THE UNIVERSITY OF CHICAGO

COMPLEXITY AND NUMERICAL STABILITY IN MATRIX COMPUTATIONS AND NONCONVEX OPTIMIZATION

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

COMMITTEE ON COMPUTATIONAL AND APPLIED MATHEMATICS

BY
ZHEN DAI

CHICAGO, ILLINOIS
JUNE 2023

To my parents and brother for their unconditional love and support.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

During my doctoral study, I received a great deal of support. Foremost, I want to thank my advisor, Prof. Lek-Heng Lim for his support in both my academic study and career development. He gave me many interesting problems in numerical linear algebra and optimization. Through his help and insightful discussion, I am able to discover many interesting results in these areas. Lek-Heng also gave me many precious advice when I was looking for an internship and a full time job. His help made my transition to industry much smoother than expected.

Besides my advisor, I also want to thank Prof. Nathan Srebro. I learned statistical learning theory from Nathan and did a project related to implicit regularization of neural networks with him. He gave me a lot of insightful advice and we were able to have some interesting findings in this area. I would also like to thank Prof. Mina Karzand for her insightful discussion and help during this project. She taught me a lot of things about academic writing and LaTeX, which are very helpful during the rest of my doctoral study.

In addition, I would like to thank Prof. Yury Makarychev and Prof. Ali Vakilian for their help and insightful discussions in a project on approximation algorithms.

I would like to thank Prof. Ke Ye for his help and insightful discussions in the project on complex matrix inversions and the project on numerical stability and tensor nuclear norm.

I would also like to thank Prof. Nick Higham for his insightful discussions in the project on numerical stability and tensor nuclear norm.

I would also like to thank Prof. Gabor Pataki for his insightful discussions in the project on rank-constrained hyperbolic programming.

I would also like to thank my committee members: Prof. Yuehaw Khoo and Prof. Ali Vakilian for their help in my Ph.D. defense. I would also like to thank Prof. Bradley J. Nelson who was in my committee for his valuable advice in both research and career development. He also gave me many helpful advice in my Ph.D. defense.

I would like to thank Lijia Zhou for his insightful discussions on many problems during the past few years. We solved a problem in statistical learning theory together recently. The discussion is very insightful and this problem solving process is very enjoyable. I would like to thank Lijia for discussing this problem with me!

I would also like to thank Zihao Wang and Hongli Zhao for their help and advice in preparation for my defense.

I would also like to thank Bradley J. Nelson, Zhenyang Zhang, Liwen Zhang, Gregory Naitzat, Pinhan Chen, Lizhen Nie, and Haoyang Liu for their advice on my career development.

I would also like to thank my internship manager Taiyu Dong for his help during my internship.

I would also like to thank Prof. Mary Silber, Zellencia Harris, and Jonathan Rodriguez for their help in general matters.

During my doctoral study, I received funding from DARPA, NSF, and IDEAL institute. I would like to thank these institutes for their funding over the past few years. I would also like to thank the Research Computing Center in the University of Chicago for providing great computational resources and prompt technical support.

In addition, I would like to thank my friends for their support over the past few years, including Zihao Wang, Lijia Zhou, Yanqing Gui, Yi Wang, Zehua Lai, Lin Gui, Yian Chen, Fuheng Cui, Wanrong Zhu, Dongyue Xie, Yuwei Luo, Weilin Chen, Jinwen Yang, Ji Xu, Yibo Jiang, Yunqi Yang, Runxin Ni, Wei Kuang, Danye Xu, Zhenyang Zhang, Weiyi Liu, Yidong Chen, Binglin Song, Xiyu Zhai, Xinze Li, Yihao Lu and many others. I spent five fantastic years at the University of Chicago and this would not be possible without them.

Last but not least, I would like to thank my parents Fengdi Ma and Xiangen Dai and my brother Jian Dai for their love and support.

ABSTRACT

In this dissertation, we study three problems in nonconvex optimization and matrix computation: rank-constrained hyperbolic programming, real and complex matrix multiplication, and complex matrix inversion. We study efficient algorithms that solve these problems. Here, we evaluate efficiency of algorithms in terms of both speed and accuracy. In terms of speed, we are looking for algorithms that use the least number of arithmetic operations. In terms of accuracy, we are looking for algorithms that induce the smallest rounding errors. Moreover, we will study the complexity of rank-constrained problems by understanding the NP-hardness of such problems.

CHAPTER 1

INTRODUCTION

Matrix computations and nonconvex optimizations are two of the most important tools in scientific applications. In this thesis, we study several important problems in these two areas by understanding the inherent limit of computations and designing efficient algorithms. To be specific, we will study rank-constrained hyperbolic programming, complex matrix multiplication, matrix multiplication, and complex matrix inversion.

For the problem of rank-constrained hyperbolic programming, we study the time complexity of various rank-constrained and sparsity-constrained problems and design efficient algorithms to do rank reduction and sparsification. We extend rank-constrained optimization to general hyperbolic programs (HP) using the notion of matroid rank. For LP and SDP respectively, this reduces to sparsity-constrained LP and rank-constrained SDP that are already well-studied. But for QCQP and SOCP, we obtain new interesting optimization problems. For example, rank-constrained SOCP includes weighted Max-Cut and nonconvex QP as special cases, and dropping the rank constraints yield the standard SOCP-relaxations of these problems. We will show (i) how to do rank reduction for SOCP and QCQP, (ii) that rank-constrained SOCP and rank-constrained QCQP are NP-hard, and (iii) an improved result for rank-constrained SDP showing that if the number of constraints is m and the rank constraint is less than $2^{1/2-\epsilon}\sqrt{m}$ for some $\epsilon > 0$, then the problem is NP-hard. We will also study sparsity-constrained HP and extend results on LP sparsification to SOCP and QCQP. In particular, we show that there always exist (a) a solution to SOCP of cardinality at most twice the number of constraints and (b) a solution to QCQP of cardinality at most the sum of the number of linear constraints and the sum of the rank of the matrices in the quadratic constraints; and both (a) and (b) can be found efficiently.

For the problem of real and complex matrix multiplication, we study the numerical stability of several algorithms and find the most stable one. We present a notion of bilinear

stability, which is to numerical stability what bilinear complexity is to time complexity. In bilinear complexity, an algorithm for evaluating a bilinear operator $\beta : \mathbb{U} \times \mathbb{V} \to \mathbb{W}$ is a decomposition $\beta = \varphi_1 \otimes \psi_1 \otimes w_1 + \cdots + \varphi_r \otimes \psi_r \otimes w_r$; the number of terms r captures the speed of the algorithm; and its smallest possible value, i.e., the tensor rank of β, quantifies the speed of a fastest algorithm. Bilinear stability introduces norms to the mix: The growth factor of the algorithm $\|\varphi_1\|_*\|\psi_1\|_*\|w_1\| + \cdots + \|\varphi_r\|_*\|\psi_r\|_*\|w_r\|$ captures the accuracy of the algorithm; and its smallest possible value, i.e., the tensor nuclear norm of β, quantifies the accuracy of a stablest algorithm. To substantiate this notion, we establish a bound for the forward error in terms of the growth factor and present numerical evidence comparing various fast algorithms for matrix and complex multiplications, showing that larger growth factors correlate with less accurate results. Compared to similar studies of numerical stability, bilinear stability is more general, applying to any bilinear operators and not just matrix or complex multiplications; is more simplistic, bounding forward error in terms of a single (growth) factor; and is truly tensorial like bilinear complexity, invariant under any orthogonal change of coordinates. As an aside, we study a new algorithm for computing complex multiplication in terms of real, much like Gauss's, but is optimally fast and stable in that it attains both tensor rank and nuclear norm.

For the problem of complex matrix inversion, we study the speed of various complex matrix inversion algorithms and find the fastest one. We analyze a complex matrix inversion algorithm first proposed by Frobenius, but largely forgotten: $(A+iB)^{-1} = (A+BA^{-1}B)^{-1} - iA^{-1}B(A+BA^{-1}B)^{-1}$ when A is invertible and $(A+iB)^{-1} = B^{-1}A(AB^{-1}A+B)^{-1} - i(AB^{-1}A+B)^{-1}$ when B is invertible. This may be viewed as an inversion analogue of the aforementioned Gauss multiplication. We proved that Frobenius inversion is optimal — it uses the least number of real matrix multiplications and inversions among all complex matrix inversion algorithms. We also showed that Frobenius inversion runs faster than the standard method based on LU decomposition if and only if the ratio of the running time

for real matrix inversion to that for real matrix multiplication is greater than 5/4. We corroborate this theoretical result with extensive numerical experiments, applying Frobenius inversion to evaluate matrix sign function, solve Sylvester equation, and compute polar decomposition, concluding that for these problems, Frobenius inversion is more efficient than LU decomposition with nearly no loss in accuracy.

Besides the three projects discussed in this thesis, I also worked on some projects in statistical learning theory and approximation algorithms over the past few years. My projects on statistical learning theory include "Representation Costs of Linear Neural Networks: Analysis and Design" and "Learning With Square Root Lipschitz Losses". My project on approximation algorithm is "Fair Representation Clustering with Several Protected Classes".

CHAPTER 2

RANK-CONSTRAINED HYPERBOLIC PROGRAMMING

2.1 Introduction

In this work, we study rank-constrained and sparsity-constrained hyperbolic programming (HP). Specifically, we consider four types of HP: linear programming (LP), quadratically constrained quadratic program (QCQP), second order cone programming (SOCP), and semidefinite programming (SDP).

Rank-constrained SDP occurs frequently in combinatorial optimization [2, 79, 84]. It is well-known that Max-Cut could be viewed as a rank-constrained SDP and dropping this rank constraint yields the standard SDP-relaxation of Max-Cut [2]. Thus, it is natural to consider when we can get a solution to SDP of small rank. In [7, 76, 93], it is shown that every feasible SDP with m linear constraints always has a solution of rank at most $(\sqrt{1+8m}-1)/2$. Furthermore, this low-rank solution can be found in polynomial time by first solving the SDP and then run a rank reduction algorithm proposed in [7, 76, 93]. Specifically, a rank reduction algorithm takes a solution to SDP as input and outputs another low-rank solution to this SDP. This result implies that rank-constrained SDP is polynomial time solvable for any rank constraint that is at least $(\sqrt{1+8m}-1)/2$.

Parallel to SDP rank reduction, LP sparsification is studied in [23, 76, 83]. For any feasible LP with m linear constraints, there always exists a solution of cardinality at most m. Moreover, this low-cardinality solution can be found in polynomial time by fist solving the LP and then run a LP sparsification algorithm [23, 76, 83]. Specifically, a LP sparsification algorithm takes a solution to LP as input and outputs another sparse solution.

For rank-constrained problems, we use rank in HP [17, 100] to define rank in LP, QCQP, SOCP, and SDP, by viewing them as special cases of HP. For each of these problems, we study the corresponding rank-constrained problem in two ways. First, we give a polynomial

time rank-reduction algorithm to show that it is always possible to get a solution of "small" rank provided that the problem is feasible. Second, we consider the complexity of these rank-constrained problems. Under certain conditions, we will show that rank-constrained LP is polynomial time solvable and rank-constrained QCQP and rank-constrained SOCP are both NP-hard. For rank-constrained SDP with m linear constraints, we consider rank constraint $r(m)$, that is a function of m. Then, we show that the complexity of rank-constrained SDP changes as $r(m)$ passes through $\sqrt{2m}$. In particular, rank-constrained SDP is NP-hard when $r(m) \ll \sqrt{2m}$ and polynomial time solvable when $r(m) \gg \sqrt{2m}$.

For sparsity-constrained problems, we extend the results in LP sparsification [23, 76, 83] to QCQP and SOCP. Previous results show that every feasible LP with m linear constraints has a solution of cardinality at most m, and we can find such a solution in polynomial time [23, 76, 83]. In addition, there are examples of LP with m constraints whose solutions have cardinality at least m [23, 76, 83], which shows that this LP sparsification result cannot be improved without further assumptions. We extend this result to QCQP and SOCP and show that our results cannot be improved without further assumptions.

2.1.1 Further related works

Rank for Lorentz cone has been studied through the lens of Euclidean Jordan algebra [39, 41, 121]. The definition of rank for points in a Lorentz cone in [39, 41] is the same as the rank for points in SOCP in our work when there is only one second order cone constraint. In this case, the rank estimation theorem in [41] gives the same result as the SOCP rank reduction result in our work.

In addition, our results on SOCP rank reduction can also be deduced from [94, 121]. Specifically, the author gives an algorithm which constructs an extreme point solution from any starting solution [94, 121] for conic LP problems. Applying this algorithm to SOCP gives a solution of small rank.

2.2 Rank-Constrained SDP

In this section, we study Semidefinite Programming (SDP) with rank constraint:

$$\begin{aligned} \underset{X\in\mathbb{S}^n}{\text{minimize}} \quad & \operatorname{tr}(AX) \\ \text{subject to} \quad & \operatorname{tr}(A_iX) = b_i,\ i = 1,\dots,m; \\ & X \geq 0; \\ & \operatorname{rank}(X) \leq r(m), \end{aligned} \tag{2.2.1}$$

where $A, A_1, \dots, A_m \in \mathbb{S}^n$, $b_1, \dots, b_m \in \mathbb{R}$, and $r(m)$ is a function in m. In this section, we begin with some examples of rank-constrained SDP. Then, we study the condition under which SDP is NP-hard. We will show that there is a phase transition in the complexity of rank-constrained SDP when $r(m)$ passes through $\sqrt{2m}$.

2.2.1 Examples of rank-constrained SDP

Rank-constrained SDP appears in many combinatorial problems such as weighted Max-Cut, clique number, and stability number. In the following, we formulate these combinatorial problems in terms of rank-constrained SDP.

Example 2.2.1. Consider a graph $G = (V, E)$ with vertex set V and edge set E. Let $w : E \longrightarrow \mathbb{R}$ be a weight function on G. Without loss of generality, we might assume that G is a complete graph (i.e., $(i, j) \in E$ for all $i, j \in V$) and some of the edges have zero weight (i.e., $w(e) = 0$ for some $e \in E$). In weighted Max-Cut problem, the goal is to find a partition of $V = V_1 \sqcup V_2$, that maximizes the sum of the weights on edges whose endpoints lie in different portions of the partition. To be specific, we want to solve the following problem:

$$\underset{V=V_1\sqcup V_2}{\text{maximize}} \sum_{i\in V_1, j\in V_2} w(i, j).$$

When $w(i,j) \in \{0,1\}$ for all $i,j \in V$, we call this problem the unweighted Max-Cut problem.

For simplicity, we identify V with $[n] := \{1,2,\ldots,n\}$. We define the weight matrix $W \in \mathbb{R}^{n\times n}$ by

$$W[i,j] = \begin{cases} w(i,j)/4 & \text{if} \quad i \neq j \\ -\sum_{k\neq i} w(i,k)/4 & \text{if} \quad i = j. \end{cases}$$

Then weighted Max-Cut is equivalent to the following rank-constrained SDP [2]:

$$\begin{aligned} \underset{X\in\mathbb{S}^n}{\text{minimize}} \quad & \operatorname{tr}(WX) \\ \text{subject to} \quad & X_{ii} = 1,\ i = 1,\ldots,n; \\ & X \geq 0; \\ & \operatorname{rank}(X) \leq 1. \end{aligned} \tag{2.2.2}$$

Example 2.2.2. Next, we consider the problem of computing clique number. Given a graph G, a clique in G is a subgraph H of G such that any two distinct vertices in H are adjacent (i.e., there is an edge between them in G). The clique number $\omega(G)$ of G is the maximum number of vertices in a clique in G.

By results in [79],

$$1 - \frac{1}{\omega(G)} = 2\max_{x\in\Delta^n} \sum_{(i,j)\in E} x_i x_j,$$

where $\Delta^n = \{x \in \mathbb{R}^n : x_1 + \cdots + x_n = 1, x_i \geq 0\}$ is the unit simplex in $\mathbb{R}^n$. Thus, to compute the clique number, it suffices to solve the follow QP:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & \sum_{(i,j)\in E} -x_i x_j \\ \text{subject to} \quad & x_i \geq 0,\ i = 1,\ldots,n; \\ & \sum_{i=1}^n x_i = 1. \end{aligned}$$

By results in [84], we can convert this QP into a rank-constrained SDP. We first homogenize it to obtain the following QP:

$$\begin{aligned} \underset{x\in\mathbb{R}^n, t\in\mathbb{R}}{\text{minimize}} \quad & \sum_{(i,j)\in E} -x_i x_j \\ \text{subject to} \quad & tx_i \geq 0,\ i = 1,\ldots,n; \\ & \sum_{i=1}^{n} tx_i = 1; \\ & t^2 = 1. \end{aligned}$$

This is clearly equivalent to the original QP by substituting tx for x. Let $A \in \mathbb{R}^{(n+1)\times(n+1)}$ be such that

$$A[i,j] = \begin{cases} -1 & \text{if } i \leq n, j \leq n, (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Then the homogenized QP is equivalent to the following rank-constrained SDP [84]:

$$\begin{aligned} \underset{X\in S^{n+1}}{\text{minimize}} \quad & \text{tr}(AX) \\ \text{subject to} \quad & X_{i(n+1)} \geq 0,\ i = 1,\ldots,n; \\ & \sum_{i=1}^{n} X_{i(n+1)} = 1; \\ & X_{(n+1)(n+1)} = 1; \\ & X \geq 0; \\ & \text{rank}(X) \leq 1. \end{aligned}$$

For any solution X, $\text{rank}(X) \neq 0$ since $X_{(n+1)(n+1)} = 1$. Thus, $X = vv^T$ for some $v \in \mathbb{R}^{n+1}$. Then $(x,t) = v$ is a solution to the homogenized QP.

Example 2.2.3. Next, we consider the problem of computing stability number. Given a

graph G, the stability number $\alpha(G)$ of G is the maximum number of vertices in G, of which no two are adjacent (i.e., there is no edge between them in G).

By results in [79],

$$1 - \frac{1}{\alpha(G)} = 2 \max_{x \in \Delta^n} \sum_{(i,j) \notin E} x_i x_j,$$

where $\Delta^n = \{x \in \mathbb{R}^n : x_1 + \cdots + x_n = 1, x_i \geq 0\}$ is the unit simplex in $\mathbb{R}^n$. Thus, to compute the clique number, it suffices to solve the following QP:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \sum_{(i,j) \notin E} -x_i x_j \\ \text{subject to} \quad & x_i \geq 0, \ i = 1, \ldots, n; \\ & \sum_{i=1}^{n} x_i = 1. \end{aligned}$$

Let $B \in \mathbb{R}^{(n+1) \times (n+1)}$ be such that

$$B[i,j] = \begin{cases} -1 & \text{if } i \leq n, j \leq n, (i,j) \notin E, \\ 0 & \text{otherwise.} \end{cases}$$

By the same argument as we used in the clique number example, this QP is equivalent to

the following rank-constrained SDP [84]:

$$\begin{aligned}
\underset{X \in S^{n+1}}{\text{minimize}} \quad & \operatorname{tr}(BX) \\
\text{subject to} \quad & X_{i(n+1)} \geq 0, \ i = 1, \ldots, n; \\
& \sum_{i=1}^{n} X_{i(n+1)} = 1; \\
& X_{(n+1)(n+1)} = 1; \\
& X \geq 0; \\
& \operatorname{rank}(X) \leq 1.
\end{aligned}$$

2.2.2 Complexity of rank-constrained SDP

In this section, we give the condition under which rank-constrained SDP is NP-hard. Recall that rank-constrained SDP is formulated as:

$$\begin{aligned}
\underset{X \in \mathbb{S}^n}{\text{minimize}} \quad & \operatorname{tr}(AX) \\
\text{subject to} \quad & \operatorname{tr}(A_i X) = b_i, \ i = 1, \ldots, m; \\
& X \geq 0; \\
& \operatorname{rank}(X) \leq r(m),
\end{aligned}$$

where $A, A_1, \ldots, A_m \in \mathbb{S}^n$, $b_1, \ldots, b_m \in \mathbb{R}$, and $r(m)$ is a function in m. In last section, we see that weighted Max-Cut can be formulated as a rank-constrained SDP with $r(m) = 1$. As a result, rank-constrained SDP is NP-hard if $r(m) = 1$ for all m. On the other hand, for any feasible SDP, we could first solve the vanilla SDP (i.e., without rank constraint) and then run a rank reduction algorithm to find another optimal solution of rank at most $(\sqrt{1+8m} - 1)/2$ [7, 76, 93]. Thus, if $r(m) \geq (\sqrt{1+8m} - 1)/2$ for all m, then we can always solve the rank-constrained SDP by the procedure we just described. Assuming that we can compute real numbers exactly, solving the vanilla SDP (without rank constraint)

and running the rank reduction algorithm can both be done in polynomial time [7, 76, 93]. Roughly speaking, when $r(m) \gg \sqrt{2m}$, rank-constrained SDP is polynomial time solvable. In the following result, we show that when $r(m) \ll \sqrt{2m}$, rank-constrained SDP is NP-hard.

