & \boldsymbol{c}_j \end{bmatrix}$;

Exchange columns $l+1$ and $l+j$ of $\boldsymbol{R}^{(l)}$ and update $\boldsymbol{\Pi}$;

Retriangularize the first $l+1$ columns of $\boldsymbol{R}^{(l)}$ by Householder reflection to get $\boldsymbol{R}^{(l+1)}$;

Update $\boldsymbol{Q}$;

end for

end procedure

The idea of the algorithm Greedy-I.1 is very simple. Recall that the objective of Problem-I is to find the k most well-conditioned columns of $\boldsymbol{M}$. Suppose now we have already find $l < k$ such 'good' columns. At the $l+1$th step, we select a column from the remaining $n-l$ candidate columns such that the smallest singular value of the matrix formed by the given l columns and the new column is maximized. At $l = 0$, the Algorithm 3.1 will select the column of $\boldsymbol{R}$ that has the largest 2-norm. We repeat k times and expect to get k well-conditioned columns of $\boldsymbol{M}$.

Note, Algorithm Greedy-I.1 examines the smallest singular value $n-l$ times at each step. The smallest singular value of a matrix $\boldsymbol{S}$ can be computed by applying inverse iteration on $\boldsymbol{S}^T\boldsymbol{S}$. Finding such j is expensive, so Algorithm Greedy-I.1 rarely appears in practical use.

3.2.2 Algorithm Greedy-I.2

Algorithm Greedy-I.2 tries to approximate the smallest singular value in order to speed up computations. Though Greedy-I.2 is an approximation for Greedy-I.1, since the greedy strategy is not guaranteed to be optimal, algorithm Greedy-I.1 is not necessary to work better than Greedy-I.2.

For $\boldsymbol{R}^{(l)}$ defined in the algorithm Greedy-I.1, we define $\gamma_i = \|\boldsymbol{c}_i\|_2$ for $1 \leq i \leq n-l$.

Lemma 3.2.1. *For $1 \leq i \leq n-l$, we have*

$$\sigma_{\min}\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \boldsymbol{c}_i \end{bmatrix} = \sigma_{\min}\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \gamma_i \end{bmatrix}$$

Proof. Block matrix multiplications gives

$$\begin{bmatrix} \boldsymbol{A}^T & \boldsymbol{0} \\ \boldsymbol{b}_i^T & \boldsymbol{c}_i^T \end{bmatrix}\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \boldsymbol{c}_i \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}^T\boldsymbol{A} & \boldsymbol{A}^T\boldsymbol{b}_i \\ \boldsymbol{A}\boldsymbol{b}_i^T & \boldsymbol{b}_i^T\boldsymbol{b}_i + \boldsymbol{c}_i^T\boldsymbol{c}_i \end{bmatrix},$$

and

$$\begin{bmatrix} \boldsymbol{A}^T & \boldsymbol{0} \\ \boldsymbol{b}_i^T & \gamma_i \end{bmatrix}\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \gamma_i \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}^T\boldsymbol{A} & \boldsymbol{A}^T\boldsymbol{b}_i \\ \boldsymbol{A}\boldsymbol{b}_i^T & \boldsymbol{b}_i^T\boldsymbol{b}_i + \gamma_i^2 \end{bmatrix}.$$

Since the singular values of a matrix $\boldsymbol{S}$ are the square root of the eigenvalues of $\boldsymbol{S}^T\boldsymbol{S}$ and since $\gamma_i^2 = \boldsymbol{c}_i^T\boldsymbol{c}_i$, the singular values of the two matrices are the same. □

Lemma 3.2.1 reduce the problem to determining the smallest singular value of a square upper triangular matrix of order $l+1$. Notice that the smallest singular value of a matrix is the reciprocal of the largest singular value of its inverse. Now, we can approximate that largest singular value by the largest 2-norm of each row vector.

Lemma 3.2.2. *Given a nonsingular matrix $\boldsymbol{S} \in \mathbb{R}^{n\times n}$. Suppose $\boldsymbol{S}^{-1}$ has the form,*

$$\boldsymbol{S}^{-1} = \begin{bmatrix} \boldsymbol{s}_1^T \\ \boldsymbol{s}_2^T \\ \vdots \\ \boldsymbol{s}_n^T \end{bmatrix}.$$

Then

$$\sigma_{\min}(\boldsymbol{S}) \leq \min_{1\leq i\leq n} \frac{1}{\|\boldsymbol{s}_i\|_2} \leq \sigma_{\min}(\boldsymbol{S})\sqrt{n}.$$

Proof. From Section 2.3.2 ([5], Golub, 2012), we have

$$\frac{\sigma_{\max}(\boldsymbol{S}^{-1})}{\sqrt{n}} \leq \max_{1\leq i\leq n} \|\boldsymbol{s}_i\|_2 \leq \sigma_{\max}(\boldsymbol{S}^{-1})$$

Since $\sigma_{\min}(\boldsymbol{S}) = 1/\sigma_{\max}(\boldsymbol{S}^{-1})$, inverse the above inequaltiy gives the desired result. □

Lemma 3.2.2 gives an estimation of the smallest singular value and this leads to algorithm Greedy-I.2. The pseudocode is shown in Algorithm 3.2.

One may notice that the Greedy-I.2 involves inverting an order $l+1$ matrix $n-l$ times for each l. Since

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \gamma_i \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{A}^{-1} & -\boldsymbol{A}^{-1}\boldsymbol{b}_i\gamma_i^{-1} \\ \boldsymbol{0} & \gamma_i^{-1} \end{bmatrix}$$

and $\boldsymbol{A}^{-1}$ is available from the previous step, we only need to compute the last column, which involves only $n-l$ times of matrix vector multiplications. So Greedy-I.2 runs much faster than Greedy-I.1 in practice.

Algorithm 3.2 RRQR Greedy-I.2

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and k.

Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

procedure GREEDYI.2($\boldsymbol{M}$,k)

$\boldsymbol{R}^{(0)} = \boldsymbol{M}$;

$\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{\Pi} = \boldsymbol{I}$;

for $l = 0, 1, ..., k-1$ **do**

Find j such that

$$\max_{1\leq i\leq n-l} \min_{h} \left\| \boldsymbol{e}_h^T \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \gamma_i \end{bmatrix}^{-1} \right\|_2^{-1} = \min_{h} \left\| \boldsymbol{e}_h^T \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_j \\ \boldsymbol{0} & \gamma_j \end{bmatrix}^{-1} \right\|_2^{-1}$$

where $\boldsymbol{e}_h$ is a row vector of size $l+1$ with all zero entries except the hth entry being one;

Exchange columns $l+1$ and $l+j$ of $\boldsymbol{R}^{(l)}$ and update $\boldsymbol{\Pi}$;

Retriangularize the first $l+1$ columns of $\boldsymbol{R}^{(l)}$ by Householder reflection to get $\boldsymbol{R}^{(l+1)}$;

Update $\boldsymbol{Q}$;

end for

end procedure

3.2.3 Algorithm Greedy-I.3

Algorithm Greedy-I.3 aims to further approximate Greedy-I.2. If our greedy algorithm works well till the lth step, then $\boldsymbol{A}$, the upper left $l \times l$ portion of $\boldsymbol{R}^{(l)}$, should be well-conditioned. This means $\sigma_{\max}(\boldsymbol{A}^{-1}) = 1/\sigma_{\min}(\boldsymbol{A})$ is small. By Lemma 3.2.2, no row of $\boldsymbol{A}^{-1}$ can have a large 2-norm. Instead of comparing the 2-norm of each row of the inverse matrix, we can compare the magnitude of each entry of the last column of the inverse matrix,

$$\left\| \boldsymbol{e}_h^T \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \gamma_i \end{bmatrix}^{-1} \right\|_2^{-1} \approx \left| \boldsymbol{e}_h^T \begin{bmatrix} -\boldsymbol{A}^{-1}\boldsymbol{b}_i\gamma_i^{-1} \\ \gamma_i^{-1} \end{bmatrix} \right|^{-1},$$

where $\boldsymbol{e}_h$ is a row vector of size $l+1$ with all elements being zero except that the hth entry being one. If adding the new column cause the new $\boldsymbol{A}$ to be deficient, then the 2-norm of some row of that new $\boldsymbol{A}_{new}^{-1}$ must be large. Since we assumed that no row of

the old $\boldsymbol{A}_{old}^{-1}$ has a large 2-norm, there must be some large element in the last column of the new $\boldsymbol{A}_{new}^{-1}$. So this approximation actually avoids selecting a bad column.

Algorithm Greedy-I.3 requires less computations than GreedyI.2. The pseudocode is shown in Algorithm 3.3.

Algorithm 3.3 RRQR Greedy-I.3

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and k.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.
procedure GREEDYI.3($\boldsymbol{M}$,k)
$\boldsymbol{R}^{(0)} = \boldsymbol{R}$;
$\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{\Pi} = \boldsymbol{I}$;
for $l = 0, 1, ..., k-1$ **do**
Find j such that

$$\max_{1\leq i\leq n-l} \min_h \left| \boldsymbol{e}_h^T \begin{bmatrix} -\boldsymbol{A}^{-1}\boldsymbol{b}_i\gamma_i^{-1} \\ \gamma_i^{-1} \end{bmatrix} \right|^{-1} = \min_h \left| \boldsymbol{e}_h^T \begin{bmatrix} -\boldsymbol{A}^{-1}\boldsymbol{b}_j\gamma_j^{-1} \\ \gamma_j^{-1} \end{bmatrix} \right|^{-1}$$

Exchange columns $l+1$ and $l+j$ of $\boldsymbol{R}^{(l)}$ and update $\boldsymbol{\Pi}$;
Retriangularize the first $l+1$ columns of $\boldsymbol{R}^{(l)}$ by Householder reflection to get $\boldsymbol{R}^{(l+1)}$;
Update $\boldsymbol{Q}$;
end for
end procedure

3.2.4 Algorithm QR with Column Pivoting

Now, we look for further approximation of algorithm Greedy-I.3. Since $\boldsymbol{A}$ is probably well-conditioned, if we assume $\boldsymbol{b}_i$ is not very large, then $\|\boldsymbol{A}^{-1}\boldsymbol{b}_i\|_2$ is not very large. If adding some column would let the new $\boldsymbol{A}_{new}$ to be ill-conditioned, then the large component of $\boldsymbol{A}_{new}^{-1}$ may probably comes from γ_i^{-1}. This gives the following approximation,

$$\min_h \left| \boldsymbol{e}_h^T \begin{bmatrix} -\boldsymbol{A}^{-1}\boldsymbol{b}_i\gamma_i^{-1} \\ \gamma_i^{-1} \end{bmatrix} \right|^{-1} \approx \gamma_i.$$

The corresponding algorithm is known as QR with column pivoting ([9], Golub, 1965; [10], Businger et al., 1965). This algorithm still avoids the selection of a bad column.

It is worth noting that QR with column pivoting is very efficient to implement be-

cause we can update the value of γ_i instead of recompute it. In practice, its running time is similar to the tradition Householder QR and Givens QR. Techniques for updating γ_i are shown in Section 5.3. The pseudocode of QR with column pivoting is presented in Algorithm 3.4.

Algorithm 3.4 QR with column pivoting

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and k.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

```
procedure ColumnPivotingQR(M,k)
    R = M;
    Q = I, Π = I;
    for l = 0, 1, ..., k − 1 do
        Find j such that max_{1≤i≤n−l} γ_i = γ_j;
        Exchange columns l + 1 and l + j of R^(l) and update Π;
        Retriangularize the first l + 1 columns of R^(l) by Householder reflection to
get R^(l+1);
        Update Q;
    end for
end procedure
```

3.2.5 Algorithm Chan

Chan's algorithm ([12], 1987) is another kind of greedy algorithm but has similar behaviors as QR with column pivoting.

Recall $\boldsymbol{C}$ is the lower right upper triangular submatrix of $\boldsymbol{R}^{(l)}$. It computes the right singular vector $\boldsymbol{v}$ of $\boldsymbol{C}$,

$$\boldsymbol{Cv} = \|\boldsymbol{C}\|_2\boldsymbol{u},$$

where $\|\boldsymbol{v}\| = \|\boldsymbol{u}\| = 1$. Then it choose column j of $\boldsymbol{C}$ such that $|\boldsymbol{v}_j| = \max_{1\leq i\leq n-l} |\boldsymbol{v}_i|$. The idea of this algorithm is also very simple. Since $\boldsymbol{v}$ is the right singular vector according to $\|\boldsymbol{C}\|_2$, then presumably its largest element would correspond to the most dominant column in $\boldsymbol{C}$. This gives

$$\gamma_j = \|\boldsymbol{Ce}_j\|_2 = \|\boldsymbol{u}\|_2\|\boldsymbol{Ce}_j\|_2 \geq |\boldsymbol{u}^T\boldsymbol{Ce}_j| = \|\boldsymbol{C}\|_2|\boldsymbol{v}_j|, \tag{3-1}$$

where the inequality comes from Cauchy-Schwartz inequality. Since vector $\boldsymbol{v}$ is of size $n-l$ and it is has unitary two norm, the largest entry of $\boldsymbol{v}$ must satisfy $\boldsymbol{v}_j \geq 1/\sqrt{n-l}$.

Now let's compare the behaviors of algorithm Chan and QR with column pivoting. Denote $\max_{i \leq i \leq n-l} \gamma_i$ by r_{l+1}, which is the lth diagonal element of the final upper triangular matrix $\boldsymbol{R}$ computed by QR with column pivoting. Plug this back to Inequality (3-1), we have

$$r_{l+1} \geq \gamma_j \geq \frac{\|\boldsymbol{C}\|_2}{\sqrt{n-l}} \geq \frac{r_{l+1}}{n-l}. \tag{3-2}$$

We see for l close to n, both Algorithm Chan and QR with column pivoting will give almost the same column if they are given the same leading l columns. The pseudocode is presented in Algorithm 3.5.

Algorithm 3.5 RRQR Chan

Input: $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ with $m \geq n$ and k.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m \times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m \times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n \times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.
procedure CHAN($\boldsymbol{M}$,k)
  $\boldsymbol{R} = \boldsymbol{M}$;
  $\boldsymbol{Q} = \boldsymbol{I}, \boldsymbol{\Pi} = \boldsymbol{I}$;
  for $l = 0, 1, ..., k-1$ **do**
    Compute the right singular vector $\boldsymbol{v}$ of $\boldsymbol{C}$ corresponding to $\|\boldsymbol{C}\|_2$.
    Find j such that $|\boldsymbol{v}_j| = \max_{1 \leq i \leq n-l} |\boldsymbol{v}_i|$.
    Exchange columns $l+1$ and $l+j$ of $\boldsymbol{R}^{(l)}$ and update $\boldsymbol{\Pi}$;
    Retriangularize the first $l+1$ columns of $\boldsymbol{R}^{(l)}$ by Householder reflection to get $\boldsymbol{R}^{(l+1)}$;
    Update $\boldsymbol{Q}$;
  end for
end procedure

It would be expensive to find $\boldsymbol{v}$ exactly. Performing a few steps of power method on $\boldsymbol{C}^T\boldsymbol{C}$ will give an approximation of $\boldsymbol{v}$ which works well in practice.

3.2.6 Algorithm GKS

Let's look back to Inequality (3-2). Combining the interlacing property, $\|\boldsymbol{C}\|_2 \geq \sigma_{l+1}(\boldsymbol{M})$, we have

$$r_{l+1} \geq \frac{\sigma_{l+1}(\boldsymbol{M})}{\sqrt{n-l}}, \quad \text{for } 0 \leq l \leq k-1. \tag{3-3}$$

However, our objective of Problem-I is just to guarantee

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{p(n,k)}.$$

In other word, the previous 5 greedy algorithms try to achieve that the lth diagonal element of the resulting $\boldsymbol{R}_{11}$ is approximately the lth largest singular value

$$|(\boldsymbol{R}_{11})_{ll}| \approx \sigma_l(\boldsymbol{M}), \quad \text{for } 1 \leq l \leq k.$$

But sometimes we may only want

$$|(\boldsymbol{R}_{11})_{ll}| \approx \sigma_k(\boldsymbol{M}), \quad \text{for } 1 \leq l \leq k.$$

Instead of restricting all diagonal elements of $\boldsymbol{R}_{11}$, we hope only the smallest diagonal element is important. This kind of relaxation will give us more freedom in selecting columns.

The algorithm GKS, named after its authors, Golub, Klema, and Stewart ([22], 1976), is an extension of QR with column pivoting based on this relaxation.

Suppose $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ is the SVD of $\boldsymbol{M}$. Now partition $\boldsymbol{V}$ as

$$\begin{matrix} & \text{k} & \text{n-k} \end{matrix}$$
$$\boldsymbol{V} = \begin{bmatrix} \boldsymbol{V}_1 & \boldsymbol{V}_2 \end{bmatrix}.$$

The idea of the algorithm is to first apply QR with column pivoting on $\boldsymbol{V}_1^T$ which will give a permutation matrix $\boldsymbol{\Pi}$. Then use this $\boldsymbol{\Pi}$ to get an RRQR factorization of the

origin $\boldsymbol{M}$. The pseudocode is presented in Algorithm 3.6.

Algorithm 3.6 RRQR GKS

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and k.
Output: Orthogonal $\bar{\boldsymbol{Q}} \in \mathbb{R}^{m\times m}$, upper triangular $\bar{\boldsymbol{R}} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$ reveals the rank deficiency of $\boldsymbol{M}$.
procedure GKS($\boldsymbol{M}$,k)
Find $\boldsymbol{V}_1$ by computing the k most dominant right singular vectors of $\boldsymbol{M}$;
Get $\boldsymbol{V}_1^T\boldsymbol{\Pi} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{V}}_1^T$ by applying QR with column pivoting on $\boldsymbol{V}_1^T$.
Compute QR factorization $\boldsymbol{M\Pi} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$.
end procedure

To illustrate why Algorithm 3.6 work, suppose $\boldsymbol{M}$ is a square matrix. If not, then we can QR factor $\boldsymbol{M}$ and work on its upper square portion. Partition the SVD of $\boldsymbol{M}$ as,

$$\boldsymbol{M} = \boldsymbol{U}\begin{bmatrix}\boldsymbol{\Sigma}_1 & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{\Sigma}_2\end{bmatrix}\begin{bmatrix}\boldsymbol{V}_1 & \boldsymbol{V}_2\end{bmatrix}^T.$$

By the QR factorization $\boldsymbol{M\Pi} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$, we have $\bar{\boldsymbol{R}} = \bar{\boldsymbol{Q}}^T\boldsymbol{M\Pi}$. Combine $\boldsymbol{V}_1^T\boldsymbol{\Pi} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{V}}_1^T$, we obtain

$$\tilde{\boldsymbol{V}}_1^T\bar{\boldsymbol{R}}^{-1} = (\tilde{\boldsymbol{Q}}^T\boldsymbol{V}_1^T\boldsymbol{\Pi})(\boldsymbol{\Pi}^T\boldsymbol{M}^{-1}\bar{\boldsymbol{Q}}) = \tilde{\boldsymbol{Q}}^T\boldsymbol{V}_1^T\boldsymbol{M}^{-1}\bar{\boldsymbol{Q}}. \tag{3-4}$$

Since

$$\boldsymbol{M}^{-1} = \begin{bmatrix}\boldsymbol{V}_1 & \boldsymbol{V}_2\end{bmatrix}\begin{bmatrix}\boldsymbol{\Sigma}_1^{-1} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{\Sigma}_2^{-1}\end{bmatrix}\boldsymbol{U}^T,$$

we have $\boldsymbol{V}_1^T\boldsymbol{M}^{-1} = \boldsymbol{\Sigma}_1^{-1}\boldsymbol{U}^T$. Plug this back into Equation (3-4), we get

$$\tilde{\boldsymbol{V}}_1^T\bar{\boldsymbol{R}}^{-1} = \tilde{\boldsymbol{Q}}^T\boldsymbol{\Sigma}_1^{-1}\boldsymbol{U}^T\bar{\boldsymbol{Q}}.$$

Because both $\bar{\boldsymbol{R}}^{-1}$ and $\tilde{\boldsymbol{V}}_1^T$ are upper triangular, we have

$$\frac{|(\tilde{\boldsymbol{V}}_1)_{ii}|}{|(\bar{\boldsymbol{R}})_{ii}|} = \frac{|(\tilde{\boldsymbol{V}}_1)_{ii}^T|}{|(\bar{\boldsymbol{R}})_{ii}|} \leq \|\tilde{\boldsymbol{V}}_1^T\bar{\boldsymbol{R}}^{-1}\|_2 = \|\boldsymbol{\Sigma}_1^{-1}\|_2 = \frac{1}{\sigma_k(\boldsymbol{M})}, \tag{3-5}$$

for $1 \leq i \leq k$. Because $\boldsymbol{V}_1^T$ is of size $k \times n$ and QR with column pivoting ensures

the diagonal entries has the largest magnitude, we have $|(\tilde{\boldsymbol{V}}_1)_{ii}| \geq 1/\sqrt{n}$. Plug this into Inequality (3-5), we obtain

$$|(\bar{\boldsymbol{R}})_{ii}| \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{n}} \quad \text{for } 1 \leq i \leq k.$$

The most computational expensive step in Algorithm GKS is finding $\boldsymbol{V}_1$. Of course we don't want to compute the full SVD of the matrix. Finding the k most dominant right singular vectors through truncated SVD or other variants of power method is still expensive.

3.2.7 Algorithm Foster

Differs from the previous 6 greedy algorithms, Algorithm Foster ([23], Foster, 1986) requires an additional parameter δ, which is a tolerance presumably about as big as $\sigma_k(\boldsymbol{M})$. The algorithm tries to achieve $\sigma_{\min}(\boldsymbol{R}_{11}) \approx \delta$ by selecting diagonal entries greater than or equal to δ. Specifically, it searches from bottom to top and focus on the rows of the lower right portion of $\boldsymbol{R}$. When it finds an element such that its magnitude is greater than δ, it adds that column into $\boldsymbol{R}_{11}$ and retriangularize the matrix. After going through n rows, the algorithm terminates. The first k diagonal elements of the resulting $\boldsymbol{R}$ will be at least δ if the algorithm succeeded in finding k such elements. The pseudocode is shown in Algorithm 3.7.

We see that algorithm Foster is very efficient. If the parameter δ is chosen correctly, then it also gives a good result in practice.

3.2.8 Bounds at each Iteration

In this section, I first show a lower bound on the smallest singular value at each step for algorithm Greedy-I.1, Greedy-I.2, Greedy-I.3 and QR with column pivoting. Then I show an exact lower bound on $\sigma_{\min}(\boldsymbol{R}_{11})$ for Algorithm QR with column pivoting, Chan and GKS.

Algorithm 3.7 RRQR Foster

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$, k and δ.

Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

procedure FOSTER($\boldsymbol{M}$,k,δ)
  $\boldsymbol{QR} = \boldsymbol{M}$ (by Algorithm 2.2);
  $i = n, l = 0$;
  for $count = 1, 2, ..., n$ **do**
    Find the element $\boldsymbol{R}_{ij}$ with maximum magnitude in row i;
    if $|\boldsymbol{R}_{ij}| \geq \delta$ **then**
      Insert column j between the lth and $l+1$st column of $\boldsymbol{R}$ and update $\boldsymbol{\Pi}$;
      Retriangularize the first $l+1$ columns of $\boldsymbol{R}$ by Householder reflection and update $\boldsymbol{Q}$;
      $l = l + 1$;
    else
      $i = i - 1$;
    end if
  end for
end procedure

Suppose we are at the lth step and partition $\boldsymbol{R}^{(l)}$ as

$$\boldsymbol{R}^{(l)} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{C} \end{bmatrix},$$

where $\boldsymbol{A}$ is an $l \times l$ upper triangular matrix. Let $\bar{\sigma}_l$ denote the smallest singular value of $\boldsymbol{A}$ for $\boldsymbol{R}^{(l)}$. At the next step, Greedy-I.1 choose column j such that

$$\bar{\sigma}_{l+1} = \max_{1\leq i\leq n-l} \sigma_{\min} \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_i \\ \boldsymbol{0} & \boldsymbol{c}_i \end{bmatrix} = \sigma_{\min} \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_j \\ \boldsymbol{0} & \boldsymbol{c}_j \end{bmatrix}$$

Now, instead of computing a lower bound for $\bar{\sigma}_{l+1}$, we compute a lower bound for the column selected by QR with column pivoting, given the same $\boldsymbol{A}$. This lower bound will also be a lower bound for the column that Greedy-I.1 would select. Assume QR with column pivoting will pick column j^*. The selection criteria implies that column j^* has the largest two norm among all columns of $\boldsymbol{C}$.

By Lemma 3.2.2, $\sigma_{\min}$ can be estimated by the reciprocal of the largest 2-norm of

the rows of the inverse matrix

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b}_{j^*} \\ 0 & \gamma_{j^*} \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{A}^{-1} & -\boldsymbol{A}^{-1}\boldsymbol{b}_{j^*}\gamma_{j^*}^{-1} \\ 0 & \gamma_{j^*}^{-1} \end{bmatrix}. \tag{3-6}$$

Lemma 3.2.2 gives a bound for the approximation error,

$$\bar{\sigma}_l \leq \min_{1\leq i\leq l} \frac{1}{\|\boldsymbol{e}_i^T \boldsymbol{A}^{-1}\|_2} \leq \bar{\sigma}_l \sqrt{l}. \tag{3-7}$$

This is equivalent to

$$\frac{1}{\bar{\sigma}_l} \geq \max_{1\leq i\leq l} \|\boldsymbol{e}_i^T \boldsymbol{A}^{-1}\|_2 \geq \frac{1}{\bar{\sigma}_l \sqrt{l}}. \tag{3-8}$$

Back to Equation (3-6), the largest row 2-norm among the leading l rows can be bounded by

$$\begin{aligned} \max_{1\leq i\leq l} \left\{ \|\boldsymbol{e}_i^T \boldsymbol{A}^{-1}\|_2 + \|\boldsymbol{e}_i^T \boldsymbol{A}^{-1}\boldsymbol{b}_{j^*}\|_2 \right\} &\leq \frac{1}{\bar{\sigma}_l} + \frac{\|\boldsymbol{b}_i\|_2}{\bar{\sigma}_l}\gamma_{j^*}^{-1} \\ &\leq \frac{2\sqrt{\gamma_i \cdot \|\boldsymbol{b}_i\|_2}}{\bar{\sigma}_l}\gamma_{j^*}^{-1} \\ &\leq \frac{\sqrt{2} \cdot \sqrt{\gamma_i^2 + \|\boldsymbol{b}_i\|_2^2}}{\bar{\sigma}_l}\gamma_{j^*}^{-1} \\ &\leq \frac{\sqrt{2} \cdot \|\boldsymbol{M}\|_2}{\bar{\sigma}_l}\gamma_{j^*}^{-1} \end{aligned} \tag{3-9}$$

The first inequality comes from Inequality (3-8) and the property of matrix norms. The second inequality comes from the inequality of arithmetic and geometric means. The third inequality comes from Cauch-Schwartz inequality. The forth inequality comes from Lemma (3.2.2) and the interlacing properties of singular values. Apply Inequality (3-7) on $l+1$ gives

$$\bar{\sigma}_l \geq \frac{1}{\sqrt{l+1} \cdot \max_{1\leq i\leq l} \|\boldsymbol{e}_1^T \boldsymbol{A}^{-1}\|_2}.$$

Plug this into Inequality (3-9) and we obtain

$$\bar{\sigma}_{l+1} \geq \frac{1}{\sqrt{l+1}} \frac{\bar{\sigma}_l}{\sigma_1(\boldsymbol{M})}\gamma_{j^*}. \tag{3-10}$$

By Lemma (3.2.2) and the interlacing property, we have

$$\gamma_{j^*} = \max_{1 \le i \le n-l} \gamma_i \ge \frac{\sigma_{l+1}(\boldsymbol{M})}{\sqrt{n-l}}.$$

Plug this back into Inequality (3-10). We get

$$\bar{\sigma}_{l+1} \ge \sigma_{l+1}(\boldsymbol{M}) \cdot \frac{\bar{\sigma}_l}{\sigma_1(\boldsymbol{M})} \cdot \frac{1}{\sqrt{2(l+1)(n-l)}} \tag{3-11}$$

Inequality (3-11) shows one shortcoming of these greedy algorithms. Even if the former l columns had beed well-selected such that $\bar{\sigma}_l$ is almost accurate, it is still possible for the greedy algorithm to fail in choosing the $l+1$st column.

This is due to the fact that the greedy strategy is not necessary to be the optimal. A 'good' column that ensures an accurate estimation for $\bar{\sigma}_l$ may not be included in an accurate estimation for $\bar{\sigma}_{l+1}$. So the main problem of greedy algorithms is that they never get rid of a selected column. Section 3.2.10 shows a pessimistic ill-conditioned matrix on which Algorithm Greedy-I.1, Greedy-I.2, Greedy-I.3 and QR with column pivoting won't perform any permutations and thus fail to reveal the rank deficiency of the matrix.

Since these greedy algorithms do not guarantee to work for all matrices, one may ask why are we interested in these algorithms? The answer is, compared with the hybrid algorithms in the following sections, greedy algorithms, especially QR with column pivoting, Chan and Foster, are typically much more efficient. And if we ignore those pessimistic matrices, in practice, some greedy algorithms work pretty well.

3.2.9 Bounds for the Final Result

Now, I show lower bounds on $\sigma_{\min}(\boldsymbol{R}_{11})$ for Algorithm QR with column pivoting, Chan and GKS.

The idea of the proof is to look for a $k \times k$ triangular $\boldsymbol{W}$ with

$$\begin{aligned} |\boldsymbol{W}_{ii}| &= 1, \quad \text{for } 1 \le i \le k, \\ |\boldsymbol{W}_{ij}| &\le 1, \quad \text{for } 1 \le i, j \le k, \end{aligned} \tag{3-12}$$

such that

$$\frac{\sigma_k(\boldsymbol{M})}{\sqrt{nk}\|\boldsymbol{W}^{-1}\|_2} \le \sigma_{\min}(\boldsymbol{R}_{11}) \le \sigma_k(\boldsymbol{M}),$$

for algorithm Chan and

$$\frac{\sigma_k(\boldsymbol{M})}{\sqrt{n}\|\boldsymbol{W}^{-1}\|_2} \le \sigma_{\min}(\boldsymbol{R}_{11}) \le \sigma_k(\boldsymbol{M}),$$

for algorithm QR with column pivoting and GKS. The matrix $\boldsymbol{W}$ is chosen in different ways for different algorithms.

Kahan ([13], 1966) gave a tight upper bound on $\|\boldsymbol{W}^{-1}\|_2$,

$$\|\boldsymbol{W}^{-1}\|_2 \le \frac{1}{3}\sqrt{4^k + 6k - 1} \le \sqrt{k}2^k, \quad k > 1$$

and the bound can be reached by the pessimistic example shown in Section 3.2.10.