Theorem 2.2.4. Let $A, A_1, \ldots, A_m \in \mathbb{S}^n$, $b_1, \ldots, b_m \in \mathbb{R}$, and $r : \mathbb{Z}^+ \longrightarrow \mathbb{Z}^+$ be given. Suppose that there exist constants $M, \epsilon > 0$, such that

$$r(m) < 2^{1/2-\epsilon}\sqrt{m}, \quad \text{for all} \quad m \geq M. \tag{2.2.3}$$

Then, the rank-constrained SDP

$$\begin{aligned} \underset{X \in \mathbb{S}^n}{\text{minimize}} \;\; & \operatorname{tr}(AX) \\ \text{subject to} \;\; & \operatorname{tr}(A_i X) = b_i, \; i = 1, \ldots, m; \\ & X \geq 0; \\ & \operatorname{rank}(X) \leq r(m), \end{aligned}$$

is NP-hard.

The above result, together with results of SDP rank reduction [7, 76, 93], show that there is a phase transition in the complexity of rank-constrained SDP as $r(m)$ passes through $\sqrt{2m}$.

Before giving the formal proof, we explain the main ideas of this proof. To begin with, consider the case $r(m) = 2$. Recall that weighted Max-Cut is equivalent to the following rank-constrained SDP:

$$\begin{aligned} \underset{X \in \mathbb{S}^n}{\text{minimize}} \;\; & \operatorname{tr}(WX) \\ \text{subject to} \;\; & X_{ii} = 1, \; i = 1, \ldots, n; \\ & X \geq 0; \\ & \operatorname{rank}(X) \leq 1. \end{aligned} \tag{2.2.4}$$

Now, we transform the above rank-constrained SDP into a new rank-constrained SDP, whose rank constraint is two. Let

$$W' = \begin{bmatrix} 0 & 0 \\ 0 & W \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)},$$

where 0 denotes vectors whose entries are zeros. Then, consider the following rank-constrained SDP:

$$\begin{aligned} \underset{X' \in S^{n+1}}{\text{minimize}} \quad & \text{tr}\Big(W'X'\Big) \\ \text{subject to} \quad & X'_{ii} = 1, \ i = 1, \ldots, n+1; \\ & X'_{1j} = 0, \ j = 2, \ldots, n+1; \\ & X' \geq 0; \\ & \text{rank}(X') \leq 2. \end{aligned} \tag{2.2.5}$$

Note that any solution X' to the above rank-constrained SDP must have the form

$$X' = \begin{bmatrix} 1 & 0 \\ 0 & X \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)},$$

where 0 denotes vectors whose entries are zeros. The rank-constrained SDP in (2.2.5) is equivalent to the one in (2.2.4) since $\text{rank}(X') = \text{rank}(X) + 1$.

By applying this "rank increment" technique, we can show that rank-constrained SDP with constant rank constraint (i.e., $r(m) = r$ for some constant r) is NP-hard. To get the NP-hardness result for r, we need $m = rn + r(r-1)/2 = r(n-1) + r(r+1)/2$ many linear constraints. In other words,

$$n = \left\lfloor \left(m - \frac{r(r+1)}{2} \right) / r \right\rfloor + 1.$$

Thus, we need $m > r(r+1)/2$. Roughly speaking, $r \ll \sqrt{2m}$. In addition, note that as long as $r \ll \sqrt{2m}$, $n = \Omega(\sqrt{m}) = \Omega(r)$. Thus, the new dimension of the problem $n + r - 1$ is polynomial in the original dimension n. Thus, any polynomial time algorithm on this transformed problem translates to a polynomial time algorithm on the original problem.

Proof. We begin with the special case that $r(m)$ is non-decreasing (i.e., $r(m+1) \geq r(m)$ for all m). Then, we will drop this additional assumption.

Special Case:

Our goal is to reduce weighted Max-Cut to rank-constrained SDP. Suppose that there is a polynomial time algorithm $\mathcal{A}$ for rank-constrained SDP with $r(m)$ satisfying equation (2.2.3). Then, we will show that we can use this algorithm to solve weighted Max-Cut in polynomial time. Let

$$\phi(m) = \left\lfloor \left(m - \frac{r(m)(r(m)+1)}{2} \right) \Big/ r(m) \right\rfloor + 1.$$

Given an input graph G with n nodes and a weight function w, we consider two cases $(i) n \geq C$ and $(ii) n < C$ separately, where C is some constant which only depends on ϵ and M. We will pick C later in the proof but it can be determined before we receive the input of weighted Max-Cut.

If $n \geq C$, our algorithm works as follows:

1. find m such that $n = \phi(m)$;
2. construct a rank-constrained SDP with m linear constrains that is equivalent to weighted Max-Cut on the weighted graph (G, w);
3. solve this rank-constrained SDP using algorithm $\mathcal{A}$.

If $n < C$, we use brute force to solve the problem. Now, we discuss each step in detail.

(1) We begin with some observations on $\phi(m)$. For $m \geq M$,

$$\begin{aligned}
\phi(m) &\geq \left(m - \frac{r(m)(r(m)+1)}{2}\right) \Big/ r(m) \\
&\overset{(a)}{\geq} \left(m - 2^{-2\epsilon}m - 2^{-1/2-\epsilon}\sqrt{m}\right) \Big/ r(m) \\
&\overset{(b)}{\geq} \delta\sqrt{m} \quad \text{for some } \delta > 0, \text{ for all } m \geq M', \text{ for some } M' > 0,
\end{aligned} \tag{2.2.6}$$

where δ and M' only depend on ϵ, and we used equation (2.2.3) in (a) and (b). Now, we pick $C = max(M, M') + 1$. Recall that we only consider input graph G with $n \geq C$ nodes.

Since $r(m)$ is non-decreasing, for all $m \geq C - 1$,

$$\begin{aligned}
&\phi(m+1) - \phi(m) \\
&= \left\lfloor \left(m + 1 - \frac{r(m+1)(r(m+1)+1)}{2}\right) \Big/ r(m+1) \right\rfloor - \left\lfloor \left(m - \frac{r(m)(r(m)+1)}{2}\right) \Big/ r(m) \right\rfloor \\
&\leq \left\lfloor \left(m + 1 - \frac{r(m)(r(m)+1)}{2}\right) \Big/ r(m) \right\rfloor - \left\lfloor \left(m - \frac{r(m)(r(m)+1)}{2}\right) \Big/ r(m) \right\rfloor \\
&\leq 1.
\end{aligned} \tag{2.2.7}$$

Note that

$$\phi(n-1) \leq n - 1 + 1 = n. \tag{2.2.8}$$

In addition, by equation (2.2.6),

$$\phi(\lceil n/\delta \rceil^2) \geq n. \tag{2.2.9}$$

By equation (2.2.7), (2.2.8), and (2.2.9), there exists some $m \in [n-1, \lceil n/\delta \rceil^2]$ such that $\phi(m) = n$. By computing $\phi(m)$ from $m = n - 1$ to $m = \lceil n/\delta \rceil^2$, we can find m with $\phi(m) = n$ in $\mathcal{O}(n^2)$ time.

(2) Recall that weighted Max-Cut is equivalent to the following rank-constrained SDP

[2]:

$$\begin{aligned} \underset{X \in \mathbb{S}^n}{\text{minimize}} \;\; & \operatorname{tr}(WX) \\ \text{subject to} \;\; & X_{ii} = 1, \; i = 1, \ldots, n; \\ & X \geq 0; \\ & \operatorname{rank}(X) \leq 1, \end{aligned} \tag{2.2.10}$$

where $W \in \mathbb{R}^{n \times n}$ is the weight matrix induced by the weighted graph (G, w):

$$W[i,j] = \begin{cases} w(i,j)/4 & \text{if} \quad i \neq j \\ -\sum_{k \neq i} w(i,k)/4 & \text{if} \quad i = j. \end{cases}$$

Now, we will construct an equivalent rank-constrained SDP of dimension $n + r(m) - 1$. Let

$$W' = \begin{bmatrix} 0 & 0 \\ 0 & W \end{bmatrix} \in \mathbb{R}^{(n+r(m)-1) \times (n+r(m)-1)},$$

where 0 denotes a matrix whose entries are zeroes. Then, we claim that the rank-constrained SDP in (2.2.10) is equivalent to the following rank-constrained SDP:

$$\begin{aligned} \underset{X' \in \mathbb{S}^{n+r(m)-1}}{\text{minimize}} \;\; & \operatorname{tr}\Big(W'X'\Big) \\ \text{subject to} \;\; & X'_{ii} = 1, \; i = 1, \ldots, n + r(m) - 1; \\ & X'_{ij} = 0, \; j = i+1, \ldots, n + r(m) - 1; \; i = 1, \ldots, r(m) - 1; \\ & X' \geq 0; \\ & \operatorname{rank}(X') \leq r(m), \end{aligned} \tag{2.2.11}$$

To see this, note that any solution X' to the rank-constrained SDP in (2.2.11) must have

the form

$$X' = \begin{bmatrix} I_{r(m)-1} & 0 \\ 0 & X \end{bmatrix} \in \mathbb{R}^{(n+r(m)-1)\times(n+r(m)-1)},$$

where $I_{r(m)-1} \in \mathbb{R}^{(r(m)-1)\times(r(m)-1)}$ is the identity matrix, $X \in \mathbb{S}^n$, and 0 denotes a matrix whose entries are zeroes. Thus, $\text{rank}(X') \leq r(m)$ if and only if $\text{rank}(X) \leq 1$ and $X' \geq 0$ if and only if $X \geq 0$. In addition, $\text{tr}\big(W'X'\big) = \text{tr}(WX)$. Thus, the rank-constrained SDP in (2.2.11) is equivalent to the one in (2.2.10). Thus, in order to solve weighted Max-Cut, it suffices to solve the rank-constrained SDP in (2.2.11). Note that the rank-constrained SDP in (2.2.11) has

$$\sum_{i=n}^{n+r(m)-1} i = \frac{(2n + r(m) - 1)r(m)}{2} = (n-1)r(m) + \frac{r(m)(r(m)+1)}{2} \tag{2.2.12}$$

many linear constraints. Since $n = \phi(m)$,

$$(n-1)r(m) + \frac{r(m)(r(m)+1)}{2} \leq m. \tag{2.2.13}$$

By adding superfluous linear constraints to (2.2.11), we get a rank-constrained SDP which is equivalent to (2.2.11) and has exactly m linear constraints.

(3) Finally, we can apply algorithm $\mathcal{A}$ to solve this rank-constrained SDP in $\text{poly}(n + r(m) - 1)$ time. Since we search for m in $[n-1, \lceil n/\delta \rceil^2]$ in step (1), $m \geq n-1$. Since $n \geq C$, $m \geq C - 1 \geq M$. Thus,

$$r(m) < 2^{1/2-\epsilon}\sqrt{m}. \tag{2.2.14}$$

Since $n = \phi(m) \geq \delta\sqrt{m}$ by equation (2.2.6),

$$r(m) = \mathcal{O}(n),$$

by equation (2.2.14). Thus, algorithm $\mathcal{A}$ solves the rank-constrained SDP in $\text{poly}(n+r(m)-1) = \text{poly}(n)$ time.

Case when $n < C$: If the number of nodes n is less than C, we solve weighted Max-Cut by brute force. We simply try each of the 2^n possible partitions of the vertex set and compute the value of the cut in each case. Then, we take the maximum one. Since $n < C$, this takes at most $\mathcal{O}(2^C)$ time, which is a constant. Thus, overall, the algorithm runs in $\text{poly}(n) + \mathcal{O}(2^C) = \text{poly}(n)$ time.

General Case:

Now, we drop the assumption that $r(m+1) \geq r(m)$ for all m. Note that the only place we used this assumption in the proof of special case is to show that

$$\phi(m+1) - \phi(m) \leq 1.$$

The way to avoid using this assumption is to change the definition of $\phi(m)$. First, note that in the proof of the special case, we used $r(m)$ only when $m \geq M$. Thus, we can assume without loss of generality that

$$r(m) < 2^{1/2-\epsilon}\sqrt{m}, \quad \text{for all} \quad m \in \mathbb{Z}^+. \tag{2.2.15}$$

Then, define $\tilde{r} : \mathbb{Z}^+ \longrightarrow \mathbb{Z}^+$ as

$$\tilde{r}(m) = \max(r(i) : i = 1, \ldots, m). \tag{2.2.16}$$

Clearly,

$$\tilde{r}(m+1) \geq \tilde{r}(m), \quad \text{for all } m. \tag{2.2.17}$$

Note that for any $m \in \mathbb{Z}^+$,

$$\begin{aligned} \tilde{r}(m) &= r(t) \quad \text{for some } t \leq m \\ &\overset{(a)}{<} 2^{1/2-\epsilon}\sqrt{t} \\ &\leq 2^{1/2-\epsilon}\sqrt{m}, \end{aligned} \tag{2.2.18}$$

where we used equation (2.2.15) in (a). Let

$$\phi(m) = \left\lfloor \left(m - \frac{\tilde{r}(m)(\tilde{r}(m)+1)}{2} \right) \Big/ \tilde{r}(m) \right\rfloor + 1. \tag{2.2.19}$$

The skeleton of the algorithm is similar as before. Given an input graph G with n nodes and a weight function w, we consider two cases $(i) n \geq C$ and $(ii) n < C$ separately, where C is some constant which only depends on ϵ and M. We will pick C later in the proof but it can be determined before we receive the input of weighted Max-Cut.

If $n \geq C$, our algorithm works as follows:

1. find m such that $n = \phi(m)$;
2. construct a rank-constrained SDP with m linear constrains that is equivalent to weighted Max-Cut on the weighted graph (G, w);
3. solve this rank-constrained SDP using algorithm $\mathcal{A}$.

If $n < C$, we use brute force to solve the problem. Now, we discuss each step in detail.

(1) By equations (2.2.18), (2.2.17), and (2.2.19), we can get

$$\begin{aligned} &\phi(m) \geq \delta\sqrt{m}; \\ &\phi(m+1) - \phi(m) \leq 1; \\ &\phi(n-1) \leq n; \\ &\phi(\lceil n/\delta \rceil^2) \geq n, \end{aligned} \tag{2.2.20}$$

in the same way as we did in the special case of the proof. Thus, this step remains unchanged.

(2) Once we find m such that $n = \phi(m)$, we consider the rank-constrained SDP in (2.2.11), in exactly the same way as we did in the special case. Note that we use $r(m)$ instead of $\tilde{r}(m)$ here. It is important to note that we use $\tilde{r}(m)$ solely to choose the value of m. After that, we only use $r(m)$. Thus, the number of linear constraints in (2.2.11) is exactly the same as we counted in equation (2.2.12), which is $(n-1)r(m) + r(m)(r(m)+1)/2$. Then,

$$\begin{aligned} & (n-1)r(m) + r(m)(r(m)+1)/2 \\ & \overset{(a)}{\leq} (n-1)\tilde{r}(m) + \tilde{r}(m)(\tilde{r}(m)+1)/2 \\ & \overset{(b)}{\leq} m, \end{aligned} \tag{2.2.21}$$

where we used equation (2.2.16) in (a) and equation (2.2.19) in (b). The rest of this step remains unchanged.

(3) Finally, we need to show that $r(m) = \mathcal{O}(n)$. This holds since $n = \phi(m) \geq \delta\sqrt{m}$ by equation (2.2.20), and $r(m) < 2^{1/2-\epsilon}\sqrt{m}$ by equation (2.2.15).

The case when $n < C$ is exactly the same as before. □

Corollary 2.2.5. Let $A_1, \ldots, A_m \in \mathbb{S}^n$, $b_1, \ldots, b_m \in \mathbb{R}$, and $r : \mathbb{Z}^+ \longrightarrow \mathbb{Z}^+$ be given. Suppose that there exist constants $M, \epsilon > 0$, such that

$$r(m) < 2^{1/2-\epsilon}\sqrt{m}, \quad \text{for all} \quad m \geq M.$$

Then, the rank-constrained SDP feasibility problem:

$$\begin{aligned} \text{Find} \;\; & X \in \mathbb{S}^n \\ \text{subject to} \;\; & \operatorname{tr}(A_i X) = b_i, \; i = 1, \ldots, m; \\ & X \geq 0; \\ & \operatorname{rank}(X) \leq r(m), \end{aligned} \tag{2.2.22}$$

is NP-hard.

Proof. Suppose that there is a polynomial time algorithm which solves (2.2.22). We call this algorithm a feasibility oracle. Then we show that we can also solve the unweighted Max-Cut problem in polynomial time using this feasibility oracle. Let $G = (V, E, w)$ be a weighted graph. Since we are solving the unweighted Max-Cut problem, we may assume that $w(i, j) \in \{0, 1\}$ for all $i, j \in V$. Let $n = |V|$ be the number of nodes in G.

By the same arguments as in the proof of Theorem 2.2.4, unweighted Max-Cut is equivalent to some rank-constrained SDP

$$\begin{aligned} \underset{X \in \mathbb{S}^N}{\text{minimize}} \quad & \operatorname{tr}(AX) \\ \text{subject to} \quad & \operatorname{tr}(A_i X) = b_i, \ i = 1, \ldots, m; \\ & X \geq 0; \\ & \operatorname{rank}(X) \leq r(m), \end{aligned} \tag{2.2.23}$$

where $N = n + r(m) - 1 = \text{poly}(n)$, when $n \geq C$ for some constant C. When $n < C$, we can use brute force to find the solution to unweighted Max-Cut problem in $2^{|E|} = \mathcal{O}(2^{C^2}) = \mathcal{O}(1)$ time. Then, it suffices to solve (2.2.23) using the feasibility oracle. To solve (2.2.23), we just write the objective in (2.2.23) as a linear constraint $\operatorname{tr}(AX) \leq c$ and try for different c by bisection search. For the unweighted Max-Cut problem, we know that the objective is bounded below by 0 and bounded above by n^2. Since we know that the solution to unweighted Max-Cut is an integer, we can find the solution by applying the feasibility oracle $O(\log n)$ times. Thus, we could solve the unweighted Max-Cut in polynomial time. Since unweighted Max-Cut is NP-hard, we are done. □

2.3 Sparsity-Constrained Problems

In this section, we study sparsity-constrained problems. We start with sparsity-constrained Linear Programming (LP):

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}}\ & c^T x \\ \text{subject to}\ & Ax = b; \\ & x \geq 0; \\ & \text{card}(x) \leq \kappa, \end{aligned} \tag{2.3.1}$$

where $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$, $\text{card}(x)$ is the cardinality of x, which is the number of nonzero entries of x, and $\kappa \geq 0$ is a constant. In this case, cardinality is the analogue of rank in SDP. To be specific, if we write LP as SDP such that x is mapped to $X = \text{diag}(x)$, where $\text{diag}(x)$ denotes the diagonal matrix whose diagonal entries are entries of x, then $\text{card}(x) = \text{rank}(X)$. Unlike rank-constrained SDP, sparsity constrained LP (2.3.1) is polynomial time solvable for any constant κ.

Theorem 2.3.1. Let $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$ be given. Then, for any constant $\kappa \geq 0$, the sparsity-constrained LP problem

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}}\ & c^T x \\ \text{subject to}\ & Ax = b; \\ & x \geq 0; \\ & \text{card}(x) \leq \kappa, \end{aligned}$$

is polynomial time solvable.

Proof. To solve this problem, it suffices to solve $\binom{n}{\kappa}$ many LP problems. We encode each LP by a set $S \subset [n]$ of size $n - \kappa$. For each S, we set the variable x_i to be 0 for all $i \in S$. Then

we solve the resulting LP defined as follows. Let $c = (c_1, \ldots, c_n)^T$ and $A = (a_1, \ldots, a_n)$, where $a_i \in \mathbb{R}^m$. Let c_S be the subvector of c obtained by dropping the entries c_i for all $i \in S$. Similarly, let A_S be the submatrix of A obtained by dropping columns a_i for all $i \in S$. Then, the LP corresponding to S is defined as

$$\begin{aligned} \underset{x' \in \mathbb{R}^{n-\kappa}}{\text{minimize}} \ & c_S^T x' \\ \text{subject to} \ & A_S x' = b; \\ & x' \geq 0. \end{aligned}$$

If the above LP is solvable, we keep its solution x_S and optimal value y_S. If it is not solvable, we do nothing. If none of the $\binom{n}{\kappa}$ LP is solvable, we claim that no solution to the sparsity-constrained LP exists. Otherwise, let S^* be the set such that y_{S^*} is minimal among all y_S's. Then, we output the solution $\tilde{x}_{S^*}$ defined by

$$\begin{cases} \tilde{x}_{S^*}[i] = 0 & \text{if } i \in S^* \\ \tilde{x}_{S^*}[i] = x_{S^*}[k_i] & \text{if } i \notin S^*, \end{cases}$$

where i is the k_ith element in $[n] - S^*$, and the optimal value y_{S^*}. Since LP is polynomial time solvable and $\binom{n}{\kappa} \leq n^\kappa$ is polynomial in n, this algorithm runs in polynomial time. $\square$

Note that the above result does not contradict the fact that finding minimum-cardinality solution for LP is NP-hard [44]. The reason is that Theorem 2.3.1 only applies to constant constraint on cardinality. However, in order to find minimum-cardinality solution for LP, we need to consider cardinality constraints that depend on n.