Lower Bound for QR with Column Pivoting

Define $\boldsymbol{W}$ by

$$\boldsymbol{R}_{11} = \boldsymbol{D}\boldsymbol{W},$$

where $\boldsymbol{D}$ is a diagonal matrix which shares its diagonal elements with $\boldsymbol{R}_{11}$. The algorithm ensures $|(\boldsymbol{R})_{ii}| \ge |(\boldsymbol{R}_{11})_{ij}|$ for $1 \le i, j \le k$. So this $\boldsymbol{W}$ satisfies Equation (3-12). Recall Inequality (3-3),

$$|(\boldsymbol{R}_{11})_{ii}| \triangleq r_i \ge \frac{\sigma_i(\boldsymbol{M})}{\sqrt{n-i+1}}, \quad \text{for } 1 \le i \le k. \tag{3-13}$$

From this, we obtain

$$\frac{1}{\sigma_{\min}(\boldsymbol{D})} = \|\boldsymbol{D}^{-1}\|_2 \le \frac{\sqrt{n}}{\sigma_k(\boldsymbol{M})},$$

where the inequality comes from Inequality (3-13) and the fact that $\boldsymbol{D}$ is a diagonal matrix. Since $\sigma_{\min}(\boldsymbol{R}_{11}) \geq \sigma_{\min}(\boldsymbol{D})\sigma_{\min}(\boldsymbol{W})$, we have the following bound

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{n}\|\boldsymbol{W}^{-1}\|_2}.$$

Lower Bound for Chan

Let $\boldsymbol{R}_{22}^{(l)}$ be the lower right $n-l+1$ portion of the final $\boldsymbol{R}$. Define an $n \times k$ auxiliary matrix $\boldsymbol{Z}$ whose columns are composed by concatenating a zero column vector of size $l-1$ with the right singular vector $\boldsymbol{v}^{(l)}$ corresponding to the largest singular value of $\boldsymbol{R}_{22}^{(l)}$. So the matrix $\boldsymbol{Z}$ is a lower trapezoidal matrix illustrated as following:

$$\begin{array}{c} \\ l-1 \\ n-l+1 \end{array}\begin{array}{c} 1 \\ \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{v}^{(l)} \end{bmatrix} \end{array} = \boldsymbol{Z}_l, \quad \boldsymbol{Z} = \begin{bmatrix} * & 0 & \cdots & \cdots & 0 \\ * & * & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ * & * & \cdots & * & 0 \\ * & * & \cdots & * & * \\ \vdots & \vdots & & \vdots & \vdots \\ * & * & \cdots & * & * \end{bmatrix}_{n\times k}$$

By definition, we have $\boldsymbol{R}_{22}^{(l)}\boldsymbol{v}^{(l)} = \|\boldsymbol{R}_{22}^{(l)}\|_2\boldsymbol{u}^{(l)}$ with $\|\boldsymbol{v}^{(l)}\|_2 = \|\boldsymbol{u}^{(l)}\|_2 = 1$. Define the order k lower triangular matrix $\boldsymbol{W}$ as

$$\boldsymbol{Z} = \begin{bmatrix} \boldsymbol{W} \\ * \end{bmatrix} \boldsymbol{D},$$

where $\boldsymbol{D}$ is a diagonal matrix that shares its diagonal elements with the upper k rows of $\boldsymbol{Z}$. Algorithm Chan ensures the first entry of $\boldsymbol{v}^{(l)}$ is always the largest in magnitude. So this $\boldsymbol{W}$ satisfies Equation (3-12). Since $\|\boldsymbol{v}^{(l)}\|_2 = 1$, we have

$$|\boldsymbol{Z}_{ll}| = |\boldsymbol{v}_1^{(l)}| \geq \frac{1}{\sqrt{n}},$$

for $1 \leq l \leq k$. By definition of $\boldsymbol{D}$, we obtain

$$\frac{1}{\|\boldsymbol{D}^{-1}\|_2} = \sigma_{\min}(\boldsymbol{D}) \geq \frac{1}{\sqrt{n}} \tag{3-14}$$

By our selection of $\boldsymbol{v}^{(l)}$ and $\boldsymbol{u}^{(l)}$, we have

$$(\boldsymbol{v}^{(l)})^T(\boldsymbol{R}_{22}^{(l)})^{-1} = \frac{1}{\|\boldsymbol{R}_{22}^{(l)}\|_2}(\boldsymbol{u}^{(l)})^T. \tag{3-15}$$

Since $\boldsymbol{R}$ is upper triangular and $\boldsymbol{Z}_l$ has value 0 in its frist $l-1$ entries, we have

$$\|\boldsymbol{Z}_l^T\boldsymbol{R}^{-1}\|_2 = \|(\boldsymbol{v}^{(l)})^T(\boldsymbol{R}_{22}^{(l)})^{-1}\|_2 \leq \frac{1}{\sigma_l(\boldsymbol{M})}, \tag{3-16}$$

where the last inequality comes from the interlacing property, $\|\boldsymbol{R}_{22}^{(l)}\|_2 \geq \sigma_l(\boldsymbol{M})$. Since $\boldsymbol{Z}^T$ and $\boldsymbol{R}$ are upper triangular and $\boldsymbol{R}_{11}$ is a submatrix of $\boldsymbol{R}$, we have

$$\|\boldsymbol{D}\boldsymbol{W}^T\boldsymbol{R}_{11}^{-1}\|_2 \leq \|\boldsymbol{Z}^T\boldsymbol{R}^{-1}\|_2 \tag{3-17}$$

Use $\boldsymbol{Z}^T$ to denote the lth row of $\boldsymbol{Z}^T$. Lemma 3.2.2 relates $\|\boldsymbol{Z}^T\boldsymbol{R}^{-1}\|_2$ with $\|\boldsymbol{Z}_l^T\boldsymbol{R}^{-1}\|_2$ by

$$\|\boldsymbol{Z}^T\boldsymbol{R}^{-1}\|_2 \leq \sqrt{k}\max_{1\leq l\leq k}\|\boldsymbol{Z}_l^T\boldsymbol{R}^{-1}\|_2 \tag{3-18}$$

Combine Inequality (3-16), (3-17) and (3-18), we obtain

$$\frac{\|\boldsymbol{R}_{11}^{-1}\|_2}{\|\boldsymbol{D}^{-1}\|_2\|\boldsymbol{W}^{-1}\|_2} \leq \frac{\sqrt{k}}{\sigma_k(\boldsymbol{M})}. \tag{3-19}$$

Plug Inequality (3-14) into (3-19), we get the desired bound

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{kn}\|\boldsymbol{W}^{-1}\|_2}.$$

Lower Bound for GKS

Let $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ be the SVD of $\boldsymbol{M}$. Partition $\boldsymbol{V}$ as

$$\boldsymbol{V} = \begin{matrix} \\ \end{matrix} \begin{matrix} k & n-k \end{matrix} \\ \begin{bmatrix} \boldsymbol{V}_1 & \boldsymbol{V}_2 \end{bmatrix}.$$

Algorithm GKS applies QR with column pivoting to $\boldsymbol{V}_1^T$ which results in

$$\boldsymbol{V}_1^T\boldsymbol{\Pi} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{V}}_1^T.$$

Identify that $\tilde{\boldsymbol{V}}_1$ is a trapezoidal matrix. The triangular matrix $\boldsymbol{W}$ is again defined by

$$\tilde{\boldsymbol{V}}_1 = \begin{bmatrix} \boldsymbol{W} \\ * \end{bmatrix} \boldsymbol{D},$$

where $\boldsymbol{D}$ is a diagonal matrix that shares its diagonal elements with the upper k rows of $\tilde{\boldsymbol{V}}_1$. Since $\tilde{\boldsymbol{V}}_1$ is the output of QR with column pivoting, the diagonal elements of $\tilde{\boldsymbol{V}}_1$ is guaranteed to have the largest magnitude in each row. So such $\boldsymbol{W}$ satisfies Equation (3-12).

Since each column of $\tilde{\boldsymbol{V}}_1$ has 2-norm 1, we have $|\tilde{\boldsymbol{V}}_{ii}| \geq 1/\sqrt{n}$. So Inequality (3-14) also holds here. Notice that $\boldsymbol{W}^T$ and the final matrix $\bar{\boldsymbol{R}}$ are upper triangular and $\bar{\boldsymbol{R}}_{11}$ is a submatrix of $\bar{\boldsymbol{R}}$. Combine Inequality (3-14) and we obtain

$$\begin{aligned} \frac{\|\bar{\boldsymbol{R}}_{11}^{-1}\|_2}{\sqrt{n}\|\boldsymbol{W}^{-1}\|_2} &\leq \frac{\|\bar{\boldsymbol{R}}_{11}^{-1}\|_2}{\|(\boldsymbol{W}^{-1})^T\|_2\|\boldsymbol{D}^{-1}\|_2} \\ &\leq \|\boldsymbol{D}\boldsymbol{W}^T\bar{\boldsymbol{R}}_{11}^{-1}\|_2 \\ &\leq \|\tilde{\boldsymbol{V}}_1^T\bar{\boldsymbol{R}}^{-1}\|_2. \end{aligned} \tag{3-20}$$

Recall Equation (3-5),

$$\|\tilde{\boldsymbol{V}}_1^T\bar{\boldsymbol{R}}^{-1}\|_2 = \|\boldsymbol{\Sigma}_{\boldsymbol{1}}^{-1}\|_2 = \frac{1}{\sigma_k(\boldsymbol{M})}.$$

Plug this back into Inequality (3-20) gives the bound

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{n}\|\boldsymbol{W}^{-1}\|_2}.$$

3.2.10 Pessimistic Example

There exist matrices on which Algorithm Greedy-I.1, Greedy-I.2, Greedy-I.3, QR with column pivoting do not perform any permutations. One well-known example is the Kahan matrix ([13], Kahna, 1966).

Definition 3.2.1. Kahan matrix of order n is defined as

$$\boldsymbol{K}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & s & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & s^{n-1} \end{bmatrix} \begin{bmatrix} 1 & -c & \cdots & -c \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -c \\ 0 & \cdots & 0 & 1 \end{bmatrix},$$

where $c^2 + s^2 = 1$.

Theorem 3.2.1. *Algorithm Greedy-I.1 does not perform any permutations for Kahan matrix.*

Proof. We prove this by induction. At the first step, since all columns of $\boldsymbol{K}_n$ has unitary 2-norm, Greedy-I.1 won't perform any permutations. Now suppose no permutations are performed at the first l steps, then we have

$$\boldsymbol{K}_n = \begin{bmatrix} \boldsymbol{K}_l & \boldsymbol{b}_1 & \cdots & \boldsymbol{b}_{n-l} \\ \boldsymbol{0} & \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_{n-l} \end{bmatrix}$$

As before, let $\gamma_i = \|\boldsymbol{c}_i\|_2$ for $1 \leq i \leq n-l$. At the $l+1$st step, the algorithm choose i from $1, ..., n-l$ such that

$$\sigma_{\min} \begin{bmatrix} \boldsymbol{K}_l & \boldsymbol{b}_i \\ \boldsymbol{0} & \boldsymbol{c}_i \end{bmatrix} = \sigma_{\min} \begin{bmatrix} \boldsymbol{K}_l & \boldsymbol{b}_i \\ \boldsymbol{0} & \gamma_i \end{bmatrix}$$

is maximized. Identify that b_i are exactly the same for all i. Since all columns of $\boldsymbol{K}_n$ has unit 2-norm, γ_i are also the same for all i. So no permutations are performed in the $l+1$st step. Thus, Algorithm Greedy-I.1 does not perform any permutations for Kahan matrix. □

For $n = 100, k = 99, c = 0.2$, we have

$$\sigma_{100}(\boldsymbol{K}_{100}) \approx 0.000000003678056, \quad \sigma_{99}(\boldsymbol{K}_{100}) \approx 0.148211206273914$$

We see that the 100th and 99th singular value of $\boldsymbol{K}_{100}$ are well separated. Set $k = 99$ and apply QR with column pivoting, I obtain

$$\sigma_{99}(\boldsymbol{R}_{11}) \approx 0.000000004504681.$$

The numerical experiments accords that QR with column pivoting fails to reveal the rank deficiency of the Kahan matrix. Section 6.1 shows the numerical results of all these greedy algorithms on different matrices.

3.3 Greedy Algorithms for Problem Type-II

3.3.1 The Unification Principle

In Section 3.2, we discussed some greedy algorithms for solving Problem-I. Now we focus on Problem-II which is

$$\min_{\boldsymbol{\Pi}} \sigma_{\max}(\boldsymbol{R}_{22})$$

such that

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M}) p_2(k, n),$$

where $p_2(k, n)$ is a function bounded by low-degree polynomials in k, n.

In this section, we show the unification principle, which enable us to convert an algorithm for solving Problem-I to a corresponding version for solving Problem-II.

Lemma 3.1.2 tells that given a matrix $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ with $m \geq n$, we can always first

QR factor $\boldsymbol{M} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$ and work on the triangular matrix $\bar{\boldsymbol{R}}$. Further, it is equivalent to work on the upper n rows of $\bar{\boldsymbol{R}}$ since the rest portion are zero. So we can assume $\bar{\boldsymbol{R}}$ is a square upper triangular matrix. Given a permutation matrix $\boldsymbol{\Pi}$, QR factorization on $\bar{\boldsymbol{R}}\boldsymbol{\Pi}$ gives

$$\bar{\boldsymbol{R}}\boldsymbol{\Pi} = \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{Q}\begin{bmatrix}\boldsymbol{R}_{11} & \boldsymbol{R}_{12}\\ \boldsymbol{0} & \boldsymbol{R}_{22}\end{bmatrix}. \tag{3-21}$$

Suppose $\bar{\boldsymbol{R}}$ is nonsingular. Inverting both sides of Equation (3-21), we obtain

$$\boldsymbol{\Pi}^T\bar{\boldsymbol{R}}^{-1} = \begin{bmatrix}\boldsymbol{R}_{11}^{-1} & -\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\boldsymbol{R}_{22}^{-1}\\ \boldsymbol{0} & \boldsymbol{R}_{22}^{-1}\end{bmatrix}\boldsymbol{Q}^T. \tag{3-22}$$

Take transpose on both sides of Equation (3-22),

$$(\bar{\boldsymbol{R}}^{-1})^T\boldsymbol{\Pi} = \boldsymbol{Q}\begin{bmatrix}(\boldsymbol{R}_{11}^{-1})^T & \boldsymbol{0}\\ -(\boldsymbol{R}_{22}^{-1})^T\boldsymbol{R}_{12}^T(\boldsymbol{R}_{11}^{-1})^T & (\boldsymbol{R}_{22}^{-1})^T\end{bmatrix}. \tag{3-23}$$

Now we can formulate Problem-II as

$$\begin{aligned}\min_{\boldsymbol{\Pi}}\sigma_{\max}(\boldsymbol{R}_{22}) &= \min_{\boldsymbol{\Pi}}\frac{1}{\sigma_{\min}(\boldsymbol{R}_{22}^{-1})}\\ &= \frac{1}{\max_{\boldsymbol{\Pi}}\sigma_{\min}(\boldsymbol{R}_{22}^{-1})}\\ &= \frac{1}{\max_{\boldsymbol{\Pi}}\sigma_{\min}((\boldsymbol{R}_{22}^{-1})^T)}.\end{aligned}$$

Now we set run an algorithm for solving Problem-I on matrix $(\bar{\boldsymbol{R}}^{-1})^T$ with k_1 given by $n-k$ (k_1 is the input parameter for the Problem-I algorithm and k is the input parameter for the Problem-II algorithm). We obtain

$$(\bar{\boldsymbol{R}}^{-1})^T\bar{\boldsymbol{\Pi}} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{P}} = \bar{\boldsymbol{Q}}\begin{bmatrix}\boldsymbol{P}_{11} & \boldsymbol{P}_{12}\\ \boldsymbol{0} & \boldsymbol{P}_{22}\end{bmatrix}, \tag{3-24}$$

where $\boldsymbol{P}_{11}$ is an upper triangular matrices of order $n-k$. Now we need to convert Equation (3-24) into an RRQR factorization of $\boldsymbol{R}$. This can be achieved by permuting

the matrix $\bar{\boldsymbol{P}}$ into a lower triangular matrix shown in Equation (3-23). Notice that row permutations can be absorbed into $\bar{\boldsymbol{Q}}$ while column permutations can be absorbed into $\bar{\boldsymbol{\Pi}}$. After updated $\bar{\boldsymbol{\Pi}}$ and $\bar{\boldsymbol{Q}}$ to $\boldsymbol{\Pi}$ and $\boldsymbol{Q}$, we get the final $\boldsymbol{R}$ by $\boldsymbol{R} = \boldsymbol{Q}^T \bar{\boldsymbol{R}} \boldsymbol{\Pi}$.

Define $\boldsymbol{J}_p$ be a permutation matrix of order p with ones on the antidiagonal entries. The following diagram illustrates the procedure to permute $\boldsymbol{P}$.

$$\begin{bmatrix} \boldsymbol{P}_{11} & \boldsymbol{P}_{12} \\ \boldsymbol{0} & \boldsymbol{P}_{22} \end{bmatrix} \rightarrow \begin{bmatrix} \boldsymbol{P}_{12} & \boldsymbol{P}_{11} \\ \boldsymbol{P}_{22} & \boldsymbol{0} \end{bmatrix} \rightarrow \begin{bmatrix} \boldsymbol{P}_{22} & \boldsymbol{0} \\ \boldsymbol{P}_{12} & \boldsymbol{P}_{11} \end{bmatrix} \rightarrow \begin{bmatrix} \boldsymbol{J}_k \boldsymbol{P}_{22} \boldsymbol{J}_k & \boldsymbol{0} \\ \boldsymbol{J}_{n-k} \boldsymbol{P}_{12} \boldsymbol{J}_k & \boldsymbol{J}_{n-k} \boldsymbol{P}_{11} \boldsymbol{J}_{n-k} \end{bmatrix}$$

Now we see running an algorithm for solving Problem-I on the transpose of the inverse of the matrix gives an algorithm for solving Problem-II. This is known as the *unification principle*. From now on, we add '-I' after the name of the algorithm to denote the version for Problem-I and '-II' to denote the version for for Problem-II.

Not all algorithms for Problem-II have to be implemented as in the *unification principle*. There may be some ways to reformulate the Problem-II algorithms so that it doesn't need to deal with inverse explicitly.

Note, the *unification principle* just says we can modify the permutation matrix to convert Problem-I algorithms to Problem-II algorithms. Since the permutations are different, solving Problem-I does not implies solving Problem-II and vice versa.

3.3.2 Bounds for the Final Result

Section 3.2.8 shows we have the following bounds

$$\frac{\sigma_k(\boldsymbol{M})}{\sqrt{nk}\|\boldsymbol{W}^{-1}\|_2} \leq \sigma_{\min}(\boldsymbol{R}_{11}) \leq \sigma_k(\boldsymbol{M}),$$

for algorithm Chan-I and,

$$\frac{\sigma_k(\boldsymbol{M})}{\sqrt{n}\|\boldsymbol{W}^{-1}\|_2} \leq \sigma_{\min}(\boldsymbol{R}_{11}) \leq \sigma_k(\boldsymbol{M}),$$

for algorithm QR with column pivoting-I and GKS-I.

The $k \times k$ matrix $\boldsymbol{W}$ should satisfy

$$|\boldsymbol{W}_{ii}| = 1, \quad \text{for } 1 \leq i \leq k,$$
$$|\boldsymbol{W}_{ij}| \leq 1, \quad \text{for } 1 \leq i, j \leq k,$$

Theorem 3.3.1. *With the same $\boldsymbol{W}$ defined as above, the following bound*

$$\sigma_{k+1}(\boldsymbol{M}) \leq \sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{n(n-k)} \cdot \|\boldsymbol{W}^{-1}\|_2,$$

holds for algorithm Chan-II and,

$$\sigma_{k+1}(\boldsymbol{M}) \leq \sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{n} \cdot \|\boldsymbol{W}^{-1}\|_2,$$

holds for algorithm QR with column pivoting-II, GKS-II.

Proof. The proof relies on the unification principle. I prove it for Chan-II. The proof for QR with column pivoting-II and GKS-II are exactly the same.

In Section 3.3.1, we see that Problem-II can be reformulated as

$$\min_{\boldsymbol{\Pi}} \sigma_{\max}(\boldsymbol{R}_{22}) = \frac{1}{\max_{\boldsymbol{\Pi}} \sigma_{\min}((\boldsymbol{R}_{22}^{-1})^T)}$$

Run Algoirhtm Chan-I on $(\boldsymbol{M}^{-1})^T$ and use the lower bound for Chan-I,

$$\begin{aligned}\frac{\sigma_{n-k}((\boldsymbol{M}^{-1})^T)}{\sqrt{n(n-k)}\|\boldsymbol{W}^{-1}\|_2} &\leq \sigma_{\min}(\boldsymbol{J}_{n-k}(\boldsymbol{R}_{22}^{-1})^T\boldsymbol{J}_{n-k}) \\ &= \sigma_{\min}(\boldsymbol{R}_{22}^{-1}) \\ &= \frac{1}{\sigma_{\max}(\boldsymbol{R}_{22})}\end{aligned}$$

Invert both sides gives

$$\begin{aligned}\sigma_{\max}(\boldsymbol{R}_{22}) &\leq \frac{\sqrt{n(n-k)}\|\boldsymbol{W}^{-1}\|_2}{\sigma_{n-k}(\boldsymbol{M}^{-1})} \\ &= \sigma_{k+1}(\boldsymbol{M})\sqrt{n(n-k)} \cdot \|\boldsymbol{W}^{-1}\|_2\end{aligned}$$

Combine the interlacing property, we obtain

$$\sigma_{k+1}(\boldsymbol{M}) \leq \sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{n(n-k)} \cdot \|\boldsymbol{W}^{-1}\|_2$$

□

From the lower bound, we see that Chan-I works well for small k while Chan-II works well when k is close to n. This property actually works on most algorithms. In practice, when k is small, algorithms for Problem-I are preferred, while algorithms for Problem-II are preferred if k is close to n.

3.3.3 Pessimistic Example

Section 3.2.10 shows that Algorithm Greedy-I.1, Greedy-I.2, Greedy-I.3, QR with column pivoting-I do not perform any permutations on Kahan matrices. By the unification principle, Algorithm Greedy-II.1, Greedy-II.2, Greedy-II.3, QR with column pivoting-II will fail to reveal the rank deficiency of the following modified Kahan matrix whose inverse is given by

$$\bar{\boldsymbol{K}}_n = \begin{bmatrix} 1 & -c & \cdots & -c \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -c \\ 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & s & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & s^{n-1} \end{bmatrix}.$$

Chapter 4 Hybrid Rank Revealing QR Factorization

Section 3 present three formulations of the RRQR factorization problems and show a set of greedy algorithms for solving Problem Type-I and Problem Type-II. An obvious shortcoming of these greedy algorithms is that they never permute columns that have already been selected and thus some of them perform very bad on some pessimistic ill-conditioned matrix.

Hybrid algorithms improve these greedy algorithms by also considering previous columns at each step. Given k, greedy algorithms will permutate k columns. However, the total number of permutations that hybrid algorithms would perform is unknown. There is no complete analysis of the worst-case operation count for hybrid algorithms, which we believe that it might be combinatorial. In practice, hybrid algorithms run very fast and always give a better result since they guarantee tighter bounds for the singular values.

This chapter is organized as follows: In Section 4.1, I present a hybrid algorithm, Hybrid-I, for solving Problem-I and it guarantees

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{k(n-k+1)}},$$

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{\min}(\boldsymbol{R}_{11})\sqrt{k(n-k+1)}.$$

By unification principle, a hybrid algorithm, Hybrid-II, for Problem-II is shown in Section 4.2 and it guarantees

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_{\max}(\boldsymbol{R}_{22})}{\sqrt{(k+1)(n-k)}},$$

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{(k+1)(n-k)}.$$

Combining Hybrid-I and Hybrid-II gives an algorithm Hybrid-III which simultaneously

solve Problem-I and Problem-II. It guarantees

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{k(n-k+1)}},$$

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{(k+1)(n-k)}.$$

4.1 Hybrid Algorithm for Problem Type-I

4.1.1 Algorithm Hybrid-I

QR with column pivoting is a very efficient algorithm for computing the RRQR factorization. The idea of algorithm Hybrid-I is to alternate between Problem-I version and Problem-II version of QR with column pivoting.

Let's first define some notations for algorithm Hybrid-I. Suppose we are given a matrix $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ such that its first k columns are upper triangular. Partition $\boldsymbol{R}$ in two different ways

$$\begin{array}{c} \\ k-1 \\ m-k+1 \end{array}\begin{array}{c} \begin{array}{cc} k-1 & n-k+1 \end{array} \\ \begin{bmatrix} \bar{\boldsymbol{A}} & \bar{\boldsymbol{B}} \\ \boldsymbol{0} & \bar{\boldsymbol{C}} \end{bmatrix} \end{array} = \bar{\boldsymbol{R}}, \qquad \begin{array}{c} \\ k \\ m-k \end{array}\begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{C} \end{bmatrix} \end{array} = \boldsymbol{R},$$

where $\bar{\boldsymbol{A}}$ is upper triangular with order $k-1$ and $\boldsymbol{A}$ is upper triangular with order k.

At each step, QR with column pivoting-I works on $\bar{\boldsymbol{C}}$, the lower right $n-k+1$ portion of the triangular matrix $\boldsymbol{R}$. It permutes the column with the largest 2-norm to the kth column of $\boldsymbol{R}$. QR with column pivoting-II works on $\bar{\boldsymbol{A}}$, the upper left k portion of $\boldsymbol{R}$. It permutes the worst column from that portion with the kth column. The algorithm halts when QR with column pivoting-I agrees that column k is the best column among all columns of $\bar{\boldsymbol{C}}$ and QR with column pivoting-II agrees that column k is the worst column among all columns of $\boldsymbol{A}$.

Algorithm Hybrid-I makes use of the fact that QR with column pivoting-I is good at approximating the largest singular value of a matrix while QR with column pivoting-II is good at approximating the smallest singular value. To see this, suppose we are

given an $m \times n$ matrix $\boldsymbol{M}$, with $m \leq n$. Apply QR with column pivoting-I and get $\boldsymbol{M\Pi} = \boldsymbol{QR}$. As shown in Section 3.2, we have

$$|r_{11}| \leq \sigma_{\max}(\boldsymbol{M}) \leq \sqrt{n}|r_{11}|, \tag{4-1}$$

where r_{11} is the first diagonal entry of $\boldsymbol{R}$. For QR with column pivoting-II, since we permute the most dominant column of the inverse to the last column, we have

$$|r_{nn}^{-1}| \leq \sigma_{\max}(\boldsymbol{M}^{-1}) \leq \sqrt{n}|r_{nn}^{-1}|. \tag{4-2}$$

This is equivalent to

$$\sigma_{\min}(\boldsymbol{M}) \leq |r_{nn}| \leq \sqrt{n}\sigma_{\min}(\boldsymbol{M}). \tag{4-3}$$

So we see that QR with column pivoting-II is good at approximating the largest singular value of $\boldsymbol{M}^{-1}$. The pseudocode of Hybrid-I is presented in Algorithm 4.1. One may notice that QR with column pivoting-II involves calculating the 2-norm of rows in $\boldsymbol{A}^{-1}$. Instead of calculating the inverse explicitly, we can update it in each iteration. Also, after applying QR with column pivoting-I, we just need to retriangularize the kth column of $\boldsymbol{R}$ because the upper left $k-1$ portion remains unchanged. If we exchange column k with column j such that $j < k$, then the upper left $k-1$ portion is no longer upper triangular. Notice, we just need column j to be at the kth position. If we left shift column $j+1, ..., k$ by one column and move column j to the kth position, then the upper left k portion is a Hessenberg matrix, which can be efficiently triangularized by Givens rotations. These implementation techniques are extremely important and they make these hybrid algorithms to be efficient enough compared with those greedy algorithms. They are shown in detail in Section 5.3.

Algorithm 4.1 RRQR Hybrid-I

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and k.

Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

procedure HYBRID-I($\boldsymbol{M}$,k)
  QR factor $\boldsymbol{M}$ up to the kth column, $\boldsymbol{QR} = \boldsymbol{M}$;
  $\boldsymbol{\Pi} = \boldsymbol{I}, permuted = true$;
  while $permuted$ **do**
    $permuted = false$;
    Find j such that $\|\bar{\boldsymbol{C}}\boldsymbol{e}_j\|_2 = \max_{1\leq i\leq n-k+1} \|\bar{\boldsymbol{C}}\boldsymbol{e}_i\|_2$;
    if $\|\bar{\boldsymbol{C}}\boldsymbol{e}_1\|_2 < \|\bar{\boldsymbol{C}}\boldsymbol{e}_j\|_2$ **then**
      $permuted = true$;
      Exchange column k and $k+j-1$ of $\boldsymbol{R}$ and update $\boldsymbol{\Pi}$;
      Retriangularize the kth column of $\boldsymbol{R}$ and update $\boldsymbol{Q}$;
    end if
    Find j such that $\|\boldsymbol{e}_j^T\boldsymbol{A}^{-1}\|_2 = \max_{1\leq i\leq k} \|\boldsymbol{e}_i^T\boldsymbol{A}^{-1}\|_2$;
    if $\|\boldsymbol{e}_k^T\boldsymbol{A}^{-1}\|_2 < \|\boldsymbol{e}_j^T\boldsymbol{A}^{-1}\|_2$ **then**
      $permuted = true$
      Left shift column $j+1, ..., k$ by one column and move column j to the kth position.
      Update $\boldsymbol{\Pi}$;
      Retriangularize the first k columns of $\boldsymbol{R}$ and update $\boldsymbol{Q}$;
    end if
  end while
end procedure

4.1.2 Analytical Bound

Partition the final matrix $\boldsymbol{R}$ in two different ways

$$\boldsymbol{R} = \begin{bmatrix} \bar{\boldsymbol{R}}_{11} & \bar{\boldsymbol{R}}_{12} \\ \mathbf{0} & \bar{\boldsymbol{R}}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \mathbf{0} & \boldsymbol{R}_{22} \end{bmatrix},$$

where $\bar{\boldsymbol{R}}_{11}$ is an upper triangular with order $k-1$ and $\boldsymbol{R}_{11}$ is upper triangular with order k.

Theorem 4.1.1. *When Algorithm Hybrid-I halts, it guarantees*

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{k(n-k+1)}},$$

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{\min}(\boldsymbol{R}_{11})\sqrt{k(n-k+1)}.$$

Proof. At the time Hybrid-I halts, QR with column pivoting-I applied to $\bar{\boldsymbol{R}}_{22}$ does not cause any column permutations. Let r_{ij} denote the element $\boldsymbol{e}_i^T \boldsymbol{R} \boldsymbol{e}_j$. By Inequality (4-1),

$$|r_{kk}| \geq \frac{\sigma_{\max}(\bar{\boldsymbol{R}}_{22})}{\sqrt{n-k+1}} \geq \frac{\sigma_{\max}(\boldsymbol{R}_{22})}{\sqrt{n-k+1}}, \tag{4-4}$$

where the last inequality comes from the fact that $\boldsymbol{R}_{22}$ is a submatrix of $\bar{\boldsymbol{R}}_{22}$. Note that QR with column pivoting-II applied to $\boldsymbol{R}_{11}$ also does not cause any permutations. By Inequality (4-3),

$$|r_{kk}| \leq \sigma_{\min}(\boldsymbol{R}_{11})\sqrt{k}. \tag{4-5}$$

Combining Inequality (4-4) and (4-5) gives the first bound

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{\min}(\boldsymbol{R}_{11})\sqrt{k(n-k+1)} \tag{4-6}$$

Apply the interlacing property on Inequality (4-4), we obtain

$$|r_{kk}| \geq \frac{\sigma_{\max}(\bar{\boldsymbol{R}}_{22})}{\sqrt{n-k+1}} \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{n-k+1}}, \tag{4-7}$$

Combining Inequality (4-5) and (4-7) gives the second bound

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{k(n-k+1)}}. \tag{4-8}$$

□

Theorem 4.1.2. *Algorithm Hybrid-I does halt given* $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ *and* $1 \leq k \leq n$ *such that* $\sigma_k(\boldsymbol{M}) > 0$.

Proof. Since $|\det(\boldsymbol{A})|$ is unique for any given permutation matrix, there are finitely many possible values for $|\det(\boldsymbol{A})|$. If $|\det(\boldsymbol{A})|$ is strictly increasing, the algorithm must halt at some step. Now it remains to show that $|\det(\boldsymbol{A})|$ does strictly increase during the algorithm.

Notice, QR with column pivoting-II won't change $\det(\boldsymbol{A})$ because permutation of columns and orthogonal transformation does not change the determinant of a matrix.

Now we focus on QR with column pivoting-I. First, if doesn't perform any permutations, then the algorithm will halt at the next loop. The proof is done.

Suppose QR with column pivoting-I permute the column j with the largest 2-norm in $\bar{\boldsymbol{C}}$ to the kth column. After retriangularization, the value of the kth diagonal element is $\|\bar{\boldsymbol{C}}\boldsymbol{e}_j\|_2$, which is strictly larger than the previous one. Since the other elements in $\boldsymbol{A}$ does not change after retriangularization and $\boldsymbol{A}$ is an upper triangular matrix, $|\det(\boldsymbol{A})|$ strictly increases at each step and this completes the halting argument. □

4.2 Hybrid Algorithm for Problem Type-II

4.2.1 Algorithm Hybrid-II

Section 4.1 present a Hybrid algorithm for Problem-I. By the unification principle, there is a corresponding Hybrid algorithm, Hybrid-II, for solving Problem-II. Actually, simply applying Hybrid-I with $k+1$ gives Hybrid-II.

Now let's define some notations for the Algorithm Hybrid-II. Suppose we are given a matrix $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ such that its first $k+1$ columns are upper triangular. Partition $\boldsymbol{R}$

in two different ways, $\boldsymbol{R} \in \mathbb{R}^{m\times n}$.

$$\begin{array}{c} \\ k \\ m-k \end{array}\begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{C} \end{bmatrix} \end{array} = \boldsymbol{R}, \qquad \begin{array}{c} \\ k+1 \\ m-k-1 \end{array}\begin{array}{c} \begin{array}{cc} k+1 & n-k-1 \end{array} \\ \begin{bmatrix} \hat{\boldsymbol{A}} & \hat{\boldsymbol{B}} \\ \boldsymbol{0} & \hat{\boldsymbol{C}} \end{bmatrix} \end{array} = \hat{\boldsymbol{R}},$$

where $\hat{\boldsymbol{A}}$ is an upper triangular matrix of order $k+1$ and $\boldsymbol{A}$ is an upper triangular matrix of order k. The pseudocode is shown in Algorithm 4.2.

Algorithm 4.2 RRQR Hybrid-II

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and k.

Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

procedure HYBRID-II($\boldsymbol{M}$,k)
  QR factor $\boldsymbol{M}$ up to the $k+1$th column, $\boldsymbol{QR} = \boldsymbol{M}$;
  $\boldsymbol{\Pi} = \boldsymbol{I}, permuted = true$;
  while $permuted$ **do**
    $permuted = false$;
    Find j such that $\|\boldsymbol{Ce}_j\|_2 = \max_{1\leq i\leq n-k} \|\boldsymbol{Ce}_i\|_2$;
    if $\|\boldsymbol{Ce}_1\|_2 < \|\boldsymbol{Ce}_j\|_2$ **then**
      $permuted = true$;
      Exchange column $k+1$ and $k+j$ of $\boldsymbol{R}$ and update $\boldsymbol{\Pi}$;
      Retriangularize the $k+1$th column of $\boldsymbol{R}$ and update $\boldsymbol{Q}$;
    end if
    Find j such that $\|\boldsymbol{e}_j^T\hat{\boldsymbol{A}}^{-1}\|_2 = \max_{1\leq i\leq k+1} \|\boldsymbol{e}_i^T\hat{\boldsymbol{A}}^{-1}\|_2$;
    if $\|\boldsymbol{e}_{k+1}^T\hat{\boldsymbol{A}}^{-1}\|_2 < \|\boldsymbol{e}_j^T\hat{\boldsymbol{A}}^{-1}\|_2$ **then**
      $permuted = true$
      Left shift column $j+1, ..., k+1$ by one column and move column j to the $k+1$th position.
      Update $\boldsymbol{\Pi}$;
      Retriangularize the first $k+1$ columns of $\boldsymbol{R}$ and update $\boldsymbol{Q}$;
    end if
  end while
end procedure

4.2.2 Analytical Bound

Partition the final matrix $\boldsymbol{R}$ in two ways

$$\begin{array}{c} \\ k \\ m-k \end{array}\begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix} \end{array} = \boldsymbol{R}, \qquad \begin{array}{c} \\ k+1 \\ m-k-1 \end{array}\begin{array}{c} \begin{array}{cc} k+1 & n-k-1 \end{array} \\ \begin{bmatrix} \hat{\boldsymbol{R}}_{11} & \hat{\boldsymbol{R}}_{12} \\ \boldsymbol{0} & \hat{\boldsymbol{R}}_{22} \end{bmatrix} \end{array} = \hat{\boldsymbol{R}},$$

where $\hat{\boldsymbol{R}}$ is an upper triangular matrix of order $k+1$ and $\boldsymbol{R}$ is an upper triangular matrix of order k.

Theorem 4.2.1. *Apply Hybrid-I with $k+1$ on the given matrix. When the algorithm halts, it guarantees*

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_{\max}(\boldsymbol{R}_{22})}{\sqrt{(k+1)(n-k)}},$$

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{(k+1)(n-k)}.$$

Proof. At the time the algorithm halts, QR with column pivoting-I does not perform any permutations in $\boldsymbol{R}_{22}$ and QR with column pivoting-II does not perform any permutations in $\hat{\boldsymbol{R}}_{11}$. By Inequality (4-1),

$$|r_{k+1,k+1}| \geq \frac{\sigma_{\max}(\boldsymbol{R}_{22})}{\sqrt{n-k}}. \tag{4-9}$$

By inequality (4-3),

$$|r_{k+1,k+1}| \leq \sigma_{\min}(\hat{\boldsymbol{R}}_{11})\sqrt{k+1} \leq \sigma_{\min}(\boldsymbol{R}_{11})\sqrt{k+1}, \tag{4-10}$$

where the last inequality comes from the fact that $\boldsymbol{R}_{11}$ is a submatrix of $\hat{\boldsymbol{R}}_{11}$. Combining Inequality (4-9) and (4-10) gives the first bound

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_{\max}(\boldsymbol{R}_{22})}{\sqrt{(k+1)(n-k)}}. \tag{4-11}$$

Apply the interlacing property on Inequality (4-10), we obtain

$$|r_{k+1,k+1}| \leq \sigma_{\min}(\hat{\boldsymbol{R}}_{11})\sqrt{k+1} \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{k+1}. \tag{4-12}$$

Combining Inequality (4-9) and (4-12) gives the second bound

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{(k+1)(n-k)}. \tag{4-13}$$

□

To prove that Hybrid-II does halt, we cannot reduce it to the proof for Hybrid-I, because we cannot guarantee $\sigma_{k+1}(\boldsymbol{M}) > 0$). However, we can use a similar idea to complete the halting argument.

Theorem 4.2.2. *Algorithm Hybrid-II does halt given* $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ *and* $1 \leq k \leq n$ *such that* $\sigma_k(\boldsymbol{M}) > 0$.

Proof. The proof is similar as for Hybrid-I. To show that the algorithm does halt, we only need to show that $|\det(\boldsymbol{A})|$ is strictly increasing during the algorithm.

Notice, QR with column pivoting-I won't change $\det(\boldsymbol{A})$ because it does not change any elements in $\boldsymbol{A}$. So we just need to focus on QR with column pivoting-II. If no permutation is performed by it, then the algorithm halts at the next loop. The proof is done.

Now suppose QR with column pivoting-II performs permutation. QR with column pivoting-II wll permute the columns of $\hat{\boldsymbol{A}}$ so that the last row of $\hat{\boldsymbol{A}}^{-1}$ has largest 2-norm among all rows. We split this procedure into two steps.

In the first step, we permute and retriangularize columns of $\boldsymbol{A}$ so that the last row of $\boldsymbol{A}^{-1}$ has largest 2-norm among all rows. This procedure won't affect the value of $\det(\boldsymbol{A})$ because it only involves permutation and orthogonal transformation.

After we update $\boldsymbol{A}$ (thus $\hat{\boldsymbol{A}}$ is also updated). Suppose the following is the submatrix

formed by the $k, k+1$th row and $k, k+1$th column of $\hat{\boldsymbol{A}}$,

$$\begin{array}{c} \\ k \\ k+1 \end{array} \begin{array}{c} \begin{array}{cc} k & k+1 \end{array} \\ \begin{bmatrix} \gamma & \beta \\ 0 & \alpha \end{bmatrix} \end{array}.$$

Since $\hat{\boldsymbol{A}}$ is upper triangular, the submatrix formed by the $k, k+1$row and $k, k+1$th column of $\hat{\boldsymbol{A}}^{-1}$ can be represented as

$$\begin{array}{c} \\ k \\ k+1 \end{array} \begin{array}{c} \begin{array}{cc} k & k+1 \end{array} \\ \begin{bmatrix} 1/\gamma & -\beta/\gamma\alpha \\ 0 & 1/\alpha \end{bmatrix} \end{array}.$$

In the second step, we permute column k with $k+1$. Since the permutation is performed only if the 2-norm of the kth row of the inverse is larger than the 2-norm of the $k+1$th row of the inverse, we have

$$\frac{1}{\gamma^2} + \frac{\beta^2}{\gamma^2\alpha^2} > \frac{1}{\alpha^2}.$$

This implies $\gamma^2 < \alpha^2 + \beta^2$. After exchange column k with column $k+1$ and re-triangularize the matrix by Givens rotation, the value of the kth diagonal element is $\sqrt{\alpha^2+\beta^2}$, which is strictly larger than the previous γ. Since all other elements in $\boldsymbol{A}$ does not change after retriangularization and $\boldsymbol{A}$ is an upper triangular matrix, $|\det(\boldsymbol{A})|$ strictly increases at each step. This completes the halting argument. □

4.3 Hybrid Algorithm for Problem Type-III

4.3.1 Algorithm Hybrid-III

Combining algorithm Hybrid-I and Hybrid-II gives algorithm Hybrid-III, which guarantees to solve Problem-III. Algorithm Hybrid-III make use of Lemma 3.1.2 and alternate between Hybrid-I and Hybrid-II until no permutations are performed. The idea is motivated by the fact that the determinant of the upper left $k \times k$ block of $\boldsymbol{R}$ is strictly increasing in both Hybrid-I and Hybrid-II. The pseudocode is presented in Algorithm

4.3.

Algorithm 4.3 RRQR Hybrid-III

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ with $m \geq n$ and k.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

```
procedure HYBRID-III(M,k)
    QR = MΠ                          ▷ Initialize by QR with column pivoting (M, k);
    permuted = true;
    while permuted do
        permuted = false;
        Q̄R̄ = RΠ̄                          ▷ Apply Hybrid-I (R,k);
        if Π is not the identity matrix then
            permuted = true;
            Π = ΠΠ̄;
            Q = QQ̄;
            R = R̄;
        end if
        Q̄R̄ = RΠ̄                          ▷ Apply Hybrid-II (R,k);
        if Π is not the identity matrix then
            permuted = true;
            Π = ΠΠ̄;
            Q = QQ̄;
            R = R̄;
        end if
    end while
end procedure
```

4.3.2 Analytical Bound

Theorem 4.3.1. *When Algorithm Hybrid-III halts, it guarantees*

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{k(n-k+1)}},$$

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{(k+1)(n-k)}.$$

Proof. When Hybrid-III halts, both Hybrid-I and Hybrid-II do not perform any permutations. So the bounds for Hybrid-I and Hybrid-II must hold simultaneously and this completes the proof. □

Theorem 4.3.2. *Algorithm Hybrid-III does halt given* $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ *and* $1 \leq k \leq n$ *such*

that $\sigma_k(\boldsymbol{M}) > 0$.

Proof. As shown in the halting argument of algorithm Hybrid-I and Hybrid-II, $|\det(\boldsymbol{A})|$, the magnitude of the determinant of the upper left $k \times k$ portion of $\boldsymbol{R}$, is strictly increasing during the procedure. Since there is finitely many possible values for $|\det(\boldsymbol{A})|$, so the algorithm will eventually halt. □

Chapter 5 Strong Rank Revealing QR Factorization

This chapter is organized as follows: Section 5.1 discusses the shortcoming of RRQR factorization and then gives the definition of strong RRQR factorization. Section 5.2 first shows algorithm SRRQR-1, which computes a strong RRQR factorization. Then it presents a more efficient algorithm SRRQR-2 which is an variation of SRRQR-1. Based on algorithm SRRQR-1, I proposed a greedy strong RRQR (GSRRQR) algorithm in Section 5.2.3. Combined with the greedy strategy, algorithm GSRRQR-1 and GSRRQR-2 run much faster than algorithm SRRQR-1 and SRRQR-2 for most large scale matrices. In subsection 5.2.4, I discuss a general method to reveal the numerical rank k and present an algorithm SRRQR-3. Section 5.3 shows some important implementation techniques that make these hybrid and strong RRQR algorithms efficient in practice.

5.1 Background

In Chapter 3 and 4, we discussed algorithms for solving RRQR factorization,

$$\boldsymbol{M\Pi} = \boldsymbol{QR} = \boldsymbol{Q}\begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix},$$

where $\boldsymbol{R}_{11}$ is an upper triangular matrix of order k. Further, we have algorithm Hybrid-III that guarantees

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{\sqrt{k(n-k+1)}},$$

$$\sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M})\sqrt{(k+1)(n-k)}.$$

When doing numerical experiments, we can measure the goodness of the RRQR factorization by measuring the ratio $\sigma_k(\boldsymbol{M})/\sigma_{\min}(\boldsymbol{R}_{11})$ and $\sigma_{\max}(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$. If both two ratios are sufficiently small, then we regard the resulting factorization as a

good result. However, in practice, when we want to apply RRQR factorization in some practical problems like null space determination, least squares and subsets selection, since the matrix is of large scale, we don't need to and also don't want to compute the singular values. Then, how good will it perform in these applications?

Suppose we want to estimate the right null space of $\boldsymbol{M}$. We hope that

$$\boldsymbol{\Pi}\begin{bmatrix} -\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12} \\ \boldsymbol{I}_{n-k} \end{bmatrix}$$

will be a good approximation because

$$\begin{aligned} \left\| \boldsymbol{M}\boldsymbol{\Pi}\begin{bmatrix} -\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12} \\ \boldsymbol{I}_{n-k} \end{bmatrix} \right\|_2 &= \left\| \boldsymbol{Q}\begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix}\begin{bmatrix} -\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12} \\ \boldsymbol{I}_{n-k} \end{bmatrix} \right\|_2 \\ &= \left\| \boldsymbol{Q}\begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{R}_{22} \end{bmatrix} \right\|_2 \\ &= \|\boldsymbol{R}_{22}\|_2 = \sigma_{\max}(\boldsymbol{R}_{22}). \end{aligned}$$

However, this approximation is not guaranteed to be stable if we use an RRQR algorithm because it has the potential probability for the elements of $\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}$ to be very large. To overcome this shortcut, Gu and Eisenstat ([2], 1996) proposed the idea of *strong RRQR factorization*, which is defined as following.

Definition 5.1.1 (Strong RRQR)**.** Given a matrix $\boldsymbol{M} \in \boldsymbol{R}^{m\times n}$ with $m \geq n$. Let $\boldsymbol{M}\boldsymbol{\Pi} = \boldsymbol{Q}\boldsymbol{R}$ be the QR factorization of $\boldsymbol{M}$ with its columns permuted according to the permutation matrix $\boldsymbol{\Pi}$. Partition $\boldsymbol{R}$ as

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix},$$

where $\boldsymbol{R}_{11}$ is an upper triangular matrix with order k. Strong RRQR factorization aims

to choose $\mathbf{\Pi}$ such that

$$\sigma_{\min}(\boldsymbol{R}_{11}) \geq \frac{\sigma_k(\boldsymbol{M})}{q_1(k,n)} \quad \text{and} \quad \sigma_{\max}(\boldsymbol{R}_{22}) \leq \sigma_{k+1}(\boldsymbol{M}) q_2(k,n), \tag{5-1}$$

and

$$\left|(\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12})_{i,j}\right| \leq q_3(k,n), \tag{5-2}$$

for $1 \leq i \leq k, 1 \leq j \leq n-k$, where $q_1(k,n)$, $q_2(k,n)$ and $q_3(k,n)$ are functions bounded by some low-degree polynomials in k and n.

5.2 Strong RRQR Factorization

5.2.1 Algorithm SRRQR-1

For simplicity, let's first define some notations. Let $\mathbf{\Pi}_{i,j}$ denotes a permutation matrix that permutes the ith and jth columns of a given matrix. Suppose

$$\boldsymbol{M} = \boldsymbol{Q}\boldsymbol{R}_k = \boldsymbol{Q}\begin{bmatrix} \boldsymbol{A}_k & \boldsymbol{B}_k \\ \boldsymbol{0} & \boldsymbol{C}_k \end{bmatrix},$$

is the partial QR factorization of $\boldsymbol{M} \in \boldsymbol{R}^{m\times n}$ such that $\boldsymbol{A}_k$ is an $k \times k$ upper triangular matrix. Define matrix operators $\mathcal{A}_k, \mathcal{C}_k, \mathcal{R}_k$ by

$$\mathcal{A}_k(\boldsymbol{M}) = \boldsymbol{A}_k, \quad \mathcal{C}_k(\boldsymbol{M}) = \boldsymbol{C}_k, \quad \mathcal{R}_k(\boldsymbol{M}) = \boldsymbol{R}_k.$$

Assume $\boldsymbol{A}_k$ is nonsingular and let $\omega_i(\boldsymbol{A}_k)$ denote the reciprocal of the 2-norm of the ith row of $\boldsymbol{A}_k^{-1}$, for $1 \leq i \leq k$. Let $\gamma_i(\boldsymbol{C}_k)$ denote the 2-norm of the jth column of $\boldsymbol{C}$, for $1 \leq j \leq n-k$. Write $\omega(\boldsymbol{A}_k) = [\omega_1(\boldsymbol{A}_k), \ldots, \omega_k(\boldsymbol{A}_k)]^T$, and $\gamma(\boldsymbol{C}_k) = [\gamma_1(\boldsymbol{C}_k), \ldots, \gamma_{n-k}(\boldsymbol{C}_k)]^T$.

A strong RRQR factorization guarantees that the smallest singular value of $\boldsymbol{R}_{11}$ is sufficiently large, the largest singular value of $\boldsymbol{R}_{22}$ is sufficiently small and every element of $\boldsymbol{A}_k^{-1}\boldsymbol{B}_k$ is bounded. One may ask, does there exist such factorization for any matrices with any numerical deficiency? In this subsection, I first present an algorithm,

SRRQR-1 and then show that it guarantees a strong RRQR factorization.

The main idea of the algorithm SRRQR-1 is motivated by the previous hybrid algorithms. Notice

$$\det(\boldsymbol{A}_k) = \prod_{i=1}^{k} \sigma_i(\boldsymbol{A}_k) = \frac{\sqrt{\det(\boldsymbol{M}^T\boldsymbol{M})}}{\prod_{i=1}^{n-k} \sigma_i(\boldsymbol{C}_k)}.$$

We see maximizing $\sigma_{\min}(\boldsymbol{A}_k)$ and minimizing $\sigma_{\max}(\boldsymbol{C}_k)$ will lead to a large $\det(\boldsymbol{A}_k)$. Thus algorithm SRRQR-1 focus on the determinant of $\boldsymbol{A}_k$ at each step and tries to permute columns so that $\det(\boldsymbol{A}_k)$ is maximized. However, finding the maximum $\det(\boldsymbol{A}_k)$ may cost combinatorial number of operations. So in practice, we set another parameter $f \geq 1$ and permute columns if and only if the new determinant of the upper left portion is f times larger than the previous one. The pseudocode is shown in Algorithm 5.1.

Algorithm 5.1 SRRQR-1

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, k and $f \geq 1$.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M}\boldsymbol{\Pi} = \boldsymbol{Q}\boldsymbol{R}$ reveals the rank deficiency of $\boldsymbol{M}$.

```
procedure SRRQR-1(M,k,f)
    QR factor M up to the kth column, QR = M;
    Π = I;
    while these exist i and j such that det(A(RΠ_{i,j+k}))/ det(A(R)) > f do
        Find such i, j;
        Permute column i and j + k of R and update Π;
        Retriangularize the first k columns of R and update Q;
    end while
end procedure
```

Theorem 5.2.1. *Algorithm SRRQR-1 does halt given $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, k and $f \geq 1$.*

Proof. Since there are only finitely many permutations, there are finitely many possible $\det(\boldsymbol{A}_k)$. The loop condition of SRRQR-1 ensures $\det(\boldsymbol{A}_k)$ is strictly increasing at each loop. So the algorithm does halt at some time. □

Now, we show that Algorithm SRRQR-1 does compute a strong RRQR factorization.

Lemma 5.2.1. *Assume $\boldsymbol{A}_k = \mathcal{A}(\boldsymbol{R}_k)$ has positive diagonal elements (Householder reflections and Givens rotations give the flexibility to choose the sign). Let $\bar{\boldsymbol{A}}_k =$*

$\mathcal{A}(\boldsymbol{R}_k\boldsymbol{\Pi}_{i,k+j})$, *where* $i \le k$ *and* $j > 1$. *Then*

$$\frac{\det(\bar{\boldsymbol{A}}_k)}{\det(\boldsymbol{A}_k)} = \sqrt{(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}^2 + \left(\frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)}\right)^2}. \tag{5-3}$$

Proof. Let $\boldsymbol{A}_k\boldsymbol{\Pi}_{i,k} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{A}}_k$ be the QR factorization of $\boldsymbol{A}_k\boldsymbol{\Pi}_{i,k}$ such that $\tilde{\boldsymbol{A}}_k$ has positive diagonal elements. Now define $\tilde{\boldsymbol{B}}_k = \tilde{\boldsymbol{Q}}^T\boldsymbol{B}_k\boldsymbol{\Pi}_{1,j}$, $\tilde{\boldsymbol{C}}_k = \boldsymbol{C}_k\boldsymbol{\Pi}_{1,j}$ and

$$\tilde{\boldsymbol{\Pi}} = \begin{bmatrix} \boldsymbol{\Pi}_{i,k} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\Pi}_{1,j} \end{bmatrix}.$$

We have

$$\boldsymbol{R}\tilde{\boldsymbol{\Pi}} = \begin{bmatrix} \boldsymbol{A}_k\boldsymbol{\Pi}_{i,k} & \boldsymbol{B}_k\boldsymbol{\Pi}_{1,j} \\ \boldsymbol{0} & \boldsymbol{C}_k\boldsymbol{\Pi}_{1,j} \end{bmatrix} = \begin{bmatrix} \tilde{\boldsymbol{Q}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_{m-k} \end{bmatrix} \begin{bmatrix} \tilde{\boldsymbol{A}}_k & \tilde{\boldsymbol{B}}_k \\ \boldsymbol{0} & \tilde{\boldsymbol{C}}_k \end{bmatrix}$$

being a partial QR factorization of $\boldsymbol{R}\tilde{\boldsymbol{\Pi}}$. Since we ensure $\boldsymbol{A}_k$ and $\tilde{\boldsymbol{A}}_k$ have positive diagonal elements, so they have the same determinant. Since $\tilde{\boldsymbol{A}}_k^{-1} = \boldsymbol{\Pi}_{i,k}^T\boldsymbol{A}_k^{-1}\tilde{\boldsymbol{Q}}$, we have

$$\tilde{\boldsymbol{A}}_k^{-1}\tilde{\boldsymbol{B}}_k = (\boldsymbol{\Pi}_{i,k}^T\boldsymbol{A}_k^{-1}\tilde{\boldsymbol{Q}})(\tilde{\boldsymbol{Q}}^T\boldsymbol{B}_k\boldsymbol{\Pi}_{1,j}) = \boldsymbol{\Pi}_{i,k}^T\boldsymbol{A}_k^{-1}\boldsymbol{B}_k\boldsymbol{\Pi}_{1,j}.$$

This means $(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j} = (\tilde{\boldsymbol{A}}_k^{-1}\tilde{\boldsymbol{B}}_k)_{k,1}$. Since right multiplication by an orthogonal matrix does not change the 2-norm of each row and $\tilde{\boldsymbol{A}}_k^{-1} = \boldsymbol{\Pi}_{i,k}^T\boldsymbol{A}_k^{-1}\tilde{\boldsymbol{Q}}$, we have $\omega_i(\boldsymbol{A}_k) = \omega_k(\tilde{\boldsymbol{A}}_k)$. By definition of $\tilde{\boldsymbol{C}}$, we have $\gamma_j(\boldsymbol{C}_k) = \gamma_1(\tilde{\boldsymbol{C}}_k)$.

Now we can assume $i = k$ and $j = 1$, otherwise we can reduce the problem by retriangularizing $\boldsymbol{R}\tilde{\boldsymbol{\Pi}}$. Suppose

$$\mathcal{R}_{k+1}(\boldsymbol{R}) = \begin{bmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{b}_1 & \boldsymbol{b}_2 & \boldsymbol{B} \\ \boldsymbol{0} & \gamma_1 & \beta & \boldsymbol{c}_1^T \\ \boldsymbol{0} & 0 & \gamma_2 & \boldsymbol{c}_2^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{C}_{k+1} \end{bmatrix}.$$

Then

$$\boldsymbol{A}_k^{-1} = \begin{bmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{b}_1 \\ \boldsymbol{0} & \gamma_1 \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{A}_{k-1}^{-1} & -\boldsymbol{A}_{k-1}^{-1}\boldsymbol{b}_1\gamma_1^{-1} \\ \boldsymbol{0} & \gamma_1^{-1} \end{bmatrix}.$$

Notice that we assume $i = k$ and $j = 1$. So $\omega_i(\boldsymbol{A}_k) = \gamma_1$, $\gamma_j(\boldsymbol{C}_k) = \gamma_2$ and $(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j} = \beta/\gamma_1$. Also since the upper left $k+1$ portion is upper triangular, we have

$$\det(\boldsymbol{A}_k) = \gamma_1 \cdot \det(\boldsymbol{A}_{k-1}).$$

After permute column i and column j and retriangularize it by Givens rotation, the k-kth element becomes $\sqrt{\beta^2+\gamma_2^2}$ and all other elements of $\bar{\boldsymbol{A}}_k$ remain unchanged. So we have

$$\det(\bar{\boldsymbol{A}}_k) = \sqrt{\beta^2+\gamma_2^2} \cdot \det(\boldsymbol{A}_{k-1}).$$

Combine it with the formula of $\det(\boldsymbol{A})k)$. We obtain

$$\frac{\det(\bar{\boldsymbol{A}}_k)}{\det(\boldsymbol{A}_k)} = \sqrt{\left(\frac{\beta}{\gamma_1}\right)^2 + \left(\frac{\gamma_2}{\gamma_1}\right)^2} = \sqrt{(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}^2 + \left(\frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)}\right)^2}.$$

□

Theorem 5.2.2. *When SRRQR-1 halts, it guarantees*

$$\left|(\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12})_{i,j}\right| \le f, \quad \textit{for } 1 \le i \le k, 1 \le j \le n-k.$$

Proof. By Lemma 5.2.1, at the time SRRQR-1 halts, we have

$$\frac{\det(\bar{\boldsymbol{A}}_k)}{\det(\boldsymbol{A}_k)} = \sqrt{(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}^2 + \left(\frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)}\right)^2} < f,$$

for $1 \le i \le k, 1 \le j \le n-k$. By definition, the final $\boldsymbol{A}_k$, $\boldsymbol{B}_k$ are defined as $\boldsymbol{R}_{11}$, $\boldsymbol{R}_{12}$. This completes the proof of the theorem. □

For simplicity, we define

$$\rho(\boldsymbol{R}, k) = \max_{1\le i\le k, 1\le j\le n-k} \sqrt{(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}^2 + \left(\frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)}\right)^2} \tag{5-4}$$

By Theorem 5.2.2, we see algorithm SRRQR-1 provides a bound on all elements of $\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}$. An interesting question is, though the determinant of $\boldsymbol{A}_k$ also strictly in-

creases during those hybrid algorithms we've discussed in Chapter 4, why hybrid algorithms cannot provide a similar bound like algorithm SRRQR-1? Recall that, in those hybrid algorithms, we exchange column k with the best column among all columns of $\boldsymbol{C}_{k-1}$ and exchange column k with the worst column among all columns of $\boldsymbol{A}_k$. The goodness of these column are decided by γ and ω. Instead of considering all pairs of i, j, algorithm Hybrid-I only consider $j = \arg\max_{1\le j\le n-k+1} \gamma_j(\boldsymbol{C}_{k-1})$ and $i = \arg\max_{1\le i\le k} \omega_i(\boldsymbol{A}_k)$. So it cannot guarantee all elements of $\boldsymbol{A}_k^{-1}\boldsymbol{B}_k$ being small.

Theorem 5.2.3. *Given $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ and apply SRRQR-1. Partition the final matrix $\boldsymbol{R}$*

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix} = \mathcal{R}_k(\boldsymbol{M}\boldsymbol{\Pi}).$$

If $\boldsymbol{R}$ satisfy $\rho(\boldsymbol{R}, k) \le f$, then

$$\begin{aligned} \sigma_i(\boldsymbol{R}_{11}) &\ge \frac{\sigma_i(\boldsymbol{M})}{\sqrt{1+f^2k(n-k)}}, && \text{for } 1 \le i \le k \\ \sigma_j(\boldsymbol{R}_{22}) &\le \sigma_{j+k}(\boldsymbol{M})\sqrt{1+f^2k(n-k)}, && \text{for } 1 \le j \le n-k. \end{aligned} \tag{5-5}$$

Proof. For simplicity, assume $\boldsymbol{M}$ has full column rank (actually we just need $\sigma_{\min}(\boldsymbol{R}_{11})$ and $\sigma_{\max}(\boldsymbol{R}_{22})$ are positive). Let $\alpha = \sigma_{\max}(\boldsymbol{R}_{22})/\sigma_{\min}(\boldsymbol{R}_{11})$. Rewrite $\boldsymbol{R}$ as a product of two matrices,

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{R}_{22}/\alpha \end{bmatrix} \begin{bmatrix} \boldsymbol{I}_k & \boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12} \\ \boldsymbol{0} & \alpha\boldsymbol{I}_{n-k} \end{bmatrix} \triangleq \tilde{\boldsymbol{R}}\tilde{\boldsymbol{W}}. \tag{5-6}$$

Then we have

$$\sigma_i(\boldsymbol{R}) \le \sigma_i(\tilde{\boldsymbol{R}})\|\tilde{\boldsymbol{W}}\|_2, \quad \text{for } 1 \le i \le n. \tag{5-7}$$

The definition of α implies $\sigma_{\min}(\boldsymbol{R}_{11}) = \sigma_{\max}(\boldsymbol{R}_{22}/\alpha)$. This means the k most dominant singular values of $\tilde{\boldsymbol{R}}$ are exactly the same as the singular values of $\boldsymbol{R}_{11}$,

$$\sigma_i(\boldsymbol{R}_{11}) = \sigma_i(\tilde{\boldsymbol{R}}), \quad \text{for } 1 \le i \le k. \tag{5-8}$$

Also $\boldsymbol{R}$ and $\boldsymbol{M}$ share the same singular values. For $\|\tilde{\boldsymbol{W}}\|_2$, we have,

$$\begin{aligned}\|\tilde{\boldsymbol{W}}\|_2^2 &\leq 1 + \|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\|_2^2 + \alpha^2 \\ &= 1 + \|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\|_2^2 + \|\boldsymbol{R}_{22}\|_2^2\|\boldsymbol{R}_{11}^{-1}\|_2^2\end{aligned} \tag{5-9}$$

We can bound the 2-norm of a matrix by its Frobenius norm,

$$\begin{aligned}\|\tilde{\boldsymbol{W}}\|_2^2 &\leq 1 + \|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\|_F^2 + \|\boldsymbol{R}_{22}\|_2^2\|\boldsymbol{R}_{11}^{-1}\|_F^2 \\ &= 1 + \sum_{i=1}^{k}\sum_{j=1}^{n-k}\left((\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12})_{i,j}^2 + \frac{\gamma_j(\boldsymbol{R}_{22})^2}{\omega_i(\boldsymbol{R}_{11})^2}\right) \\ &\leq 1 + f^2k(n-k)\end{aligned} \tag{5-10}$$

Combining Inequality (5-7), (5-8) and (5-10) gives the first bound

$$\sigma_i(\boldsymbol{R}_{11}) \geq \frac{\sigma_i(\boldsymbol{M})}{\sqrt{1 + f^2k(n-k)}}, \quad \text{for } 1 \leq i \leq k.$$

For the second bound, we define $\hat{\boldsymbol{R}}$ and $\hat{\boldsymbol{W}}$ by

$$\hat{\boldsymbol{R}} \triangleq \begin{bmatrix}\alpha\boldsymbol{R}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{R}_{22}\end{bmatrix} = \begin{bmatrix}\boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22}\end{bmatrix}\begin{bmatrix}\alpha\boldsymbol{I}_k & -\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{I}_{n-k}\end{bmatrix} \triangleq \boldsymbol{R}\hat{\boldsymbol{W}}.$$

Then we have

$$\sigma_i(\hat{\boldsymbol{R}}) \leq \sigma_i(\boldsymbol{R})\|\hat{\boldsymbol{W}}\|_2 = \sigma_i(\boldsymbol{M})\|\hat{\boldsymbol{W}}\|_2, \quad \text{for } 1 \leq i \leq n. \tag{5-11}$$

Since $\sigma_{\min}(\alpha\boldsymbol{R}_{11}) = \sigma_{\max}(\boldsymbol{R}_{22})$, we have

$$\sigma_i(\boldsymbol{R}_{22}) = \sigma_{i+k}(\hat{\boldsymbol{R}}), \quad \text{for } 1 \leq i \leq n-k. \tag{5-12}$$

From Inequality (5-9) and (5-10), we see $\hat{\boldsymbol{W}}$ also satisfies

$$\|\hat{\boldsymbol{W}}\|_2 \leq \sqrt{1 + f^2k(n-k)}. \tag{5-13}$$

Combining Inequality (5-11), (5-12) and (5-13) gives the second bound

$$\sigma_j(\boldsymbol{R}_{22}) \leq \sigma_{j+k}(\boldsymbol{M})\sqrt{1+f^2k(n-k)}, \quad \text{for } 1 \leq j \leq n-k.$$

□

Theorem 5.2.2 and Theorem 5.2.3 together show that SRRQR-1 does compute a strong RRQR factorization. Since the halting argument requires $f \geq 1$, so there exist a strong RRQR factorization under

$$q_1(k,n) = q_2(k,n) = \sqrt{1+k(n-k)} \quad \text{and} \quad q_3(k,n) = 1.$$

However, setting $f = 1$ will guarantee that $|\det \boldsymbol{R}_{11}|$ reaches the exact maximum. This may lead to combinatorial operations.