Parallel to the rank reduction results in SDP [7, 76, 93], every feasible LP has a solution x^* whose cardinality is at most m [23, 76, 83]. Next, we extend the sparsification techniques in LP to Quadratically Constrained Quadratic Program (QCQP) and Second Order Cone Programming (SOCP). Before considering QCQP and SOCP sparsification, we make a simple

observation on LP sparsification, which will be used in QCQP and SOCP sparsification. Consider the following LP:

$$\begin{aligned} &\underset{x\in\mathbb{R}^n}{\text{minimize}} \ c^T x \\ &\text{subject to} \ Ax = b; \\ &\qquad x \geq 0, \end{aligned} \tag{2.3.2}$$

where $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$. Given a solution y to (2.3.2), there is an efficient algorithm to find another solution x^* to (2.3.2), whose cardinality is at most m [23, 76, 83]. Now, we observe that the same result holds without the condition $x \geq 0$.

Lemma 2.3.2. Let $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$. Suppose that the following LP

$$\begin{aligned} &\underset{x\in\mathbb{R}^n}{\text{minimize}} \ c^T x \\ &\text{subject to} \ Ax = b; \end{aligned} \tag{2.3.3}$$

has a finite optimal value (i.e., the problem is feasible and the objective is bounded). Then, there exists a solution x^* to (2.3.3) such that $\text{card}(x^*) \leq m$. Moreover, x^* can be found in polynomial time.

Proof. Let y be a solution to (2.3.3). Let $S = \{i \in [n] : y_i < 0\}$. Let $c = (c_1, \ldots, c_n)^T$ and $A = (a_1, \ldots, a_n)$, where $a_i \in \mathbb{R}^m$. Let $\tilde{c} \in \mathbb{R}^n$ be defined as

$$\tilde{c}_i = \begin{cases} c_i & \text{if} \quad i \notin S; \\ -c_i & \text{if} \quad i \in S. \end{cases}$$

Similarly, let $\tilde{A} = (\tilde{a}_1, \ldots, \tilde{a}_n)$ be defined as

$$\tilde{a}_i = \begin{cases} a_i & \text{if} \quad i \notin S; \\ -a_i & \text{if} \quad i \in S. \end{cases}$$

Now, consider the following LP:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \ & \tilde{c}^T x \\ \text{subject to} \ & \tilde{A}x = b; \\ & x \geq 0. \end{aligned} \tag{2.3.4}$$

Let $\tilde{y} = |y|$, where $|\cdot|$ is applied entry-wise. We claim that $\tilde{y}$ is a solution to (2.3.4). To see this, note that $\tilde{A}\tilde{y} = Ay = b$ by definition. Suppose that there exists a solution $\tilde{z} \in \mathbb{R}^n$ to (2.3.4) such that $\tilde{c}^T\tilde{z} < \tilde{c}^T\tilde{y}$. Let $z \in \mathbb{R}^n$ be defined as

$$z_i = \begin{cases} \tilde{z}_i & \text{if} \quad i \notin S; \\ -\tilde{z}_i & \text{if} \quad i \in S. \end{cases}$$

Then, $Az = \tilde{A}\tilde{z} = b$ and $c^T z = \tilde{c}^T\tilde{z} < \tilde{c}^T\tilde{y} = c^T y$, which contradicts the fact that y is a solution to (2.3.3). Thus, $\tilde{y}$ is a solution to (2.3.4).

Now, we apply the LP sparsification algorithm to $\tilde{y}$ to get a solution $\tilde{x}$ of (2.3.4) such that $\text{card}(\tilde{x}) \leq m$. Let $x^* \in \mathbb{R}^n$ be defined as

$$x_i^* = \begin{cases} \tilde{x}_i & \text{if} \quad i \notin S; \\ -\tilde{x}_i & \text{if} \quad i \in S. \end{cases}$$

Then, $Ax^* = \tilde{A}\tilde{x} = b$ and $c^T x^* = \tilde{c}^T\tilde{x} = \tilde{c}^T\tilde{y} = c^T y$. Thus, x^* is a solution to (2.3.3). Note that $\text{card}(x^*) = \text{card}(\tilde{x}) \leq m$. □

2.3.1 QCQP sparsification

Consider the following Quadratically Constrained Quadratic Program (QCQP):

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}}\ & x^TQ_0x + c_0^Tx \\ \text{subject to}\ & x^TQ_ix + c_i^Tx + d_i \le 0,\ \ i = 1,\dots k; \\ & Ax = b, \end{aligned}$$

where $Q_i \in \mathbb{S}_+^n$, $c_i \in \mathbb{R}^n$ for each $i = 0, 1, \dots, k$, $d_i \in \mathbb{R}^n$ for each $i = 1, \dots, k$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$. In this section, we first give a sparsification result on QCQP. Then, we show that this sparsification result cannot be improved without additional assumptions.

Theorem 2.3.3. Let $Q_i \in \mathbb{S}_+^n$, $c_i \in \mathbb{R}^n$ for each $i = 0, 1, \dots, k$, $d_i \in \mathbb{R}^n$ for each $i = 1, \dots, k$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$ be given. Suppose the following QCQP:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}}\ & x^TQ_0x + c_0^Tx \\ \text{subject to}\ & x^TQ_ix + c_i^Tx + d_i \le 0,\ \ i = 1,\dots k; \\ & Ax = b, \end{aligned} \tag{2.3.5}$$

is feasible. Then there exists a solution x^* to (2.3.5) such that

$$\text{card}(x^*) \le m - 1 + \sum_{i=0}^{k}(\text{rank}(Q_i) + 1).$$

Moreover, x^* can be found in polynomial time.

Proof. For each $i = 0, 1, \dots, k$, let $r_i = \text{rank}(Q_i)$. Since $Q_i \in \mathbb{S}_+^n$, there exist an orthogonal matrix U_i and a diagonal matrix D_i such that $Q_i = U_i^TD_iU_i$ and

$$D_i = \begin{bmatrix} B_i^2 & 0 \\ 0 & 0 \end{bmatrix},$$

where $B_i \in \mathbb{R}^{r_i \times r_i}$ is a diagonal matrix. Let

$$P_i = \begin{bmatrix} B_i & 0 \end{bmatrix} U_i \in \mathbb{R}^{r_i \times n}.$$

Then, $Q_i = P_i^T P_i$ and $x^T Q_i x = \|P_i x\|_2^2$. Let $y \in \mathbb{R}^n$ be a solution to the QCQP (2.3.5). Now, consider the following LP:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; & c_0^T x \\ \text{subject to} \;\; & P_i x = P_i y, \;\; i = 0, \ldots, k; \\ & c_i^T x = c_i^T y, \;\; i = 1, \ldots, k; \\ & Ax = b. \end{aligned} \tag{2.3.6}$$

Note that the above LP has a finite optimal value and y is a solution to it. To see this, first y is clearly feasible. Second, if there is a solution z such that $c_0^T z < c_0^T y$, then z is a feasible point of the QCQP (2.3.5) and $z^T Q_i z + c_0^T z = \|P_i z\|_2^2 + c_0^T z < \|P_i y\|_2^2 + c_0^T y = y^T Q_i y + c_0^T y$, which contradicts the fact that y is a solution to the QCQP (2.3.5). Thus, y is a solution to the LP (2.3.6). Now, we can find a solution x^* to the LP (2.3.6) of cardinality at most

$$m - 1 + \sum_{i=0}^{k} (\text{rank}(Q_i) + 1),$$

by Lemma 2.3.2. Since x^* and y are both solution of (2.3.6),

$$c_0^T x^* = c_0^T y.$$

Thus,

$$x^{*T} Q_0 x^* + c_0^T x^* = \|P_0 x^*\|_2^2 + c_0^T y = \|P_0 y\|_2^2 + c_0^T y = y^T Q_0 y + c_0^T y.$$

Thus, x^* is a solution to the QCQP (2.3.5). Since LP sparsification can be done in polynomial

time, we can find x^* in polynomial time by the above procedure. $\square$

Theorem 2.3.3 gives an upper bound on the minimal cardinality of solutions of QCQP. In the following example, we show that this bound is tight, which means that it cannot be improved without additional assumptions.

Example 2.3.4. Let $n, m, \operatorname{rank}(Q_0), \ldots, \operatorname{rank}(Q_k) \in \mathbb{R}$ be given. To make the bound in Theorem 2.3.3 nontrivial, assume that

$$m - 1 + \sum_{i=0}^{k} (\operatorname{rank}(Q_i) + 1) < n.$$

For each $i = 0, 1, \ldots, k$, let

$$r_i = \operatorname{rank}(Q_i), \quad \text{and} \quad s_i = \sum_{j=0}^{i-1} (r_j + 1).$$

Note that $s_0 = 0$ since it is an empty sum. For each $i = 0, 1, \ldots, k$, let

$$Q_i = \begin{bmatrix} 0_{s_i \times s_i} & 0_{s_i \times r_i} & 0_{s_i \times (n-r_i-s_i)} \\ 0_{r_i \times s_i} & I_{r_i} & 0_{r_i \times (n-r_i-s_i)} \\ 0_{(n-r_i-s_i) \times s_i} & 0_{(n-r_i-s_i) \times r_i} & 0_{(n-r_i-s_i) \times (n-r_i-s_i)} \end{bmatrix} \in \mathbb{R}^{n \times n},$$

where $I_{r_i} \in \mathbb{R}^{r_i \times r_i}$ is the identity matrix and $0_{s,t} \in \mathbb{R}^{s \times t}$ denotes a matrix whose entries are zeros. For each $i = 1, \ldots, k$, let

$$c_i = -e_{s_i + r_i + 1} - 2 \sum_{j=s_i+1}^{s_i+r_i} e_j,$$

where $e_{s_i+r_i+1} = (0, \ldots, 0, 1, 0, \ldots, 0)^T \in \mathbb{R}^n$ is the $s_i + r_i + 1$th standard basis vector. Let

$$c_0 = -2\sum_{i=1}^{r_0} e_i + \sum_{i=1}^{k} e_{s_i+r_i+1}.$$

Let

$$A = \begin{bmatrix} 0_{m\times s_{k+1}} & I_m & 0_{m\times(n-m-s_{k+1})} \end{bmatrix}.$$

Consider the following QCQP:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \;& x^T Q_0 x + c_0^T x \\ \text{subject to} \;& x^T Q_i x + c_i^T x + r_i + 1 \leq 0, \;\; i = 1, \ldots k; \\ & Ax = \mathbb{1}_m, \end{aligned} \tag{2.3.7}$$

where $\mathbb{1}_m \in \mathbb{R}^m$ is a vector whose entries are ones. We claim that any solution to the above QCQP has at least $m - 1 + \sum_{i=0}^{k}(\text{rank}(Q_i) + 1)$ nonzero entries. Let $z = (z_1, \ldots, z_n) \in \mathbb{R}^n$ be a feasible point of the QCQP (2.3.7). Then, for each $i = 1, \ldots, k$,

$$z^T Q_i z + c_i^T z + r_i + 1 \leq 0,$$

which implies

$$\sum_{j=s_i+1}^{s_i+r_i} z_j^2 - 2\sum_{j=s_i+1}^{s_i+r_i} z_j - z_{s_i+r_i+1} + r_i + 1 \leq 0,$$

which implies

$$z_{s_i+r_i+1} \geq 1 + \sum_{j=s_i+1}^{s_i+r_i} (z_j^2 - 2z_j + 1) \geq 1. \tag{2.3.8}$$

Then,

$$z^T Q_0 z + c_0^T z = \sum_{i=1}^{r_0}(z_i^2 - 2z_i) + \sum_{i=1}^{k} z_{s_i+r_i+1} \geq -r_0 + \sum_{i=1}^{k} 1 = k - r_0, \tag{2.3.9}$$

where the last step follows from equation (2.3.8). Let $x^* = (1, 1, \ldots, 1, 0, \ldots, 0) \in \mathbb{R}^n$ be a vector whose first $m - 1 + \sum_{i=0}^{k}(r_i + 1)$ entries are ones and the rest are zeros. Then, x^* satisfies all the constraints in the QCQP (2.3.7) and

$$x^{*T} Q_0 x^* + c_0^T x^* = k - r_0. \tag{2.3.10}$$

Thus, by equations (2.3.9) and (2.3.10), the optimal value of the QCQP (2.3.7) is $k - r_0$.

Now, let $y \in \mathbb{R}^n$ be a solution to the QCQP (2.3.7). We will show that $\text{card}(y) \geq m - 1 + \sum_{i=0}^{k}(\text{rank}(Q_i) + 1)$. By equation (2.3.8), we have

$$y_{s_i + r_i + 1} \geq 1,$$

for all $i = 1, \ldots, k$. Since y is optimal, we have

$$k - r_0 = y^T Q_0 y + c_0^T y = \sum_{i=1}^{r_0}(y_i^2 - 2y_i) + \sum_{i=1}^{k} y_{s_i + r_i + 1} \geq -r_0 + \sum_{i=1}^{k} 1 = k - r_0,$$

which implies that

$$y_{s_i + r_i + 1} = 1, \tag{2.3.11}$$

for all $i = 1, \ldots, k$ and

$$y_i = 1, \tag{2.3.12}$$

for all $i = 1, \ldots, r_0$. By equations (2.3.11) and (2.3.8),

$$y_j = 1 \tag{2.3.13}$$

for all $j = s_i + 1, \ldots, s_i + r_i$, for all $i = 1, \ldots, k$. Then, equations (2.3.11), (2.3.12), (2.3.13),

together with the fact that $Ay = \mathbb{1}_m$ implies that

$$y_i = 1$$

for all $i = 1, \ldots, m - 1 + \sum_{i=0}^{k}(\text{rank}(Q_i) + 1)$. Thus,

$$\text{card}(y) \geq m - 1 + \sum_{i=0}^{k}(\text{rank}(Q_i) + 1).$$

This shows that our bound on Theorem 2.3.3 is tight.

2.3.2 SOCP sparsification

Consider the following Second Order Cone Programming (SOCP):

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;& c^T x \\ \text{subject to} \;& \|A_i x + b_i\|_2 \leq c_i^T x + d_i, \;\; i = 1, \ldots, k; \\ & Fx = g, \end{aligned}$$

where $A_i \in \mathbb{R}^{m_i \times n}, b_i \in \mathbb{R}^{m_i}, c_i \in \mathbb{R}^n$, and $d_i \in \mathbb{R}$ for each $i = 1, \ldots, k$, $F \in \mathbb{R}^{m \times n}, c \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$. In this section, we first give a sparsification result on SOCP. Then, we show that this sparsification result cannot be improved without additional assumptions.

Theorem 2.3.5. Let $A_i \in \mathbb{R}^{m_i \times n}, b_i \in \mathbb{R}^{m_i}, c_i \in \mathbb{R}^n$, and $d_i \in \mathbb{R}$ for each $i = 1, \ldots, k$, $F \in \mathbb{R}^{m \times n}, c \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$ be given. Suppose the following SOCP:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;& c^T x \\ \text{subject to} \;& \|A_i x + b_i\|_2 \leq c_i^T x + d_i, \;\; i = 1, \ldots, k; \\ & Fx = g, \end{aligned} \tag{2.3.14}$$

is feasible. Then there exists a solution x^* to (2.3.14) such that

$$\mathrm{card}(x^*) \leq m + \sum_{i=1}^{k}(m_i + 1).$$

Moreover, x^* can be found in polynomial time.

Proof. Let y be a solution to the SOCP (2.3.14). Then, consider the following LP:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \;& c^T x \\ \text{subject to} \;& A_i x = A_i y, \;\; i = 1, \ldots, k; \\ & c_i^T x = c_i^T y, \;\; i = 1, \ldots, k; \\ & Fx = g. \end{aligned} \tag{2.3.15}$$

Note that the above LP has a finite optimal value and y is a solution to it. To see this, first y is clearly feasible. Second, if there is a solution z such that $c^T z < c^T y$, then z is a feasible point of the SOCP (2.3.14) and $c^T z < c^T y$, which contradicts the fact that y is a solution to the SOCP (2.3.14). Thus, y is a solution to the LP (2.3.15). Now, we can find a solution x^* to the LP (2.3.15) of cardinality at most

$$m + \sum_{i=1}^{k}(m_i + 1),$$

by Lemma 2.3.2. Since x^* and y are both solutions of (2.3.15),

$$c^T x^* = c^T y.$$

Thus, x^* is a solution to the SOCP (2.3.14). Since LP sparsification can be done in polynomial time, we can find x^* in polynomial time by the above procedure. □

Theorem 2.3.5 gives an upper bound on the minimal cardinality of solutions of SOCP.

In the following example, we show that this bound is tight, which means that it cannot be improved without additional assumptions.

Example 2.3.6. Let $n, m, m_1, m_2, \ldots, m_k \in \mathbb{R}$ be given. To make the bound in Theorem 2.3.5 nontrivial, assume that

$$m + \sum_{i=1}^{k} (m_i + 1) < n.$$

For each $i = 1, \ldots, k+1$, let

$$r_i = \sum_{j=1}^{i-1} (m_j + 1).$$

Note that $r_1 = 0$ since it is an empty sum. For each $i = 1, \ldots, k$, let

$$c_i = e_{r_i + m_i + 1}$$

and

$$E_i = \begin{bmatrix} 0_{m_i \times r_i} & I_{m_i} & 0_{m_i \times (n - r_{i+1} + 1)} \end{bmatrix} \in \mathbb{R}^{m_i \times n},$$

where $I_{m_i} \in \mathbb{R}^{m_i \times m_i}$ is the identity matrix, $0_{s,t} \in \mathbb{R}^{s \times t}$ denotes a matrix whose entries are zeros, and $e_{r_i + m_i + 1} = (0, \ldots, 0, 1, 0, \ldots, 0)^T \in \mathbb{R}^n$ is the $r_i + m_i + 1$th standard basis vector. Let

$$F = \begin{bmatrix} 0_{m \times r_{k+1}} & I_m & 0_{m \times (n - m - r_{k+1})} \end{bmatrix}.$$

Let

$$c = \sum_{i=1}^{k} c_i.$$

Consider the following SOCP:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; & c^T x \\ \text{subject to} \;\; & \|E_i x - \mathbb{1}_{m_i}\|_2 \leq c_i^T x - 1, \;\; i = 1, \ldots, k; \\ & Fx = \mathbb{1}_m, \end{aligned} \tag{2.3.16}$$

where $\mathbb{1}_d \in \mathbb{R}^d$ is a vector whose entries are ones. We claim that any solution to the above SOCP has at least $m + \sum_{i=1}^{k}(m_i + 1)$ nonzero entries. Let z be a feasible point of the SOCP (2.3.16). Then, for each $i = 1, \ldots, k$,

$$z_{r_i+m_i+1} - 1 = c_i^T z - 1 \geq \|E_i z - \mathbb{1}_{m_i}\|_2 \geq 0,$$

which implies that

$$z_{r_i+m_i+1} \geq 1 \tag{2.3.17}$$

for all $i = 1, \ldots, k$. Thus,

$$c^T z = \sum_{i=1}^{k} z_{r_i+m_i+1} \geq k. \tag{2.3.18}$$

Let $x^* = (1, 1, \ldots, 1, 0, \ldots, 0) \in \mathbb{R}^n$ be a vector whose first $m + \sum_{i=1}^{k}(m_i + 1)$ entries are ones and the rest are zeros. Then, x^* satisfies all the constraints in the SOCP (2.3.16) and

$$c^T x^* = k. \tag{2.3.19}$$

Thus, by equations (2.3.18) and (2.3.19), the optimal value of the SOCP (2.3.16) is k.

Now, let $y \in \mathbb{R}^n$ be a solution to the SOCP (2.3.16). We will show that $\text{card}(y) \geq m + \sum_{i=1}^{k}(m_i + 1)$. By equation (2.3.17), we have

$$y_{r_i+m_i+1} \geq 1, \tag{2.3.20}$$

for all $i = 1, \ldots, k$. Since y is optimal, we have

$$k = c^T y = \sum_{i=1}^{k} y_{r_i+m_i+1}. \tag{2.3.21}$$

By equations (2.3.20) and (2.3.21),

$$y_{r_i+m_i+1} = 1, \tag{2.3.22}$$

for all $i = 1, \ldots, k$. This implies that for each $i = 1, \ldots, k$,

$$\|E_i y - \mathbb{1}_{m_i}\|_2 = 0, \tag{2.3.23}$$

which implies that

$$E_i y = \mathbb{1}_{m_i}. \tag{2.3.24}$$

Then, equations (2.3.22) and (2.3.24), together with the fact that $Fy = \mathbb{1}_m$, imply that

$$y_i = 1$$

for all $i = 1, \ldots, m + \sum_{j=1}^{k}(m_j + 1)$. Thus,

$$\text{card}(y) \geq m + \sum_{j=1}^{k}(m_j + 1).$$

This implies that our bound in Theorem 2.3.5 is tight.

2.4 Rank-Constrained Hyperbolic Programming

In this section, we consider rank-constrained Hyperbolic Programming (HP), which unifies rank-constrained SDP and sparsity-constrained LP. We extend rank reduction techniques to rank-constrained QCQP and rank-constrained SOCP, which are special cases of rank-constrained HP. In addition, we study the complexity of these two optimization problems.

We first recall the definition of a hyperbolic polynomial [17, 48, 100].

Definition 2.4.1. A homogeneous polynomial $p : \mathbb{R}^n \longrightarrow \mathbb{R}$ is hyperbolic if there exists a direction $e \in \mathbb{R}^n$ such that $p(e) \neq 0$ and for each $x \in \mathbb{R}^n$ the univariate polynomial $t \mapsto p(x - te)$ has only real roots. The polynomial p is said to be hyperbolic in direction e.

Then, we recall the definition of characteristic polynomial and eigenvalues in HP [17, 48, 100].

Definition 2.4.2. Given $x \in \mathbb{R}^n$ and a polynomial $p : \mathbb{R}^n \longrightarrow \mathbb{R}$ that is hyperbolic in direction $e \in \mathbb{R}^n$, the characteristic polynomial of x with respect to p in direction e is the univariate polynomial $\lambda \mapsto p(x - \lambda e)$. The roots of the characteristic polynomial are the eigenvalues of x.

Next, we recall the definition of hyperbolic programming [17, 48, 100].

Definition 2.4.3. Given a hyperbolic polynomial $p : \mathbb{R}^n \longrightarrow \mathbb{R}$ that is hyperbolic in direction $e \in \mathbb{R}^n$, the hyperbolic cone for p in direction e is defined as

$$\Lambda_{++} := \{x \in \mathbb{R}^n : \lambda_{\min}(x) > 0\},$$

where $\lambda_{\min}(x)$ is the minimum eigenvalue of x. Let

$$\Lambda_{+} := \{x \in \mathbb{R}^n : \lambda_{\min}(x) \geq 0\}$$

be the closure of Λ_{++}.

Definition 2.4.4. Given a hyperbolic polynomial $p : \mathbb{R}^n \longrightarrow \mathbb{R}$ that is hyperbolic in direction $e \in \mathbb{R}^n$, a hyperbolic program is an optimization problem of the form

$$\begin{aligned} &\underset{x \in \mathbb{R}^n}{\text{minimize}} \ c^T x \\ &\text{subject to } \ Ax = b; \\ &\qquad\qquad x \in \Lambda_{+}, \end{aligned} \tag{2.4.1}$$

where $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$ are given.

Finally, we recall the definition of rank in HP [17, 48, 100].

Definition 2.4.5. Let p be a hyperbolic polynomial that is hyperbolic in direction $e \in \mathbb{R}^n$. The rank of $x \in \mathbb{R}^n$ is defined as $\text{rank}(x) := \deg p(e + tx)$, where t is the indeterminate. Equivalently, $\text{rank}(x)$ is the number of non-zero eigenvalues of x.

Note that HP includes SDP and LP as special cases. Moreover, rank-constrained HP includes rank-constrained SDP and sparsity-constrained LP as special cases. To be specific, when $p(X) = \det(X)$ and $e = I$ (in this case, the domain of p is the set of symmetric matrices which can be identified as $\mathbb{R}^{n(n+1)/2}$), HP becomes SDP and $\text{rank}(X)$ is simply the usual rank of a matrix [17, 48, 100]. In addition, when $p(x) = \prod_{i=1}^n x_i$, where x_i is the ith component of x and $e = (1, 1, \ldots, 1)$, HP becomes LP and $\text{rank}(x) = \text{card}(x)$ [17, 48, 100].

2.4.1 Rank-Constrained SOCP

In this section, we study rank-constrained SOCP. We first define the rank of SOCP by viewing it as a HP. Then, we give a rank reduction result for SOCP. Next, we show that Max-Cut can be written as a rank-constrained SOCP. Finally, we study the complexity of rank-constrained SOCP and show that it is NP-hard in certain circumstances.

SOCP rank reduction

We consider the following SOCP:

$$\begin{aligned} &\underset{x\in\mathbb{R}^n}{\text{minimize}} \ c^T x \\ &\text{subject to } \|A_i x + b_i\|_2 \le c_i^T x + d_i, \ \ i = 1, \ldots, k; \\ &\qquad\qquad Fx = g, \end{aligned} \tag{2.4.2}$$

where $A_i \in \mathbb{R}^{m_i \times n}, b_i \in \mathbb{R}^{m_i}, c_i \in \mathbb{R}^n$, and $d_i \in \mathbb{R}$ for each $i = 1, \ldots, k$, $F \in \mathbb{R}^{m \times n}, c \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$. We first show that it can be written as a HP. This step is similar to the Lorentz cone example in [9]. First, we associate each second order cone constraint $\|A_i x + b_i\|_2 \leq c_i^T x + d_i$ with a new variable y_i. Let $y = (y_1, \ldots, y_k)$. Let $z = (x, y)$. For each $j \in [m_i]$, let $a_{i,j}^T$ be the jth row of A_i, $b_{i,j}$ be the jth entry of b_i, and let $q_{i,j}(x) = a_{i,j}^T x + b_{i,j}$. Then let

$$p_i(z) = y_i^2 - \sum_{j=1}^{m_i} q_{i,j}(x)^2.$$

Now let

$$p(z) = \prod_{i=1}^{k} p_i(z).$$

Let $e = (0, \ldots, 0, 1, \ldots, 1)$ where there are n zeros followed by k ones. For each i,

$$p_i(z - te) = (y_i - t)^2 - \sum_{j=1}^{m_i} q_{i,j}(x)^2 = t^2 - 2y_i t + \left(y_i^2 - \sum_{j=1}^{m_i} q_{i,j}(x)^2 \right). \tag{2.4.3}$$

The discriminant

$$\Delta_i = 4y_i^2 - 4\left(y_i^2 - \sum_{j=1}^{m_i} q_{i,j}(x)^2 \right) = 4 \sum_{j=1}^{m_i} q_{i,j}(x)^2 \geq 0.$$

Thus, p_i is hyperbolic in e for all i. Hence, p is hyperbolic in e. Note that roots of $p(z - te)$ are positive if and only if roots of $p_i(z - te)$ are positive for all i. From equation (2.4.3), we see that this holds if and only if

$$y_i \geq 0 \quad \text{and} \quad y_i^2 \geq \sum_{j=1}^{m_i} q_{i,j}(x)^2.$$

This is equivalent to

$$y_i \geq \sqrt{\sum_{j=1}^{m_i} q_{i,j}(x)^2} = \|A_i x + b_i\|_2.$$

Now for each i, we add the linear constraint

$$y_i = c_i^T x + d_i.$$

Then the resulting HP with the original linear constraint $Fx = g$ and the new linear constraints $y_i = c_i^T x + d_i$ is equivalent to the SOCP problem (2.4.2).

Next, we consider the rank. Note that

$$p_i(e + tz) = (ty_i + 1)^2 - \sum_{j=1}^{m_i} q_{i,j}(x)^2 t^2 = \left(y_i^2 - \sum_{j=1}^{m_i} q_{i,j}(x)^2 \right) t^2 + 2y_i t + 1.$$

Then we have

$$\deg(p_i) = \begin{cases} 0 & \text{if } y_i^2 = \sum_{j=1}^{m_i} q_{i,j}(x)^2 = 0 \\ 1 & \text{if } y_i^2 = \sum_{j=1}^{m_i} q_{i,j}(x)^2 \neq 0 \\ 2 & \text{otherwise.} \end{cases}$$

Let $s(x)$ be the number of second order cone constraints that are satisfied with equality. Let $e(x)$ be the number of second order cone constraints that are satisfied with equality and both sides of the constraints are zero. Since

$$\deg(p(e + tz)) = \sum_{i=1}^{k} \deg(p_i(e + tz)),$$

we have

$$\operatorname{rank}(z) = 2k - s(x) - e(x).$$

In addition, we define

$$\operatorname{rank}(x) = \operatorname{rank}(z).$$

Now we consider SOCP rank reduction.

Theorem 2.4.6. Let $A_i \in \mathbb{R}^{m_i \times n}, b_i \in \mathbb{R}^{m_i}, c_i \in \mathbb{R}^n$, and $d_i \in \mathbb{R}$ for each $i = 1, \ldots, k$,

$F \in \mathbb{R}^{m\times n}, c \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$ be given. Suppose the following SOCP:

$$\begin{aligned} &\underset{x\in\mathbb{R}^n}{\text{minimize}} \ c^T x \\ &\text{subject to } \|A_i x + b_i\|_2 \le c_i^T x + d_i, \ \ i = 1, \dots, k; \\ &\qquad\qquad Fx = g, \end{aligned} \tag{2.4.4}$$

is feasible and

$$\bigcap_{i\in[k+1]} \ker B_i = \{0\},$$

where

$$B_i = \begin{bmatrix} A_i \\ c_i^T \end{bmatrix} \quad \text{for each } \ i \in [k], \ \text{ and } \quad B_{k+1} = \begin{bmatrix} F \\ c^T \end{bmatrix}.$$

Then there exists a solution x^* to 2.4.4 such that

$$\operatorname{rank}(x) \le 2k - \left\lceil \frac{n}{\max(m, m_1, m_2, \dots, m_k) + 1} \right\rceil + 1.$$

Moreover, x^* can be found in polynomial time.

Proof. It suffices to show that we can find a solution x such that

$$s(x) \ge \left\lceil \frac{n}{\max(m, m_1, m_2, \dots, m_k) + 1} \right\rceil - 1.$$

Let

$$m' = \max(m, m_1, m_2, \dots, m_k).$$

Let $x^{(0)}$ be a solution to the SOCP. We define an algorithm iteratively with the inductive hypothesis that at the beginning of step i,

$$\|A_j x^{(i-1)} + b_j\|_2 = c_j^T x^{(i-1)} + d_j,$$

for all $j \in S_{i-1}$, where $x^{(i)}$ is the value of x at the end of the ith iteration and S_i is a set of size i which is updated in each iteration. This clearly holds for $i = 1$ with $S_0 = \emptyset$. At step i, we do the following. Let $S_{i-1} = \{u_1, \ldots, u_{i-1}\}$ and

$$M_i = \begin{bmatrix} B_{k+1} \\ B_{u_1} \\ \vdots \\ B_{u_{i-1}} \end{bmatrix}$$

Then $M_i \in \mathbb{R}^{t_i \times n}$ for some $t_i \leq (m' + 1)i$. Suppose that

$$i \leq \left\lceil \frac{n}{m' + 1} \right\rceil - 1.$$

Then $t_i \leq (m' + 1)i < n$. Thus, $\ker M_i \neq \{0\}$. Take $v \in \ker M_i$ such that $v \neq 0$. Since

$$\bigcap_{i \in [k+1]} \ker B_i = \{0\},$$

there exists $u \in [k]$, such that $B_u v \neq 0$. Clearly, $u \notin S_{i-1}$. Now we consider two cases. If $c_u^T v = 0$, then $A_u v \neq 0$. Thus,

$$\|A_u(x^{(i-1)} + \lambda v) + b_u\|_2 - (c_u^T(x^{(i-1)} + \lambda v) + d_u) \longrightarrow \infty,$$

as $\lambda \longrightarrow \infty$. Since the above expression is less than or equal to 0 when $\lambda = 0$, there exists λ^* such that

$$\|A_u(x^{(i-1)} + \lambda^* v) + b_u\|_2 - (c_u^T(x^{(i-1)} + \lambda^* v) + d_u) = 0.$$

Now if $c_u^T v \neq 0$, then multiplying v by -1 if necessary, we can assume that $c_u^T v < 0$. Thus,

$$\|A_u(x^{(i-1)} + \lambda v) + b_u\|_2 - (c_u^T(x^{(i-1)} + \lambda v) + d_u) \longrightarrow \infty,$$

as $\lambda \longrightarrow \infty$. Then again we have there exists λ^* such that

$$\|A_u(x^{(i-1)} + \lambda^* v) + b_u\|_2 - (c_u^T(x^{(i-1)} + \lambda^* v) + d_u) = 0.$$

Now let

$$E = \{u : B_u v \neq 0\}.$$

For each $u \in E$, let

$$\lambda_u = \arg\min_{\lambda \in \mathbb{R}} \left\{ |\lambda| : \|A_u(x^{(i-1)} + \lambda v) + b_u\|_2 = (c_u^T(x^{(i-1)} + \lambda v) + d_u) \right\}.$$

Now let

$$u_i \in \arg\min_{u \in E} |\lambda_u|.$$

Update

$$x^{(i)} = x^{(i-1)} + \lambda_{u_i} v$$

$$S_i = S_{i-1} \cup \{u_i\}.$$

Then clearly

$$\|A_j x^{(i-1)} + b_j\|_2 = c_j^T x^{(i-1)} + d_j,$$

for all $j \in S_i$ and

$$|S_i| = |S_{i-1}| + 1 = i.$$

Note that $x^{(i)}$ is still a feasible solution since for all $u \in E$,

$$\|A_u(x^{(i-1)} + \lambda^* v) + b_u\|_2 - (c_u^T(x^{(i-1)} + \lambda^* v) + d_u) \leq 0,$$

by minimality of $|\lambda_{u_i}|$. This process stop when $i = \lceil n/(m'+1) \rceil$. Then we have

$$\|A_j x^{(i-1)} + b_j\|_2 = c_j^T x^{(i-1)} + d_j,$$

for all $j \in S_{\lceil n/(m'+1) \rceil - 1}$ and thus

$$s(x^{(i-1)}) \geq \left\lceil \frac{n}{m'+1} \right\rceil - 1.$$

□

Complexity of rank-constrained SOCP

In this section, we study the complexity of rank-constrained SOCP. First, we will show that Max-Cut could be formulated as rank-constrained SOCP. Then, we will use this reduction to show that rank-constrained SOCP is NP-hard under certain circumstances. Our reduction of Max-Cut to rank-constrained SOCP is a slight modification of the SOCP relaxation results in [91].

Example 2.4.7. First, we consider a nonconvex Quadratically Constrained Linear Program (QCLP):

$$\begin{aligned} &\underset{x \in \mathbb{R}^n}{\text{minimize}} \ c^T x \\ &\text{subject to } \ x^T Q_i x + g_i^T x + f_i = 0, \ \ i = 1, \ldots, m, \end{aligned} \tag{2.4.5}$$

where $Q_i \in \mathbb{S}^n$, $g_i \in \mathbb{R}^n$, and $f_i \in \mathbb{R}$. We will first show that the above nonconvex QCLP is

equivalent to a rank-constrained SOCP. Then, since Max-Cut can be written as a nonconvex QCLP, Max-Cut is also equivalent to a rank-constrained SOCP.

QCLP to rank-constrained SOCP: Since

$$x^T Q_i x = \mathrm{tr}\Big(Q_i x x^T\Big),$$

the QCLP in (2.4.5) is clearly equivalent to

$$\begin{aligned} \underset{x\in\mathbb{R}^n, X\in\mathbb{S}^n}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & \mathrm{tr}(Q_i X) + g_i^T x + f_i = 0, \;\; i = 1, \ldots, m, \\ & X - xx^T = 0. \end{aligned}$$

The above is an LP with the additional constraint $X - xx^T = 0$. Thus, it suffices to write that constraint as a second order cone constraint. Let $C_1, \ldots, C_r$ be an orthonormal basis for $\mathbb{S}^n$. Then $X - xx^T = 0$ if and only if

$$\mathrm{tr}\Big(C_i(X - xx^T)\Big) = 0,$$

for all $i \in [r]$. For each i, there exists λ_i such that $C_i + \lambda_i I \in \mathbb{S}^n_+$. Let $\tilde{C}_i = C_i + \lambda_i I$. Then for all $i \in [r]$, $\mathrm{tr}\Big(C_i(X - xx^T)\Big) = 0$ if

$$\mathrm{tr}\Big(\tilde{C}_i(X - xx^T)\Big) = 0, \;\; \text{and} \;\; \mathrm{tr}\Big(I(X - xx^T)\Big) = 0. \tag{2.4.6}$$

On the other hand, if $X - xx^T = 0$, then equation (2.4.6) certainly holds for all $i \in [r]$. Thus, $X - xx^T = 0$ if and only if equation (2.4.6) holds for all $i \in [r]$.

Now let $A \in \mathbb{S}^n_+$. Then $A = VV^T$ for some $V \in \mathbb{R}^{n\times n}$. Note that

$$\mathrm{tr}\Big(A(X - xx^T)\Big) = \mathrm{tr}(AX) - x^T VV^T x = a - u^T u, \tag{2.4.7}$$

where $a = \text{tr}(AX), u = V^T x$. Now note that

$$a - u^T u = 0$$

if and only if

$$(a+1)^2 = (a-1)^2 + 4u^T u.$$

Let $w = \begin{bmatrix} a-1 \\ 2u \end{bmatrix}$, then the above equation is equivalent to

$$\|w\|_2 = a + 1.$$

In other words, we get

$$\left\| \begin{bmatrix} \text{tr}(AX) - 1 \\ 2V^T x \end{bmatrix} \right\|_2 = \text{tr}(AX) + 1, \tag{2.4.8}$$

which is a second order cone constraint with equality. Note that the left hand side and right hand side of (2.4.8) cannot both be 0. Otherwise, we would have $tr(AX) = 1$ from the left hand side and $tr(AX) = -1$ from the right hand side, which is a contradiction. Thus,

$$e(x) = 0,$$

for all x that satisfies (2.4.8), where $e(x)$ is the number of second order cone constrained that are satisfied with equality and both sides are zero. Then, we need

$$s(x) = k \qquad \text{and} \qquad e(x) = 0,$$

where

$$k = 1 + \frac{n(n+1)}{2}$$

is the number of second order cone constraints and $s(x)$ is the number of second order cone constrained that are satisfied with equality. Thus, the original QCLP could be written as a SOCP problem with the additional rank constraint

$$\operatorname{rank}(x) \leq k.$$

Max-Cut to QCLP: It is well known that Max-Cut can be written in the form [91]

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; & x^T Q x \\ \text{subject to} \;\; & x_i^2 - 1 = 0, \;\; i = 1, \ldots, n, \end{aligned}$$

for some $Q \in \mathbb{S}^n$. Then it is equivalent to

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; & t \\ \text{subject to} \;\; & x_i^2 - 1 = 0, \;\; i = 1, \ldots, n, \\ & x^T Q x - t = 0, \end{aligned}$$

which is then in the form of the nonconvex QCLP in (2.4.5). Thus, Max-Cut is equivalent to a rank-constrained SOCP.

From the above example, we see that rank-constrained SOCP includes some interesting combinatorial problems, which motivates the study of rank-constrained SOCP:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; & c^T x \\ \text{subject to} \;\; & \|A_i x + b_i\|_2 \leq c_i^T x + d_i, \;\; i = 1, \ldots, k; \\ & Fx = g, \\ & \operatorname{rank}(x) \leq r(k) \end{aligned}$$

where $r(k)$ is a function in k, $A_i \in \mathbb{R}^{m_i \times n}, b_i \in \mathbb{R}^{m_i}, c_i \in \mathbb{R}^n$, and $d_i \in \mathbb{R}$ for each $i = 1, \ldots, k$,

$F \in \mathbb{R}^{m\times n}, c \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$. Next, we give a complexity result of this problem.

Theorem 2.4.8. Let $A_i \in \mathbb{R}^{m_i\times n}, b_i \in \mathbb{R}^{m_i}, c_i \in \mathbb{R}^n$, and $d_i \in \mathbb{R}$ for each $i = 1, \ldots, k$, $F \in \mathbb{R}^{m\times n}, c \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$ be given. Let $s \geq 0$ be a constant. Then the following rank-constrained SOCP:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \;& c^T x \\ \text{subject to} \;& \|A_i x + b_i\|_2 \leq c_i^T x + d_i, \;\; i = 1, \ldots, k; \\ & Fx = g, \\ & \text{rank}(x) \leq k + s, \end{aligned}$$

is NP-hard.