5.2.2 Algorithm SRRQR-2

In practice, instead of computing $\rho(\boldsymbol{R}, k)$, we compute

$$\hat{\rho}(\boldsymbol{R}, k) = \max_{1\leq i\leq k, 1\leq j\leq n-k} \max\left\{ \left|(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}\right|, \frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)} \right\}. \tag{5-14}$$

This gives algorithm SRRQR-2. The pseudocode is shown in Algorithm 5.2.

Algorithm 5.2 SRRQR-2

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, k and $f \geq 1$.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

```
procedure SRRQR-2(M,k,f)
    QR factor M up to the kth column, QR = M;
    Π = I;
    while ρ̂(R, k) > f do
        Find i, j such that |(A_k^{-1} B_k)_{i,j}| > f or γ_j(C_k)/ω_i(A_k) > f;
        Permute column i and j + k of R and update Π;
        Retriangularize the first k columns of R and update Q;
    end while
end procedure
```

When SRRQR-2 halts, we have $\hat{\rho}(\boldsymbol{R}, k) \leq f$, which implies $\rho(\boldsymbol{R}, k) \leq \sqrt{2}f$. Then SRRQR-2 computes a strong RRQR factorization under

$$q_1(k,n) = q_2(k,n) = \sqrt{1 + 2f^2k(n-k)} \quad \text{and} \quad q_3(k,n) = \sqrt{2}f.$$

5.2.3 Algorithm GSRRQR

Gu ([2], 1996) pointed out that we can speed up algorithm SRRQR-1 and SRRQR-2 by initializing the origin matrix by algorithm QR with column pivoting. This strategy does improve the time efficiency. However, by embedding the greedy strategy into the strong RRQR algorithms, we can speed up the algorithm further.

In this subsection, I present greedy strong RRQR (GSRRQR) algorithms, GSRRQR-1 and GSRRQR-2. GSRRQR algorithms try to improve our selection of the columns that are going to be permuted by the idea of QR with column pivoting.

Notice that in algorithm SRRQR-1, we select any i-j pair such that

$$\sqrt{(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}^2 + \left(\frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)}\right)^2} > f. \tag{5-15}$$

Though any i-j pair satisfying Inequality 5-15 will increase $|\det(\boldsymbol{A}_k)|$ at least f times after permuting column i with column $j+k$, we still want to choose a better i-j pair such that they can increase $|\det(\boldsymbol{A}_k)|$ much more. Choosing the best i-j pair is computational expensive, but we can approximate it through greedy strategy.

Recall that algorithm QR with column pivoting-I inserts the column, which has the largest column two-norm among all columns of $\boldsymbol{C}_k$, into $\boldsymbol{A}_k$. Algorithm QR with column pivoting-II removes the column, which has the largest column two-norm among all columns of $(\boldsymbol{A}_k^{-1})^T$, from $\boldsymbol{A}_k$. Inspired by this, algorithm GSRRQR-1 first consider $i = \arg\max_{1\leq l\leq n-k} \gamma_l(\boldsymbol{C}_k)$ and $j = \arg\min_{1\leq l\leq k} \omega_l(\boldsymbol{A}_k)$ at each cycle. If such choice of i and j cannot ensure Inequality (5-15), we then pick an arbitrary i-j pair that satisfying Inequality (5-15) so that GSRRQR-1 still guarantees a strong RRQR factorization. The pseudocode is presented in Algorithm 5.3.

Computing the maximum of $\{\gamma_1, ..., \gamma_{n-k}\}$ and the minimum of $\{\omega_1, ..., \omega_k\}$ re-

Algorithm 5.3 GSRRQR-1

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, k and $f \geq 1$.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.

```
procedure GSRRQR-1(M,k,f)
    QR factor M up to the kth column, QR = M;
    Π = I;
    while ρ(R, k) > f do
        if max_{1≤l≤n−k} γ_l(C_k) / min_{1≤l≤k} ω_l(A_k) > f then
            Set i = arg max_{1≤l≤n−k} γ_l(C_k), j = arg min_{1≤l≤k} ω_l(A_k);
        else
            Find i, j such that sqrt((A_k^{-1} B_k)_{i,j}^2 + (γ_j(C_k)/ω_i(A_k))^2) > f;
        end if
        Permute column i and j + k of R and update Π;
        Retriangularize the first k columns of R and update Q;
    end while
end procedure
```

quires $\mathcal{O}(\max(n-k,k))$ operations. The leading order of the time complexity remains unchanged since computing $\rho(\boldsymbol{R},k)$ requires $\mathcal{O}((n-k)k)$ operations.

At this time, it's hard to say how much iterations does algorithm GSRRQR-1 guarantee to save from algorithm SRRQR-1. Numerical results in Section 6.1 show that GSRRQR-1 is much faster than SRRQR-1 especially for large scale matrices.

We can also relax algorithm GSRRQR-1 by replacing $\rho(\boldsymbol{R},k)$ by $\hat{\rho}(\boldsymbol{R},k)$. This gives algorithm GSRRQR-2 and it is shown in Algorithm 5.4.

5.2.4 Algorithm SRRQR-3

In previous discussion, we assume the algorithm has k pre-given. However, this may not be realistic assumption in some circumstances. In this subsection, I present a general method for revealing the numerical rank k when computing the (strong) RRQR factorization.

Since QR with column pivoting-I is good at approximating the largest singular value, the idea is to run QR with column pivoting-I in the outer loop and run the (strong) RRQR algorithm in the inner loop.

We start at $k = 1$ and increase k by 1 at each execution of the outer loop. The

Algorithm 5.4 GSRRQR-2

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, k and $f \geq 1$.
Output: Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.
procedure GSRRQR-2($\boldsymbol{M}$,k,f)
  QR factor $\boldsymbol{M}$ up to the kth column, $\boldsymbol{QR} = \boldsymbol{M}$;
  $\boldsymbol{\Pi} = \boldsymbol{I}$;
  while $\hat{\rho}(\boldsymbol{R}, k) > f$ **do**
    if $\max_{1\leq l\leq n-k} \gamma_l(\boldsymbol{C}_k)/\min_{1\leq l\leq k} \omega_l(\boldsymbol{A}_k) > f$ **then**
      Set $i = \arg\max_{1\leq l\leq n-k} \gamma_l(\boldsymbol{C}_k)$, $j = \arg\min_{1\leq l\leq k} \omega_l(\boldsymbol{A}_k)$;
    else
      Find i, j such that $\left|(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}\right| > f$ or $\gamma_j(\boldsymbol{C}_k)/\omega_i(\boldsymbol{A}_k) > f$;
    end if
    Permute column i and $j+k$ of $\boldsymbol{R}$ and update $\boldsymbol{\Pi}$;
    Retriangularize the first k columns of $\boldsymbol{R}$ and update $\boldsymbol{Q}$;
  end while
end procedure

outer loop terminate if it detects the largest singular value of the lower right portion is sufficiently small. For example, combining algorithm SRRQR-2 with QR with column pivoting-I gives Algorithm SRRQR-3. The pseudocode is presented in Algorithm 5.5.

Algorithm 5.5 SRRQR-3

Input: $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, δ, $f \geq 1$.
Output: The numerical rank k of $\boldsymbol{M}$; Orthogonal $\boldsymbol{Q} \in \mathbb{R}^{m\times m}$, upper triangular $\boldsymbol{R} \in \mathbb{R}^{m\times n}$ and a permutation matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ such that $\boldsymbol{M\Pi} = \boldsymbol{QR}$ reveals the rank deficiency of $\boldsymbol{M}$.
procedure SRRQR-3($\boldsymbol{M}$,δ, f)
  $k = 0$, $\boldsymbol{R} = \boldsymbol{M}$, $\boldsymbol{\Pi} = \boldsymbol{Q} = \boldsymbol{I}$;
  while $\max_{1\leq i\leq n-k} \gamma_i(\boldsymbol{C}_k) \geq \delta$ **do**
    Find j such that $\gamma_j(\boldsymbol{C}_k) = \max_{1\leq i\leq n-k} \gamma_i(\boldsymbol{C}_k)$;
    $k = k + 1$;
    Permute column k and $k + j - 1$ of $\boldsymbol{R}$ and update $\boldsymbol{\Pi}$;
    Retriangularize the first k columns of $\boldsymbol{R}$ and update $\boldsymbol{Q}$;
    while $\hat{\rho}(\boldsymbol{R}, k) > f$ **do**
      Find i, j such that $\left|(\boldsymbol{A}_k^{-1}\boldsymbol{B}_k)_{i,j}\right| > f$ or $\gamma_j(\boldsymbol{C}_k)/\omega_i(\boldsymbol{A}_k) > f$;
      Permute column i and $j+k$ of $\boldsymbol{R}$ and update $\boldsymbol{\Pi}$;
      Retriangularize the first k columns of $\boldsymbol{R}$ and update $\boldsymbol{Q}$;
    end while
  end while
end procedure

The outer loop will terminate if $\sigma_{\max}(\boldsymbol{C})$ is sufficiently small. Sometimes we also want to detect the gap between the kth and $k+1$st singular values and terminate the loop when the gap is sufficiently large. Measuring the quotient of $\sigma_{k+1}(\boldsymbol{M})$ over $\sigma_k(\boldsymbol{M})$ gives

$$\begin{aligned}\frac{\sigma_{k+1}(\boldsymbol{M})}{\sigma_k(\boldsymbol{M})} &\geq \frac{1}{q_1(k,n)\,q_2(k,n)} \cdot \frac{\sigma_{\max}(\boldsymbol{C}_k)}{\sigma_{\min}(\boldsymbol{A}_k)} \\ &\geq \frac{1}{q_1(k,n)\,q_2(k,n)} \max_{1\leq i\leq k, 1\leq j\leq n-k} \frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)},\end{aligned} \tag{5-16}$$

where the first inequality is guaranteed by SRRQR-II and the second inequality comes from Inequality (4-1) and (4-3). Also, applying the interlacing property gives

$$\begin{aligned}\frac{\sigma_{k+1}(\boldsymbol{M})}{\sigma_k(\boldsymbol{M})} &\leq \frac{\sigma_{\max}(\boldsymbol{C}_k)}{\sigma_{\min}(\boldsymbol{A}_k)} \\ &\leq \sqrt{k(n-k)} \max_{1\leq i\leq k, 1\leq j\leq n-k} \frac{\gamma_j(\boldsymbol{C}_k)}{\omega_i(\boldsymbol{A}_k)}.\end{aligned} \tag{5-17}$$

Inequality (5-16) and (5-17) together shows that $\max_{1\leq i\leq k,1\leq j\leq n-k} \gamma_j(\boldsymbol{C}_k)/\omega_i(\boldsymbol{A}_k)$ is good at approximating the gap between the kth and $k+1$st singular value of $\boldsymbol{M}$. It is sometimes useful to add this stopping criteria into the loop condition of the outer loop.

5.3 Implementation Techniques

Many algorithms we have discussed before involves the computation of $\gamma(\boldsymbol{C}_k)$, $\omega(\boldsymbol{A}_k)$ and $\boldsymbol{A}_k^{-1}\boldsymbol{B}_k$. Obviously, we don't want to compute them directly at each iteration. In this section, I present how to update these information efficiently. Due to these techniques, algorithm QR with column pivoting-I, QR with column pivoting-II, Hybrid-I, Hybrid-II, Hybrid-III, SRRQR-1, SRRQR-2, SRRQR-3 can run very fast in practice.

Since these information are updated at each iteration instead of recomputed, one may care about the stability issue, whether the cumulating error will affect out results. Note that since we only use these information to determine which column to permute, it won't cause stability problem to our factorization. One can recompute these information after they have been updated sufficiently many times.

Subsection 5.3.1 consider how to update these information when we increase k to $k+1$. Subsection 5.3.2 shows how to reduce a general case to a special case when we are going to permute two columns. Finally, Subsection 5.3.3 shows how to modify these information under this special case.

5.3.1 Updating Formula

Partition $\boldsymbol{R}_{k-1}$ and $\boldsymbol{R}_k \triangleq \mathcal{R}(\boldsymbol{R}_{k-1})$ as

$$\boldsymbol{R}_{k-1} = \begin{bmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{B}_{k-1} \\ \boldsymbol{0} & \boldsymbol{C}_{k-1} \end{bmatrix}, \qquad \boldsymbol{R}_k = \begin{bmatrix} \boldsymbol{A}_k & \boldsymbol{B}_k \\ \boldsymbol{0} & \boldsymbol{C}_k \end{bmatrix}.$$

Suppose we already have $\boldsymbol{A}_{k-1}$, $\boldsymbol{B}_{k-1}$, $\boldsymbol{C}_{k-1}$, $\omega(\boldsymbol{A}_{k-1})$, $\gamma(\boldsymbol{C}_{k-1})$ and $\boldsymbol{A}_{k-1}^{-1}\boldsymbol{B}_{k-1}$. Now we want to compute $\omega(\boldsymbol{A}_k)$, $\gamma(\boldsymbol{C}_k)$ and $\boldsymbol{A}_k^{-1}\boldsymbol{B}_k$.

Let $\boldsymbol{H}$ be a Householder matrix of order $m-k+1$ so that it zeroes out all but the first entry of the first column of $\boldsymbol{C}_{k-1}$,

$$\boldsymbol{H}\boldsymbol{C}_{k-1} = \begin{bmatrix} \gamma & \boldsymbol{c}^T \\ \boldsymbol{0} & \boldsymbol{C} \end{bmatrix},$$

where $\gamma = \gamma_1(\boldsymbol{C}_{k-1})$. Partition $\boldsymbol{B}_{k-1}$ as $\boldsymbol{B}_{k-1} = \begin{bmatrix} \boldsymbol{b} & \boldsymbol{B} \end{bmatrix}$. Then

$$\boldsymbol{R}_k = \begin{bmatrix} \boldsymbol{A}_k & \boldsymbol{B}_k \\ \boldsymbol{0} & \boldsymbol{C}_k \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{b} & \boldsymbol{B} \\ \boldsymbol{0} & \gamma & \boldsymbol{c}^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{C} \end{bmatrix}.$$

So we get

$$\boldsymbol{A}_k = \begin{bmatrix} \boldsymbol{A}_{k-1} & b \\ \boldsymbol{0} & \gamma \end{bmatrix}, \quad \boldsymbol{B}_k = \begin{bmatrix} \boldsymbol{B} \\ \boldsymbol{c}^T \end{bmatrix}, \quad \boldsymbol{C}_k = \boldsymbol{C}.$$

Suppose $\boldsymbol{A}_{k-1}^{-1}\boldsymbol{B}_{k-1} = \begin{bmatrix}\boldsymbol{u} & \boldsymbol{U}\end{bmatrix}$, then $\boldsymbol{A}_{k-1}^{-1}\boldsymbol{b} = \boldsymbol{u}$ and $\boldsymbol{A}_{k-1}^{-1}\boldsymbol{B} = \boldsymbol{U}$. Then

$$\boldsymbol{A}_k^{-1} = \begin{bmatrix}\boldsymbol{A}_{k-1}^{-1} & -\boldsymbol{u}/\gamma \\ \boldsymbol{0} & 1/\gamma\end{bmatrix}, \quad \boldsymbol{A}_k^{-1}\boldsymbol{B}_k = \begin{bmatrix}\boldsymbol{U} - \boldsymbol{u}\boldsymbol{c}^T/\gamma \\ \boldsymbol{c}^T/\gamma\end{bmatrix}.$$

Also $\omega(\boldsymbol{A}_k)$ and $\gamma(\boldsymbol{C}_k)$ can be updated according to

$$\omega_k(\boldsymbol{A}_k) = |\gamma|, \quad \text{and} \quad 1/\omega_i(\boldsymbol{A}_k)^2 = 1/\omega_i(\boldsymbol{A}_{k-1})^2 + \boldsymbol{u}_i^2/\gamma^2 \quad \text{for } 1 \le i \le k-1,$$

and

$$\gamma_j(\boldsymbol{C}_k)^2 = \gamma_{j+1}(\boldsymbol{C}_{k-1})^2 - \boldsymbol{c}_j^2, \quad \text{for } 1 \le j \le n-k.$$

5.3.2 Reduction from a General Case to a Special Case

Suppose we are going to permute column i with column $j+k$. In this subsection, I show how to reduce this general case into a special case, that is $i = k$ and $j = 1$.

If $j > 1$, then we exchange the column $k+1$ and column $k+j$ of $\boldsymbol{R}$. After the permutation, $\boldsymbol{A}_k$ and $\omega(\boldsymbol{A}_k)$ remain unchanged and only the corresponding column 1 and j of $\boldsymbol{B}_k$, $\boldsymbol{C}_k$, $\gamma(\boldsymbol{C}_k)$ and $\boldsymbol{A}_k^{-1}\boldsymbol{B}_k$ are permuted. So now we assume $j = 1$ and $i < k$.

Partition $\boldsymbol{A}_k$ as

$$\boldsymbol{A}_k \triangleq \begin{bmatrix}\boldsymbol{A}_{11} & \boldsymbol{a}_1 & \boldsymbol{A}_{12} \\ \boldsymbol{0} & \alpha & \boldsymbol{a}_2^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{A}_{22}\end{bmatrix},$$

where $\boldsymbol{A}_{11}$ and $\boldsymbol{A}_{22}$ are upper triangular of order $i-1$ and $k-i$. Let $\boldsymbol{\Pi}_k$ be a permutation matrix corresponding to left shifting columns $i+1, ..., k-1$ of $\boldsymbol{A}_k$ by one column and moving column i to the kth position. Then we have

$$\boldsymbol{A}_k\boldsymbol{\Pi}_k = \begin{bmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \boldsymbol{a}_1 \\ \boldsymbol{0} & \boldsymbol{a}_2^T & \alpha \\ \boldsymbol{0} & \boldsymbol{A}_{22} & \boldsymbol{0}\end{bmatrix}.$$

Now the matrix $\boldsymbol{A}_k\boldsymbol{\Pi}_k$ is a Hessenberg matrix. Use Givens rotations to zero out the

nonzero elements on the off-diagonal entries. Let $\boldsymbol{Q}_k^T$ be the product of these Givens rotations. Then $\boldsymbol{Q}_k^T \boldsymbol{A}_k \boldsymbol{\Pi}_k$ is upper triangular.

Now we define $\boldsymbol{\Pi}$ as

$$\boldsymbol{\Pi} \triangleq \begin{bmatrix} \boldsymbol{\Pi}_k & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_{n-k} \end{bmatrix}.$$

This gives

$$\boldsymbol{R}\boldsymbol{\Pi} = \begin{bmatrix} \boldsymbol{A}_k \boldsymbol{\Pi}_k & \boldsymbol{B}_k \\ \boldsymbol{0} & \boldsymbol{C}_k \end{bmatrix} \quad \text{and} \quad \mathcal{R}(\boldsymbol{R}\boldsymbol{\Pi}) = \begin{bmatrix} \boldsymbol{Q}_k^T \boldsymbol{A}_k \boldsymbol{\Pi}_k & \boldsymbol{Q}_k^T \boldsymbol{B}_k \\ \boldsymbol{0} & \boldsymbol{C}_k \end{bmatrix}.$$

For simplicity, denote the matrix after permutation by adding bar. Then we have

$$\bar{\boldsymbol{A}}_k = \boldsymbol{Q}_k^T \boldsymbol{A}_k \boldsymbol{\Pi}_k, \quad \bar{\boldsymbol{B}}_k = \boldsymbol{Q}_k^T \boldsymbol{B}_k, \quad \bar{\boldsymbol{C}}_k = \boldsymbol{C}_k.$$

Also since $\bar{\boldsymbol{A}}_k^{-1} = \boldsymbol{\Pi}_k^T \boldsymbol{A}_k^{-1} \boldsymbol{Q}_k$ and right multiplication by an orthogonal matrix does not change the 2-norm of each row, we have

$$\omega(\bar{\boldsymbol{A}}_k) = \boldsymbol{\Pi}_k^T \omega(\boldsymbol{A}), \quad \gamma(\bar{\boldsymbol{C}}_k) = \gamma(\boldsymbol{C}_k), \quad \bar{\boldsymbol{A}}_k^{-1} \bar{\boldsymbol{B}}_k = \boldsymbol{\Pi}_k^T \boldsymbol{A}_k^{-1} \boldsymbol{B}_k.$$

5.3.3 Modifying Formula for a Special Case

In this subsection, we discuss how to modify the formula after exchanging column k with $k+1$ and retriangularization. Partition $\boldsymbol{R}_k$ and $\bar{\boldsymbol{R}}_k \triangleq \mathcal{R}(\boldsymbol{R}_k \boldsymbol{\Pi}_{k,k+1})$ as

$$\boldsymbol{R}_k \triangleq \begin{bmatrix} \boldsymbol{A}_k & \boldsymbol{B}_k \\ \boldsymbol{0} & \boldsymbol{C}_k \end{bmatrix}, \qquad \bar{\boldsymbol{R}}_k \triangleq \begin{bmatrix} \bar{\boldsymbol{A}}_k & \bar{\boldsymbol{B}}_k \\ \boldsymbol{0} & \bar{\boldsymbol{C}}_k \end{bmatrix}.$$

Suppose we already have $\boldsymbol{A}_k$, $\boldsymbol{B}_k$, $\boldsymbol{C}_k$, $\omega(\boldsymbol{A}_k)$, $\gamma(\boldsymbol{C}_k)$ and $\boldsymbol{A}_k^{-1}\boldsymbol{B}_k$, and we want to compute $\omega(\bar{\boldsymbol{A}}_k)$, $\gamma(\bar{\boldsymbol{C}}_k)$ and $\bar{\boldsymbol{A}}_k^{-1}\bar{\boldsymbol{B}}_k$.

Further partition $\boldsymbol{R}_k$ as

$$\boldsymbol{R}_k \triangleq \begin{bmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{b}_1 & \boldsymbol{b}_2 & \boldsymbol{B} \\ \boldsymbol{0} & \gamma & \gamma\mu & \boldsymbol{c}_1^T \\ \boldsymbol{0} & \boldsymbol{0} & \gamma\nu & \boldsymbol{c}_2^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{C}_{k+1} \end{bmatrix}.$$

After permutation, retriagularize $\boldsymbol{R}_k\boldsymbol{\Pi}_{k,k+1}$ by Givens rotation and obtain

$$\bar{\boldsymbol{R}}_k = \begin{bmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{b}_2 & \boldsymbol{b}_1 & \boldsymbol{B} \\ \boldsymbol{0} & \bar{\gamma} & \gamma\mu/\rho & \bar{\boldsymbol{c}}_1^T \\ \boldsymbol{0} & \boldsymbol{0} & \gamma\nu/\rho & \bar{\boldsymbol{c}}_2^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{C}_{k+1} \end{bmatrix},$$

where $\rho = \sqrt{\mu^2+\nu^2}, \bar{\gamma} = \gamma\rho, \bar{\boldsymbol{c}}_1 = (\mu\boldsymbol{c}_1 + \nu\boldsymbol{c}_2)/\rho, \bar{\boldsymbol{c}}_2 = (\nu\boldsymbol{c}_1 + \mu\boldsymbol{c}_2)/\rho$. This gives $\bar{\boldsymbol{A}}_k, \bar{\boldsymbol{B}}_k, \bar{\boldsymbol{C}}_k$. From the partition of $\boldsymbol{R}_k$, we have

$$\boldsymbol{A}_k^{-1} = \begin{bmatrix} \boldsymbol{A}_{k-1}^{-1} & -\boldsymbol{A}_{k-1}^{-1}\boldsymbol{b}_1/\gamma \\ \boldsymbol{0} & 1/\gamma \end{bmatrix}.$$

Define $\boldsymbol{u} = \boldsymbol{A}_{k-1}^{-1}\boldsymbol{b}_1$. Define $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{U}$ by

$$\boldsymbol{A}_k^{-1}\boldsymbol{B}_k = \begin{bmatrix} \boldsymbol{A}_{k-1}^{-1} & -\boldsymbol{u}/\gamma \\ \boldsymbol{0} & 1/\gamma \end{bmatrix} \begin{bmatrix} \boldsymbol{b}_2 & \boldsymbol{B} \\ \gamma\mu & \boldsymbol{c}_1^T \end{bmatrix} \triangleq \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{U} \\ \boldsymbol{\mu} & \boldsymbol{u}_2^T \end{bmatrix}.$$

Then we have

$$\boldsymbol{A}_{k-1}^{-1}\boldsymbol{b}_2 = \boldsymbol{u}_1 + \mu\boldsymbol{u}, \quad \boldsymbol{A}_{k-1}^{-1}\boldsymbol{B} = \boldsymbol{U} + \boldsymbol{u}\boldsymbol{c}_1^T/\gamma.$$

By the partition of $\bar{\boldsymbol{R}}_k$, we have

$$\bar{\boldsymbol{A}}_k^{-1} = \begin{bmatrix} \boldsymbol{A}_{k-1}^{-1} & -\boldsymbol{A}_{k-1}^{-1}\boldsymbol{b}_2/\bar{\gamma} \\ \boldsymbol{0} & 1/\bar{\gamma} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{k-1}^{-1} & -(\boldsymbol{u}_1 + \mu\boldsymbol{u})/\bar{\gamma} \\ \boldsymbol{0} & 1/\bar{\gamma} \end{bmatrix}.$$

Compute $\bar{\boldsymbol{A}}_k^{-1}\bar{\boldsymbol{B}}_k$. We obtain

$$\begin{aligned}
\bar{\boldsymbol{A}}_k^{-1}\bar{\boldsymbol{B}}_k &= \begin{bmatrix} \boldsymbol{A}_{k-1}^{-1} & -(\boldsymbol{u}_1+\mu\boldsymbol{u})/\bar{\gamma} \\ \boldsymbol{0} & 1/\bar{\gamma} \end{bmatrix} \begin{bmatrix} \boldsymbol{b}_1 & \boldsymbol{B} \\ \gamma\mu/\rho & \bar{\boldsymbol{c}}_1^T \end{bmatrix} \\
&= \begin{bmatrix} (1-\gamma\mu^2/(\bar{\gamma}\rho))\boldsymbol{u}-(\gamma\mu/(\bar{\gamma}\rho))\boldsymbol{u}_1 & \boldsymbol{A}_{k-1}^{-1}\boldsymbol{B}-(\boldsymbol{u}_1+\mu\boldsymbol{u})\bar{\boldsymbol{c}}_1^T/\bar{\gamma} \\ \gamma\mu/(\bar{\gamma}\rho) & \bar{\boldsymbol{c}}_1^T/\bar{\gamma} \end{bmatrix} \\
&= \begin{bmatrix} (1-\mu^2/\rho^2)\boldsymbol{u}-(\mu/\rho^2)\boldsymbol{u}_1 & \boldsymbol{U}+(\boldsymbol{u}\boldsymbol{c}_1^T/\gamma-\mu\boldsymbol{u}\bar{\boldsymbol{c}}_1^T/\bar{\gamma})-\boldsymbol{u}_1\bar{\boldsymbol{c}}_1^T/\bar{\gamma} \\ \mu/\rho^2 & \bar{\boldsymbol{c}}_1^T/\bar{\gamma} \end{bmatrix} \\
&= \begin{bmatrix} (\nu^2\boldsymbol{u}-\mu\boldsymbol{u}_1)/\rho^2 & \boldsymbol{U}+(\nu\boldsymbol{u}\bar{\boldsymbol{c}}_2^T-\boldsymbol{u}_1\bar{\boldsymbol{c}}_1^T)/\bar{\gamma} \\ \mu/\rho^2 & \bar{\boldsymbol{c}}_1^T/\bar{\gamma} \end{bmatrix}
\end{aligned}.$$

Now we focus on $\omega(\boldsymbol{A}_k)$ and $\gamma(\boldsymbol{A}_k)$. For simplicity, define

$$\begin{aligned}
\boldsymbol{u} &= [s_1,...,s_{k-1}]^T, & \boldsymbol{c}_2 &= [t_1,...,t_{n-k}]^T, \\
\boldsymbol{u_1}+\mu\boldsymbol{u} &= [\bar{s}_1,...,\bar{s}_{k-1}]^T, & \bar{\boldsymbol{c}}_2 &= [\bar{t}_1,...,\bar{t}_{n-k}]^T.
\end{aligned}$$

So we have

$$\omega_k(\bar{\boldsymbol{A}}_k)=|\bar{\gamma}|, \quad \text{and} \quad 1/\omega_i(\bar{\boldsymbol{A}}_k)^2=1/\omega_i(\boldsymbol{A}_k)^2+\bar{s}^2/\bar{\gamma}^2-s^2/\gamma^2 \quad \text{for } 1\le i\le k-1,$$

and

$$\gamma_1(\bar{\boldsymbol{C}}_k)=|\gamma\nu/\rho|, \quad \text{and} \quad \gamma_j(\bar{\boldsymbol{C}}_k)^2=\gamma_j(\boldsymbol{C}_k)^2+\bar{t}_j^2-t_j^2 \quad \text{for } 2\le j\le n-k.$$

Chapter 6 Applications and Numerical Experiments

This chapter is organzied as follows: Section 6.1 compares the numerical performance of 15 QR algorithms we've discussed on 4 different kinds of matrices. Section 6.2 show how to apply RRQR factorization in rank deficient least squares problem and compare its solution with the truncated SVD-based solution. Section 6.3 presents the application in subset selection problem and compare its solution with the SVD-based solution. Section 6.4 shows the application in matrix approximation and compare its solution with the truncated SVD-based solution. It further shows the performance of RRQR factorization in image compression.

6.1 Revealing Matrix Rank Deficiency

In this section, I report the numerical results of applying these (strong) RRQR algorithms to some matrices. The algorithms I've tested includs algorithm Greedy-I.1, Greedy-I.2, Greedy-I.3, QR with column pivoting-I, Chan-I, GKS-I, Foster-I, Hybrid-I, Hybrid-II, Hybrid-III, SRRQR-1, SRRQR-2, GSRRQR-1 and GSRRQR-2. Since the Type-II version of algorithm QR with column pivoting, Chan, GKS and Foster are not tested, I emit '-I' for simplicity. Also, I test the result of using tradition QR factorization without any permutations.