Proof. From the example at the beginning of this section, we see that rank-constrained SOCP with $r(k) = k$ is NP-hard since we can reduce Max-Cut to it. In other words, the following problem:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \;& c^T x \\ \text{subject to} \;& \|A_i x + b_i\|_2 \leq c_i^T x + d_i, \;\; i = 1, \ldots, k; \\ & Fx = g, \\ & \text{rank}(x) \leq k, \end{aligned} \tag{2.4.9}$$

is NP-hard. Now, we show how to increase $r(k)$ from k to $k + 1$. Consider the following

problem:

$$\begin{aligned} \underset{x\in\mathbb{R}^{n+1}}{\text{minimize}}\ & \tilde{c}^T x \\ \text{subject to}\ & \left\|\tilde{A}_i x + \tilde{b}_i\right\|_2 \leq \tilde{c_i}^T x + d_i,\ \ i = 1,\ldots,k; \\ & \tilde{F}x = g, \\ & x_{n+1} = 1, \\ & \|Ex\|_2 \leq 2, \\ & \text{rank}(x) \leq (k+1)+1, \end{aligned} \tag{2.4.10}$$

where $\tilde{A}_i = \begin{bmatrix} A_i \\ 0 \end{bmatrix}, \tilde{c} = \begin{bmatrix} c \\ 0 \end{bmatrix}, \tilde{b}_i = \begin{bmatrix} b_i \\ 0 \end{bmatrix}, \tilde{c}_i = \begin{bmatrix} c_i \\ 0 \end{bmatrix}, \tilde{F} = [\, F\ 0\,], E = e_{n+1}e_{n+1}^T$, and $e_{n+1} = (0,\ldots,0,1)$. Note that since $x_{n+1} = 1$,

$$\|Ex\|_2 = 1 < 2.$$

Thus, this second order cone constraint will contribute 2 to the rank. Thus, (2.4.10) is equivalent to (2.4.9). Note that there are $k+1$ second order cone constraints in (2.4.10). Using this argument s times finishes the proof. □

2.4.2 Rank-Constrained QCQP

In this section, we study rank-constrained QCQP. We first define the rank of QCQP by viewing it as a SOCP. Then, we apply the results in the previous section to do rank reduction on QCQP. Next, we show that Max-Cut can also be written as a rank-constrained QCQP. Finally, we show that rank-constrained QCQP is NP-hard in certain circumstances.

QCQP rank reduction

Consider the following QCQP:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}}\ & x^TQ_0x + c_0^Tx \\ \text{subject to}\ & x^TQ_ix + c_i^Tx + d_i \leq 0,\ \ i = 1,\dots k; \\ & Ax = b, \end{aligned} \tag{2.4.11}$$

where $Q_i \in \mathbb{S}_+^n$, $c_i \in \mathbb{R}^n$ for each $i = 0, 1, \dots, k$, $d_i \in \mathbb{R}^n$ for each $i = 1, \dots, k$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$. First, we write QCQP (2.4.11) as a SOCP (2.4.2). To do this, we first consider the epigraph version of (2.4.11):

$$\begin{aligned} \underset{x\in\mathbb{R}^n, t\in\mathbb{R}}{\text{minimize}}\ & t \\ \text{subject to}\ & x^TQ_0x + c_0^Tx - t \leq 0 \\ & x^TQ_ix + c_i^Tx + d_i \leq 0,\ \ i = 1,\dots k; \\ & Ax = b. \end{aligned} \tag{2.4.12}$$

Then, it suffices to convert a quadratic constraint

$$x^TQ_ix + c_i^Tx + d_i \leq 0 \tag{2.4.13}$$

to a second order cone constraint. For each $i = 0, 1, \dots, k$, let $r_i = \text{rank}(Q_i)$. Then there exists $P_i \in \mathbb{R}^{r_i\times r_i}$ such that $Q_i = P_i^TP_i$ since $Q_i \in \mathbb{S}_+^n$. Then, equation (2.4.13) becomes

$$\|P_ix\|_2^2 \leq -c_i^Tx - d_i. \tag{2.4.14}$$

Since

$$-c_i^Tx - d_i = (1/4 - c_i^Tx - d_i)^2 - (1/4 + c_i^Tx + d_i)^2,$$

equation (2.4.14) is equivalent to

$$\|P_i x\|_2^2 + (1/4 + c_i^T x + d_i)^2 \leq (1/4 - c_i^T x - d_i)^2,$$

which is equivalent to

$$\left\| \begin{bmatrix} P_i x \\ 1/4 + c_i^T x + d_i \end{bmatrix} \right\|_2 \leq \|1/4 - c_i^T x - d_i\|_2, \tag{2.4.15}$$

which is a second order cone constraint. Note that the left hand side and right hand side of equation (2.4.15) cannot both be zeroes, since $1/4 + c_i^T x + d_i$ and $1/4 - c_i^T x - d_i$ cannot both be zeroes. Thus, according to the definition of rank for SOCP, the rank in QCQP (2.4.11) is defined as

$$\text{rank}(x) := 2k + 1 - s(x),$$

where $s(x)$ is the number of quadratic constraints in (2.4.11) that are satisfied with equality (i.e., $x^T Q_i x + c_i^T x + d_i = 0$). Note that by writing QCQP as SOCP, we have $k + 1$ second order cone constraints. However, $x^T Q_0 x + c_0^T x - t = 0$ for any x that attains the optimal value. Note that in QCQP,

$$\text{rank}(x) \geq k + 1$$

for all $x \in \mathbb{R}^n$.

Now, we do rank reduction on QCQP. By viewing QCQP as a SOCP and applying Theorem 2.4.6, we get the following result.

Theorem 2.4.9. Let $Q_i \in \mathbb{S}_+^n$, $c_i \in \mathbb{R}^n$ for each $i = 0, 1, \ldots, k$, $d_i \in \mathbb{R}^n$ for each $i = 1, \ldots, k$,

$A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$ be given. Suppose the following QCQP:

$$\begin{aligned} &\underset{x\in\mathbb{R}^n}{\text{minimize}} \;\; x^T Q_0 x + c_0^T x \\ &\text{subject to} \;\; x^T Q_i x + c_i^T x + d_i \leq 0, \;\; i = 1, \dots k; \\ &\qquad\qquad Ax = b, \end{aligned} \tag{2.4.16}$$

is feasible and

$$\bigcap_{i=0}^{k+1} \ker B_i = \{0\},$$

where

$$B_i = \begin{bmatrix} Q_i \\ c_i^T \end{bmatrix} \quad \text{for each } \; i = 0, 1, \dots, k, \;\; \text{and} \quad B_{k+1} = A.$$

Then there exists a solution x^* to 2.4.16 such that

$$\operatorname{rank}(x^*) \leq 2k + 3 - \left\lceil \frac{n}{\max(m-1, \operatorname{rank}(Q_0), \operatorname{rank}(Q_1), \dots, \operatorname{rank}(Q_k)) + 1} \right\rceil.$$

Moreover, x^* can be found in polynomial time.

Proof. First, we write the QCQP as a SOCP. Then we have $k+1$ second order cone constraints. Note that for B_{k+1}, we don't need to concatenate A with c_0 as we did in the SOCP case since the objective $x^T Q_0 x + c_0^T x$ becomes a second order cone constraint. In addition, if $Q_i = P_i^T P_i$, then $\ker(Q_i) = \ker(P_i)$. In addition, for $i = 1, \dots, k$ instead of defining B_i as a concatenation of Q_i, c_i^T and c_i^T, we just need to define it as a concatenation of Q_i and c_i^T since they have the same kernel. Applying Theorem 2.4.6 to the resulting SOCP finishes the proof. □

Complexity of rank-constrained QCQP

In this section, we study the complexity of rank-constrained QCQP. First, note that Max-Cut can be written as a rank-constrained QCQP. This follows from the fact that equation (2.4.7) can be seen as a quadratic constraint

$$x^T Ax - \operatorname{tr}(AX) \leq 0,$$

that holds with equality. Thus, Max-Cut can be written as a rank-constrained QCQP where the constraint on rank is $\operatorname{rank}(x) \leq k+1$. With similar techniques as we use in Theorem 2.4.8, we obtain the following result.

Theorem 2.4.10. Let $Q_i \in \mathbb{S}^n_+$, $c_i \in \mathbb{R}^n$ for each $i = 0, 1, \ldots, k$, $d_i \in \mathbb{R}^n$ for each $i = 1, \ldots, k$, $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$ be given. Let $s \geq 1$ be a constant. Then the following rank-constrained QCQP:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; & x^T Q_0 x + c_0^T x \\ \text{subject to} \;\; & x^T Q_i x + c_i^T x + d_i \leq 0, \;\; i = 1, \ldots k; \\ & Ax = b, \\ & \operatorname{rank}(x) \leq k + s, \end{aligned}$$

is NP-hard.

Proof. Since we can write Max-Cut as a rank-constrained QCQP with the rank constraint

$\text{rank}(x) \leq k+1$, the following problem

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \;& x^T Q_0 x + c_0^T x \\ \text{subject to} \;& x^T Q_i x + c_i^T x + d_i \leq 0, \;\; i = 1, \dots k; \\ & Ax = b, \\ & \text{rank}(x) \leq k+1, \end{aligned} \tag{2.4.17}$$

is NP-hard. Now, we show how to increase the rank constraint from $k+1$ to $k+2$. For each $i = 0, \dots, k$, $Q_i = P_i^T P_i$ for some $P_i \in \mathbb{R}^{n\times n}$. Consider the following problem:

$$\begin{aligned} \underset{x\in\mathbb{R}^{n+1}}{\text{minimize}} \;& x^T \tilde{P_0}^T \tilde{P_0} x + \tilde{c_0}^T x \\ \text{subject to} \;& x^T \tilde{P_i}^T \tilde{P_i} x + \tilde{c_i}^T x + d_i \leq 0, \;\; i = 1, \dots k; \\ & \tilde{A}x = b, \\ & x_{k+1} = 1, \\ & x^T E x \leq 2, \\ & \text{rank}(x) \leq (k+1) + 2, \end{aligned} \tag{2.4.18}$$

where $\tilde{A} = [\, A \; 0 \,], \tilde{P_i} = [\, P_i \; 0 \,], \tilde{c_i} = \left[\begin{smallmatrix} c_i \\ 0 \end{smallmatrix}\right], E = e_{n+1} e_{n+1}^T$, and $e_{n+1} = (0, \dots, 0, 1)$. Note that since $x_{n+1} = 1$,

$$x^T E x = 1 < 2.$$

Thus, this quadratic constraint will contribute 2 to the rank. Thus, (2.4.18) is equivalent to (2.4.17). Note that there are $k+1$ quadratic constraints in (2.4.18). Using this argument $s-1$ times finishes the proof. □

2.5 Conclusion

In this work, we study rank-constrained and sparsity-constrained HP. For rank-constrained HP, we design algorithms for rank reduction and study the complexity of rank-constrained HP. We showed that both rank-constrained QCQP and rank-constrained SOCP are NP-hard. In addition, we show that there is a phase transition in the complexity of rank-constrained SDP with m linear constraints when the rank constraint $r(m)$ passes through $\sqrt{2m}$.

For sparsity-constrained HP, we extend results on LP sparsification to QCQP and SOCP and show that our results give tight upper bounds on minimal cardinality solutions to QCQP and SOCP.

CHAPTER 3

NUMERICAL STABILITY AND TENSOR NUCLEAR NORM

3.1 Introduction

More than fifty years ago, Volker Strassen announced an astounding result: A pair of 2×2 matrices may be multiplied with seven multiplications [111]. A consequence is that linear systems can be solved in $O(n^{\log_2 7})$ time complexity, a surprise at that time as existing works such as [68] purportedly showed that $O(n^3)$ was the lowest possible.

Strassen's algorithm is in the spirit of the well-known algorithm, often attributed to Gauss,[1] for multiplying a pair of complex numbers with three real multiplications [55],

$$(a + bi)(c + di) = (ac - bd) + i[(a + b)(c + d) - ac - bd], \tag{3.1.1}$$

but is notable in that Strassen's applies to a noncommutative product (matrix multiplication) as opposed to a commutative one (complex scalar multiplication). It led to a plethora of followed-up works and ultimately to the realization that there is a unified framework underlying the algorithms of Gauss and Strassen, namely, in evaluating a bilinear operator $\beta\colon \mathbb{U} \times \mathbb{V} \to \mathbb{W}$, viewed as a 3-tensor in $\mathbb{U}^* \otimes \mathbb{V}^* \otimes \mathbb{W}$, any decomposition

$$\beta = \varphi_1 \otimes \psi_1 \otimes w_1 + \cdots + \varphi_r \otimes \psi_r \otimes w_r \tag{3.1.2}$$

into linear functionals $\varphi_i\colon \mathbb{U} \to \mathbb{R}$, $\psi_i : \mathbb{V} \to \mathbb{R}$, and vectors $w_i \in \mathbb{W}$, $i = 1, \ldots, r$, gives us an algorithm for computing β. Furthermore, the number of terms r in such a decomposition counts precisely the number of multiplications, and thus the minimal value of r, i.e., the tensor rank of β, gives the optimal complexity for evaluating β in an appropriate sense [112] (see Section 3.2). Both Gauss's and Strassen's algorithms are the fastest possible according

1. See [88, p. 37], [103, p. 8] for example.

to this measure, that is, they attain the tensor ranks of complex multiplication (three) and 2×2 matrix product (seven) respectively [120].

Well-known to readers of this journal, speed is not all that matters in an algorithm, numerical stability is arguably more important in finite-precision computations as rounding errors may result in an unstable algorithm producing no correct digits. While the stability of algorithms for evaluating bilinear operators has been studied for specific algorithms or operators in isolation, e.g., for Gauss's algorithm in [55], Strassen's algorithm in [18], and other fast matrix multiplication algorithms in [5, 13], there has been no unfied treatment that applies to all bilinear operators β as in the case of speed. There is no analysis that quantifies stability in terms of some tensorial property of β analogous to how speed is quantified in terms of its tensor rank. The goal of the present article is to fill this gap. We will show that just as the number of terms r in the decomposition (3.1.2) controls the speed of the algorithm, the growth factor, defined as

$$\|\varphi_1\|_*\|\psi_1\|_*\|w_1\| + \cdots + \|\varphi_r\|_*\|\psi_r\|_*\|w_r\|, \tag{3.1.3}$$

controls the stability of the algorithm; and just as the tensor rank of β measures the optimal speed, the tensor nuclear norm of β, defined as

$$\|\beta\|_\nu \coloneqq \inf\Big\{\sum_{i=1}^r \|\varphi_i\|_*\|\psi_i\|_*\|w_i\| : \beta = \sum_{i=1}^r \varphi_i \otimes \psi_i \otimes w_i\Big\}, \tag{3.1.4}$$

measures the optimal stability, the precise meaning of which we will state in due course.

Although we have alluded to the relation between tensor nuclear norm and numerical stability in earlier works [80, 125, 43], we have never stated a precise relation nor carried out numerical experiments to demonstrate the relation. This article provides both. Theorem 3.3.3 gives a general relation between the growth factor of a bilinear algorithm and its forward error, from which a relation between tensor nuclear norm and forward error may be

deduced as in Corollary 3.3.4. We then perform a range of numerical experiments involving Gauss's and Strassen's algorithms to substantiate our theoretical findings:

Matrix multiplication We compare Strassen's algorithm with a well-known variant due to Winograd [58, 69]. While both attain the optimal seven multiplications, Winograd's variant is often favored because it requires only fifteen additions, compared to Strassen's eighteen. Nevertheless we will show that Strassen's algorithm has a growth factor of $12+2\sqrt{2} \approx 14.83$ whereas Winograd's variant has a growth factor of $7+4\sqrt{2}+3\sqrt{3} \approx 17.85$. For comparison, the conventional algorithm for 2×2 matrix product has eight multiplications and a growth factor of 8. Our numerical experiments confirm that in terms of accuracy Winograd's is indeed worse than Strassen's, which is in turn worse than the conventional algorithm, as Theorem 3.3.3 indicates.

Complex multiplication We compare the regular algorithm for complex multiplication, which requires four real multiplications and has a growth factor of 4; Gauss's algorithm, which requires three real multiplications but has a larger growth factor of $2(1+\sqrt{2}) \approx 4.83$; and a new algorithm:

$$\begin{aligned}(a+bi)(c+di) = \frac{1}{2}\left[\left(a+\frac{1}{\sqrt{3}}b\right)\left(c+\frac{1}{\sqrt{3}}d\right)+\left(a-\frac{1}{\sqrt{3}}b\right)\left(c-\frac{1}{\sqrt{3}}d\right)-\frac{8}{3}bd\right] \\ + \frac{i\sqrt{3}}{2}\left[\left(a+\frac{1}{\sqrt{3}}b\right)\left(c+\frac{1}{\sqrt{3}}d\right)-\left(a-\frac{1}{\sqrt{3}}b\right)\left(c-\frac{1}{\sqrt{3}}d\right)\right].\end{aligned} \tag{3.1.5}$$

This new algorithm has the best features of both the regular and Gauss's algorithms, requiring three real multiplications and yet has the smaller (in fact, smallest, as we will see) growth factor of 4. Again the results are consistent with the prediction of Theorem 3.3.3.

For the uninitiated, we would like to stress that the aforementioned algorithms only begin to make a difference when they are applied recursively, or applied to matrices, or both. For instance, Gauss's algorithm (3.1.1) is really quite useless for multiplying a pair

of complex numbers, whether 'by hand' or on a computer. It only becomes useful when applied recursively in the form of Karatsuba's algorithm [63] for integer multiplication, with i replaced by the number base; or when applied to complex matrices [38]:

$$(A + iB)(C + iD) = (AC - BD) + i[(A + B)(C + D) - AC - BD], \tag{3.1.6}$$

with $A+iB, C+iD \in \mathbb{C}^{n\times n}$, $A, B, C, D \in \mathbb{R}^{n\times n}$. As multiplication of matrices is much more expensive than addition of matrices, so (3.1.6) really does represent an enormous savings in speed over the regular algorithm:

$$(A + iB)(C + iD) = (AC - BD) + i(BC + AD). \tag{3.1.7}$$

Likewise, our new algorithm (3.1.5) only begins to make a difference when the quantities involved are matrices. For the same reasons, the algorithms of Strassen and Winograd are only worth the trouble when applied recursively to a product of $n \times n$ matrices partitioned recursively into 2×2 blocks.

To address another related point early on, a surprisingly common complaint among early feedbacks is that there are a lot of $\sqrt{3}$'s in our algorithm (3.1.5). Certainly, if one computes these products 'by hand,' it would be easier to use the regular or Gauss's algorithm since they do not involve irrational coefficients. But when performed by a computer this is completely immaterial. In case it is not clear, it does not matter whether we multiply by 3 or by $\sqrt{3}$; to a computer (or any IEEE 754-compliant equipment) both are binary strings of 0's and 1's and arithmetic takes one flop regardless. Maybe there would be some minor savings when a constant happens to be a power of 2, because of binary arithmetic; but aside from that, it makes no difference what constants we have in our algorithm.

For the matrix multiplication experiments, our goal is simply to illustrate Theorem 3.3.3 by comparing the known algorithms of Strassen and Winograd. But for the complex multi-

plication experiments, we also have the additional goal of testing, for the first time, the new algorithm (3.1.5) applied to multiply complex matrices, which we will see is

- nearly as fast as Gauss's algorithm (3.1.6), and
- nearly as accurate as the regular algorithm (3.1.7).

To substantiate these claims, we perform more extensive experiments to compare (3.1.5), (3.1.6), and (3.1.7), including three practical applications: evaluation of matrix polynomials via Horner's method [59], unitary transform, and complex-valued neural networks [1, 8, 26, 105, 117, 126]. All our codes are available from `https://github.com/zhen06/Complex-Matrix-Multiplication`.

Notations

To reduce notational clutter, we denote norms on different vector spaces $\mathbb{U}, \mathbb{V}, \mathbb{W}$ by the same $\|\cdot\|$. There is no cause for confusion since we always use it in a form like $\|v\|$ for some $v \in \mathbb{V}$, where it is clear from context that $\|\cdot\|$ refers to a norm on $\mathbb{V}$. Likewise the corresponding dual norms on $\mathbb{U}^*, \mathbb{V}^*, \mathbb{W}^*$ will be denoted by the same $\|\cdot\|_*$. Recall that for $\varphi \in \mathbb{V}^*$, i.e., $\varphi : \mathbb{V} \to \mathbb{R}$ is a linear functional, this is defined by

$$\|\varphi\|_* \coloneqq \sup\{|\varphi(v)| : \|v\| \leq 1\}.$$

3.2 Bilinear Complexity

We provide a brief review of bilinear complexity, usually studied in Algebraic Computational Complexity [16, 21, 75, 114], for numerical analysts. Our goals here are to (i) highlight certain departures from typical practice in numerical linear algebra; and (ii) show a parallel with our notion of bilinear stability in the next section.

Let $\mathbb{U}$, $\mathbb{V}$, $\mathbb{W}$ be finite-dimensional vector spaces, assume to be over $\mathbb{R}$ for simplicity. Let $\beta : \mathbb{U} \times \mathbb{V} \to \mathbb{W}$ be a bilinear operator. Depending on one's definition of a tensor, we have $\beta \in \mathbb{U}^* \otimes \mathbb{V}^* \otimes \mathbb{W}$ either through definition [80, Definition 3.3] or by the universal mapping property [80, Equation 4.88]. A bilinear algorithm for evaluating β is a decomposition of the form (3.1.2). In other words, for any $u \in \mathbb{U}$ and $v \in \mathbb{V}$, we evaluate $\beta(u, v)$ by performing the algorithm given by the decomposition on the right:

$$\beta(u, v) = \sum_{i=1}^{r} \varphi_i(u)\psi_i(v)w_i. \tag{3.2.1}$$

In practice, the vector spaces involved are usually Euclidean spaces of vectors $\mathbb{R}^n$ or matrices $\mathbb{R}^{m\times n}$. Riesz representation theorem guarantees that any linear functional $\varphi : \mathbb{R}^n \to \mathbb{R}$ must take the form $\varphi(x) = a^{\mathsf{T}}x$ for some $a \in \mathbb{R}^n$ and likewise any functional $\varphi : \mathbb{R}^{m\times n} \to \mathbb{R}$ must take the form $\varphi(X) = \operatorname{tr}\big(A^{\mathsf{T}}X\big)$ for some $A \in \mathbb{R}^{m\times n}$.

Each rank-one term $\varphi_i(u)\psi_i(v)w_i$ in (3.2.1) accounts for one multiplication but herein lies a pitfall — the 'multiplication' refers to the product of $\varphi_i(u)$ and $\psi_i(v)$; note that this a variable product, i.e., the value depends on variables u and v, as opposed to a scalar product. Take a randomly made-up example[2] with $\mathbb{U} = \mathbb{R}^{2\times 2}$, $\mathbb{V} = \mathbb{R}^2$, $\mathbb{W} = \mathbb{R}^3$, and

$$\begin{aligned}
&\varphi_i\big(\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]\big) = \operatorname{tr}\big(\left[\begin{smallmatrix} -1 & 0 \\ 1 & 2 \end{smallmatrix}\right]^{\mathsf{T}}\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]\big) = -a + c + 2d,\\
&\psi_i\big(\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]\big) = \left[\begin{smallmatrix} 3 \\ -1/2 \end{smallmatrix}\right]^{\mathsf{T}}\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right] = 3x - y/2,\\
&w_i = \begin{bmatrix} -3 \\ 4 \\ \sqrt{5} \end{bmatrix},
\end{aligned}$$

then there is exactly one multiplication in

$$\varphi_i(u)\psi_i(v)w_i = \begin{bmatrix} -3(-a+c+2d)(3x-y/2) \\ 4(-a+c+2d)(3x-y/2) \\ \sqrt{5}(-a+c+2d)(3x-y/2) \end{bmatrix}.$$

2. Genuine examples to follow in Sections 3.4 and 3.5.

The scalar products like $2d$ or $-y/2$ or $\sqrt{5}t$ are discounted in Strassen's model of bilinear complexity [112, 113] and for good reasons — these constants coefficients are fixed in the algorithm and can be hardcoded or hardwired, unlike the product between $-a+c+2d$ and $3x-y/2$, which depends on the variable inputs $u = \left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$ and $v = \left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]$. In particular, Strassen's measure of speed, called bilinear complexity, is independent of the values of these constant coefficients, but we will show in the next section that these will affect numerical stability of the algorithm.

To emphasize its distinction from scalar products, Strassen calls a variable product in the above sense a nonscalar product [113]. In other words, bilinear complexity measures speed purely in terms of the number of nonscalar products. The bilinear complexity of the algorithm in (3.2.1) is given by the number terms in the decomposition r and the optimal speed of evaluating β is therefore given by the tensor rank [112]

$$\operatorname{rank}(\beta) \coloneqq \min\left\{ r : \beta = \sum_{i=1}^{r} \varphi_i \otimes \psi_i \otimes w_i \right\}. \tag{3.2.2}$$

A tensor rank decomposition of β, i.e., one that attains its tensor rank, is then a fastest algorithm in the context of bilinear complexity.

In realistic scenarios, storage and computations both have finite-precision. Given u and v, we do not need to know $\beta(u, v)$ exactly; in fact computing anything beyond 16 decimal digits of accuracy is wasted effort since we do not store more than 16 digits in IEEE double precision. So the tensor rank of β is less relevant than the border rank [14] of β, which is the smallest r so that

$$\|\beta - \varphi_1^\varepsilon \otimes \psi_1^\varepsilon \otimes w_1^\varepsilon - \varphi_2^\varepsilon \otimes \psi_2^\varepsilon \otimes w_2^\varepsilon - \cdots - \varphi_r^\varepsilon \otimes \psi_r^\varepsilon \otimes w_r^\varepsilon\| < \varepsilon$$

for all $\varepsilon > 0$, or, formally,

$$\overline{\operatorname{rank}}(\beta) \coloneqq \min\left\{ r : \beta = \lim_{\varepsilon \to 0^+} \sum_{i=1}^{r} \varphi_i^{\varepsilon} \otimes \psi_i^{\varepsilon} \otimes w_i^{\varepsilon} \right\}. \tag{3.2.3}$$

For the two problems studied in our article, namely, matrix multiplication,

$$\beta_{m,n,p} : \mathbb{R}^{m \times n} \times \mathbb{R}^{n \times p} \to \mathbb{R}^{m \times p}, \quad (A, B) \mapsto AB,$$

and complex multiplication,

$$\beta_{\mathbb{C}} : \mathbb{C} \times \mathbb{C} \to \mathbb{C}, \quad (w, z) \mapsto wz,$$

(noting that $\mathbb{C}$ is a two-dimensional real vector space), we have [73, 120]

$$\operatorname{rank}(\beta_{2,2,2}) = \overline{\operatorname{rank}}(\beta_{2,2,2}) = 7, \qquad \operatorname{rank}(\beta_{\mathbb{C}}) = \overline{\operatorname{rank}}(\beta_{\mathbb{C}}) = 3.$$

It is in general difficult to find such exact values. For instance, the values of $\operatorname{rank}(\beta_{3,3,3})$ and $\overline{\operatorname{rank}}(\beta_{3,3,3})$ are still unknown. Most of the efforts in studying matrix multiplication go towards determining the asymptotic value $\omega \coloneqq \inf\{p \in \mathbb{R} \colon \operatorname{rank}(\beta_{n,n,n}) = O(n^p)\}$, called the exponent of matrix multiplication. An advantage is that asymptotically, the full arithmetic complexity, i.e., counting all operations and not just nonscalar multiplications, is also $O(n^{\omega})$. More importantly, the role of ω stretches far beyond matrix multiplication, governing the full arithmetic complexity of computing inverse, determinant, null basis, linear systems, LU/QR/eigenvalue/Hessenberg decompositions, characteristic polynomials, sparsification, and even linear programming — note in particular that none of these are bilinear operations [113] (see also [21, Chapter 16] and [80, Examples 3.10 and 4.40]).

3.3 Bilinear Stability

We would like to state at the outset that numerical stability is a moderately complicated issue that depends on many factors and cannot be completely represented by any single number. Designing numerically stable algorithms is as much an art as it is a science. However the six Higham guidelines for numerical stability [58, Section 1.18] capture the most salient aspects. Among them, the second guideline to "minimize the size of intermediate quantities relative to the final solution" is one of the most unequivocal, lends itself to precise quantification, and is what we will focus on in this section. Consideration of Higham's second guideline for bilinear algorithms leads us naturally to the notion of bilinear stability, which relates to accuracy the way bilinear complexity relates to speed. More precisely, the growth factor (3.1.3) and tensor nuclear norm (3.1.4) are to accuracy in bilinear stability what the number of rank-one terms in (3.2.1) and the tensor rank (3.2.2) are to speed in bilinear complexity. Here accuracy refers to the size of relative forward error.

Bilinear stability differs from existing studies of numerical stability of bilinear algorithms such as those in [5, 13, 18, 55] in three ways: (i) it is more general, applying to any bilinear operators as opposed to specific ones like matrix multiplication; (ii) it is more simplistic, relating forward error to just growth factor as opposed to two or three different factors in the approaches of [5, 13]; (iii) it is truly tensorial, as growth factor and tensor nuclear norm are invariant under any orthogonal change-of-coordinates, just as tensor rank is invariant under any invertible change-of-coordinates. The factors (i) and (ii), i.e., generality and simplicity, may often be sacrificed for better bounds: Given any specific bilinear operator, we may often obtain smaller forward error bounds by performing a more precise analysis tailored to that given operator. We will do see this in Section 3.5.2.

One difference between bilinear complexity and bilinear stability is that the latter requires a norm. While there are many excellent treatises on tensor norms [27, 29, 104], they are excessive for our purpose. All the reader needs to know is that for a vector space $\mathbb{V}_i$ with

norm $\|\cdot\|_i$, $i = 1, \dots, d$, a tensor norm $\|\cdot\|$ on $\mathbb{V}_1 \otimes \mathbb{V}_2 \otimes \cdots \otimes \mathbb{V}_d$ satisfies the multiplicativity property for rank-one tensors:

$$\|v_1 \otimes v_2 \otimes \cdots \otimes v_d\| = \|v_1\|_1 \|v_2\|_2 \cdots \|v_d\|_d,$$

where $v_i \in \mathbb{V}_i$. In particular, the spectral, Frobenius (also called Hilbert–Schmidt), nuclear norms [80, p. 561 and Example 4.17] are all equal on rank-one tensors in $\mathbb{U}^* \otimes \mathbb{V}^* \otimes \mathbb{W}$, i.e.,

$$\|\varphi \otimes \psi \otimes w\|_\sigma = \|\varphi \otimes \psi \otimes w\|_{\mathsf{F}} = \|\varphi \otimes \psi \otimes w\|_\nu = \|\varphi\|_* \|\psi\|_* \|w\|$$

for all $\varphi \in \mathbb{U}^*$, $\psi_i \in \mathbb{V}^*$, $w \in \mathbb{W}$. Consequently, when we speak of the norm of a rank-one tensor $\varphi \otimes \psi \otimes w$, it does not matter which of these three norms we choose, and we will simply write

$$\|\varphi \otimes \psi \otimes w\| \coloneqq \|\varphi\|_* \|\psi\|_* \|w\|.$$

We first present a straightforward heurstic that motivates our definition of the growth factor, deferring the more formal forward error analysis to Theorem 3.3.3. If we apply the rank-one bilinear operator $\varphi_i \otimes \psi_i \otimes w_i$ to u and v,

$$\begin{aligned} \|(\varphi_i \otimes \psi_i \otimes w_i)(u, v)\| &= \|\varphi_i(u)\psi_i(v)w_i\| \\ &= |\varphi_i(u)||\psi_i(v)|\|w_i\| \\ &\leq \|\varphi_i\|_* \|u\| \|\psi_i\|_* \|v\| \|w_i\| \\ &= \|\varphi_i \otimes \psi_i \otimes w_i\| \|u\| \|v\|. \end{aligned}$$

So $\varphi_i \otimes \psi_i \otimes w_i$ magnifies the errors in u and v by an amount bounded by its tensor norm $\|\varphi_i \otimes \psi_i \otimes w_i\|$. Therefore, in a bilinear algorithm given by the right side of (3.2.1) for

evaluating β, triangle inequality gives

$$\|\beta(u,v)\| = \left\|\sum_{i=1}^{r}(\varphi_i \otimes \psi_i \otimes w_i)(u,v)\right\| \leq \left[\sum_{i=1}^{r}\|\varphi_i \otimes \psi_i \otimes w_i\|\right]\|u\|\|v\|.$$

The ith step of the algorithm magnifies the error in the inputs (u, v) by an amount bounded by $\|\varphi_i \otimes \psi_i \otimes w_i\|$ and over the course of r steps in the algorithm, the accumulated error is bounded by a factor of

$$\sum_{i=1}^{r}\|\varphi_i \otimes \psi_i \otimes w_i\| = \sum_{i=1}^{r}\|\varphi_i\|_*\|\psi_i\|_*\|w_i\|, \tag{3.3.1}$$

which we will define as the growth factor of the algorithm or decomposition (3.2.1). Its minimum value over all possible bilinear algorithms for evaluating β or, equivalently, over all decomposition of β as a 3-tensor is therefore given by the nuclear norm (3.1.4). This idea was first floated in [125, Section 3.2]. Note that the growth factor depends on the algorithm/decomposition for β but the nulcear norm depends only on β.

We now state a formal definition to make precise the terms used in the preceding discussions.

Definition 3.3.1. Let $\mathbb{U}, \mathbb{V}, \mathbb{W}$ be three finite-dimensional real vector spaces. A decomposition of a bilinear operator $\beta : \mathbb{U} \times \mathbb{V} \to \mathbb{W}$ is a sequence $\mathcal{D} = (\varphi_i, \psi_i, w_i)_{i=1}^{r}$ with

$$\beta = \sum_{i=1}^{r} \varphi_i \otimes \psi_i \otimes w_i, \tag{3.3.2}$$

where $\varphi_i : \mathbb{U} \to \mathbb{R}$ and $\psi_i : \mathbb{V} \to \mathbb{R}$ are linear functionals and $w_i \in \mathbb{W}$, $i = 1, \ldots, r$. An algorithm $\widehat{\beta}_{\mathcal{D}}$ given by the decomposition $\mathcal{D}$ takes $(u, v) \in \mathbb{U} \times \mathbb{V}$ as inputs and computes the output $\beta(u, v)$ in three steps:

(i) computes $\varphi_i(u)$ and $\psi_i(v)$, $i = 1, \ldots, r$;

(ii) computes $\varphi_i(u)\psi_i(v)w_i$, $i = 1, \dots, r$;

(iii) computes $\sum_{i=1}^r \varphi_i(u)\psi_i(v)w_i$.

The growth factor of the algorithm $\widehat{\beta}_{\mathcal{D}}$ is defined as

$$\gamma(\widehat{\beta}_{\mathcal{D}}) \coloneqq \sum_{i=1}^r \|\varphi_i \otimes \psi_i \otimes w_i\| = \sum_{i=1}^r \|\varphi_i\|_* \|\psi_i\|_* \|w_i\|.$$

As noted in Section 3.2, only the variable multiplication in step (ii) counts in bilinear complexity; the other two steps comprising scalar multiplications and additions are discounted. In bilinear stability all three steps contribute to the growth factor.

Proposition 3.3.2. The minimal growth factor is given by nuclear norm of the β, i.e.,

$$\min_{\mathcal{D}} \gamma(\widehat{\beta}_{\mathcal{D}}) = \|\beta\|_\nu,$$

with $\mathcal{D}$ running over all decomposition. Furthermore, there is always an algorithm that attains the minimal growth factor.

The above equality is just stating (3.1.4) in terms of the growth factor. That there is always an algorithm attaining the minimal growth factor, justifying our writing min instead of inf, follows from the existence of a nuclear decomposition [43, Proposition 3.1], i.e., a decomposition that attains the nuclear norm. Just as a rank decomposition of β represents a fastest algorithm in bilinear complexity, a nuclear decomposition of β represents a stablest algorithm in bilinear stability.

We next establish a rigorous relationship between growth factor and numerical stability by proving a forward error bound in terms of the growth factor of a bilinear algorithm. We assume a system of floating point arithmetic obeying the standard model as in [58]: For $x, y \in \mathbb{R}$

$$\mathsf{fl}(x \,\mathsf{op}\, y) = (x \,\mathsf{op}\, y)(1 + \delta), \qquad |\delta| \le \mathsf{u}, \quad \mathsf{op} = +, -, *, / \tag{3.3.3}$$

with u the unit roundoff. We assume that $\mathbb{U}, \mathbb{V}, \mathbb{W}$ are vector spaces of dimensions m, n, p and that appropriate computational bases have been chosen on them so that we may identify $\mathbb{U} \cong \mathbb{R}^m$, $\mathbb{V} \cong \mathbb{R}^n$, $\mathbb{W} \cong \mathbb{R}^p$. The computational bases do not need to be the standard bases and may instead be Fourier, Krylov, Haar, wavelet bases, etc. This is another reason why we cast our discussions in terms of abstract vector spaces and do not choose bases until absolutely necessary. However, once a choice of bases has been made, the result below depends only on the dimensions of $\mathbb{U}, \mathbb{V}, \mathbb{W}$; if say, $\mathbb{U} = \mathbb{R}^{m\times n}$, then only the fact that it has dimension mn matters, i.e., $\mathbb{U} \cong \mathbb{R}^{mn}$.

Theorem 3.3.3 (Growth factor and forward error)**.** Let $\beta : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^p$ be a bilinear operator, $\mathcal{D} = (\varphi_i, \psi_i, w_i)_{i=1}^r$ a decomposition, and $\widehat{\beta}_{\mathcal{D}}$ the corresponding algorithm. If $\widehat{\beta}_{\mathcal{D}}(u, v)$ is the output of $\widehat{\beta}_{\mathcal{D}}$ with $u \in \mathbb{R}^m$ and $v \in \mathbb{R}^n$ as inputs, then

$$\|\beta(u, v) - \widehat{\beta}_{\mathcal{D}}(u, v)\|_\infty \leq (m + n + r + 1)\gamma(\widehat{\beta}_{\mathcal{D}})\|u\|\|v\|\mathsf{u} + O(\mathsf{u}^2).$$

Proof. We first show that the result reduces to the case $p = 1$. It suffices to show that

$$|\beta(u, v)_k - \widehat{\beta}_{\mathcal{D}}(u, v)_k| \leq (m + n + r + 1)\gamma(\widehat{\beta}_{\mathcal{D}})\|u\|\|v\|\mathsf{u} + O(\mathsf{u}^2) \tag{3.3.4}$$

for all $k = 1, \dots, p$, where the subscript k refers to the kth coordinate of a vector in $\mathbb{R}^p$. Since

$$\gamma(\widehat{\beta}_{\mathcal{D}}) = \sum_{i=1}^r \|\varphi_i\|_*\|\psi_i\|_*\|w_i\| \geq \sum_{i=1}^r \|\varphi_i\|_*\|\psi_i\|_*|w_{ik}|,$$

with w_{ik} the kth coordinate of $w_i \in \mathbb{R}^p$, to show (3.3.4), it suffices to show

$$|\beta(u, v)_k - \widehat{\beta}_{\mathcal{D}}(u, v)_k| \leq (m + n + r + 1)\left[\sum_{i=1}^r \|\varphi_i\|_*\|\psi_i\|_*|w_{ik}|\right]\|u\|\|v\|\mathsf{u} + O(\mathsf{u}^2),$$

which is equivalent to the case $p = 1$. In the following, we will assume that $p = 1$.