All these algorithms are implemented in MATLAB R2016a, where the machine precision is $\epsilon = 2.220446049250313 \times 10^{-16}$. I use the following square test matrices of order n:

Matrix-I A random matrix, typically of full rank, whose elements are from a uniform distribution in $[0, 1]$;

Matrix-II Modify Matrix-I by scaling its ith row by a factor $\eta^{i/n}$, where $\eta = 20\epsilon$;

Matrix-III GKS matrix, an upper triangular matrix with value $1/\sqrt{i}$ on its ith diagonal entry and $-1/\sqrt{j}$ at the i-j for $j > i$;

Matrix-IV Kahan matrix (Definition 3.2.1) with $c = 0.2$;

In algorithm Greedy-I.1 and algorithm Chan which involve computing eigenvectors, the power method is terminated until the error of the singular value is less than 0.01. The parameter δ is set to 1e-04 in algorithm Foster. In SRRQR-1 and SRRQR-2, I set $f = \sqrt{k(n-k)+\min(k,n-k)}/\sqrt{k(n-k)}$ so that these strong RRQR algorithms have similar upper bounds on $\sigma_k(\boldsymbol{M})/\sigma_{\min}(\boldsymbol{R}_{11})$ and $\sigma_{\max}(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$. In algorithm GKS, I use SVD to compute the first k dominant singular vectors.

6.1.1 Matrix-I

Table 6.1 shows the numerical results of 15 different algorithms for Matrix-I of size 50 and 100.

For $n = 50$ and $k = 40$, SRRQR algorithms guarantee

$$\max\left\{\frac{\sigma_k(\boldsymbol{M})}{\sigma_k(\boldsymbol{R}_{11})}, \frac{\sigma_1(\boldsymbol{R}_{22})}{\sigma_{k+1}(\boldsymbol{M})}\right\} < \sqrt{1+k(n-k)+\min(k,n-k)} = 20.2731$$

and

$$\max|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}| < \frac{\sqrt{k(n-k)+\min(k,n-k)}}{\sqrt{k(n-k)}} = 1.0124$$

For $n = 100$ and $k = 80$, the upper bounds are $\max\{\frac{\sigma_k(\boldsymbol{M})}{\sigma_k(\boldsymbol{R}_{11})}, \frac{\sigma_1(\boldsymbol{R}_{22})}{\sigma_{k+1}(\boldsymbol{M})}\} < 40.2616$ and $\max|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}| < 1.0062$.

Since the Matrix-I is nonsinglar and the condition number is of order 10^3, $\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$ and $\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$ are much smaller than the upper bound even for the tradition QR factorization which does not consider any permutations.

Among all the algorithms, Greedy-I.1 is the most expensive since it evolves computing the smallest singular values for $n-l$ different matrices at step l. Since the running time of Greedy-I.1 is much more than computing the SVD, this algorithm is rarely used in practice. Also as we've expected, Greedy-I.2 runs faster than Greedy-I.1 and Greedy-I.3 runs faster than Greedy-I.2. Though Greedy-I.3 costs similar running time as hybrid and strong RRQR algorithms, the performance of Greedy-I.3 is consistently worse than hybrid and strong RRQR algorithms. So Greedy-I.2 and Greedy-I.3

Table 6.1 Numerical Results for Matrix-I

	$n = 50$, rank$(\boldsymbol{M}) = 50$, cond$(\boldsymbol{M})$ =8.5108e+03, set $k = 40$			
	Time (s)	$\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$	$\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$	$\max \lvert\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\rvert$
SVD	0.0117			
QR	0.0014	2.8876	2.5026	0.9842
Greedy-I.1	0.1703	1.9364	2.2639	0.7649
Greedy-I.2	0.0165	2.5609	2.8417	0.8744
Greedy-I.3	0.0098	2.7971	2.6270	0.9207
ColumnPivot	0.0023	2.7971	2.6270	0.9207
Chan	0.0070	1.8310	2.0008	0.9166
GKS	0.0146	2.3144	2.2450	0.8617
Foster	0.0030	2.8362	2.4851	0.8639
Hybrid-I	0.0042	1.9115	1.8641	0.6485
Hybrid-II	0.0062	1.5410	2.3129	0.7179
Hybrid-III	0.0113	1.8934	1.8975	0.6797
SRRQR-1	0.0107	1.3935	1.8560	0.8002
SRRQR-2	0.0119	1.6703	2.0053	0.7267
GSRRQR-1	0.0066	1.5599	2.1660	0.6978
GSRRQR-2	0.0072	1.4833	1.9371	0.7247
	$n = 100$, rank$(\boldsymbol{M}) = 100$, cond$(\boldsymbol{M})$ =1.9331e+03, set $k = 80$			
	Time (s)	$\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$	$\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$	$\max \lvert\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\rvert$
SVD	0.0846			
QR	0.0060	2.7047	2.9856	0.9140
Greedy-I.1	1.0971	2.2114	2.9711	0.7298
Greedy-I.2	0.0654	2.2680	2.8928	0.8912
Greedy-I.3	0.0370	2.1525	2.7229	0.7355
ColumnPivot	0.0065	2.5955	2.8867	0.8141
Chan	0.0228	2.3599	2.0632	0.7345
GKS	0.0838	2.6569	2.8946	0.7804
Foster	0.0127	2.5422	2.6426	0.8276
Hybrid-I	0.0143	1.7832	2.3184	0.6388
Hybrid-II	0.0143	1.8311	2.5306	0.6431
Hybrid-III	0.0490	1.6072	2.4058	0.6656
SRRQR-1	0.0594	1.7842	2.1876	0.6077
SRRQR-2	0.0491	1.6519	2.4636	0.6844
GSRRQR-1	0.0272	1.8350	2.2535	0.5681
GSRRQR-2	0.0218	1.6624	2.1590	0.6187

also have no practical use.

As we can see, hybrid and strong RRQR algorithms give better results even for nonsingular matrices. However, the running time of algorithm SRRQR-1 and SRRQR-2 are close to the time for computing SVD. The new greedy strong RRQR algorithms reduce the running time by half which make them even faster than algorithm Hybrid-III.

Table 6.2 shows the number of iterations they performed for the two matrices. We can see by embedding the greedy strategy into the strong RRQR algorithms, the number of iterations is significantly reduced.

Table 6.2 Matrix-I: Number of Iterations of strong RRQR algorithms

	Num. of Iterations at $n = 50$	Num. of Iterations at $n = 100$
SRRQR-1	26	66
GSRRQR-1	10	27
SRRQR-2	14	37
GSRRQR-2	6	16

Algorithm QR with column pivoting is the most efficient RRQR algorithm. The time cost is only a little more than the traditional QR factorization. Gu ([2], 1996) suggested that instead of using traditional QR factorization to initialize $\boldsymbol{R}$, using QR with column pivoting may reduce the total number of iterations. Table 6.3 presents the total running time and the number of iterations performed of algorithm SRRQR-1 and SRRQR-2 on Matrix-I with size 1000×1000 before and after initialized by QR with column pivoting.

Table 6.3 Matrix-I: Time efficiency of strong RRQR algorithms

Matrix-I of size 1000×1000, $k = 500$	Time (s)	Number of Iterations
SVD	69.2855	
SRRQR-1	30.1796	888
SRRQR-1 (Initialized by ColumnPivot)	35.8319	868
GSRRQR-1	11.7054	269
SRRQR-2	27.5884	805
SRRQR-2 (Initialized by ColumnPivot)	33.5999	791
GSRRQR-2	11.5738	266

First, we see for large matrices, these strong RRQR algorithms are much faster than

SVD. For Matrix-I, the advantage of this kind of initialization is not clear because the total running time does increase in the numerical results. But we can see that initializing by QR with column pivoting decreases the number of iterations that SRRQR-1 and SRRQR-2 will perform. However it's still much larger than the number of iterations that GSRRQR-1 and GSRRQR-2 will perform. This reveals the superiority of GSRRQR algorithms.

6.1.2 Matrix-II

Table 6.4 shows the numerical results on Matrix-II. For $n = 50$, since $\sigma_1(\boldsymbol{M})/\sigma_{16}(\boldsymbol{M}) > 10^2$ and $\sigma_{16}(\boldsymbol{M}) < 0.0001$, I select $k = 15$. For $n = 100$, since $\sigma_1(\boldsymbol{M})/\sigma_{31}(\boldsymbol{M}) > 10^2$ and $\sigma_{31}(\boldsymbol{M}) < 0.0001$, I select $k = 30$.

For $n = 50$ and $k = 15$, SRRQR algorithms guarantee

$$\max\left\{\frac{\sigma_k(\boldsymbol{M})}{\sigma_k(\boldsymbol{R}_{11})}, \frac{\sigma_1(\boldsymbol{R}_{22})}{\sigma_{k+1}(\boldsymbol{M})}\right\} < \sqrt{1 + k(n-k) + \min(k, n-k)} = 23.2594$$

and

$$\max|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}| < \frac{\sqrt{k(n-k) + \min(k, n-k)}}{\sqrt{k(n-k)}} = 1.0142$$

For $n = 100$ and $k = 30$, the upper bounds are $\max\{\frac{\sigma_k(\boldsymbol{M})}{\sigma_k(\boldsymbol{R}_{11})}, \frac{\sigma_1(\boldsymbol{R}_{22})}{\sigma_{k+1}(\boldsymbol{M})}\} < 46.1628$ and $\max|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}| < 1.0071$.

This time, the condition numbers of these two matrices are much higher than Matrix-I. We see that algorithm Foster even failed to give a good RRQR factorization. Algorithm Greedy-I.1 is again the most expensive algorithm. Algorithm Chan gets the best results among all greedy algorithms while it also runs very fast (only slightly a little slower than QR with column pivoting).

Also, we see that only SRRQR algorithms, including SRRQR-1, SRRQR-2, GSRRQR-1 and GSRRQR-2, succeeded in achieving the above bounds. Hybrid algorithms failed to control $\max|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}|$. The best $\sigma_k(\boldsymbol{R}_{11})$ and $\sigma_1(\boldsymbol{R}_{22})$ is also given by SRRQR algorithms. This shows that it is meaningful to design SRRQR algorithms.

The result also shows that GSRRQR algorithms reduce the time cost of the ori-

Table 6.4 Numerical Results for Matrix-II

	$n = 50$, rank$(\boldsymbol{M}) = 46$, cond$(\boldsymbol{M})$ =4.9411e+15, set $k = 15$			
	Time (s)	$\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$	$\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$	$\max \lvert\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\rvert$
SVD	0.0074			
QR	0.0008	5.6597	4.3694	3.7278
Greedy-I.1	0.0269	10.7276	11.9016	8.4359
Greedy-I.2	0.0062	5.2200	4.5652	3.5922
Greedy-I.3	0.0034	5.2200	4.5652	3.5922
ColumnPivot	0.0008	5.2200	4.5652	3.5922
Chan	0.0015	3.7424	2.2174	1.2127
GKS	0.0090	6.0064	5.2097	8.1821
Foster	0.0020	9.8869	3.0568	3.5607
Hybrid-I	0.0020	3.1786	2.7320	1.2694
Hybrid-II	0.0020	3.0228	2.4930	1.5080
Hybrid-III	0.0043	1.9504	2.4807	1.3777
SRRQR-1	0.0067	2.3777	2.0571	0.9829
SRRQR-2	0.0089	2.4725	1.6376	0.9491
GSRRQR-2	0.0033	2.3935	1.7470	0.9836
GSRRQR-2	0.0048	1.9415	1.8019	0.9558
	$n = 100$, rank$(\boldsymbol{M}) = 90$, cond$(\boldsymbol{M})$ =5.8480e+15 ,set $k = 30$			
	Time (s)	$\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$	$\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$	$\max \lvert\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\rvert$
SVD	0.0511			
QR	0.0050	10.7159	7.2143	5.5968
Greedy-I.1	0.1731	7.4402	4.6532	3.7944
Greedy-I.2	0.0739	13.2902	5.1055	6.4425
Greedy-I.3	0.0170	11.6392	4.6704	3.9468
ColumnPivot	0.0049	11.6392	4.6704	3.9468
Chan	0.0084	4.2990	2.0698	1.5668
GKS	0.0548	7.6746	5.7229	4.0032
Foster	0.0112	263.9728	4.2487	37.3676
Hybrid-I	0.0076	3.3815	1.7790	1.4796
Hybrid-II	0.0116	3.0524	2.6152	1.2677
Hybrid-III	0.0263	3.3815	1.7790	1.4796
SRRQR-1	0.0310	2.4026	1.6851	0.9525
SRRQR-2	0.0310	2.8079	1.9037	0.9491
GSRRQR-1	0.0200	3.6254	2.1377	0.9848
GSRRQR-2	0.0223	2.4766	1.6020	0.9903

gin SRRQR algorithms almost by half. This make strong RRQR factorization much more engaging comparing with SVD. Table 6.5 show the running time and the number of iterations these algorithms performed for Matrix-II with size 1000×1000. Since σ_1/σ_{288} >1e+04, I choose $k = 287$ in this experiment.

Table 6.5 Matrix-II: Time efficiency of strong RRQR algorithms

Matrix-I of size 1000×1000, $k = 287$	Time (s)	Number of Iterations
SVD	42.8328	
SRRQR-1	13.6208	399
SRRQR-1 (Initialized by ColumnPivot)	18.1257	423
GSRRQR-1	8.0523	207
SRRQR-2	9.8827	234
SRRQR-2 (Initialized by ColumnPivot)	12.4373	172
GSRRQR-2	5.5030	66

The results show that greedy SRRQR algorithms again reduce the number of iterations of the origin SRRQR algorithms approximately by half while their resulting factorization is as good as the one provided by the origin SRRQR algorithms.

Also from the results, we see that initialization by QR with column pivoting does not guarantee to reduce the number of iterations that the following SRRQR algorithms will perform.

6.1.3 Matrix-III

Table 6.6 shows the numerical results on Matrix-III. The numerical rank of this matrix is 49. The condition number of this matrix is 7.6873e+18. For all algorithms, I set $k = 48$ and want to test whether the algorithm could split out the single bad column from others.

The theoretical bounds for SRRQR algorithms are

$$\max\left\{\frac{\sigma_k(\boldsymbol{M})}{\sigma_k(\boldsymbol{R}_{11})}, \frac{\sigma_1(\boldsymbol{R}_{22})}{\sigma_{k+1}(\boldsymbol{M})}\right\} < \sqrt{1 + k(n-k) + \min(k, n-k)} = 9.9499$$

and

$$\max|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}| < \frac{\sqrt{k(n-k) + \min(k, n-k)}}{\sqrt{k(n-k)}} = 1.0104$$

Table 6.6 Numerical Results for Matrix-III

	$n = 50$, rank$(\boldsymbol{M}) = 49$, cond$(\boldsymbol{M})$ =7.6873e+18, set $k = 48$			
	Time (s)	$\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$	$\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$	$\max \lvert\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\rvert$
SVD	0.0132			
QR	0.0017	2.4059e+13	1.0574	2.0105e+13
Greedy-I.1	0.1142	4.4050e+04	1.6907	5.8773e+04
Greedy-I.2	0.0259	5.6010e+03	1.7015	8.1890e+03
Greedy-I.3	0.0076	1.4006e+03	1.3302	2.1894e+03
ColumnPivot	0.0021	9.3993e+10	1.1611	8.6922e+10
Chan	0.0078	1.0197	1.1665	2.3094
GKS	0.0144	5.8525e+12	1.1656	5.0024e+12
Foster	0.0038	1.6923e+13	1.0504	1.3527e+13
Hybrid-I	0.0034	1.0040	1.1611	0.7071
Hybrid-II	0.0042	1.0050	1.1776	0.7071
Hybrid-III	0.0101	1.0040	1.1611	0.7071
SRRQR-1	0.0168	1.0040	1.1611	0.7071
SRRQR-2	0.0045	1.0129	1.3034	0.7071
GSRRQR-1	0.0135	1.0040	1.1611	0.7071
GSRRQR-2	0.0024	1.0050	1.1776	0.7071

This matrix is much more tough than the previous two. From the numerical results, we see that all greedy algorithms except Chan failed to find a good RRQR factorization. Algorithm Chan, though give a very good $\sigma_k(\boldsymbol{R}_{11})$ and $\sigma_1(\boldsymbol{R}_{22})$, failed to control the magnitude of $\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}$.

Table 6.7 Matrix-III: Number of Iterations of hybrid algorithms

Matrix-III of size 50×50, $k = 48$	Number of Iterations
Hybrid-I	2
Hybrid-II	2
Hybrid-III	2

This matrix shows the deficiency of greedy algorithms. All hybrid algorithms and strong RRQR algorithms succeed in finding a strong RRQR factorization. We also see that the theoretical bound on $\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$ and $\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$ is far from reached for both hybrid algorithms and strong RRQR algorithms. And again, greedy SRRQR algorithms run faster than the origin SRRQR algorithms.

From Table 6.7, we see that hybrid algorithms only use 2 iterations to find a good permutation. Since algorithm Hybrid-III involves alternating between Hybrid-I and

Hybrid-II, we see that after algorithm Hybrid-III performs Hybrid-I for the first time, Hybrid-II agrees with its solution and so no more iterations are required.

6.1.4 Matrix-IV

Finally, the numerical results for the Kahan matrix with size 50 is presented in Table 6.8. The condition number of this matrix is 4.9910e+04, which is much smaller than the preivous Matrix-III. We again choose $k = 48$. So the theoretical bounds for SRRQR algorithms are the same as for Matrix-III, which are

$$\max\left\{\frac{\sigma_k(\boldsymbol{M})}{\sigma_k(\boldsymbol{R}_{11})}, \frac{\sigma_1(\boldsymbol{R}_{22})}{\sigma_{k+1}(\boldsymbol{M})}\right\} < \sqrt{1 + k(n-k) + \min(k, n-k)} = 9.9499$$

and

$$\max|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}| < \frac{\sqrt{k(n-k) + \min(k, n-k)}}{\sqrt{k(n-k)}} = 1.0104$$

Table 6.8 Numerical Results for Matrix-IV

	$n = 50$, rank$(\boldsymbol{M}) = 49$, rank$(\boldsymbol{M})$ =4.9910e+04, $k = 48$			
	Time (s)	$\sigma_k(\boldsymbol{M})/\sigma_k(\boldsymbol{R}_{11})$	$\sigma_1(\boldsymbol{R}_{22})/\sigma_{k+1}(\boldsymbol{M})$	$\max\|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}\|$
SVD	0.0129			
QR	0.0018	3.0303e+03	1.0000	1.0533e+03
Greedy-I.1	0.2349	1.0058	1.0954	0.8333
Greedy-I.2	0.0148	1.2472e+03	1.1993	654.2344
Greedy-I.3	0.0077	1.2472e+03	1.1993	654.2344
ColumnPivot	0.0017	1.2472e+03	1.1993	654.2344
Chan	0.0042	3.0303e+03	1.0202	1.0533e+03
GKS	0.0136	1.0058	1.0954	0.8333
Foster	0.0026	3.0303e+03	1.0202	0.8333
Hybrid-I	0.0025	1.0058	1.0954	0.8333
Hybrid-II	0.0043	1.0058	1.0954	0.8333
Hybrid-III	0.0102	1.0058	1.0954	0.8333
SRRQR-1	0.0032	1.0058	1.0954	0.8333
SRRQR-2	0.0032	1.0058	1.0954	0.8333
GSRRQR-1	0.0031	1.0058	1.0954	0.8333
GSRRQR-2	0.0030	1.0058	1.0954	0.8333

Only two greedy algorithms, Greedy-I.1 and GKS, succeeded in finding a good

permutation for the Kahan matrix. Kahan matrix shows the deficiency of these greedy RRQR algorithms again.

In Section 3.2.10, we showed that exact execution of Algorithm Greedy-I.1, Greedy-I.2, Greedy-I.3 and QR with column pivoting won't perform any permutation for Kahan matrices. However, due to the error in computing the smallest singular value (by inverse iteration), as I shown in the results, Algorithm Greedy-I.1 does succeed in finding a good permutation.

Table 6.9 Matrix-IV: Number of iterations of strong RRQR algorithms

Matrix-IV of size 50×50	Number of Iterations
SRRQR-1	1
GSRRQR-1	1
SRRQR-2	1
GSSRRQR-2	1

Table 6.9 show the number of iterations performed by these strong RRQR algorithms. Though the origin SRRQR algorithms only require 1 iteration, greedy SRRQR algorithms run faster because they use heuristic, or say greedy strategy, to find that i-j pair.

6.2 Rank Deficient Least Squares Problem

6.2.1 Theoretical Analysis

In this section, we consider the linear least squares problem

$$\min \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2,$$

where $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is ill-conditioned. The usual least squares method first QR factor $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$ and then solve the linear least squares problem $\min \|\boldsymbol{R}\boldsymbol{x} - \boldsymbol{Q}^T\boldsymbol{b}\|_2$. Let $\tilde{bm}R$ denote the matrix formed by the first n rows of $\boldsymbol{R}$ and $\tilde{\boldsymbol{b}}$ be the vector formed by the first n rows of $\boldsymbol{Q}^T\boldsymbol{b}$, then the solution is given by $\boldsymbol{x} = \tilde{\boldsymbol{R}}^{-1}\tilde{b}$. However, this solution is unstable since $\tilde{R}$ is nearly singular. The solution is extremely sensitive the small perturbations of $\boldsymbol{b}$.

One practical approach for solving the problem is by the truncated SVD (TSVD). TSVD only keeps the information of the k most dominant singular values of the origin matrix $\boldsymbol{A}$, where k is the numerical rank of $\boldsymbol{A}$. We have

$$\boldsymbol{A} = \sum_{i=1}^{k} \boldsymbol{u}_i \sigma_i \boldsymbol{v}_i^T,$$

where $\boldsymbol{u}_i$ ($\boldsymbol{v}_i$) is the left (right) singular vector of $\boldsymbol{A}$ corresponding to σ_i. Then the solution $\boldsymbol{x}_{TSVD}$ is given by

$$\boldsymbol{x}_{TSVD} = \sum_{i=1}^{k} \frac{\boldsymbol{u}_i^T \boldsymbol{b} \boldsymbol{v}_i}{\sigma_i}.$$

Instead of using SVD, we can also use (strong) RRQR factorization to give a solution for the least squares problem. For convention, we change our notation here. We use $\boldsymbol{A}$ instead of $\boldsymbol{M}$ as the input of RRQR factorization. Suppose (strong) RRQR factorization gives

$$\boldsymbol{A}\boldsymbol{\Pi} = \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix}.$$

Since $\sigma_{\max}(\boldsymbol{R}_{22})$ is small, we neglect the submatrix $\boldsymbol{R}_{22}$. Now our problem turns to

$$\min \left\| \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \boldsymbol{\Pi}^T \boldsymbol{x} - \boldsymbol{Q}^T \boldsymbol{b} \right\|_2.$$

Find an orthogonal transformation $\hat{\boldsymbol{Q}}$ such that

$$\hat{\boldsymbol{Q}}^T \begin{bmatrix} \boldsymbol{R}_{11}^T \\ \boldsymbol{R}_{12}^T \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{R}}_{11}^T \\ \boldsymbol{0} \end{bmatrix}.$$

This $\hat{\boldsymbol{Q}}$ also gives

$$\begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \hat{\boldsymbol{Q}} = \begin{bmatrix} \hat{\boldsymbol{R}}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}.$$

Now the problem can be restated as

$$\min \left\| \begin{bmatrix} \hat{\boldsymbol{R}}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \hat{\boldsymbol{Q}}^T \boldsymbol{\Pi}^T \boldsymbol{x} - \boldsymbol{Q}^T \boldsymbol{b} \right\|_2,$$

and the truncated RRQR solution $\boldsymbol{x}_{TQR}$ is given by

$$\boldsymbol{x}_{TQR} = \boldsymbol{\Pi} \hat{\boldsymbol{Q}} \begin{bmatrix} \hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \boldsymbol{Q}^T \boldsymbol{b}.$$

Another way to compute the solution, which is more efficient, is to also neglect the submatrix $\boldsymbol{R}_{12}$. The *basic solution* $\boldsymbol{x}_{BQR}$ is then given by

$$\boldsymbol{x}_{BQR} = \boldsymbol{\Pi} \begin{bmatrix} \boldsymbol{R}_{11}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \boldsymbol{Q}^T \boldsymbol{b}.$$

Now we analyze the difference and goodness of these three solutions.

Definition 6.2.1. The residual vectors are defined as

$$\begin{aligned} \boldsymbol{r}_{TSVD} &= \boldsymbol{A}\boldsymbol{x}_{TSVD} - \boldsymbol{b} \\ \boldsymbol{r}_{TQR} &= \boldsymbol{A}\boldsymbol{x}_{TQR} - \boldsymbol{b} \\ \boldsymbol{r}_{BQR} &= \boldsymbol{A}\boldsymbol{x}_{BQR} - \boldsymbol{b} \end{aligned}$$

Theorem 6.2.1. *The differences between $\boldsymbol{x}_{TSVD}$, $\boldsymbol{x}_{TQR}$, $\boldsymbol{x}_{BQR}$ can be controlled by*

$$\begin{aligned} \|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{TQR}\|_2 &\leq \|\boldsymbol{R}_{22}\|_2 \|\boldsymbol{R}_{11}^{-1}\|_2 \left(2\|\boldsymbol{x}_{TSVD}\|_2 + \frac{\|\boldsymbol{r}_{TSVD}\|_2}{\sigma_k} \right) \\ \|\boldsymbol{x}_{TQR} - \boldsymbol{x}_{BQR}\|_2 &\leq \frac{1+\sqrt{5}}{2} \|\boldsymbol{R}_{11}^{-1}\|_2^2 \|\boldsymbol{R}_{12}\|_2 \|\boldsymbol{b}\|_2. \end{aligned} \tag{6-1}$$

The differences between the residual vectors satisfy

$$\begin{aligned} \|\boldsymbol{r}_{TSVD} - \boldsymbol{r}_{TQR}\|_2 &\leq \|\boldsymbol{R}_{22}\|_2 \left(\|\boldsymbol{x}_{TSVD}\|_2 + \frac{\|\boldsymbol{r}_{TSVD}\|_2}{\sigma_k} \right) \\ \|\boldsymbol{r}_{TQR} - \boldsymbol{r}_{BQR}\|_2 &\leq \|\boldsymbol{R}_{22}\|_2 \|\boldsymbol{R}_{11}^{-1}\|_2 \|\boldsymbol{b}\|_2 \end{aligned} \tag{6-2}$$

Proof. Note that solution $\boldsymbol{x}_{TQR}$ is identical to the TSVD solution $\boldsymbol{x}_{TSVD}$ applied to the problem

$$\min \left\| \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \boldsymbol{\Pi}^T \boldsymbol{x} - \boldsymbol{b} \right\|_2$$

We can regard $\boldsymbol{x}_{TQR}$ as a TSVD solution for the perturbed matrix $\tilde{\boldsymbol{A}}$ with perturbation $\boldsymbol{E}$,

$$\tilde{\boldsymbol{A}} = \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \boldsymbol{\Pi}^T, \quad \boldsymbol{E} = \boldsymbol{Q} \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix} \boldsymbol{\Pi}^T \tag{6-3}$$

From the derivation of $\boldsymbol{x}_{TQR}$, we see the pseudo-inverse of $\tilde{\boldsymbol{A}}$ is

$$\tilde{\boldsymbol{A}}^+ \triangleq \begin{bmatrix} \hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}.$$

Then we have

$$\|\tilde{\boldsymbol{A}}^+\|_2 = \|\hat{\boldsymbol{R}}_{11}^{-1}\|_2 \leq \|\boldsymbol{R}_{11}^{-1}\|_2, \tag{6-4}$$

where the last inequality comes from the derivation of $\hat{\boldsymbol{R}}_{11}$. Also by Equation (6-3), we have $\|\boldsymbol{E}\|_2 = \|\boldsymbol{R}_{22}\|_2$ and $\sigma_{k+1}(\tilde{\boldsymbol{A}}) = 0$.

Hansen ([24], 1987) gave general perturbation bounds for the TSVD solutions:

$$\begin{aligned} \|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{TQR}\|_2 &\leq \|\tilde{\boldsymbol{A}}^+\|_2 \|\boldsymbol{E}\|_2 \left(\|\boldsymbol{x}_{TSVD}\|_2 + \frac{\|\boldsymbol{r}_{TSVD}\|_2}{\sigma_k} \right) + \frac{\|\boldsymbol{E}\|_2 \|\boldsymbol{x}_{TSVD}\|_2}{\sigma_k} \\ \|\boldsymbol{r}_{TSVD} - \boldsymbol{r}_{TQR}\|_2 &\leq \|\boldsymbol{E}\|_2 \|\boldsymbol{x}_{TSVD}\|_2 + \frac{\|\boldsymbol{E}\|_2 \|\boldsymbol{r}_{TSVD}\|_2}{\sigma_k}, \end{aligned} \tag{6-5}$$

where σ_k denotes the kth largest singular value of $\boldsymbol{A}$. The interlacing property tells

$$\sigma_k^{-1} \leq \|\boldsymbol{R}_{11}^{-1}\|_2. \tag{6-6}$$

Plugging Inequality (6-4) and (6-6) into Inequality (6-5) gives

$$\|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{TQR}\|_2 \leq \|\boldsymbol{R}_{22}\|_2 \|\boldsymbol{R}_{11}^{-1}\|_2 \left(2\|\boldsymbol{x}_{TSVD}\|_2 + \frac{\|\boldsymbol{r}_{TSVD}\|_2}{\sigma_k}\right)$$

$$\|\boldsymbol{r}_{TSVD} - \boldsymbol{r}_{TQR}\|_2 \leq \|\boldsymbol{R}_{22}\|_2 \left(\|\boldsymbol{x}_{TSVD}\|_2 + \frac{\|\boldsymbol{r}_{TSVD}\|_2}{\sigma_k}\right)$$

By definition of $\boldsymbol{x}_{BQR}$, we have

$$\|\boldsymbol{x}_{TQR} - \boldsymbol{x}_{BQR}\|_2 \leq \|\boldsymbol{\Pi}\|_2 \cdot \left\| \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}^+ - \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}^+ \right\|_2 \cdot \|\boldsymbol{Q}\|_2 \cdot \|\boldsymbol{b}\|_2$$

Now define

$$\bar{\boldsymbol{E}} \triangleq \begin{bmatrix} 0 & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}, \quad \bar{\boldsymbol{A}} \triangleq \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}.$$

By the property of perturbed pseudo-inverse (Thm 5.3 in [25], Björck, 1990), we have

$$\|(\bar{\boldsymbol{A}} + \bar{\boldsymbol{E}})^+ - \bar{\boldsymbol{A}}^+\|_2 \leq \frac{1+\sqrt{5}}{2} \|\bar{\boldsymbol{A}}^+\|_2 \|(\bar{\boldsymbol{A}} + \bar{\boldsymbol{E}})^+\|_2 \|\bar{\boldsymbol{E}}\|_2 \tag{6-7}$$

This gives the bound

$$\|\boldsymbol{x}_{TQR} - \boldsymbol{x}_{BQR}\|_2 \leq \frac{1+\sqrt{5}}{2} \|\boldsymbol{R}_{11}^{-1}\|_2^2 \|\boldsymbol{R}_{12}\|_2 \|\boldsymbol{b}\|_2.$$

Finally, by definition of the residual vectors, we have

$$\begin{aligned} \boldsymbol{r}_{TQR} - \boldsymbol{r}_{BQR} &= \boldsymbol{A}(\boldsymbol{x}_{TQR} - \boldsymbol{x}_{BQR}) \\ &= \boldsymbol{Q}\boldsymbol{R}\boldsymbol{\Pi}^T\boldsymbol{\Pi} \left(\hat{\boldsymbol{Q}} \begin{bmatrix} \hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} - \begin{bmatrix} \boldsymbol{R}_{11}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \right) \boldsymbol{Q}^T\boldsymbol{b} \end{aligned} \tag{6-8}$$

Partition $\hat{\boldsymbol{Q}}$ as

$$\hat{\boldsymbol{Q}} = \begin{bmatrix} \hat{\boldsymbol{Q}}_{11} & \hat{\boldsymbol{Q}}_{12} \\ \hat{\boldsymbol{Q}}_{21} & \hat{\boldsymbol{Q}}_{22} \end{bmatrix},$$

where $\hat{\boldsymbol{Q}}_{11}$ is of order k. The definition of $\hat{\boldsymbol{Q}}$ tells

$$\hat{\boldsymbol{R}}_{11} = \boldsymbol{R}_{11}\hat{\boldsymbol{Q}}_{11} + \boldsymbol{R}_{12}\hat{\boldsymbol{Q}}_{21} \tag{6-9}$$

Now follow Equation (6-8),

$$\begin{aligned} \boldsymbol{r}_{TQR} - \boldsymbol{r}_{BQR} &= \boldsymbol{Q}\boldsymbol{R}\left(\begin{bmatrix} \hat{\boldsymbol{Q}}_{11}\hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \\ \hat{\boldsymbol{Q}}_{21}\hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \end{bmatrix} - \begin{bmatrix} \boldsymbol{R}_{11}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}\right)\boldsymbol{Q}^T\boldsymbol{b} \\ &= \boldsymbol{Q}\left(\begin{bmatrix} (\boldsymbol{R}_{11}\hat{\boldsymbol{Q}}_{11} + \boldsymbol{R}_{12}\boldsymbol{H}_{21})\hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \\ \boldsymbol{R}_{22}\hat{\boldsymbol{Q}}_{21}\hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \end{bmatrix} - \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}\right)\boldsymbol{Q}^T\boldsymbol{b} \\ &= \boldsymbol{Q}\begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{R}_{22}\hat{\boldsymbol{Q}}_{21}\hat{\boldsymbol{R}}_{11}^{-1} & \boldsymbol{0} \end{bmatrix}\boldsymbol{Q}^T\boldsymbol{b} \end{aligned} \tag{6-10}$$

Taking two norm on both sides of Inequality (6-10) gives

$$\|\boldsymbol{r}_{TQR} - \boldsymbol{r}_{BQR}\|_2 \leq \|\boldsymbol{R}_{22}\|_2\|\boldsymbol{R}_{11}^{-1}\|_2\|\boldsymbol{b}\|_2$$

□

Since most RRQR algorithms ensure $\|\boldsymbol{R}_{22}\|_2 \leq \sigma_{k+1}(\boldsymbol{A})\, q(n,k)$ and since $\|\boldsymbol{R}_{11}^{-1}\|_2$ is small as long as $\sigma_{k+1}(\boldsymbol{A})$ is small, Theorem 6.2.1 tells that solution $\boldsymbol{x}_{TSVD}$ and $\boldsymbol{x}_{TQR}$ are approximately the same. But $\boldsymbol{x}_{TQR}$ is cheaper to compute than $\boldsymbol{x}_{TSVD}$.

Theorem 6.2.1 also says that the residual $\boldsymbol{r}_{BQR}$ is approximately the same as $\boldsymbol{r}_{TQR}$ and $\boldsymbol{r}_{TSVD}$, though the solution $\boldsymbol{x}_{BQR}$ may be very different from the other two ($\boldsymbol{x}_{BQR}$ may have a large component in the null space of $\boldsymbol{A}$).

6.2.2 Numerical Experiment

In Section 6.1, we see hybrid algorithms and SRRQR algorithms always give a good permutation for RRQR factorization. Since RRQR algorithms guarantee $|\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12}|$ to be small, I use algorithm GSRRQR-2 in the experiment.

The dimension of the system is $m = n = 100$ and the matrix $\boldsymbol{A}$ is chosen such that the first k singular values of $\boldsymbol{A}$ span linearly between 1000 and 1, and the rest $n-k$ singular values of $\boldsymbol{A}$ are chosen to be the same. Then I pick a random unitary vector $\boldsymbol{x}_{exact}$ to generate $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}_{exact}$. I also compute the solution given by full SVD, $\boldsymbol{x}_{FSVD} = \boldsymbol{R}^{-1}\boldsymbol{Q}^T\boldsymbol{b}$, and its residual $\boldsymbol{r}_{FSVD} = \boldsymbol{A}\boldsymbol{x}_{FSVD} - \boldsymbol{b}$. Table 6.10 and 6.11 show the numerical results under different k and ratio σ_k/σ_{k+1}.

Table 6.10 Norm of Residuals

Matrix $n = 100, k = 50$			Results			
σ_1	σ_{50}	$\sigma_{51}, ..., \sigma_{100}$	$\|\boldsymbol{r}_{TSVD}\|_2$	$\|\boldsymbol{r}_{TQR}\|_2$	$\|\boldsymbol{r}_{BQR}\|_2$	$\|\boldsymbol{r}_{FSVD}\|_2$
1000	1	10^{-1}	0.0680	0.0680	0.1711	6.2423e-13
1000	1	10^{-4}	7.8631e-05	7.8631e-05	1.2905e-04	1.0609e-12
1000	1	10^{-7}	6.8505e-08	6.8505e-08	1.5457e-07	1.1350e-12
Matrix $n = 100, k = 90$			Results			
σ_1	σ_{90}	$\sigma_{91}, ..., \sigma_{100}$	$\|\boldsymbol{r}_{TSVD}\|_2$	$\|\boldsymbol{r}_{TQR}\|_2$	$\|\boldsymbol{r}_{BQR}\|_2$	$\|\boldsymbol{r}_{FSVD}\|_2$
1000	1	10^{-1}	0.0271	0.0270	0.0919	8.7051e-13
1000	1	10^{-4}	3.3660e-05	3.3660e-05	7.6489e-05	2.0268e-12
1000	1	10^{-7}	2.6954e-08	2.6954e-08	1.0541e-07	1.1925e-12

Table 6.11 Differences between Solutions

Matrix $n = 100, k = 50$			Results		
σ_1	σ_{50}	$\sigma_{51}, ..., \sigma_{100}$	$\|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{exact}\|_2$	$\|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{TQR}\|_2$	$\|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{BQR}\|_2$
1000	1	10^{-1}	0.7113	0.0043	0.9879
1000	1	10^{-4}	0.7315	2.0382e-09	1.0418
1000	1	10^{-7}	0.6347	2.2103e-14	1.1225
Matrix $n = 100, k = 90$			Results		
σ_1	σ_{90}	$\sigma_{91}, ..., \sigma_{100}$	$\|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{exact}\|_2$	$\|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{TQR}\|_2$	$\|\boldsymbol{x}_{TSVD} - \boldsymbol{x}_{BQR}\|_2$
1000	1	10^{-1}	0.3421	0.0018	0.9667
1000	1	10^{-4}	0.2000	8.6806e-11	0.7315
1000	1	10^{-7}	0.3685	4.4359e-14	0.7650

From Table 6.10, we see that RRQR-based solution $\boldsymbol{x}_{TQR}$ and $\boldsymbol{x}_{BQR}$ give compa-

rably small residuals as the TSVD-based solution $\boldsymbol{x}_{TSVD}$ when the linear system goes ill-conditioned. Also, we see the residuals $\boldsymbol{r}_{TSVD}$, $\boldsymbol{r}_{TQR}$, $\boldsymbol{r}_{BQR}$ are of the same order as σ_k/σ_{k+1} and they are irrelevant to the numerical rank k.

Table 6.11 checks the similarity between $\boldsymbol{x}_{TSVD}$, $\boldsymbol{x}_{TQR}$, $\boldsymbol{x}_{BQR}$ and $\boldsymbol{x}_{exact}$. We see that the similarity between $\boldsymbol{x}_{TSVD}$ and $\boldsymbol{x}_{TQR}$ depends on the ratio $\sigma_k(\boldsymbol{A})/\sigma_{k+1}(\boldsymbol{A})$ instead of the numerical rank k. This accords with our analytical statement.

Also we see since the system is ill-conditioned, there may be many good approximate solutions. Our $\boldsymbol{x}_{TSVD}$ and $\boldsymbol{x}_{TQR}$ may be very different from $\boldsymbol{x}_{exact}$. Also, we see the solution $\boldsymbol{x}_{BQR}$ is very different from both $\boldsymbol{x}_{TQR}$ and $\boldsymbol{x}_{TSVD}$.

So if one wants to get a faster solution for this system, I suggest $\boldsymbol{x}_{BQR}$ which offers the same order of residual as $\boldsymbol{x}_{TSVD}$. If one wants to get a solution that TSVD will give, I suggest $\boldsymbol{x}_{TQR}$, which is nearly the same as $\boldsymbol{x}_{TSVD}$.

6.3 Subset Selection Problem

6.3.1 Theoretical Analysis

In this section, we consider the subset selection problem, which aims to determine the k most linearly independent columns of the given matrix $\boldsymbol{A}$, where k is the numerical rank of $\boldsymbol{A}$. Specifically, we try to find a permutation matrix $\boldsymbol{\Pi}$ such that the condition number of the submatrix formed by the first k columns of $\boldsymbol{A\Pi}$ is maximized.

Golub, Klema and Stewart ([22], 1976) proposed an SVD-based algorithm for solving the subset selection problem. The algorithm has two steps. Given a matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, at the first step, we compute the SVD of $\boldsymbol{A}$,

$$\boldsymbol{A} = \boldsymbol{U\Sigma V}^T, \quad \boldsymbol{V} = \begin{bmatrix} \boldsymbol{V}_{11} & \boldsymbol{V}_{12} \\ \boldsymbol{V}_{21} & \boldsymbol{V}_{22} \end{bmatrix}.$$

At the second step, we apply algorithm QR with column pivoting-I on matrix $\begin{bmatrix} \boldsymbol{V}_{11}^T & \boldsymbol{V}_{21}^T \end{bmatrix}$ and get a permutation matrix $\boldsymbol{\Pi}_{SVD}$. Then the first k columns of $\boldsymbol{A\Pi}_{SVD}$ give the SVD-based solution for the problem.

Theorem 6.3.1. *Let $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ be the SVD factorization of $\boldsymbol{A}$. Partition $\boldsymbol{A}\boldsymbol{\Pi}$ and $\boldsymbol{\Pi}^T\boldsymbol{V}$, where $\boldsymbol{\Pi}$ is a permutation matrix,*

$$\boldsymbol{A}\boldsymbol{\Pi} = \begin{bmatrix} \boldsymbol{B}_1 & \boldsymbol{B}_2 \end{bmatrix}, \quad \boldsymbol{\Pi}^T\boldsymbol{V} = \begin{bmatrix} \tilde{\boldsymbol{V}}_{11} & \tilde{\boldsymbol{V}}_{12} \\ \tilde{\boldsymbol{V}}_{21} & \tilde{\boldsymbol{V}}_{22} \end{bmatrix}.$$

If $\tilde{\boldsymbol{V}}_{11} \in \mathbb{R}^{k\times k}$ is nonsingular and $k \leq$ rank$(\boldsymbol{A})$, then

$$\frac{\sigma_k(\boldsymbol{A})}{\|\tilde{\boldsymbol{V}}_{11}^{-1}\|_2} \leq \sigma_k(\boldsymbol{B}_1) \leq \sigma_k(\boldsymbol{A}) \tag{6-11}$$

Proof. The interlacing property gives the upper bound of $\sigma_k(\boldsymbol{B}_1)$. Partition the diagonal matrix $\boldsymbol{\Sigma}$ as

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\Sigma}_2 \end{bmatrix},$$

where $\boldsymbol{\Sigma}_1$ is of order k. Now let $\boldsymbol{v} \in \mathbb{R}^k$ be the right singular vector of $\boldsymbol{B}_1$ corresponding to the least dominant singular value so that $\|\boldsymbol{B}_1\boldsymbol{v}\|_2 = \sigma_k(\boldsymbol{B}_1)$ and $\|\boldsymbol{v}\|_2 = 1$. Then we have

$$\sigma_k(\boldsymbol{B}_1)^2 = \|\boldsymbol{B}_1\boldsymbol{v}\|_2^2 = \left\| \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\boldsymbol{\Pi} \begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{0} \end{bmatrix} \right\|_2^2 = \|\boldsymbol{\Sigma}_1\tilde{\boldsymbol{V}}_{11}^T\boldsymbol{v}\|_2^2 + \|\boldsymbol{\Sigma}_2\tilde{\boldsymbol{V}}_{12}^T\boldsymbol{v}\|_2^2. \tag{6-12}$$

Notice that

$$\sigma_k(\boldsymbol{A}) = \|\boldsymbol{\Sigma}_1\boldsymbol{v}\|_2 = \|(\tilde{\boldsymbol{V}}_{11}^T)^{-1}\boldsymbol{\Sigma}_1\tilde{\boldsymbol{V}}_{11}^T\boldsymbol{w}\|_2 \leq \|\tilde{\boldsymbol{V}}_{11}^{-1}\|_2 \cdot \|\boldsymbol{\Sigma}_1\tilde{\boldsymbol{V}}_{11}^T\boldsymbol{w}\|_2 \tag{6-13}$$

Combining Equation (6-12) and Inequality (6-13) gives the lower bound

$$\frac{\sigma_k(\boldsymbol{A})}{\|\tilde{\boldsymbol{V}}_{11}^{-1}\|_2} \leq \sigma_k(\boldsymbol{B}_1)$$

□

Notice that the submatrix $\tilde{\boldsymbol{V}}_{11}^T$ can be computed by $\boldsymbol{V}_{11}^T\boldsymbol{\Pi} + \boldsymbol{V}_{21}^T\boldsymbol{\Pi}$. For the SVD-based algorithm, since $\boldsymbol{\Pi}$ is found by the QR with column-pivoting-I factorization of

$\begin{bmatrix} \boldsymbol{V}_{11}^T & \boldsymbol{V}_{21}^T \end{bmatrix}$, it ensures $\boldsymbol{V}_{11}^T\boldsymbol{\Pi} + \boldsymbol{V}_{21}^T\boldsymbol{\Pi}$ to be well-conditioned. Then by Theorem 6.3.1, we have a lower bound for the smallest singular value of the first k columns.

We can also use (strong) RRQR factorization to give a solution $\boldsymbol{\Pi}_{QR}$ for the subset selection problem. Suppose the (strong) RRQR factorization of the matrix $\boldsymbol{A}$ is given by

$$\boldsymbol{A}\boldsymbol{\Pi}_{QR} = \boldsymbol{Q}\boldsymbol{R}, \quad \boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix},$$

where $\boldsymbol{R}_{11}$ is an upper triangular matrix of order k. The permutation matrix $\boldsymbol{\Pi}_{QR}$ gives the RRQR-based solution for the problem.

To see how good the subspace is, we measure the principle angle between the subspace that the algorithm has selected with the subspace spanned by the k most dominant left singular vectors.

Theorem 6.3.2. *Let $\mathbb{V}_U$ denote the subspace spanned by the first k left singular vectors $\boldsymbol{u}_1, ..., \boldsymbol{u}_k$. Let $\mathbb{V}_{QR}$ denote the subspace spanned by the first k columns of $\boldsymbol{A}\boldsymbol{\Pi}_{QR}$ and let $\mathbb{V}_{SVD}$ denote the subspace spanned by the first k columns of $\boldsymbol{A}\boldsymbol{\Pi}_{SVD}$. Let $\theta(\mathbb{V}_1, \mathbb{V}_2)$ denote the principle angle between subspace $\mathbb{V}_1$ and $\mathbb{V}_2$. Then*

$$\begin{aligned} \sin(\theta(\mathbb{V}_U, \mathbb{V}_{SVD})) &\leq \frac{\sigma_{k+1}(\boldsymbol{A})}{\sigma_k(\boldsymbol{A})}\|\tilde{\boldsymbol{V}}_{11}^{-1}\|_2 \\ \sin(\theta(\mathbb{V}_U, \mathbb{V}_{QR})) &\leq \sigma_{k+1}(\boldsymbol{A})\|\boldsymbol{R}_{11}^{-1}\|_2 \end{aligned} \tag{6-14}$$

Proof. Let $\mathbb{V}_{U^*}$ denote the subspace spanned by the last $m-k$ left singular vectors $\boldsymbol{u}_{k+1}, ..., \boldsymbol{u}_m$. Since $\boldsymbol{U}$ is an orthogonal matrix, we have $\mathbb{V}_{U^*} \perp \mathbb{V}_U$. This gives $\sin(\theta(\mathbb{V}_U, \mathbb{V}_{SVD})) = \cos(\theta(\mathbb{V}_{U^*}, \mathbb{V}_{SVD}))$ and $\sin(\theta(\mathbb{V}_U, \mathbb{V}_{QR})) = \cos(\theta(\mathbb{V}_{U^*}, \mathbb{V}_{QR}))$. For simplicity, partition $\boldsymbol{U}$, $\boldsymbol{A}\boldsymbol{\Pi}_{SVD}$ as

$$\boldsymbol{U} = \begin{matrix} \\ \end{matrix}\begin{matrix} k & m-k \end{matrix}\begin{bmatrix} \boldsymbol{U}_1 & \boldsymbol{U}_2 \end{bmatrix}, \quad \boldsymbol{A}\boldsymbol{\Pi}_{SVD} = \begin{matrix} k & m-k \end{matrix}\begin{bmatrix} \boldsymbol{B}_1 & \boldsymbol{B}_2 \end{bmatrix}$$

Then we have

$$\begin{aligned}\cos(\theta(\mathbb{V}_{U^*}, \mathbb{V}_{SVD})) &\leq \left\| \frac{\boldsymbol{U}_2^T \boldsymbol{B}_1}{(\boldsymbol{B}_1^T \boldsymbol{B}_1)^{1/2}} \right\|_2 \\ &\leq \|\boldsymbol{U}_2^T \boldsymbol{B}_1\|_2 \|(\boldsymbol{B}_1^T \boldsymbol{B}_1)^{-1/2}\|_2\end{aligned} \tag{6-15}$$

Note that

$$\begin{aligned}\boldsymbol{U}_2^T \boldsymbol{A}\boldsymbol{\Pi}_{SVD} &= \boldsymbol{U}_2^T \begin{bmatrix}\boldsymbol{U}_1 & \boldsymbol{U}_2\end{bmatrix} \Sigma \boldsymbol{V}^T \Pi_{SVD} \\ &= \begin{bmatrix}\boldsymbol{0} & \boldsymbol{I}\end{bmatrix} \begin{bmatrix}\boldsymbol{\Sigma_1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\Sigma_2}\end{bmatrix} \boldsymbol{V}^T \Pi_{SVD} \\ &= \begin{bmatrix}\boldsymbol{0} & \boldsymbol{\Sigma}_2\end{bmatrix} \boldsymbol{V}^T \Pi_{SVD}.\end{aligned} \tag{6-16}$$

Inequality (6-15) can be written as

$$\cos(\theta(\mathbb{V}_{U^*}, \mathbb{V}_{SVD})) \leq \frac{\sigma_{k+1}(\boldsymbol{A})}{\sigma_k(\boldsymbol{B}_1)} \leq \frac{\sigma_{k+1}(\boldsymbol{A})}{\sigma_k(\boldsymbol{A})} \|\tilde{\boldsymbol{V}}_{11}^{-1}\|_2, \tag{6-17}$$

where the last inequality comes from Theorem 6.3.1. Let $\bar{\boldsymbol{B}}_1$ be the matrix formed by the first k columns of $\boldsymbol{A}\boldsymbol{\Pi}_{QR}$. We have

$$\bar{\boldsymbol{B}}_1 = \boldsymbol{Q} \begin{bmatrix}\boldsymbol{R}_{11} \\ \boldsymbol{0}\end{bmatrix}. \tag{6-18}$$

By the same argument, we have

$$\cos(\theta(\mathbb{V}_{U^*}, \mathbb{V}_{QR})) \leq \frac{\sigma_{k+1}(\boldsymbol{A})}{\sigma_k(\bar{\boldsymbol{B}}_1)} = \frac{\sigma_{k+1}(\boldsymbol{A})}{\sigma_k(\boldsymbol{R}_{11})} = \sigma_{k+1}(\boldsymbol{A}) \|\boldsymbol{R}_{11}^{-1}\|_2 \tag{6-19}$$

□

Since RRQR algorithms guarantee

$$\|\boldsymbol{R}_{11}^{-1}\|_2 = \frac{1}{\sigma_{\min}(\boldsymbol{R}_{11})} \leq \frac{q(k,n)}{\sigma_k(\boldsymbol{A})}.$$

Combine this with Theorem 6.3.2, we see the sine of both angles has the same order

as $\sigma_{k+1}(\boldsymbol{A})/\sigma_k(\boldsymbol{A})$. This means both subspace $\mathbb{V}_{SVD}$ and $\mathbb{V}_{QR}$ are close to $\mathbb{V}_U$ and therefore they represent approximately the same subspace, although they may not use the same permutation matrix.

Notice that the basic solution $\boldsymbol{x}_{BQR}$ in Section 6.2 is the least squares solution in the subspace $\mathbb{V}_{QR}$.

6.3.2 Numerical Experiment

The matrix $\boldsymbol{A}$ is selected according to the same criteria as in Section 6.2.2. $\mathbb{V}_{QR}$ is generated using Algorithm GSRRQR-2. I also define $\text{Sim}(\mathbb{V}_{SVD}, \mathbb{V}_{QR})$ to be the number of same columns these two algorithms have picked among the first k columns of $\boldsymbol{A}$. The numerical results are presented in Table 6.12.

Table 6.12 Sine of the angle between $\mathbb{V}_U$, $\mathbb{V}_{SVD}$ and $\mathbb{V}_{QR}$

$n = 500, k = 200$			Results		
σ_1	σ_{200}	$\sigma_{201}, ..., \sigma_{500}$	$\sin(\theta(\mathbb{V}_U, \mathbb{V}_{SVD}))$	$\sin(\theta(\mathbb{V}_U, \mathbb{V}_{QR}))$	$\text{Sim}(\mathbb{V}_{SVD}, \mathbb{V}_{QR})$
1000	1	10^{-1}	0.5838	0.8375	99
1000	1	10^{-4}	4.3261e-04	0.0029	82
1000	1	10^{-7}	3.0032e-07	9.9717e-06	79
$n = 500, k = 400$			Results		
σ_1	σ_{400}	$\sigma_{401}, ..., \sigma_{500}$	$\sin(\theta(\mathbb{V}_U, \mathbb{V}_{SVD}))$	$\sin(\theta(\mathbb{V}_U, \mathbb{V}_{QR}))$	$\text{Sim}(\mathbb{V}_{SVD}, \mathbb{V}_{QR})$
1000	1	10^{-1}	0.2935	0.9710	330
1000	1	10^{-4}	2.3591e-04	0.0035	318
1000	1	10^{-7}	2.1639e-07	2.1654e-06	318

Table 6.13 Time for computing $\mathbb{V}_{SVD}$ and $\mathbb{V}_{QR}$

$n = 500, k = 200$			Time (s)	
σ_1	σ_{200}	$\sigma_{201}, ..., \sigma_{500}$	$\mathbb{V}_{QR}$	$\mathbb{V}_{SVD}$
1000	1	10^{-1}	0.9417	5.4794
1000	1	10^{-4}	1.8715	5.4611
1000	1	10^{-7}	1.7896	5.4190
$n = 500, k = 400$			Time (s)	
σ_1	σ_{400}	$\sigma_{401}, ..., \sigma_{500}$	$\mathbb{V}_{QR}$	$\mathbb{V}_{SVD}$
1000	1	10^{-1}	1.4389	7.7955
1000	1	10^{-4}	2.1173	7.6238
1000	1	10^{-7}	2.2196	7.6246

The results accord with our former analysis, that the sine of both angles is dominated $\sigma_{k+1}(\boldsymbol{A})/\sigma_k(\boldsymbol{A})$, though the theoretical upper bound of SRRQR algorithms depend on k. Also, the columns these two algorithms picked are different. An interesting observation is that the subspace $\mathbb{V}_{QR}$ is closer to $\mathbb{V}_U$ in all the results. This means in all our numerical results, the RRQR-based solution perform better than the SVD-based solution. This is not that surprising. Notice that the SVD-based solution use QR with column pivoting to find permutation matrix and we've already seen in Section 6.1 that algorithm SRRQR-1 outperforms QR with column pivoting. This may be the reason for why RRQR-based solution performs better than the SVD-based solution.

Also, note that the SVD solution is generated by first running truncated SVD and then apply QR with column pivoting. We can see from Table 6.13 that computing $\mathbb{V}_{SVD}$ costs three times more than computing $\mathbb{V}_{QR}$ by using algorithm GSRRQR-2 for matrix of size 500×500.

6.4 Matrix Approximation and Image Compression

6.4.1 Theoretical Analysis

In this section, we discuss the matrix approximation problem, which has potential application to image compression. Given a matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, we want to find $\boldsymbol{A}_k$ with rank k such that $\|\boldsymbol{A} - \boldsymbol{A}_k\|_2$ is minimized.

This problem is solved by truncated SVD,

$$\boldsymbol{A}_k = \sum_{i=1}^{k} \boldsymbol{u}_i \sigma_i \boldsymbol{v}_i^T$$

And the solution gives

$$\|\boldsymbol{A} - \boldsymbol{A}_k\|_2 = \sigma_{k+1}$$

Suppose $\boldsymbol{A\Pi} = \boldsymbol{QR}$ is an RRQR factorization of $\boldsymbol{A}$. Consider the approximation

given by truncating the RRQR factorization,

$$\boldsymbol{B}_k = \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \boldsymbol{\Pi}^T.$$

Theorem 6.4.1. *For $\boldsymbol{B}_k$ defined as above, we have*

$$\|\boldsymbol{A} - \boldsymbol{B}_k\|_2 \leq \sigma_{k+1}(\boldsymbol{A}) q_2(k, n),$$

where $q_2(k, n)$ is given by the RRQR algorithm.

Proof. Since

$$\|\boldsymbol{A} - \boldsymbol{B}_k\|_2 = \left\| \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix} \boldsymbol{\Pi}^T - \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \boldsymbol{\Pi}^T \right\|_2 = \|\boldsymbol{R}_{22}\|_2,$$

and since RRQR algorithm promises

$$\|\boldsymbol{R}_{22}\|_2 \leq \sigma_{k+1}(\boldsymbol{A}) q_2(k, n),$$

we have

$$\|\boldsymbol{A} - \boldsymbol{B}_k\|_2 \leq \sigma_{k+1}(\boldsymbol{A}) q_2(k, n).$$

□

From Theorem 6.4.1, we see that RRQR-based algorithm also gives a good rank-k approximation. From triangular inequality, we see the difference $\|\boldsymbol{A}_k - \boldsymbol{B}_k\|_2$ is bounded by $(1 + q_2(k, n))\sigma_{k+1}(\boldsymbol{A})$. When $\sigma_{k+1}(\boldsymbol{A})$ is sufficiently small, the difference between $\boldsymbol{A}_k$ and $\boldsymbol{B}_k$ is negligible. Also since the bound does not depend on the gap between $\sigma_k(\boldsymbol{A})$ and $\sigma_{k+1}(\boldsymbol{A})$, the RRQR-based algorithm works for any matrix independently of its singular values. This gives it the potential for application in image compressing.

6.4.2 Numerical Experiment

We choose the same matrix $\boldsymbol{A}$ as in Section 6.2.2 and select GSRRQR-2 as the rank revealing algorithm. Table 6.14 shows the numerical results.

Table 6.14 Matrix approximation using SVD, GSRRQR-2

Matrix $n = 500, k = 200$			Error		Time (s)	
σ_1	σ_{200}	$\sigma_{201}, ..., \sigma_{500}$	$\|\boldsymbol{A} - \boldsymbol{B}_k\|_2$	$\|\boldsymbol{A} - \boldsymbol{A}_k\|_2$	$\boldsymbol{B}_k$	$\boldsymbol{A}_k$
1000	1	10^{-1}	2.0257	0.1000	1.1274	5.3695
1000	1	10^{-4}	0.0018	1.0000e-04	1.6541	5.3798
1000	1	10^{-7}	1.9152e-06	1.0000e-07	1.7952	5.3417
Matrix $n = 500, k = 400$			Error		Time (s)	
σ_1	σ_{400}	$\sigma_{401}, ..., \sigma_{500}$	$\|\boldsymbol{A} - \boldsymbol{B}_k\|_2$	$\|\boldsymbol{A} - \boldsymbol{A}_k\|_2$	$\boldsymbol{B}_k$	$\boldsymbol{A}_k$
1000	1	10^{-1}	1.6829	0.1000	1.9864	7.4138
1000	1	10^{-4}	0.0017	1.0000e-04	2.1029	7.4044
1000	1	10^{-7}	1.6289e-06	1.0000e-07	2.0819	7.2584

We can see that the approximation error for RRQR-based solution is about ten times σ_{k+1}. So as long as σ_{k+1}, the approximation is good. Also we see, computing an RRQR-based solution is much faster than computing an SVD-based solution.

Now let's see the application of rank revealing algorithms in image compression. The origin image is shown in Figure 6.1. In Section 6.1, we see all strong RRQR algorithms will guarantee a good RRQR factorization. In practice, what we really care is their time efficiency. Table 6.15 shows the time cost of these strong RRQR algorithms under different settings of k. We can see that in practice, greedy SRRQR algorithms run significantly faster than the origin SRRQR algorithms.

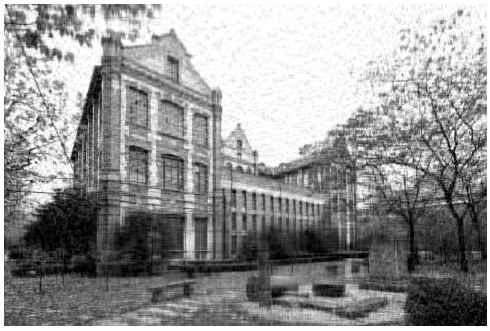
Figure 6.1 Origin figure (608×1024)

Table 6.15 Time efficiency of strong RRQR algorithms on Figure 6.1

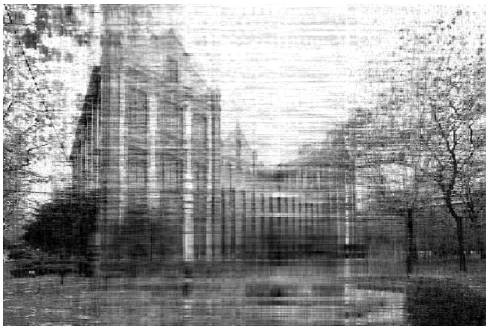
	Time (s)	Number of Iterations
SVD	32.2644	
$k = 50$	Time (s)	Number of Iterations
SRRQR-1	5.6498	370
GSRRQR-1	1.0632	68
SRRQR-2	3.6098	313
GSRRQR-2	0.8495	42
$k = 100$	Time (s)	Number of Iterations
SRRQR-1	8.6829	626
GSRRQR-1	1.9944	117
SRRQR-2	7.6097	512
GSRRQR-2	1.8059	74
$k = 150$	Time (s)	Number of Iterations
SRRQR-1	14.9104	933
GSRRQR-1	3.0059	179
SRRQR-2	10.9214	752
GSRRQR-2	2.3592	112
$k = 250$	Time (s)	Number of Iterations
SRRQR-1	31.6790	1480
GSRRQR-1	5.1366	249
SRRQR-2	22.6242	1155
GSRRQR-2	3.9903	179

Figure 6.2 shows the compressed image using SVD and GSRRQR-2.

As expected, we see when k is small, the RRQR-based approximation is bad. As we increase k, the difference between the RRQR-based approximation and the SVD-based approximation becomes negligible. Since we typically don't want to loose much information after compression which means k is not too small, rank revealing algorithm can be used as a more efficient alternative to SVD.