Since φ_i and ψ_i are linear functionals on $\mathbb{R}^m$ and $\mathbb{R}^n$, there exist $u_i \in \mathbb{R}^m$ and $v_i \in \mathbb{R}^n$

such that

$$\varphi_i(u) = u_i^{\mathsf{T}} u \quad \text{and} \quad \psi_i(v) = v_i^{\mathsf{T}} v,$$

for all $u \in \mathbb{R}^m$ and $v \in \mathbb{R}^n$. By [58, equation 3.7],

$$|x^{\mathsf{T}} y - \mathsf{fl}(x^{\mathsf{T}} y)| \leq n|x|^{\mathsf{T}}|y|\mathsf{u} + O(\mathsf{u}^2),$$

for any $x, y \in \mathbb{R}^n$ where $|\cdot|$ and $\leq$ apply coordinatewise. So for each $i = 1, \ldots, r$,

$$\begin{aligned} |\varphi_i(u) - \mathsf{fl}(\varphi_i(u))| = |u_i^{\mathsf{T}} u - \mathsf{fl}(u_i^{\mathsf{T}} u)| &\leq m|u_i|^{\mathsf{T}}|u|\mathsf{u} + O(\mathsf{u}^2) \\ &\leq m\|u_i\|\|u\|\mathsf{u} + O(\mathsf{u}^2) = m\|\varphi_i\|_*\|u\|\mathsf{u} + O(\mathsf{u}^2). \end{aligned} \tag{3.3.5}$$

Likewise, for each $i = 1, \ldots, r$,

$$|\psi_i(v) - \mathsf{fl}(\psi_i(v))| \leq n\|\psi_i\|_*\|v\|\mathsf{u} + O(\mathsf{u}^2). \tag{3.3.6}$$

Let $\Delta_{1,i} = \mathsf{fl}(\varphi_i(u)) - \varphi_i(u)$ and $\Delta_{2,i} = \mathsf{fl}(\psi_i(v)) - \psi_i(v)$. By (3.3.5) and (3.3.6),

$$|\Delta_{1,i}| \leq m\|\varphi_i\|_*\|u\|\mathsf{u} + O(\mathsf{u}^2), \quad |\Delta_{2,i}| \leq n\|\psi_i\|_*\|v\|\mathsf{u} + O(\mathsf{u}^2). \tag{3.3.7}$$

Let $c_i = \varphi_i(u)\psi_i(v)$ and $\widehat{c}_i$ be its computed value. By (3.3.7), there exists δ_i with $|\delta_i| \leq \mathsf{u}$ such that

$$\begin{aligned} \widehat{c}_i &= (\varphi_i(u) + \Delta_{1,i})(\psi_i(v) + \Delta_{2,i})(1 + \delta_i) \\ &= \varphi_i(u)\psi_i(v) + \Delta_{1,i}\psi_i(v) + \varphi_i(u)\Delta_{2,i} + \delta_i\varphi_i(u)\psi_i(v) + O(\mathsf{u}^2). \end{aligned} \tag{3.3.8}$$

By (3.3.7) and (3.3.8),

$$\begin{aligned} |c_i - \widehat{c}_i| &\le m\|\varphi_i\|_*\|u\||\psi_i(v)|\mathsf{u} + |\varphi_i(u)|n\|\psi_i\|_*\|v\|\mathsf{u} + |\varphi_i(u)\psi_i(v)|\mathsf{u} + O(\mathsf{u}^2) \\ &\le (m+n+1)\|\varphi_i\|_*\|\psi_i\|_*\|u\|\|v\|\mathsf{u} + O(\mathsf{u}^2). \end{aligned} \tag{3.3.9}$$

Let $\Delta_i = \widehat{c}_i - c_i$. By (3.3.9),

$$|\Delta_i| \le (m+n+1)\|\varphi_i\|_*\|\psi_i\|_*\|u\|\|v\|\mathsf{u} + O(\mathsf{u}^2). \tag{3.3.10}$$

Let $d_i = c_i w_i$ and $\widehat{d}_i$ be the computed value of d_i. By (3.3.10), there exists δ_i' with $|\delta_i'| \le \mathsf{u}$ such that

$$\widehat{d}_i = (c_i + \Delta_i)w_i(1+\delta_i') = c_i w_i + \Delta_i w_i + \delta_i' c_i w_i + O(\mathsf{u}^2). \tag{3.3.11}$$

Let $\Delta_i' = \widehat{d}_i - d_i$. By (3.3.10) and (3.3.11),

$$\begin{aligned} |\Delta_i'| &\le (m+n+1)\|\varphi_i\|_*\|\psi_i\|_*\|u\|\|v\|\|w_i\|\mathsf{u} + |\varphi_i(u)\psi_i(v)|\|w_i\|\mathsf{u} + O(\mathsf{u}^2) \\ &\le (m+n+2)\|\varphi_i\|_*\|\psi_i\|_*\|w_i\|\|u\|\|v\|\mathsf{u} + O(\mathsf{u}^2). \end{aligned} \tag{3.3.12}$$

Finally, let $a = \sum_{i=1}^r \varphi_i(u)\psi_i(v)w_i$ and $\widehat{a}$ be the computed value of a. By (3.3.12), there exists δ with $|\delta| \le \mathsf{u}$ such that

$$\widehat{a} = \widehat{d}_1(1+\delta)^{r-1} + \widehat{d}_2(1+\delta)^{r-1} + \widehat{d}_3(1+\delta)^{r-2} + \cdots + \widehat{d}_r(1+\delta),$$

where we compute the sum $\widehat{d}_1 + \widehat{d}_2 + \ldots \widehat{d}_r$ from left to right. Hence we obtain

$$\begin{aligned} |a - \widehat{a}| &\le (m+n+2)\|u\|\|v\| \sum_{i=1}^r \|\varphi_i\|_*\|\psi_i\|_*\|w_i\|\mathsf{u} + (r-1)\left\| \sum_{i=1}^r c_i w_i \right\|\mathsf{u} + O(\mathsf{u}^2) \\ &\le (m+n+r+1)\|u\|\|v\| \sum_{i=1}^r \|\varphi_i\|_*\|\psi_i\|_*\|w_i\|\mathsf{u} + O(\mathsf{u}^2) \\ &= (m+n+r+1)\gamma(\widehat{\beta}_{\mathcal{D}})\|u\|\|v\|\mathsf{u} + O(\mathsf{u}^2). \end{aligned}$$

□

Theorem 3.3.3 essentially says that that algorithms with small growth factors have small forward errors. Combined with Proposition 3.3.2, we see that the optimally stable algorithm in this context is the one corresponding to a nuclear decomposition of β.

Corollary 3.3.4 (Tensor nuclear norm and forward error)**.** Let $\beta : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^p$ be a bilinear operator, $\mathcal{D} = (\varphi_i, \psi_i, w_i)_{i=1}^r$ a nuclear decomposition, and $\widehat{\beta}_{\mathcal{D}}$ the corresponding algorithm. Then

$$\|\beta(u, v) - \widehat{\beta}_{\mathcal{D}}(u, v)\|_\infty \leq (m + n + r + 1)\|\beta\|_\nu \|u\| \|v\| \mathsf{u} + O(\mathsf{u}^2).$$

In principle, there is no reason to expect there to be an algorithm that is both fastest in the sense of Section 3.2 and stablest in the sense of this section, i.e., having a decomposition that attains both tensor rank and nuclear norm. In Section 3.5, we will see that such an algorithm exists for complex multiplication and we will study its properties when applied to complex matrix multiplication.

3.4 Fast Matrix Multiplications

As an illustration of bilinear stability in the last section, we will calculate the growth factors of Strassen's algorithm [111] and Winograd's variant [58, 69] for fast matrix multiplication and compare their stability empirically. We will see that the growth factor of Strassen's algorithm is smaller than that of Winograd's variant, and, consistent with the prediction of Theorem 3.3.3, numerical experiments indeed show that the former gives more accurate results.

3.4.1 Bilinear stability of Strassen multiplication

Given two block matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

Strassen's algorithm [111] first computes

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), & M_5 &= (A_{11} + A_{12})B_{22}, \\ M_2 &= (A_{21} + A_{22})B_{11}, & M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\ M_3 &= A_{11}(B_{12} - B_{22}), & M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}), \\ M_4 &= A_{22}(B_{21} - B_{11}), \end{aligned}$$

and then computes the product via

$$AB = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}.$$

Note that this may be applied recursively. Let $\widehat{\beta}_{\mathsf{S}} : \mathbb{R}^{2\times 2} \times \mathbb{R}^{2\times 2} \to \mathbb{R}^{2\times 2}$ denote the Strassen's algorithm for 2×2 matrices. It is routine to check that for $A, B \in \mathbb{R}^{2\times 2}$,

$$\widehat{\beta}_{\mathsf{S}}(A, B) = \sum_{i=1}^{7} \varphi_i(A)\psi_i(B)W_i,$$

where $\varphi_i(A) = \operatorname{tr}\big(U_i^{\mathsf{T}} A\big)$ and $\psi_i(B) = \operatorname{tr}\big(V_i^{\mathsf{T}} B\big)$ with

$$
\begin{aligned}
U_1 &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & V_1 &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & W_1 &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \\
U_2 &= \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, & V_2 &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & W_2 &= \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix}; \\
U_3 &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & V_3 &= \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}, & W_3 &= \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}; \\
U_4 &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, & V_4 &= \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}, & W_4 &= \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}; \\
U_5 &= \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, & V_5 &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, & W_5 &= \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}; \\
U_6 &= \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}, & V_6 &= \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, & W_6 &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}; \\
U_7 &= \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}, & V_7 &= \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, & W_7 &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.
\end{aligned}
$$

For simplicity we will use the Frobenius norm on $\mathbb{R}^{2\times 2}$ since it is self dual. The growth factor of Strassen's algorithm is then given by

$$\gamma(\widehat{\beta}_{\mathsf{S}}) = \sum_{i=1}^{7} \|\varphi_i\|_* \|\psi_i\|_* \|W_i\| = \sum_{i=1}^{7} \|U_i\|_{\mathsf{F}} \|V_i\|_{\mathsf{F}} \|W_i\|_{\mathsf{F}} = 12 + 2\sqrt{2} \approx 14.83. \tag{3.4.1}$$

3.4.2 Bilinear stability of Winograd multiplication

Winograd's algorithm [58, 69] computes a different set of intermediate quantities

$$
\begin{aligned}
M'_1 &= (A_{21} + A_{22} - A_{11})(B_{11} + B_{22} - B_{12}), & M'_5 &= (A_{21} + A_{22})(B_{12} - B_{11}),\\
M'_2 &= A_{11}B_{11}, & M'_6 &= (A_{11} + A_{12} - A_{21} - A_{22})B_{22},\\
M'_3 &= A_{12}B_{21}, & M'_7 &= A_{22}(B_{11} + B_{22} - B_{12} - B_{21}),\\
M'_4 &= (A_{11} - A_{21})(B_{22} - B_{12}),
\end{aligned}
$$

and then compute the product via

$$
AB = \begin{bmatrix} M'_2 + M'_3 & M'_1 + M'_2 + M'_5 + M'_6 \\ M'_1 + M'_2 + M'_4 - M'_7 & M'_1 + M'_2 + M'_4 + M'_5 \end{bmatrix}.
$$

Again this can be applied recursively. Let $\widehat{\beta}_{\mathsf{W}} : \mathbb{R}^{2\times 2} \times \mathbb{R}^{2\times 2} \to \mathbb{R}^{2\times 2}$ denote the Winograd's algorithm for 2×2 matrices. It is again routine to check that for $A, B \in \mathbb{R}^{2\times 2}$,

$$
\widehat{\beta}_{\mathsf{W}}(A, B) = \sum_{i=1}^{7} \varphi'_i(A)\psi'_i(B)W'_i,
$$

where $\varphi_i'(A) = \operatorname{tr}\big(U_i'^{\mathsf{T}} A\big)$ and $\psi_i'(B) = \operatorname{tr}\big(V_i'^{\mathsf{T}} B\big)$ with

$$
\begin{aligned}
U_1' &= \begin{bmatrix} -1 & 0 \\ 1 & 1 \end{bmatrix}, & V_1' &= \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, & W_1' &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}; \\
U_2' &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & V_2' &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & W_2' &= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}; \\
U_3' &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, & V_3' &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, & W_3' &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}; \\
U_4' &= \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix}, & V_4' &= \begin{bmatrix} 0 & -1 \\ 0 & 1 \end{bmatrix}, & W_4' &= \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}; \\
U_5' &= \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, & V_5' &= \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}, & W_5' &= \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}; \\
U_6' &= \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}, & V_6' &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, & W_6' &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}; \\
U_7' &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, & V_7' &= \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, & W_7' &= \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix}.
\end{aligned}
$$

With respect to the Frobenius norm, the growth factor of Winograd's algorithm is

$$\gamma(\widehat{\beta}_{\mathsf{W}}) = \sum_{i=1}^{7} \|\varphi_i'\|_* \|\psi_i'\|_* \|W_i'\|_{\mathsf{F}} = \sum_{i=1}^{7} \|U_i'\|_{\mathsf{F}} \|V_i'\|_{\mathsf{F}} \|W_i'\|_{\mathsf{F}} = 7 + 4\sqrt{2} + 3\sqrt{3} \approx 17.85. \quad (3.4.2)$$

3.4.3 Bilinear stability of conventional matrix multiplication

For completeness we state the growth factor of the conventional algorithm for matrix multiplication $\widehat{\beta}_{\mathsf{C}} : \mathbb{R}^{2\times 2} \times \mathbb{R}^{2\times 2} \to \mathbb{R}^{2\times 2}$,

$$\widehat{\beta}_{\mathsf{C}}(A, B) = \sum_{i,j,k=1}^{2} \operatorname{tr}\Big(E_{ij}^{\mathsf{T}} A\Big) \operatorname{tr}\Big(E_{jk}^{\mathsf{T}} B\Big) E_{ik},$$

where $E_{ij} \in \mathbb{R}^{2\times 2}$ denotes the standard basis matrix. Its growth factor is easily seen to be

$$\gamma(\widehat{\beta}_{\mathsf{C}}) = \sum_{i,j,k=1}^{2} \|E_{ij}\|_{\mathsf{F}} \|E_{jk}\|_{\mathsf{F}} \|E_{ik}\|_{\mathsf{F}} = 8.$$

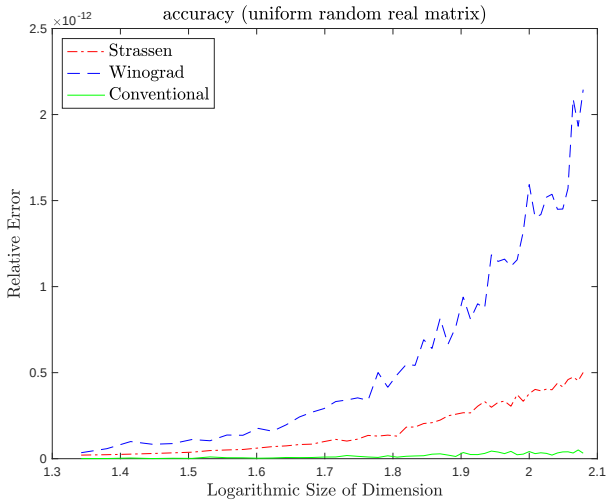
From (3.4.1) and (3.4.2), we see that

$$\gamma(\widehat{\beta}_{\mathsf{W}}) > \gamma(\widehat{\beta}_{\mathsf{S}}) > \gamma(\widehat{\beta}_{\mathsf{C}}). \tag{3.4.3}$$

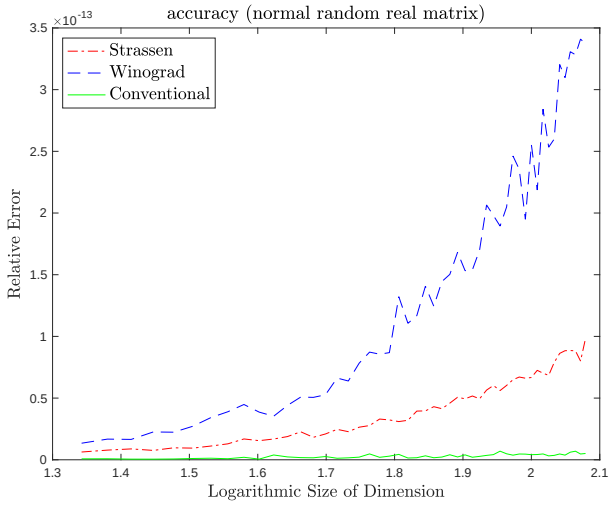
The first inequality will be verified in the numerical experiments below; the second is consistent with the well-known fact [58] that Strassen's algorithm is less accurate than conventional multiplication.

3.4.4 Numerical experiments for fast matrix multiplications

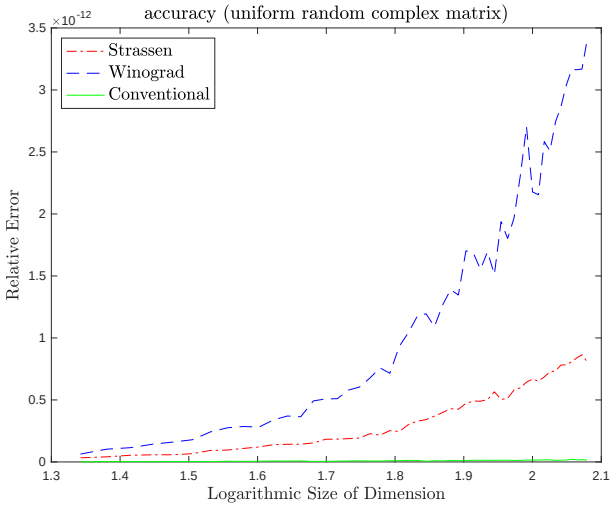
By Theorem 3.3.3 and the sizes of the growth factors in (3.4.3), we expect Strassen's algorithm to give more accurate results than Winograd's variant since it has a smaller growth factor. We test this statement with random matrices generated in three different ways: with (a) real entries drawn from the uniform distribution on $[-1, 1]$, (b) real entries drawn from the standard normal distribution, (c) complex entries whose real and imaginary parts are drawn from the uniform distribution on $[-1, 1]$. In the last case, note that our earlier discussions over $\mathbb{R}$ apply verbatim over $\mathbb{C}$ with the same growth factors.



(a) Real random matrices $\mathcal{U}[-1,1]$.



(b) Real random matrices $\mathcal{N}(0,1)$.



(c) Complex random matrices $\mathcal{U}[-1,1]+\mathcal{U}[-1,1]i$.

Figure 3.1: Accuracy of Strassen's algorithm and Winograd's variant.

In all cases, we compute $\widehat{\beta}_{\mathsf{S}}(A, B)$ and $\widehat{\beta}_{\mathsf{W}}(A, B)$ using Strassen's algorithm and Winograd's variant respectively and compare the results against the exact value $\beta(A, B) = AB$ computed using the MATLAB symbolic toolbox. From Figure 3.1, we see that Strassen's algorithm is indeed more accurate than Winograd's variant, substantiating Theorem 3.3.3. Even though the 14.83 growth factor of Strassen's algorithm appears to differ only moderately from the 17.85 growth factor of Winograd's variant, the effect is magnified multifold as a result of recursion — these algorithms are applied recursively to an $n \times n$ matrix as a block 2×2 matrix $\lfloor \log_2 n \rfloor$ times. The conventional algorithm, which has a growth factor of 8, is included in these plots for comparison.

3.5 Complex Multiplication

As described towards the end of Section 3.2, complex multiplication is an $\mathbb{R}$-bilinear operator $\beta_{\mathbb{C}} \in \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ when we identify $\mathbb{C} \cong \mathbb{R}^2$, with the standard basis vectors in $\mathbb{R}^2$

$$e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

corresponding to $1, i \in \mathbb{C}$. We write $e_1^*, e_2^* : \mathbb{R}^2 \to \mathbb{R}$ for the dual basis, i.e., linear functionals with

$$e_1^*\left(\begin{bmatrix} a \\ b \end{bmatrix}\right) = a, \qquad e_2^*\left(\begin{bmatrix} a \\ b \end{bmatrix}\right) = b.$$

We will denote the regular algorithm $(a + bi)(c + di) = (ac - bd) + i(bc + ad)$, Gauss's algorithm (3.1.1), and our new algorithm (3.1.5) by $\widehat{\beta}_{\mathsf{R}}, \widehat{\beta}_{\mathsf{G}}, \widehat{\beta}_{\mathsf{N}}$ respectively. For easy refer-

ence,

$$\widehat{\beta}_{\mathsf{R}}\left(\begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} c \\ d \end{bmatrix}\right) = \begin{bmatrix} ac - bd \\ bc + ad \end{bmatrix}, \qquad \widehat{\beta}_{\mathsf{G}}\left(\begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} c \\ d \end{bmatrix}\right) = \begin{bmatrix} ac - bd \\ (a+b)(c+d) - ac - bd \end{bmatrix},$$

$$\widehat{\beta}_{\mathsf{N}}\left(\begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} c \\ d \end{bmatrix}\right) = \begin{bmatrix} \frac{1}{2}\left[\left(a + \frac{1}{\sqrt{3}}b\right)\left(c + \frac{1}{\sqrt{3}}d\right) + \left(a - \frac{1}{\sqrt{3}}b\right)\left(c - \frac{1}{\sqrt{3}}d\right) - \frac{8}{3}bd\right] \\ \frac{i\sqrt{3}}{2}\left[\left(a + \frac{1}{\sqrt{3}}b\right)\left(c + \frac{1}{\sqrt{3}}d\right) - \left(a - \frac{1}{\sqrt{3}}b\right)\left(c - \frac{1}{\sqrt{3}}d\right)\right] \end{bmatrix}.$$

They correspond to the decompositions

$$\widehat{\beta}_{\mathsf{R}} = (e_1^* \otimes e_1^* - e_2^* \otimes e_2^*) \otimes e_1 + (e_1^* \otimes e_2^* + e_2^* \otimes e_1^*) \otimes e_2, \tag{3.5.1}$$

$$\widehat{\beta}_{\mathsf{G}} = (e_1^* + e_2^*) \otimes (e_1^* + e_2^*) \otimes e_2 + e_1^* \otimes e_1^* \otimes (e_1 - e_2) - e_2^* \otimes e_2^* \otimes (e_1 + e_2), \tag{3.5.2}$$

$$\begin{aligned} \widehat{\beta}_{\mathsf{N}} = \frac{4}{3}\Bigg(& \left[\frac{\sqrt{3}}{2}e_1^* + \frac{1}{2}e_2^*\right] \otimes \left[\frac{\sqrt{3}}{2}e_1^* + \frac{1}{2}e_2^*\right] \otimes \left[\frac{1}{2}e_1 + \frac{\sqrt{3}}{2}e_2\right] \\ & + \left[\frac{\sqrt{3}}{2}e_1^* - \frac{1}{2}e_2^*\right] \otimes \left[\frac{\sqrt{3}}{2}e_1^* - \frac{1}{2}e_2^*\right] \otimes \left[\frac{1}{2}e_1 - \frac{\sqrt{3}}{2}e_2\right] - e_2^* \otimes e_2^* \otimes e_1\Bigg). \end{aligned} \tag{3.5.3}$$

3.5.1 Bilinear stability of complex multiplication algorithms

Recall from Section 3.2 that $\operatorname{rank}(\beta_{\mathbb{C}}) = 3 = \overline{\operatorname{rank}}(\beta_{\mathbb{C}})$, i.e., both Gauss's algorithm and our new algorithm have optimal bilinear complexity whether in the exact or approximate sense. One may also show that $\beta_{\mathbb{C}}$ has nuclear norm [43, Lemma 6.1] is given by

$$\|\beta_{\mathbb{C}}\|_\nu = 4.$$

The growth factor of the regular algorithm (3.5.1) attains this minimum value,

$$\begin{aligned} \gamma(\widehat{\beta}_{\mathsf{R}}) &= \|e_1^*\|_*\|e_1^*\|_*\|e_1\| + \|-e_2^*\|_*\|e_2^*\|_*\|e_1\| + \|e_1^*\|_*\|e_2^*\|_*\|e_2\| + \|e_2^*\|_*\|e_1^*\|_*\|e_2\| \\ &= 4 = \|\beta_{\mathbb{C}}\|_\nu, \end{aligned}$$

as does our new algorithm (3.5.3),

$$\begin{aligned}\gamma(\widehat{\beta}_{\mathsf{N}}) &= \frac{4}{3}\bigg(\left\|\frac{\sqrt{3}}{2}e_1^* + \frac{1}{2}e_2^*\right\|_*\left\|\frac{\sqrt{3}}{2}e_1^* + \frac{1}{2}e_2^*\right\|_*\left\|\frac{1}{2}e_1 + \frac{\sqrt{3}}{2}e_2\right\| \\ &\qquad + \left\|\frac{\sqrt{3}}{2}e_1^* - \frac{1}{2}e_2^*\right\|_*\left\|\frac{\sqrt{3}}{2}e_1^* - \frac{1}{2}e_2^*\right\|_*\left\|\frac{1}{2}e_1 - \frac{\sqrt{3}}{2}e_2\right\| + \|e_2^*\|_*\|e_2^*\|_*\|e_1\|\bigg) \\ &= 4 = \|\beta_{\mathbb{C}}\|_\nu,\end{aligned}$$

but not Gauss's algorithm (3.5.2),

$$\begin{aligned}\gamma(\widehat{\beta}_{\mathsf{G}}) &= \|e_1^* + e_2^*\|_*\|e_1^* + e_2^*\|_*\|e_2\| + \|e_1^*\|_*\|e_1^*\|_*\|e_1 - e_2\| + \|-e_2^*\|_*\|e_2^*\|_*\|e_1 + e_2\| \\ &= 2(1+\sqrt{2}) > \|\beta_{\mathbb{C}}\|_\nu.\end{aligned}$$

So Gauss's algorithm $\widehat{\beta}_{\mathsf{G}}$ is faster (by bilinear complexity) but less stable (by bilinear stability) than the regular algorithm. Our new algorithm $\widehat{\beta}_{\mathsf{N}}$ on the other hand is optimal in both measures, attaining both $\operatorname{rank}(\beta_{\mathbb{C}})$ and $\|\beta_{\mathbb{C}}\|_\nu$.