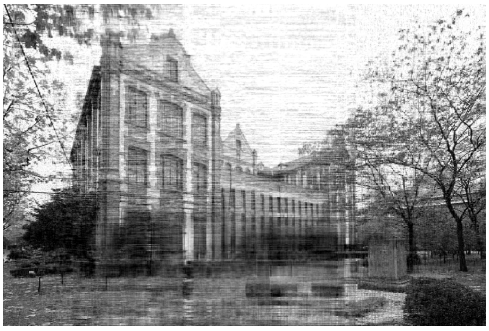


(a) $k = 50$**, SVD**



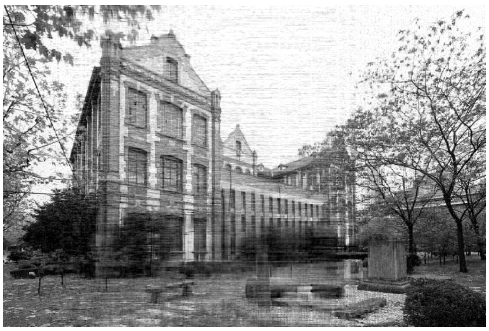
(b) $k = 50$**, GSRRQR-2**

(c) $k = 100$**, SVD**



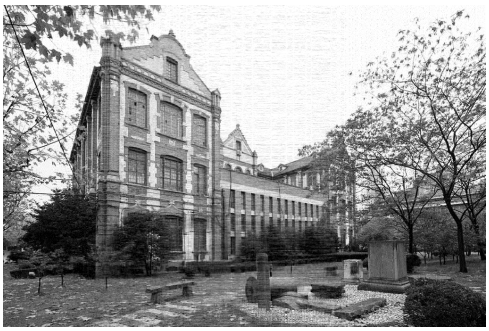
(d) $k = 100$**, GSRRQR-2**

(e) $k = 150$**, SVD**



(f) $k = 150$**, GSRRQR-2**

(g) $k = 250$**, SVD**



(h) $k = 250$**, GSRRQR-2**

Figure 6.2 Image compression using SVD and GSRRQR-2

Chapter 7 Conclusions

7.1 Summary of the Thesis

This thesis focuses on the rank revealing QR (RRQR) factorization and aims to give a survey of almost all famous algorithms for computing the RRQR factorization. This is the *first* literature that compares these algorithms both analytically and numerically. In this thesis, I propose a greedy strong RRQR (GSRRQR) algorithm for computing a strong RRQR factorization. This thesis also discusses several potential applications of the RRQR factorization.

I designed numerical experiments on matrices of different kinds and different sizes and the results show that the new proposed algorithm, GSRRQR, runs significantly faster than the origin SRRQR algorithm. Also, GSRRQR algorithm guarantees the same analytical bounds as the origin SRRQR algorithm. So I suggest that one should use GSRRQR algorithm as the default algorithm for computing a strong RRQR factorization.

Gu ([2], 1996) and Ipsen ([3], 1994) suggested that using algorithm QR with column pivoting to initialize the given matrix could improve the time efficiency of hybrid and strong RRQR algorithms. However, since GSRRQR algorithm is embedded with the greedy strategy, this initialization step is useless for GSRRQR.

Besides algorithm GSRRQR, I also complete some theoretical work in this thesis. I extends several theorems and completes some parts of the theoretical analysis independently (see Section 1.2 for detail). I also corrects some error from previous literatures.

RRQR factorization are designed to substitute SVD in rank deficient problems.

For rank deficient least squares, we see that not only the 2-norm of the residuals of RRQR-based solution and SVD-based solution are similar, but also the difference between RRQR-based solution and SVD-based solution is negligible as the system goes more and more ill-conditioned.

For subset selection problem, by using algorithm GSRRQR-2, we see the RRQR-

based solution gives a better result than the SVD-based solution while they are much more efficient to compute.

For matrix rank-k approximation, the difference between the approximation error of RRQR-based approximation and SVD-based approximation has the same order as the $k+1$st largest singular value of the origin matrix.

7.2 Future Work

Numerical results show algorithm GSRRQR runs much faster than the origin SRRQR. However, we still lack a theoretical analysis for its time efficiency. How can we bound the number of iterations that algorithm GSRRQR will perform by some formula related to the given matrix? It's hard to show this bound since till now, we still don't have a bound for the number of iterations that hybrid algorithms will perform.

Greedy strong RRQR algorithms improve the time efficiency of the origin SRRQR algorithms by first consider the greedy i-j pair. However, when dealing with a sparse matrix, we really want to maintain the sparsity of the matrix at each iteration. GSRRQR algorithms cannot guarantee this, neither can these hybrid and strong RRQR algorithms presented in this thesis.

Note, by the halting argument of strong RRQR algorithm only require that the determinant of the $k \times k$ principal submatrix should increase strictly at each step. We can compromise the selection of the pivoting column as long as it satisfies this condition. To find such column that also maintains sparsity of the matrix is the research with lots of potential at this current stage.

Parallel implementation is of great importance for matrices of very large scale. Householder transformation can be extended into block Householder transformation which enables parallel implementation of QR factorization. It's not clear how to modify the hybrid and strong RRQR algorithms so that they can be implemented in parallel. This is another potential area of research.

REFERENCES

[1] Tony F Chan and Per Christian Hansen. Some applications of the rank revealing qr factorization. *SIAM Journal on Scientific and Statistical Computing*, 13(3):727–741, 1992.

[2] Ming Gu and Stanley C Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, 1996.

[3] Shivkumar Chandrasekaran and Ilse CF Ipsen. On rank-revealing factorisations. *SIAM Journal on Matrix Analysis and Applications*, 15(2):592–622, 1994.

[4] John GF Francis. The qr transformation a unitary analogue to the lr transformationpart 1. *The Computer Journal*, 4(3):265–271, 1961.

[5] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[6] Alan M Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[7] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[8] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[9] Gene Golub. Numerical methods for solving linear least squares problems. *Numerische Mathematik*, 7(3):206–216, 1965.

[10] Peter Businger and Gene H Golub. Linear least squares solutions by householder transformations. *Numerische Mathematik*, 7(3):269–276, 1965.

[11] William B Gragg and Gilbert W Stewart. A stable variant of the secant method for solving nonlinear equations. *SIAM Journal on Numerical Analysis*, 13(6):889–903, 1976.

[12] Tony F Chan. Rank revealing QR factorizations. *Linear Algebra and its Applications*, 88-89:67–82, 1987.

[13] William Kahan. Numerical linear algebra. *Canadian Math. Bulletin*, 9(757-801):103, 1966.

[14] Yoo Pyo Hong and C-T Pan. Rank-revealing factorizations and the singular value decomposition. *Mathematics of Computation*, 58(197):213–232, 1992.

[15] Tony F Chan. On the existence and computation of -factorizations with small pivots. *Mathematics of computation*, 42(166):535–547, 1984.

[16] Tsung-Min Hwang, Wen-Wei Lin, and Eugene K Yang. Rank revealing LU factorizations. *Linear Algebra and its Applications*, 175:115–141, 1992.

[17] L Miranian and M Gu. Strong rank revealing LU factorizations. *Linear Algebra and its Applications*, 367:1–16, 2003.

[18] VE Kane, RC Ward, and GJ Davis. Assessment of linear dependencies in multivariate data. *SIAM Journal on Scientific and statistical computing*, 6(4):1022–1032, 1985.

[19] Sabine Van Huffel and Joos Vandewalle. Subset selection using the total least squares approach in collinearity problems with errors in the variables. *Linear algebra and its applications*, 88:695–714, 1987.

[20] Christian H Bischof, John G Lewis, and Daniel J Pierce. Incremental condition estimation for sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 11(4):644–659, 1990.

[21] Pierre Comon and Gene H Golub. Tracking a few extreme singular values and vectors in signal processing. *Proceedings of the IEEE*, 78(8):1327–1343, 1990.

[22] Gene Golub, Virginia Klema, and Gilbert W Stewart. Rank degeneracy and least squares problems. Technical report, DTIC Document, 1976.

[23] Leslie V Foster. Rank and null space calculations using matrix decomposition without column interchanges. *Linear Algebra and its Applications*, 74:47–71, 1986.

[24] Per Christian Hansen. The truncatedsvd as a method for regularization. *BIT Numerical Mathematics*, 27(4):534–553, 1987.

[25] Åke Björck. Least squares methods. *Handbook of numerical analysis*, 1:465–652, 1990.

ACKNOWLEDGEMENT

It has been such a long journey and an extraordinary experience working on this graduation research with all the people that have helped me. I would mainly like to acknowledge my advisor Prof. Zengqi Wang for the continuous support of my study and research. Although I was studying abroad during the whole semester and it was sometimes inconvenient to keep in touch by email, she was always so nice with patience and always willing to lend a helping hand.

I also want to thank the rest of my thesis committee: Prof. Xiaoming Wang and Prof. Jinyan Fan for their insightful comments and questions.

I thank Prof. Shi Jin, my advisor in University of Wisconsin, and my dear friend, Yisi Liu. I could not have imagined how I can tild over this busy semester without their encouragement.

I would like to express my sincere gratitude to Prof. Amos Ron in University of Wisconsin for leading me to the field of numerical linear algebra.

Last but not the least, I would like to thank my family: my parents Junzhou Bao and Xibei Huo, for giving birth to me at the first place and supporting me spiritually throughout my life.

Appendix A MATLAB Code for RRQR Factorization

A.1 QR Factorization

A.1.1 Householder Transformation

```
function v=house(x)
    v = x;
    if x(1) > 0;
        v(1) = v(1)+norm(x);
    else
        v(1) = v(1)-norm(x);
    end
end
```

house.m

A.1.2 Givens Rotation

```
function [c,s] = givens(a,b)
    if abs(a) > abs(b)
        t = b/a;
        c = 1/sqrt(1+t^2);
        s = t*c;
    else
        t = a/b;
        s = 1/sqrt(1+t^2);
        c = t*s;
    end
end
```

givens.m

A.1.3 Householder QR

```
function [Q,R]=QR_Householder(A)
    [m,n] = size(A);
    Q = eye(m);
    R = A;
    for j = 1:n,
```

```
        v = house(R(j:m,j));
        c = 2/(v'*v);
        R(j:m,:) = R(j:m,:) - c*v*(v'*R(j:m,:));
        Q(:,j:m) = Q(:,j:m) - c*(Q(:,j:m)*v)*v';
    end
end
```

QR_Householder.m

```
function [Q,R]=TQR_Householder(A,k)
    [m,n] = size(A);
    Q = eye(m);
    R = A;
    for j = 1:k,
        v = house(R(j:m,j));
        c = 2/(v'*v);
        R(j:m,:) = R(j:m,:) - c*v*(v'*R(j:m,:));
        Q(:,j:m) = Q(:,j:m) - c*(Q(:,j:m)*v)*v';
    end
end
```

TQR_Householder.m

A.1.4 Givens QR

```
function [Q,R]=QR_Givens(A)
    [m,n] = size(A);
    Q = eye(m);
    R = A;
    for j = 1:n
        for i = m:-1:j+1
            [c,s] = givens(R(i-1,j),R(i,j));
            G = [c, -s; s, c];
            R([i-1,i],j:n) = G'*R([i-1,i],j:n);
            Q(:,[i-1,i]) = Q(:,[i-1,i])*G;
        end
    end
end
```

QR_Givens.m

A.2 Greedy RRQR Factorization

A.2.1 Power Method

```
function [lambda, vnew] = powerMethod(A, delta)
    vold = rand(length(A),1);
    vnew = A*vold;
    lambda = vold'*vnew/norm(vold);
    while norm(vnew-lambda*vold)/norm(vnew) > delta
        vnew = vnew/norm(vnew);
        vold = vnew;
        vnew = A*vold;
        lambda = vold'*vnew/(vold'*vold);
    end
    vnew = vnew/norm(vnew);
end
```

powerMethod.m

A.2.2 Inverse Power Method

```
function [lambda, vnew] = inversePowerMethod(A, delta)
    [L,U,P] = lu(A); %PA=LU
    vold = rand(length(A),1);
    tmp = L\(P*vold);
    vnew = U\tmp;
    lambda = vold'*vnew/norm(vold);
    while norm(vnew-lambda*vold)/norm(vnew) > delta
        vnew = vnew/norm(vnew);
        vold = vnew;
        tmp = L\(P*vold);
        vnew = U\tmp;
        lambda = vold'*vnew/(vold'*vold);
    end
    vnew = vnew/norm(vnew);
    lambda = 1/lambda;
end
```

inversePowerMethod.m

A.2.3 Algorithm Greedy-I.1

```
function [Q, R, Pi] = QR_Greedy1(M, k)
    [m,n] = size(M);
    Pi = eye(n);
    Q = eye(m);
    R = M;
    p = 1:n;
    gamma = sqrt(sum(R.^2,1));
    for l = 0:k-1
        lambda_max = realmin;
        for i = 1:n-l
            tmp = [R(1:l,1:l),R(1:l,l+i);zeros(1,l),gamma(i)];
            [lambda,~] = inversePowerMethod(tmp'*tmp, 0.01);
            if (lambda > lambda_max)
                lambda_max = lambda;
                j = i;
            end
        end
        p([l+1,l+j]) = p([l+j,l+1]);
        R(:,[l+1,l+j]) = R(:,[l+j,l+1]);
        v = house(R(l+1:m,l+1));
        c = 2/(v'*v);
        R(l+1:m,:) = R(l+1:m,:) - c*v*(v'*R(l+1:m,:));
        Q(:,l+1:m) = Q(:,l+1:m) - c*(Q(:,l+1:m)*v)*v';
        gamma = gamma(2:end);
        gamma = sqrt(gamma.^2-R(l+1,l+2:n).^2);
    end
    Pi = Pi(:,p);
end
```

QR_Greedy1.m

A.2.4 Algorithm Greedy-I.2

```
function [Q, R, Pi] = QR_Greedy2(M, k)
    [m,n] = size(M);
    Pi = eye(n);
    Q = eye(m);
    R = M;
    p = 1:n;
    gamma = sqrt(sum(R.^2,1));
    invA = [];
    for l = 0:k-1
        lambda_max = realmin;
        for i = 1:n-l
```

```
            tmp = [invA, -invA*R(1:l,i+l)/gamma(i);zeros(1,l),1/gamma
    (i)];
            lambda = 1/max(sqrt(sum(tmp.^2,2)));
            if (lambda > lambda_max)
                lambda_max = lambda;
                j = i;
            end
        end
        invA = [invA, -invA*R(1:l,j+l)/gamma(i);zeros(1,l),1/gamma(j)
    ];
        p([l+1,l+j]) = p([l+j,l+1]);
        R(:,[l+1,l+j]) = R(:,[l+j,l+1]);
        v = house(R(l+1:m,l+1));
        c = 2/(v'*v);
        R(l+1:m,:) = R(l+1:m,:) - c*v*(v'*R(l+1:m,:));
        Q(:,l+1:m) = Q(:,l+1:m) - c*(Q(:,l+1:m)*v)*v';
        gamma = gamma(2:end);
        gamma = sqrt(gamma.^2-R(l+1,l+2:n).^2);
    end
    Pi = Pi(:,p);
end
```

QR_Greedy2.m

A.2.5 Algorithm Greedy-I.3

```
function [Q, R, Pi] = QR_Greedy3(M, k)
    [m,n] = size(M);
    Pi = eye(n);
    Q = eye(m);
    R = M;
    p = 1:n;
    gamma = sqrt(sum(R.^2,1));
    invA = [];
    for l = 0:k-1
        lambda_max = realmin;
        for i = 1:n-l
            tmp = abs([-invA*R(1:l,i+l)/gamma(i);1/gamma(i)]);
            lambda = 1/max(tmp);
            if (lambda > lambda_max)
                lambda_max = lambda;
                j = i;
            end
        end
```

```
        invA = [invA, -invA*R(1:l,j+l)/gamma(i);zeros(1,l),1/gamma(j)
    ];
        p([l+1,l+j]) = p([l+j,l+1]);
        R(:,[l+1,l+j]) = R(:,[l+j,l+1]);
        v = house(R(l+1:m,l+1));
        c = 2/(v'*v);
        R(l+1:m,:) = R(l+1:m,:) - c*v*(v'*R(l+1:m,:));
        Q(:,l+1:m) = Q(:,l+1:m) - c*(Q(:,l+1:m)*v)*v';
        gamma = gamma(2:end);
        gamma = sqrt(gamma.^2-R(l+1,l+2:n).^2);
    end
    Pi = Pi(:,p);
end
```

QR_Greedy3.m

A.2.6 Algorithm QR with Column Pivoting

```
function [Q, R, Pi] = QR_ColumnPivoting(M, k)
    [m,n] = size(M);
    Pi = eye(n);
    Q = eye(m);
    R = M;
    p = 1:n;
    gamma = sqrt(sum(R.^2,1));
    for l = 0:k-1
        [~,j] = max(gamma);
        p([l+1,l+j]) = p([l+j,l+1]);
        R(:,[l+1,l+j]) = R(:,[l+j,l+1]);
        v = house(R(l+1:m,l+1));
        c = 2/(v'*v);
        R(l+1:m,:) = R(l+1:m,:) - c*v*(v'*R(l+1:m,:));
        Q(:,l+1:m) = Q(:,l+1:m) - c*(Q(:,l+1:m)*v)*v';
        gamma = gamma(2:end);
        gamma = sqrt(gamma.^2-R(l+1,l+2:n).^2);
    end
    Pi = Pi(:,p);
end
```

QR_ColumnPivoting.m

A.2.7 Algorithm Chan

```
function [Q, R, Pi] = QR_Chan(M, k)
    [m,n] = size(M);
    Pi = eye(n);
    Q = eye(m);
    R = M;
    p = 1:n;
    for l = 0:k-1
        C = R(l+1:m,l+1:n);
        [~,v] = powerMethod(C'*C,0.01);
        [~,j] = max(abs(v));
        p([l+1,l+j]) = p([l+j,l+1]);
        R(:,[l+1,l+j]) = R(:,[l+j,l+1]);
        v = house(R(l+1:m,l+1));
        c = 2/(v'*v);
        R(l+1:m,:) = R(l+1:m,:) - c*v*(v'*R(l+1:m,:));
        Q(:,l+1:m) = Q(:,l+1:m) - c*(Q(:,l+1:m)*v)*v';
    end
    Pi = Pi(:,p);
end
```

QR_Chan.m

A.2.8 Algorithm GKS

```
function varargout = mySVD(A,varargin)
% Author:       Brian Moore
  MAX_SVD_ITER = 30;
  % Input check
  if nargin > 1
      mode = varargin{1};
  else
      mode = 'full';
  end
  [m n] = size(A);
  if (n > m)
      [Ut St Vt] = mySVD(A',mode);
      if (nargout == 3)
          varargout{1} = Vt;
          varargout{2} = St';
          varargout{3} = Ut;
      else
          if (min(size(St)) == 1)
            varargout{1} = St(1,1);
          else
```

```
            varargout{1} = diag(St);
        end
    end
    return;
end

% Initialize variables
Ufull = zeros(m);
Ufull(1:m,1:n) = A;
svals = zeros(n,1);
Vfull = zeros(n);
vect = zeros(n,1);
l = 0;
mn = 0;
g = 0;
scale = 0.0;
norm = 0.0;

% Householder reduction to bidiagonal form
for i = 1:n
  l = i + 1;
  vect(i) = scale * g;
  g = 0.0;
  s = 0.0;
  scale = 0.0;
  for k = i:m
      scale = scale + abs(Ufull(k, i));
  end
  if (scale ~= 0)
    for k = i:m
      Ufull(k, i) = Ufull(k, i) / scale;
      s = s + Ufull(k, i) * Ufull(k, i);
    end
    f = Ufull(i, i);
    g = -sqrt(s) * sign(f);
    h = f * g - s;
    Ufull(i, i) = f - g;
    for j = l:n
      s = 0.0;
      for k = i:m
          s = s + Ufull(k, i) * Ufull(k, j);
      end
      f = s / h;
      for k = i:m
          Ufull(k, j) = Ufull(k, j) + f * Ufull(k, i);
```

```
      end
    end
    for k = i:m
        Ufull(k, i) = Ufull(k, i) * scale;
    end
  end
  svals(i) = scale * g;
  g = 0.0;
  s = 0.0;
  scale = 0.0;
  for k = l:n
      scale = scale + abs(Ufull(i, k));
  end
  if (scale ~= 0)
    for k = l:n
      Ufull(i, k) = Ufull(i, k) / scale;
      s = s + Ufull(i, k) * Ufull(i, k);
    end
    f = Ufull(i, l);
    g = -sqrt(s) * sign(f);
    h = f * g - s;
    Ufull(i, l) = f - g;
    for k = l:n
        vect(k) = Ufull(i, k) / h;
    end
    for j = l:m
      s = 0.0;
      for k = l:n
          s = s + Ufull(j, k) * Ufull(i, k);
      end
      for k = l:n
          Ufull(j, k) = Ufull(j, k) + s * vect(k);
      end
    end
    for k = l:n
        Ufull(i, k) = Ufull(i, k) * scale;
    end
  end
  norm = max(norm, (abs(svals(i)) + abs(vect(i))));
end

% Accumulate right-hand transformations
for i = n:-1:1
  if (g ~= 0)
    % Double division to avoid possible underflow
```

```
      for j = l:n
          Vfull(j, i) = (Ufull(i, j) / Ufull(i, l)) / g;
      end
      for j = l:n
        s = 0.0;
        for k = 1:n
            s = s + Ufull(i, k) * Vfull(k, j);
        end
        for k = l:n
            Vfull(k, j) = Vfull(k, j) + s * Vfull(k, i);
        end
      end
    end
    for j = l:n
        Vfull(i, j) = 0.0;
        Vfull(j, i) = 0.0;
    end
    Vfull(i, i) = 1.0;
    g = vect(i);
    l = i;
  end

  % Accumulate left-hand transformations
  for i = n:-1:1
    l = i + 1;
    g = svals(i);
    for j = l:n
        Ufull(i, j) = 0.0;
    end
    if (g ~= 0)
      g = 1.0 / g;
      for j = l:n
        s = 0.0;
        for k = l:m
            s = s + Ufull(k, i) * Ufull(k, j);
        end
        f = (s / Ufull(i, i)) * g;
        for k = i:m
            Ufull(k, j) = Ufull(k, j) + f * Ufull(k, i);
        end
      end
      for j = i:m
          Ufull(j, i) = Ufull(j, i) * g;
      end
    else
```

```
      for j = i:m
          Ufull(j, i) = 0.0;
      end
    end
    Ufull(i, i) = Ufull(i, i) + 1;
  end

  % Diagonalize the bidiagonal form
  for k = n:-1:1 % loop over all singular values
    for iters = 1:MAX_SVD_ITER % loop over allowed iterations
      flag = 1;
      % Test for splitting
      for l = k:-1:1
        mn = l - 1;
        % Note: vect(1) = 0, always
        if ((abs(vect(l)) + norm) == norm)
          flag = 0;
          break;
        end
        if ((abs(svals(mn)) + norm) == norm)
            break;
        end
      end
      if (flag ~= 0)
        % Cancel vect(l), l > 1
        c = 0.0;
        s = 1.0;
        for i = l:k
          f = s * vect(i);
          vect(i) = c * vect(i);
          if ((abs(f) + norm) == norm)
              break;
          end
          g = svals(i);
          h = SafeDistance(f, g);
          svals(i) = h;
          h = 1.0 / h;
          c = g * h;
          s = -f * h;
          for j = 1:m
            y = Ufull(j, mn);
            z = Ufull(j, i);
            Ufull(j, mn) = y * c + z * s;
            Ufull(j, i) = z * c - y * s;
          end
```

```
201       end
202     end
203     z = svals(k);
204     if (l == k) % We converged!
205       % Make singular value nonnegative
206       if (z < 0.0)
207         svals(k) = -z;
208         for j = 1:n
209             Vfull(j, k) = -Vfull(j, k);
210         end
211       end
212       break;
213     end
214     if (iters == MAX_SVD_ITER)
215       disp(['mySVD() reached maximum number of iterations: '
    num2str(MAX_SVD_ITER)]);
216     end
217
218     % Shift from bottom 2 x 2 minor
219     x = svals(l);
220     mn = k - 1;
221     y = svals(mn);
222     g = vect(mn);
223     h = vect(k);
224     f = ((y - z) * (y + z) + (g - h) * (g + h)) / (2.0 * h * y);
225     g = SafeDistance(f, 1.0);
226     f = ((x - z) * (x + z) + h * ((y / (f + abs(g) * sign(f))) - h)
    ) / x;
227
228     % Perform next QR decomposition
229     c = 1.0;
230     s = 1.0;
231     for j = l:mn
232       i = j + 1;
233       g = vect(i);
234       y = svals(i);
235       h = s * g;
236       g = c * g;
237       z = SafeDistance(f, h);
238       vect(j) = z;
239       c = f / z;
240       s = h / z;
241       f = x * c + g * s;
242       g = g * c - x * s;
243       h = y * s;
```

```
        y = y * c;
        for jj = 1:n
          x = Vfull(jj, j);
          z = Vfull(jj, i);
          Vfull(jj, j) = x * c + z * s;
          Vfull(jj, i) = z * c - x * s;
        end
        z = SafeDistance(f, h);
        svals(j) = z;
        if (z ~= 0) % Note: Rotation can be arbitrary if z = 0
          z = 1.0 / z;
          c = f * z;
          s = h * z;
        end
        f = c * g + s * y;
        x = c * y - s * g;
        for jj = 1:m
          y = Ufull(jj, j);
          z = Ufull(jj, i);
          Ufull(jj, j) = y * c + z * s;
          Ufull(jj, i) = z * c - y * s;
        end
      end
      vect(l) = 0.0;
      vect(k) = f;
      svals(k) = x;
    end
  end

  if strcmpi(mode,'compact')
    % Compute singular value tolerance
    SVD_TOL = max(m,n) * max(svals) * eps(class(A));

   % Determine rank to working tolerance
    r = 0;
    for i = 1:n
      if (svals(i) > SVD_TOL)
        r = r + 1;
      end
    end

    % Sort ascending
    [svals inds] = sort(svals);

    % Return compact SVD
```

```
    if (nargout ~= 3)
        varargout{1} = svals(n:-1:(n-r+1));
    else
        % Populate compact r-dimensional entries
        U = zeros(m, r);
        S = zeros(r);
        V = zeros(n, r);
        for i = 1:r
          U(:, i) = Ufull(:, inds(n + 1 - i));
          S(i, i) = svals(n + 1 - i);
          V(:, i) = Vfull(:, inds(n + 1 - i));
        end

        % Return SVD matrices
        varargout{1} = U;
        varargout{2} = S;
        varargout{3} = V;
    end
  else
    % Sort ascending
    [svals inds] = sort(svals);

    % Return full SVD
    if (nargout ~= 3)
        varargout{1} = svals(n:-1:1);
    else
        % Populate full n-dimensional entries
        U = zeros(m);
        S = zeros(m,n);
        V = zeros(n);
        for i = 1:n
          U(:, i) = Ufull(:, inds(n + 1 - i));
          S(i, i) = svals(n + 1 - i);
          V(:, i) = Vfull(:, inds(n + 1 - i));
        end

        % Return SVD matrices
        varargout{1} = U;
        varargout{2} = S;
        varargout{3} = V;
    end
  end
end

function dist = SafeDistance(a,b)
```

```
  abs_a = abs(a);
  abs_b = abs(b);
  if (abs_a > abs_b)
    dist = abs_a * sqrt(1.0 + (abs_b / abs_a)^2);
  else
    if (abs_b == 0)
      dist = 0;
    else
      dist = abs_b * sqrt(1.0 + (abs_a / abs_b)^2);
    end
  end
end
```

mySVD.m

```
function [Q, R, Pi] = QR_GKS(M, k)
    [~,~,V] = mySVD(M);
    [~,~,Pi] = QR_ColumnPivoting(V(:,1:k)',k);
    [Q,R] = QR_Householder(M*Pi);
end
```

QR_GKS.m

A.2.9 Algorithm Foster

```
function [Q, R, Pi] = QR_Foster(M, k)
    delta = 0.1; %tolerance
    [m,n] = size(M);
    Pi = eye(n);
    [Q,R] = QR_Householder(M);
    p = 1:n;
    i = n;
    l = 0;
    for count = 1:n
        if max(abs(R(i,i:n))) > delta
            [~,j] = max(abs(R(i,i:n)));
            j = i+j-1;
            p([l+1,j]) = p([j,l+1]);
            R(:,[l+1,j]) = R(:,[j,l+1]);
            v = house(R(l+1:m,l+1));
            c = 2/(v'*v);
            R(l+1:m,:) = R(l+1:m,:) - c*v*(v'*R(l+1:m,:));
            Q(:,l+1:m) = Q(:,l+1:m) - c*(Q(:,l+1:m)*v)*v';
            l = l+1;
        else
```

```
            i = i-1;
        end
    end
    Pi = Pi(:,p);
end
```

QR_Foster.m

A.3 Hybrid RRQR Factorization

A.3.1 Algorithm Hybrid-I

```
function [Q, R, Pi] = QR_Hybrid1(M, k)
    [m,n] = size(M);
    Pi = eye(n);
    [Q,R] = TQR_Householder(M,k);
    p = 1:n;
    gamma = sqrt(sum(R(k:m,k:n).^2,1))';
    permuted = true;
    invAk = inv(R(1:k,1:k));
    omega = sqrt(sum(invAk.^2,2));
    invA = invAk(1:k-1, 1:k-1);
    while permuted == true
        permuted = false;
        % QR with column pivoting I
        [~,j] = max(gamma);
        if gamma(1) < gamma(j)
            permuted = true;
            if j > 2
                p([k+1,k-1+j]) = p([k-1+j,k+1]);
                R(:,[k+1,k-1+j]) = R(:,[k-1+j,k+1]);
                gamma([2,j]) = gamma([j,2]);
                v = house(R(k+1:m,k+1));
                c = 2/(v'*v);
                R(k+1:m,:) = R(k+1:m,:) - c*v*(v'*R(k+1:m,:));
                Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
            end
            g = R(k,k);
            mu = R(k,k+1)/g;
            nu = R(k+1,k+1)/g;
            rho = sqrt(mu^2+nu^2);
            gbar = g*rho;
            b1 = R(1:k-1,k);
            b2 = R(1:k-1,k+1);
```

```
            u = invA*b1;
            invAb2 = invA*b2;
            omega(1:k-1) = sqrt(omega(1:k-1).^2+ invAb2.^2/gbar^2-u
    .^2/g^2);
            gamma([1 2]) = gamma([2 1]);
            [c,s] = givens(R(k,k+1), R(k+1,k+1));
            G = [c, -s; s, c];
            p([k+1,k]) = p([k,k+1]);
            R(:,[k+1,k]) = R(:,[k,k+1]);
            R([k,k+1],k:n) = G'*R([k,k+1],k:n);
            Q(:,[k,k+1]) = Q(:,[k,k+1])*G;
            omega(k) = abs(1/R(k,k));
        end
        % QR with column pivoting II
        [~,j] = max(omega);
        if omega(k) < omega(j)
            permuted = true;
            if j < k-1
                p([j:k-2 k-1]) = p([j+1:k-1 j]);
                omega([j:k-2 k-1]) = omega([j+1:k-1 j]);
                R(:,[j:k-2 k-1]) = R(:,[j+1:k-1 j]);
                Qk = eye(k-1);
                for t = j:k-2
                    [c,s] = givens(R(t,t),R(t+1,t));
                    G = [c, -s; s, c];
                    R([t,t+1],t:n) = G'*R([t,t+1],t:n);
                    Q(:,[t,t+1]) = Q(:,[t,t+1])*G;
                    Qk(:,[t,t+1]) = Qk(:,[t,t+1])*G;
                end
                invA = invA*Qk;
                invA([j:k-2 k-1],:) = invA([j+1:k-1 j],:);
            end
            invAm = invA(1:k-2,1:k-2);
            g = R(k-1,k-1);
            mu = R(k-1,k)/g;
            nu = R(k,k)/g;
            rho = sqrt(mu^2+nu^2);
            b2 = R(1:k-2,k);
            c2 = R(k,k+1:n)';
            invAb2 = invAm*b2;
            omega([k,k-1]) = omega([k-1 k]);
            p([k,k-1]) = p([k-1,k]);
            R(:,[k,k-1]) = R(:,[k-1,k]);
            [c,s] = givens(R(k-1,k-1), R(k,k-1));
            G = [c, -s; s, c];
```

```
            R([k-1,k],k-1:n) = G'*R([k-1,k],k-1:n);
            gamma(1) = abs(g*nu/rho);
            gamma(2:end) = sqrt(gamma(2:end).^2+(R(k,k+1:n)').^2-c2
    .^2);
            Q(:,[k-1,k]) = Q(:,[k-1,k])*G;
            invA = [invAm, -invAb2/R(k-1,k-1);zeros(1,k-2),1/R(k-1,k
    -1)];
        end
    end
    Pi = Pi(:,p);
end
```

QR_Hybrid1.m

A.3.2 Algorithm Hybrid-II

```
function [Q, R, Pi] = QR_Hybrid2(M, k)
    [Q,R,Pi] = QR_Hybrid1(M,k+1);
end
```

QR_Hybrid2.m

A.3.3 Algorithm Hybrid-III

```
function [Q, R, Pi] = QR_Hybrid3(M, k)
    permuted = true;
    [Q,R] = TQR_Householder(M,k);
    [~,n] = size(M);
    Pi = eye(n);
    while permuted
        permuted = false;
        [tildeQ,tildeR,P] = QR_Hybrid1(R,k);
        if norm(P-eye(n),1) > 0.1
            permuted = true;
            Pi = Pi*P;
            Q = Q*tildeQ;
            R = tildeR;
        end
        [tildeQ,tildeR,P] = QR_Hybrid2(R,k);
        if norm(P-eye(n),1) > 0.1
            permuted = true;
            Pi = Pi*P;
            Q = Q*tildeQ;
```

```
            R = tildeR;
        end
    end
end
```

QR_Hybrid3.m

A.4 Strong RRQR Factorization

A.4.1 Algorithm SRRQR-1

```
function [Q, R, Pi] = QR_Strong1(M, k)
    [m,n] = size(M);
    f = 10*sqrt(n);
    Pi = eye(n);
    [Q,R] = TQR_Householder(M,k);
    p = 1:n;
    gamma = sqrt(sum(R(k+1:m,k+1:n).^2,1))';
    invAk = inv(R(1:k,1:k));
    omega = sqrt(sum(invAk.^2,2));
    invAkB = R(1:k,1:k)\R(1:k,k+1:n);
    while 1
%         Debug
%         invAk = inv(R(1:k,1:k));
%         norm(gamma-sqrt(sum(R(k+1:m,k+1:n).^2,1))')
%         norm(omega-sqrt(sum(invAk.^2,2)))
%         norm(invAkB-R(1:k,1:k)\R(1:k,k+1:n))
        gammaOmega = (gamma*ones(1,k)).*(ones(n-k,1)*omega');
        if max(max(abs(invAkB).^2+(gammaOmega').^2)) <= f^2
            break
        end
        flag = false;
        for s = 1:k
            for t = 1:n-k
                if abs(invAkB(s,t))^2+gammaOmega(t,s)^2 > f^2
                    i = s;
                    j = t;
                    flag = true;
                    break
                end
            end
            if flag
                break
            end
```

```
        end

        if i < k
            p([i:k-1 k]) = p([i+1:k i]);
            omega([i:k-1 k]) = omega([i+1:k i]);
            R(:,[i:k-1 k]) = R(:,[i+1:k i]);
            for t = i:k-1
                [c,s] = givens(R(t,t),R(t+1,t));
                G = [c, s; s, -c];
                R([t,t+1],t:n) = G'*R([t,t+1],t:n);
                Q(:,[t,t+1]) = Q(:,[t,t+1])*G;
            end
            invAkB([i:k-1 k],:) = invAkB([i+1:k i],:);
        end

        if j > 1
            p([k+1,k+j]) = p([k+j,k+1]);
            R(:,[k+1,k+j]) = R(:,[k+j,k+1]);
            gamma([1,j]) = gamma([j,1]);
            v = house(R(k+1:m,k+1));
            c = 2/(v'*v);
            R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
    ;
            Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
            invAkB(:,[1,j]) = invAkB(:,[j,1]);
        elseif j == 1
            v = house(R(k+1:m,k+1));
            c = 2/(v'*v);
            R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
    ;
            Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
        end
        g = R(k,k);
        mu = R(k,k+1)/g;
        nu = R(k+1,k+1)/g;
        rho = sqrt(mu^2+nu^2);
        b1 = R(1:k-1,k);
        c2 = R(k+1,k+2:n)';
        u = R(1:k-1,1:k-1)\b1;
        u1 = invAkB(1:k-1,1);
        invAb2 = u1+mu*u;
        U = invAkB(1:k-1,2:end);
        [c,s] = givens(R(k,k+1), R(k+1,k+1));
        G = [c, s; s, -c];
        p([k+1,k]) = p([k,k+1]);
```

```
        R(:,[k+1,k]) = R(:,[k,k+1]);
        R([k,k+1],k:n) = G'*R([k,k+1],k:n);
        Q(:,[k,k+1]) = Q(:,[k,k+1])*G;
        omega(k) = 1/abs(R(k,k));
        omega(1:k-1) = sqrt(omega(1:k-1).^2+invAb2.^2/R(k,k)^2-u.^2/g
    ^2);
        gamma(1) = abs(R(k+1,k+1));
        if k < n-1
            gamma(2:n-k) = sqrt(gamma(2:n-k).^2+(R(k+1,k+2:n)').^2-c2
    .^2);
        end
        invAkB = [(nu^2*u-mu*u1)/rho^2, U+(nu*u*R(k+1,k+2:n)-u1*R(k,k
    +2:n))/R(k,k);mu/rho^2,R(k,k+2:n)/R(k,k)];
    end
    Pi = Pi(:,p);
    norm(Q*Q'-eye(m))
    norm(M*Pi-Q*R)
end
```

QR_Strong1.m

A.4.2 Algorithm SRRQR-2

```
function [Q, R, Pi] = QR_Strong2(M, k)
    [m,n] = size(M);
    %f = 2;%10*sqrt(n);
    f = sqrt(k*(n-k)+min(k,n-k))/sqrt(k*(n-k));
    Pi = eye(n);
    [Q,R] = TQR_Householder(M,k);
    p = 1:n;
    gamma = sqrt(sum(R(k+1:m,k+1:n).^2,1))';
    invAk = inv(R(1:k,1:k));
    omega = sqrt(sum(invAk.^2,2));
    invAkB = R(1:k,1:k)\R(1:k,k+1:n);
    cnt = 0;
    while 1
%         Debug
%         invAk = inv(R(1:k,1:k));
%         norm(gamma-sqrt(sum(R(k+1:m,k+1:n).^2,1))')
%         norm(omega-sqrt(sum(invAk.^2,2)))
%         norm(invAkB-R(1:k,1:k)\R(1:k,k+1:n))
        gammaOmega = (gamma*ones(1,k)).*(ones(n-k,1)*omega');
        if max(max(max(abs(invAkB))), max(max(gammaOmega))) <= f
            break
```

```
        end
        flag = false;
        for s = 1:k
            for t = 1:n-k
                if max(abs(invAkB(s,t)), gammaOmega(t,s)) > f
                    i = s;
                    j = t;
                    flag = true;
                    break
                end
            end
            if flag
                break
            end
        end
        cnt = cnt + 1;

        if i < k
            p([i:k-1 k]) = p([i+1:k i]);
            omega([i:k-1 k]) = omega([i+1:k i]);
            R(:,[i:k-1 k]) = R(:,[i+1:k i]);
            for t = i:k-1
                [c,s] = givens(R(t,t),R(t+1,t));
                G = [c, s; s, -c];
                R([t,t+1],t:n) = G'*R([t,t+1],t:n);
                Q(:,[t,t+1]) = Q(:,[t,t+1])*G;
            end
            invAkB([i:k-1 k],:) = invAkB([i+1:k i],:);
        end

        if j > 1
            p([k+1,k+j]) = p([k+j,k+1]);
            R(:,[k+1,k+j]) = R(:,[k+j,k+1]);
            gamma([1,j]) = gamma([j,1]);
            v = house(R(k+1:m,k+1));
            c = 2/(v'*v);
            R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
    ;
            Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
            invAkB(:,[1,j]) = invAkB(:,[j,1]);
        elseif j == 1
            v = house(R(k+1:m,k+1));
            c = 2/(v'*v);
            R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
    ;
```

```
            Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
        end
        g = R(k,k);
        mu = R(k,k+1)/g;
        nu = R(k+1,k+1)/g;
        rho = sqrt(mu^2+nu^2);
        b1 = R(1:k-1,k);
        c2 = R(k+1,k+2:n)';
        u = R(1:k-1,1:k-1)\b1;
        u1 = invAkB(1:k-1,1);
        invAb2 = u1+mu*u;
        U = invAkB(1:k-1,2:end);
        [c,s] = givens(R(k,k+1), R(k+1,k+1));
        G = [c, s; s, -c];
        p([k+1,k]) = p([k,k+1]);
        R(:,[k+1,k]) = R(:,[k,k+1]);
        R([k,k+1],k:n) = G'*R([k,k+1],k:n);
        Q(:,[k,k+1]) = Q(:,[k,k+1])*G;
        omega(k) = 1/abs(R(k,k));
        omega(1:k-1) = sqrt(omega(1:k-1).^2+invAb2.^2/R(k,k)^2-u.^2/g
    ^2);
        gamma(1) = abs(R(k+1,k+1));
        if k < n-1
            gamma(2:n-k) = sqrt(gamma(2:n-k).^2+(R(k+1,k+2:n)').^2-c2
    .^2);
        end
        invAkB = [(nu^2*u-mu*u1)/rho^2, U+(nu*u*R(k+1,k+2:n)-u1*R(k,k
    +2:n))/R(k,k);mu/rho^2,R(k,k+2:n)/R(k,k)];
    end
    cnt
    Pi = Pi(:,p);
end
```

QR_Strong2.m

A.4.3 Algorithm GSRRQR-1

```
function [Q, R, Pi] = QR_GStrong1(M, k)
    [m,n] = size(M);
    f = 2;%10*sqrt(n);
    f = sqrt(k*(n-k)+min(k,n-k))/sqrt(k*(n-k));
    Pi = eye(n);
    [Q,R] = TQR_Householder(M,k);
    p = 1:n;
```

```
    gamma = sqrt(sum(R(k+1:m,k+1:n).^2,1))';
    invAk = inv(R(1:k,1:k));
    omega = sqrt(sum(invAk.^2,2));
    invAkB = R(1:k,1:k)\R(1:k,k+1:n);
    cnt = 0;
    gcnt = 0;
    while 1
%         Debug
%         invAk = inv(R(1:k,1:k));
%         norm(gamma-sqrt(sum(R(k+1:m,k+1:n).^2,1))')
%         norm(omega-sqrt(sum(invAk.^2,2)))
%         norm(invAkB-R(1:k,1:k)\R(1:k,k+1:n))
        if (max(gamma)*max(omega) > f)
            [~,j] = max(gamma);
            [~,i] = max(omega);
            gcnt = gcnt+1;
        else
            gammaOmega = (gamma*ones(1,k)).*(ones(n-k,1)*omega');
            if max(max(abs(invAkB).^2+(gammaOmega').^2)) <= f^2
                break
            end
            flag = false;
            for s = 1:k
                for t = 1:n-k
                    if abs(invAkB(s,t))^2+gammaOmega(t,s)^2 > f^2
                        i = s;
                        j = t;
                        flag = true;
                        break
                    end
                end
                if flag
                    break
                end
            end
        end
        cnt = cnt + 1;
        if i < k
            p([i:k-1 k]) = p([i+1:k i]);
            omega([i:k-1 k]) = omega([i+1:k i]);
            R(:,[i:k-1 k]) = R(:,[i+1:k i]);
            for t = i:k-1
                [c,s] = givens(R(t,t),R(t+1,t));
                G = [c, s; s, -c];
                R([t,t+1],t:n) = G'*R([t,t+1],t:n);
```

```
                Q(:,[t,t+1]) = Q(:,[t,t+1])*G;
            end
            invAkB([i:k-1 k],:) = invAkB([i+1:k i],:);
        end

        if j > 1
            p([k+1,k+j]) = p([k+j,k+1]);
            R(:,[k+1,k+j]) = R(:,[k+j,k+1]);
            gamma([1,j]) = gamma([j,1]);
            v = house(R(k+1:m,k+1));
            c = 2/(v'*v);
            R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
    ;
            Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
            invAkB(:,[1,j]) = invAkB(:,[j,1]);
        elseif j == 1
            v = house(R(k+1:m,k+1));
            c = 2/(v'*v);
            R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
    ;
            Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
        end
        g = R(k,k);
        mu = R(k,k+1)/g;
        nu = R(k+1,k+1)/g;
        rho = sqrt(mu^2+nu^2);
        b1 = R(1:k-1,k);
        c2 = R(k+1,k+2:n)';
        u = R(1:k-1,1:k-1)\b1;
        u1 = invAkB(1:k-1,1);
        invAb2 = u1+mu*u;
        U = invAkB(1:k-1,2:end);
        [c,s] = givens(R(k,k+1), R(k+1,k+1));
        G = [c, s; s, -c];
        p([k+1,k]) = p([k,k+1]);
        R(:,[k+1,k]) = R(:,[k,k+1]);
        R([k,k+1],k:n) = G'*R([k,k+1],k:n);
        Q(:,[k,k+1]) = Q(:,[k,k+1])*G;
        omega(k) = 1/abs(R(k,k));
        omega(1:k-1) = sqrt(omega(1:k-1).^2+invAb2.^2/R(k,k)^2-u.^2/g
    ^2);
        gamma(1) = abs(R(k+1,k+1));
        if k < n-1
            gamma(2:n-k) = sqrt(gamma(2:n-k).^2+(R(k+1,k+2:n)').^2-c2
    .^2);
```

```
94          end
95          invAkB = [(nu^2*u-mu*u1)/rho^2, U+(nu*u*R(k+1,k+2:n)-u1*R(k,k
    +2:n))/R(k,k);mu/rho^2,R(k,k+2:n)/R(k,k)];
96      end
97      Pi = Pi(:,p);
98      cnt
99      gcnt
100 end
```

QR_GStrong1.m

A.4.4 Algorithm GSRRQR-2

```
1  function [Q, R, Pi] = QR_GStrong2(M, k)
2      [m,n] = size(M);
3      %f = 2;%10*sqrt(n);
4      f = sqrt(k*(n-k)+min(k,n-k))/sqrt(k*(n-k));
5      Pi = eye(n);
6      [Q,R] = TQR_Householder(M,k);
7      p = 1:n;
8      gamma = sqrt(sum(R(k+1:m,k+1:n).^2,1))';
9      invAk = inv(R(1:k,1:k));
10     omega = sqrt(sum(invAk.^2,2));
11     invAkB = R(1:k,1:k)\R(1:k,k+1:n);
12     cnt = 0;
13     gcnt = 0;
14     while 1
15 %          Debug
16 %          invAk = inv(R(1:k,1:k));
17 %          norm(gamma-sqrt(sum(R(k+1:m,k+1:n).^2,1))')
18 %          norm(omega-sqrt(sum(invAk.^2,2)))
19 %          norm(invAkB-R(1:k,1:k)\R(1:k,k+1:n))
20          if (max(gamma)*max(omega) > f)
21              [~,j] = max(gamma);
22              [~,i] = max(omega);
23              gcnt = gcnt+1;
24          else
25              if max(max(abs(invAkB))) <= f
26                  break
27              end
28              flag = false;
29              for s = 1:k
30                  for t = 1:n-k
31                      if abs(invAkB(s,t)) > f
```

```
                    i = s;
                    j = t;
                    flag = true;
                    break
                end
            end
            if flag
                break
            end
        end
    end
    cnt = cnt + 1;
    if i < k
        p([i:k-1 k]) = p([i+1:k i]);
        omega([i:k-1 k]) = omega([i+1:k i]);
        R(:,[i:k-1 k]) = R(:,[i+1:k i]);
        for t = i:k-1
            [c,s] = givens(R(t,t),R(t+1,t));
            G = [c, s; s, -c];
            R([t,t+1],t:n) = G'*R([t,t+1],t:n);
            Q(:,[t,t+1]) = Q(:,[t,t+1])*G;
        end
        invAkB([i:k-1 k],:) = invAkB([i+1:k i],:);
    end

    if j > 1
        p([k+1,k+j]) = p([k+j,k+1]);
        R(:,[k+1,k+j]) = R(:,[k+j,k+1]);
        gamma([1,j]) = gamma([j,1]);
        v = house(R(k+1:m,k+1));
        c = 2/(v'*v);
        R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
;
        Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
        invAkB(:,[1,j]) = invAkB(:,[j,1]);
    elseif j == 1
        v = house(R(k+1:m,k+1));
        c = 2/(v'*v);
        R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k+1:n))
;
        Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
    end
    g = R(k,k);
    mu = R(k,k+1)/g;
    nu = R(k+1,k+1)/g;
```

```
        rho = sqrt(mu^2+nu^2);
        b1 = R(1:k-1,k);
        c2 = R(k+1,k+2:n)';
        u = R(1:k-1,1:k-1)\b1;
        u1 = invAkB(1:k-1,1);
        invAb2 = u1+mu*u;
        U = invAkB(1:k-1,2:end);
        [c,s] = givens(R(k,k+1), R(k+1,k+1));
        G = [c, s; s, -c];
        p([k+1,k]) = p([k,k+1]);
        R(:,[k+1,k]) = R(:,[k,k+1]);
        R([k,k+1],k:n) = G'*R([k,k+1],k:n);
        Q(:,[k,k+1]) = Q(:,[k,k+1])*G;
        omega(k) = 1/abs(R(k,k));
        omega(1:k-1) = sqrt(omega(1:k-1).^2+invAb2.^2/R(k,k)^2-u.^2/g
    ^2);
        gamma(1) = abs(R(k+1,k+1));
        if k < n-1
            gamma(2:n-k) = sqrt(gamma(2:n-k).^2+(R(k+1,k+2:n)').^2-c2
    .^2);
        end
        invAkB = [(nu^2*u-mu*u1)/rho^2, U+(nu*u*R(k+1,k+2:n)-u1*R(k,k
    +2:n))/R(k,k);mu/rho^2,R(k,k+2:n)/R(k,k)];
    end
    cnt
    gcnt
    Pi = Pi(:,p);
end
```

QR_GStrong2.m

A.4.5 Algorithm SRRQR-3

```
function [Q, R, Pi, k] = QR_Strong3(M)
    delta = 0.0001;
    [m,n] = size(M);
    f = 10*sqrt(n);
    Pi = eye(n);
    Q = eye(m);
    p = 1:n;
    R = M;
    k = 0;
    gamma = sqrt(sum(R(k+1:m,k+1:n).^2,1))';
    omega = [];
```

```
    invAkB = [];
    while max(gamma) >= delta
        [~,j] = max(gamma);
        k = k + 1;
        p([k,k+j-1]) = p([k+j-1,k]);
        R(:,[k,k+j-1]) = R(:,[k+j-1,k]);
        gamma([1,j]) = gamma([j,1]);
        if k > 1
            invAkB(:,[1,j]) = invAkB(:,[j,1]);
        end

        v = house(R(k:m,k));
        c = 2/(v'*v);
        R(k:m,:) = R(k:m,:) - c*v*(v'*R(k:m,:));
        Q(:,k:m) = Q(:,k:m) - c*(Q(:,k:m)*v)*v';
        omega(k) = 1/abs(R(k,k));
        if k > 1
            omega(1:k-1) = sqrt(omega(1:k-1).^2+invAkB(:,1).^2/R(k,k)
^2);
        end

        if k == n
            gamma = [];
        else
            gamma = sqrt(gamma(2:end).^2-(R(k,k+1:n)').^2);
        end

        if k > 1
            invAkB = [invAkB(:,2:end)-invAkB(:,1)*R(k,k+1:n)/R(k,k);R
(k,k+1:n)/R(k,k)];
        else
            invAkB = [R(k,k+1:n)/R(k,k)];
        end

        omega = omega(:);

        while length(gamma) > 0
            gammaOmega = (gamma*ones(1,k)).*(ones(n-k,1)*omega');
            if max(max(max(abs(invAkB))), max(max(gammaOmega))) <= f
                break
            end
            flag = false;
            for s = 1:k
                for t = 1:n-k
                    if max(abs(invAkB(s,t)), gammaOmega(t,s)) > f
```

```
                        i = s;
                        j = t;
                        flag = true;
                        break
                    end
                end
                if flag
                    break
                end
            end

            if i < k
                p([i:k-1 k]) = p([i+1:k i]);
                omega([i:k-1 k]) = omega([i+1:k i]);
                R(:,[i:k-1 k]) = R(:,[i+1:k i]);
                for t = i:k-1
                    [c,s] = givens(R(t,t),R(t+1,t));
                    G = [c, s; s, -c];
                    R([t,t+1],t:n) = G'*R([t,t+1],t:n);
                    Q(:,[t,t+1]) = Q(:,[t,t+1])*G;
                end
                invAkB([i:k-1 k],:) = invAkB([i+1:k i],:);
            end

            if j > 1
                p([k+1,k+j]) = p([k+j,k+1]);
                R(:,[k+1,k+j]) = R(:,[k+j,k+1]);
                gamma([1,j]) = gamma([j,1]);
                v = house(R(k+1:m,k+1));
                c = 2/(v'*v);
                R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k
+1:n));
                Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
                invAkB(:,[1,j]) = invAkB(:,[j,1]);
            elseif j == 1
                v = house(R(k+1:m,k+1));
                c = 2/(v'*v);
                R(k+1:m,k+1:n) = R(k+1:m,k+1:n) - c*v*(v'*R(k+1:m,k
+1:n));
                Q(:,k+1:m) = Q(:,k+1:m) - c*(Q(:,k+1:m)*v)*v';
            end
            g = R(k,k);
            mu = R(k,k+1)/g;
            nu = R(k+1,k+1)/g;
            rho = sqrt(mu^2+nu^2);
```

```
98              b1 = R(1:k-1,k);
99              c2 = R(k+1,k+2:n)';
100             u = R(1:k-1,1:k-1)\b1;
101             u1 = invAkB(1:k-1,1);
102             invAb2 = u1+mu*u;
103             U = invAkB(1:k-1,2:end);
104             [c,s] = givens(R(k,k+1), R(k+1,k+1));
105             G = [c, s; s, -c];
106             p([k+1,k]) = p([k,k+1]);
107             R(:,[k+1,k]) = R(:,[k,k+1]);
108             R([k,k+1],k:n) = G'*R([k,k+1],k:n);
109             Q(:,[k,k+1]) = Q(:,[k,k+1])*G;
110             omega(k) = 1/abs(R(k,k));
111             omega(1:k-1) = sqrt(omega(1:k-1).^2+invAb2.^2/R(k,k)^2-u
    .^2/g^2);
112             gamma(1) = abs(R(k+1,k+1));
113             if k < n-1
114                 gamma(2:n-k) = sqrt(gamma(2:n-k).^2+(R(k+1,k+2:n)')
    .^2-c2.^2);
115             end
116             invAkB = [(nu^2*u-mu*u1)/rho^2, U+(nu*u*R(k+1,k+2:n)-u1*R
    (k,k+2:n))/R(k,k);mu/rho^2,R(k,k+2:n)/R(k,k)];
117         end
118
119 %           %           Debug
120 %           if k < n
121 %               invAk = inv(R(1:k,1:k));
122 %               norm(gamma-sqrt(sum(R(k+1:m,k+1:n).^2,1))')
123 %               norm(omega-sqrt(sum(invAk.^2,2)))
124 %               norm(invAkB-R(1:k,1:k)\R(1:k,k+1:n))
125 %           end
126
127     end
128     Pi = Pi(:,p);
129 %     norm(Q*Q'-eye(m))
130 %     norm(M*Pi-Q*R)
131 end
```

QR_Strong3.m

Appendix B MATLAB Code for Numerical Experiments

B.1 Revealing Matrix Rank Deficiency

```
function M = matrixGKS(n)
    % Generate a GKS Matrix
    M = zeros(n,n);
    for i = 1:n
        M(i,i) = 1/sqrt(i);
        for j = i+1:n
            M(i,j) = -1/sqrt(j);
        end
    end
end
```

matrixGKS.m

```
function M = matrixKahan(n,s,c)
    % Generate a 'Kahan' Matrix M=S*K.
    S = diag(s.^(0:n-1));
    K = eye(n) + triu(ones(n,n)*-c, 1);
    M = S * K;
end
```

matrixKahan.m

```
function M = matrixScaled(n)
    % Generate a scaled Matrix
    eta = 20*eps;
    M = rand(n);
    for i = 1:n
        M(i,:) = M(i,:)*eta^(i/n);
    end
end
```

matrixScaled.m

```
n = 100;
k = 50;
% M = rand(n,n);
% M = matrixScaled(n);
% M = matrixGKS(n);
% M = matrixKahan(50,sqrt(1-0.2^2),0.2);
```

```
r = rank(M);
% measurements:
T = zeros(1,15); % running time averaged by 5 times
Q1 = T; % sigma_k(M)/min(sigma(R11)), averaged by 5, smaller is
    better
Q2 = T; % max(sigma(R22)) / sigma_k+1(M), averaged by 5, smaller is
    better
Q3 = T; % max(R(11)^{-1}*R_12), averaged by 5, smaller is better
V = svd(M);
skM = V(k);
sk1M = V(k+1);

b1 = sqrt(1+k*(n-k)+min(k,n-k)) % bound 1
b2 = sqrt(k*(n-k)+min(k,n-k))/sqrt(k*(n-k)) % bound 2

for t = 1:1
    tic;[~,R] = TQR_Householder(M,k);
    T(1) = toc;
    V1 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(1) = skM/V1(k);
    Q2(1) = V1(k+1)/sk1M;
    Q3(1) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Greedy1(M,k);
    T(2) = toc;
    V2 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(2) = skM/V2(k);
    Q2(2) = V2(k+1)/sk1M;
    Q3(2) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Greedy2(M,k);
    T(3) = toc;
    V3 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(3) = skM/V3(k);
    Q2(3) = V3(k+1)/sk1M;
    Q3(3) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Greedy3(M,k);
    T(4) = toc;
    V4 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(4) = skM/V4(k);
    Q2(4) = V4(k+1)/sk1M;
    Q3(4) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_ColumnPivoting(M,k);
```

```
    T(5) = toc;
    V5 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(5) = skM/V5(k);
    Q2(5) = V5(k+1)/sk1M;
    Q3(5) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Chan(M,k);
    T(6) = toc;
    V6 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(6) = skM/V6(k);
    Q2(6) = V6(k+1)/sk1M;
    Q3(6) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_GKS(M,k);
    T(7) = toc;
    V7 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(7) = skM/V7(k);
    Q2(7) = V7(k+1)/sk1M;
    Q3(7) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Foster(M,k);
    T(8) = toc;
    V8 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(8) = skM/V8(k);
    Q2(8) = V8(k+1)/sk1M;
    Q3(8) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Hybrid1(M,k);
    T(9) = toc;
    V9 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(9) = skM/V9(k);
    Q2(9) = V9(k+1)/sk1M;
    Q3(9) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Hybrid2(M,k);
    T(10) = toc;
    V10 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(10) = skM/V10(k);
    Q2(10) = V10(k+1)/sk1M;
    Q3(10) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Hybrid3(M,k);
    T(11) = toc;
    V11 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(11) = skM/V11(k);
```

```
    Q2(11) = V11(k+1)/sk1M;
    Q3(11) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Strong1(M,k);
    T(12) = toc;
    V12 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(12) = skM/V12(k);
    Q2(12) = V12(k+1)/sk1M;
    Q3(12) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_Strong2(M,k);
    T(13) = toc;
    V13 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(13) = skM/V13(k);
    Q2(13) = V13(k+1)/sk1M;
    Q3(13) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_GStrong1(M,k);
    T(14) = toc;
    V14 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(14) = skM/V14(k);
    Q2(14) = V14(k+1)/sk1M;
    Q3(14) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));

    tic;[~,R,~] = QR_GStrong2(M,k);
    T(15) = toc;
    V15 = [svd(R(1:k,1:k));svd(R(k+1:n,k+1:n))];
    Q1(15) = skM/V15(k);
    Q2(15) = V15(k+1)/sk1M;
    Q3(15) = max(max(abs(R(1:k,1:k)\R(1:k,k+1:n))));
end
```

DeficiencyRevealing.m

B.2 Rank Deficient Least Square Problems

```
function [M] = matrixCustom(n,k)
    d = [linspace(1000,1,k)';ones(n-k,1)./10];
    P = orth(randn(n,n));
    M = P'*diag(d)*P;
end
```

matrixCustom.m

```
n = 100;
k = 90;
A = matrixCustom(n, k);
xexact = rand(n,1);
xexact = xexact/norm(xexact);
b = A*xexact;
[U,S,V] = svd(A);
% FSVD solution (exact solution)
xfull = zeros(n,1);
for i = 1:n
    xfull = xfull + U(:,i)'*b*V(:,i)/S(i,i);
end
rfull = A*xfull-b;

% TSVD solution
xsvd = zeros(n,1);
for i = 1:k
    xsvd = xsvd + U(:,i)'*b*V(:,i)/S(i,i);
end
rsvd = A*xsvd-b;

% BQR solution based on SRRQR-I
[Q,R,P] = QR_Strong1(A,k);
xbqr = P*[inv(R(1:k,1:k)),zeros(k,n-k);zeros(n-k,k),zeros(n-k,n-k)]*Q
    '*b;
rbqr = A*xbqr-b;

% TQR solution
R11 = R(1:k,1:k);
R12 = R(1:k,k+1:n);
[H,tildeR] = QR_Householder([R11';R12']);
xtqr = P*H*[inv(tildeR(1:k,1:k)'),zeros(k,n-k);zeros(n-k,k),zeros(n-k
    ,n-k)]*Q'*b;
rtqr = A*xtqr-b;

res1 = [norm(xsvd-xexact),norm(xtqr-xsvd),norm(xsvd-xbqr)];
res2 = [norm(rsvd),norm(rtqr),norm(rbqr),norm(rfull)];
```

LeastSquare.m

B.3 Subset Selection Problem

```
function [cnt] = ComparePermutation(P1, P2,k)
    cnt = 0;
```

```
    for  i = 1:k
        for j = 1:k
            if P1(:,i) == P2(:,j)
                cnt = cnt+1;
            end
        end
    end
end
```

ComparePermutation.m

```
n = 500;
k = 400;
A = matrixCustom(n, k);
% svd
tic;
[U,S,V] = mySVD(A);
[~,~,Psvd] = QR_ColumnPivoting([V(1:k,1:k)', V(k+1:n,1:k)'],k);
toc
Bsvd = A*Psvd;
B1svd = Bsvd(:,1:k);

% qr
tic;
[~,~,Pqr] = QR_GStrong2(A,k);
toc
Bqr = A*Pqr;
B1qr = Bqr(:,1:k);

 sin(subspace(B1svd, U(:,1:k)))
 sin(subspace(B1qr, U(:,1:k)))
 ComparePermutation(Psvd,Pqr,k)
%norm(Psvd-Pqr)
```

SubsetSelection.m

B.4 Matrix Approximation and Image Compression

```
n = 500;
k = 400;
A = matrixCustom(n, k);

% SVD
tic;[U,S,V] = mySVD(A);toc
Ak = zeros(n,n);
```

```
for i = 1:k
    Ak = Ak + U(:,i)*S(i,i)*V(:,i)';
end

% RRQR
tic;[Q,R,P] = QR_GStrong2(A,k);toc
Bk = Q*[R(1:k,:);zeros(n-k,n)]*P';

norm(A-Bk)
norm(A-Ak)
```

MatrixApproximation.m

```
function [A] = imageCompression(path, method, k)
    RGB = imread(path);
    A = rgb2gray(RGB);
    A = double(A);
    [m,n] = size(A);
    if strcmp(method,'svd')
        [U,S,V] = svd(A);
        A = U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
    else
        [Q,R,P] = QR_GStrong2(A,k);
        A = Q*[R(1:k,:);zeros(m-k,n)]*P';
    end
    figure
    imshow(uint8(A))
end
```

imageCompression.m