We stress that numerical stability is too complicated an issue to be completely covered by the simple framework of bilinear stability. For instance, from the perspective of cancellation errors, our new algorithm also suffers from the issue pointed out in [58, Section 23.2.4] for Gauss's algorithm. By choosing $z = w$ and $b = \sqrt{3}/a$, our algorithm (3.5.3) computes

$$\frac{1}{2}\Big[\Big(a+\frac{1}{a}\Big)^2 + \Big(a-\frac{1}{a}\Big)^2 - \frac{8}{a^2}\Big] + \frac{i\sqrt{3}}{2}\Big[\Big(a+\frac{1}{a}\Big)^2 - \Big(a-\frac{1}{a}\Big)^2\Big] =: x + iy.$$

There will be cancellation error in the computed real part $\widehat{x}$ when $|a|$ is small and likewise in the computed imaginary part $\widehat{y}$ when $|a|$ is large. Nevertheless, as discussed in [58, Section 23.2.4], the new algorithm (3.5.3) is still stable in the weaker sense of having acceptably small $|x-\widehat{x}|/|z|$ and $|y-\widehat{y}|/|z|$ even if $|x-\widehat{x}|/|x|$ or $|y-\widehat{y}|/|y|$ might be large.

3.5.2 Error analysis of new algorithm applied to matrices

While using Gauss's algorithm or our new algorithm for multiplying of complex numbers is a pointless overkill, they become useful when applied to the multiplication of complex matrices. Note that any complex matrices $A + iB, C + iD \in \mathbb{C}^{n\times n}$ may be multiplied via their real and imaginary parts $A, B, C, D \in \mathbb{R}^{n\times n}$:

$$(A + iB)(C + iD) = (AC - BD) + i[AD + BC], \tag{3.5.4}$$

allowing us to focus our attention on designing algorithms for real matrix products. In this regard, Gauss's algorithm applied in the form

$$(A + iB)(C + iD) = (AC - BD) + i[(A + B)(C + D) - AC - BD] \tag{3.5.5}$$

reduces the number of real matrix products from four to three at the expense of more matrix additions. This represents an enormous saving as matrix products are invariably orders of magnitude more costly than matrix additions — note that this statement remains true even if the exponent of matrix multiplication ω is 2. Our new algorithm (3.1.5) likewise applies in the form

$$\begin{aligned}(A + iB)(C + iD) = \frac{1}{2}\left[\left(A + \frac{1}{\sqrt{3}}B\right)\left(C + \frac{1}{\sqrt{3}}D\right) + \left(A - \frac{1}{\sqrt{3}}B\right)\left(C - \frac{1}{\sqrt{3}}D\right) - \frac{8}{3}BD\right] \\ + \frac{i\sqrt{3}}{2}\left[\left(A + \frac{1}{\sqrt{3}}B\right)\left(C + \frac{1}{\sqrt{3}}D\right) - \left(A - \frac{1}{\sqrt{3}}B\right)\left(C - \frac{1}{\sqrt{3}}D\right)\right],\end{aligned} \tag{3.5.6}$$

trading expensive matrix products for inexpensive scalar multiplications and additions.

The following is an error analysis of (3.5.6), i.e., our new algorithm applied to complex matrix multiplication. We emulate a similar analysis for Gauss's algorithm in [55, 58], assuming in particular that the real matrix multiplications involved are performed using the

conventional algorithm (as opposed to Strassen's or Winograd's). We remind the reader that conventional matrix multiplication has the simple error bound

$$|AB - \mathsf{fl}(AB)| \leq n|A||B|\mathsf{u} + O(\mathsf{u}^2) \tag{3.5.7}$$

for $A, B \in \mathbb{R}^{n\times n}$.

Theorem 3.5.1 (Error analysis for our new algorithm)**.** Let $(A+iB)(C+iD) = F+iG$ with $F, G \in \mathbb{R}^{n\times n}$ and let $\widehat{F}_{\mathsf{N}}, \widehat{G}_{\mathsf{N}}$ be computed via (3.5.6) in floating point arithmetic satisfying (3.3.3). Then

$$|F - \widehat{F}_{\mathsf{N}}| \leq (n+7)\left(|A| + \frac{1}{\sqrt{3}}|B|\right)\left(|C| + \frac{1}{\sqrt{3}}|D|\right)\mathsf{u} + \left(\frac{4}{3}n + 4\right)|B||D|\mathsf{u} + O(\mathsf{u}^2), \tag{3.5.8}$$

$$|G - \widehat{G}_{\mathsf{N}}| \leq \sqrt{3}(n+6)\left(|A| + \frac{1}{\sqrt{3}}|B|\right)\left(|C| + \frac{1}{\sqrt{3}}|D|\right)\mathsf{u} + O(\mathsf{u}^2), \tag{3.5.9}$$

where the inequality $\leq$ and absolute value $|\cdot|$ both apply in a coordinatewise sense.

Proof. Following [58], we use the same letter δ to denote the error incurred in each step of our algorithm. So, for example,

$$\mathsf{fl}(B/\sqrt{3}) = B/\sqrt{3} + \delta B/\sqrt{3}.$$

In the following we will define matrices H_i and let $\widehat{H}_i$ be its computed value, $i = 1, \ldots, 8$.

Let $H_1 \coloneqq A + B/\sqrt{3}$. Then

$$\begin{aligned}
\widehat{H}_1 &= \mathsf{fl}(A + B/\sqrt{3} + \delta B/\sqrt{3}) = (A + B/\sqrt{3} + \delta B/\sqrt{3})(1+\delta) \\
&= A + B/\sqrt{3} + \delta(A + 2B/\sqrt{3}) + O(\mathsf{u}^2) \\
&= H_1 + 2\Delta_1 + O(\mathsf{u}^2), \qquad |\Delta_1| \leq (|A| + |B|/\sqrt{3})\mathsf{u}.
\end{aligned}$$

Similarly $H_2 \coloneqq C + D/\sqrt{3}$ satisfies

$$\widehat{H}_2 = H_2 + 2\Delta_2 + O(\mathsf{u}^2), \qquad |\Delta_2| \le (|C| + |D|/\sqrt{3})\mathsf{u}.$$

Let $H_3 \coloneqq (A + B/\sqrt{3})(C + D/\sqrt{3})$. By (3.5.7),

$$\widehat{H}_3 = (A + B/\sqrt{3} + 2\Delta_1)(C + D/\sqrt{3} + 2\Delta_2) + n\Delta_3 + O(\mathsf{u}^2) \tag{3.5.10}$$

where

$$\begin{aligned}
|\Delta_3| &\le |(A + B/\sqrt{3} + 2\Delta_1)||(C + D/\sqrt{3} + 2\Delta_2)|\mathsf{u} \\
&\le (|A| + |B|/\sqrt{3} + 2|\Delta_1|)(|C| + |D|/\sqrt{3} + 2|\Delta_2|)\mathsf{u} \\
&\le (|A| + |B|/\sqrt{3} + 2\mathsf{u}(|A| + |B|/\sqrt{3}))(|C| + |D|/\sqrt{3} + 2\mathsf{u}(|C| + |D|/\sqrt{3}))\mathsf{u} \\
&\le (|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3})\mathsf{u} + O(\mathsf{u}^2).
\end{aligned} \tag{3.5.11}$$

By (3.5.10) and (3.5.11),

$$\begin{aligned}
\widehat{H}_3 &= (A + B/\sqrt{3})(C + D/\sqrt{3}) + 2\Delta_1(C + D/\sqrt{3}) \\
&\qquad + 2(A + B/\sqrt{3})\Delta_2 + n\Delta_3 + O(\mathsf{u}^2) \\
&= H_3 + (n+4)\Delta_4 + O(\mathsf{u}^2)
\end{aligned} \tag{3.5.12}$$

where

$$|\Delta_4| \le (|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3})\mathsf{u}.$$

Similarly $H_4 \coloneqq (A - B/\sqrt{3})(C - D/\sqrt{3})$ satisfies

$$\widehat{H}_4 = H_4 + (n+4)\Delta_5 + O(\mathsf{u}^2) \tag{3.5.13}$$

where

$$|\Delta_5| \leq (|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3})\mathsf{u}.$$

Let $H_5 \coloneqq (A + B/\sqrt{3})(C + D/\sqrt{3}) + (A - B/\sqrt{3})(C - D/\sqrt{3})$. By (3.5.12) and (3.5.13),

$$\begin{aligned} \widehat{H}_5 &= [H_3 + (n+4)\Delta_4 + H_4 + (n+4)\Delta_5](1+\delta) + O(u^2) \\ &= H_5 + (2n+10)\Delta_6 + O(\mathsf{u}^2) \end{aligned} \tag{3.5.14}$$

where

$$|\Delta_6| \leq \mathsf{u}(|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3}).$$

Let $H_6 \coloneqq 8/3BD$. Then

$$\begin{aligned} \widehat{H}_6 &= \mathsf{fl}(8/3(BD + n\Delta_7)) + O(\mathsf{u}^2) \\ &= 8/3(BD + n\Delta_7)(1+\delta) + O(\mathsf{u}^2) \\ &= H_6 + 8/3(n+1)\Delta_8 + O(\mathsf{u}^2) \end{aligned} \tag{3.5.15}$$

where

$$|\Delta_7| \leq |B||D|\mathsf{u}, \qquad |\Delta_8| \leq |B||D|\mathsf{u}.$$

Let $H_7 \coloneqq H_5 - H_6$. By (3.5.14) and (3.5.15),

$$\begin{aligned} \widehat{H}_7 &= [H_5 + (2n+10)\Delta_6 - H_6 - 8/3(n+1)\Delta_8](1+\delta) + O(\mathsf{u}^2) \\ &= H_7 + (2n+12)\Delta_9 + 8/3(n+2)\Delta_{10} + O(\mathsf{u}^2) \end{aligned}$$

where

$$|\Delta_9| \leq (|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3})\mathsf{u}, \qquad |\Delta_{10}| \leq |B||D|\mathsf{u}.$$

Then

$$\begin{aligned}\widehat{F}_{\mathsf{N}} &= (1+\delta)[H_7 + (2n+12)\Delta_9 + 8/3(n+2)\Delta_{10}]/2 + O(\mathsf{u}^2) \\ &= F + (n+7)\Delta_{11} + 4/3(n+3)\Delta_{12} + O(\mathsf{u}^2)\end{aligned}$$

where

$$|\Delta_{11}| \leq (|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3})\mathsf{u}, \qquad |\Delta_{12}| \leq |B||D|\mathsf{u},$$

and from which we obtain (3.5.8).

Let $H_8 \coloneqq (A + B/\sqrt{3})(C + D/\sqrt{3}) - (A - B/\sqrt{3})(C - D/\sqrt{3})$. Similar to (3.5.14), we have

$$\widehat{H}_8 = H_8 - (2n+10)\Delta_{13} + O(\mathsf{u}^2)$$

where

$$|\Delta_{13}| \leq (|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3})\mathsf{u}.$$

Then

$$\begin{aligned}\widehat{G}_{\mathsf{N}} &= \sqrt{3}/2[H_8 - (2n+10)\Delta_{13}](1+\delta) + O(\mathsf{u}^2) \\ &= G + \sqrt{3}(n+6)\Delta_{14} + O(\mathsf{u}^2)\end{aligned}$$

where

$$|\Delta_{14}| \leq (|A| + |B|/\sqrt{3})(|C| + |D|/\sqrt{3})\mathsf{u},$$

from which we obtain (3.5.9). □

If we compute the matrices F, G in Theorem 3.5.1 using Gauss's algorithm (3.5.5) with floating point arithmetic and let the results be $\widehat{F}_{\mathsf{G}}$ and $\widehat{G}_{\mathsf{G}}$, then the corresponding error

bounds [55, 58] are

$$\begin{aligned}
|F - \widehat{F}_{\mathsf{G}}| &\leq (n+1)(|A||C| + |B||D|)\mathsf{u} + O(\mathsf{u}^2), \\
|G - \widehat{G}_{\mathsf{G}}| &\leq (n+4)[(|A| + |B|)(|C| + |D|) + |A||C| + |B||D|]\mathsf{u} + O(\mathsf{u}^2).
\end{aligned} \tag{3.5.16}$$

When $n \to \infty$, we have $n + c \approx n$ for any constant c. Hence the errors in (3.5.8) and (3.5.9) are dominated by

$$\begin{aligned}
|F - \widehat{F}_{\mathsf{N}}| &\sim n\left[|A||C| + \frac{5}{3}|B||D| + \frac{1}{\sqrt{3}}|A||D| + \frac{1}{\sqrt{3}}|B||C|\right]\mathsf{u}, \\
|G - \widehat{G}_{\mathsf{N}}| &\sim n\left[\sqrt{3}|A||C| + |B||C| + |A||D| + \frac{1}{\sqrt{3}}|B||D|\right]\mathsf{u},
\end{aligned}$$

whereas those in (3.5.16) are dominated by

$$\begin{aligned}
|F - \widehat{F}_{\mathsf{G}}| &\sim n(|A||C| + |B||D|)\mathsf{u}, \\
|G - \widehat{G}_{\mathsf{G}}| &\sim n(2|A||C| + 2|B||D| + |A||D| + |B||C|)\mathsf{u}.
\end{aligned}$$

For easy comparison suppose the magnitudes of the entries in A, B, C, D are all approximately θ, then these reduce to

$$\begin{aligned}
|F - \widehat{F}_{\mathsf{N}}| &\sim 3.8n^2\theta^2, \qquad & |G - \widehat{G}_{\mathsf{N}}| &\sim 4.3n^2\theta^2, \\
|F - \widehat{F}_{\mathsf{G}}| &\sim 2n^2\theta^2, \qquad & |G - \widehat{G}_{\mathsf{G}}| &\sim 6n^2\theta^2.
\end{aligned} \tag{3.5.17}$$

So Gauss's algorithm gives an imaginary part that is three times less accurate than its real part. Note the the imaginary part of Gauss's algorithm accounts for all its computational savings; the real part is just the regular algorithm. On the other hand, our algorithm balances the accuracy of both the real and imaginary parts by spreading out the computational savings across both parts.

To quantify this, we use the max norm. For a complex matrix $A + iB \in \mathbb{C}^{n\times n}$, this is

$$\|A + iB\|_{\max} \coloneqq \max\{|a_{ij}|, |b_{ij}| : i, j = 1, \ldots, n\}. \tag{3.5.18}$$

The max norm differs from the usual matrix ∞-norm given by maximum row sum used in [55, 58]. We favor the max norm as it is the strictest measure of numerical accuracy — a small max norm error implies that each entry is accurate as opposed to accurate on average.

If we denote the matrices resulting from Gauss's algorithm and our new algorithm by

$$\widehat{E}_{\mathsf{G}} \coloneqq \widehat{F}_{\mathsf{G}} + i\widehat{G}_{\mathsf{G}}, \qquad \widehat{E}_{\mathsf{N}} \coloneqq \widehat{F}_{\mathsf{N}} + i\widehat{G}_{\mathsf{N}}$$

respectively, we expect $\|E - \widehat{E}_{\mathsf{N}}\|_{\max}$ to be smaller than $\|E - \widehat{E}_{\mathsf{G}}\|_{\max}$. The extensive experiments in the next section will attest to this.

3.6 Experiments for New Complex Matrix Multiplication Algorithm

The goal of this section is to provide numerical evidence to show that our new algorithm (3.5.6) for complex matrix multiplication is

- nearly as accurate as the regular algorithm (3.5.4), and
- nearly as fast as Gauss's algorithm (3.5.5).

We begin with routine experiments comparing the three algorithms (3.5.4), (3.5.5), (3.5.6) on random matrices, and move on to three actual applications: matrix polynomial evaluations, unitary transformations, and the increasingly popular complex-valued neural networks. The results, we think, show that our new algorithm can be a realistic replacement for Gauss's algorithm in engineering applications.

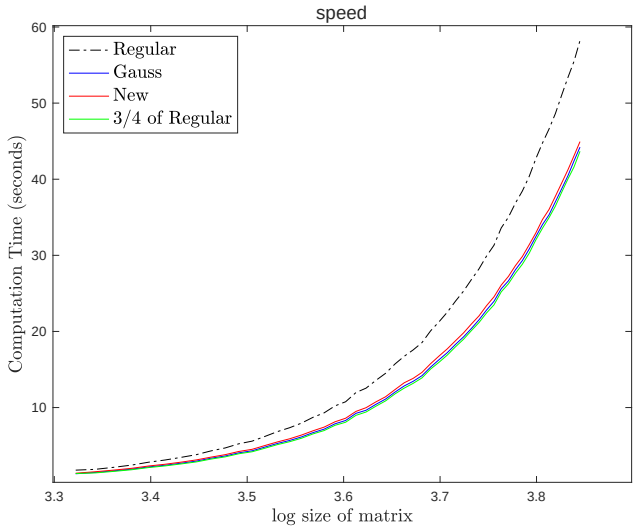


Figure 3.2: Speed of the three algorithms for complex matrix multiplication.

3.6.1 Speed of the algorithms

We generate random $A + iB, C + iD \in \mathbb{C}^{n\times n}$ with entries of A, B, C, D drawn uniformly in $[-1, 1]$; the results with standard normal are similar and omitted. We increase n from 2100 to 7000 in steps of 100. The product $(A+iB)(C+iD)$ is computed numerically with the regular algorithm (3.5.4), Gauss's algorithm (3.5.5), and our new algorithm (3.5.6). For each n, we generate ten different matrices and record the average time taken for each algorithm and plot these in Figure 3.2, with wall time (in seconds) for vertical axis and $\log_{10}(n)$ for horizontal axis. The time taken by MATLAB's internal function for complex matrix multiplication is virtually indistinguishable from that of the regular algorithm and therefore omitted.

Consistent with the predictions of bilinear complexity, our new algorithm has roughly the same computation time as Gauss's algorithm, at roughly 3/4 the time taken by the regular algorithm. We will perform more speed experiments in conjunction with our accuracy experiments in Section 3.6.2.

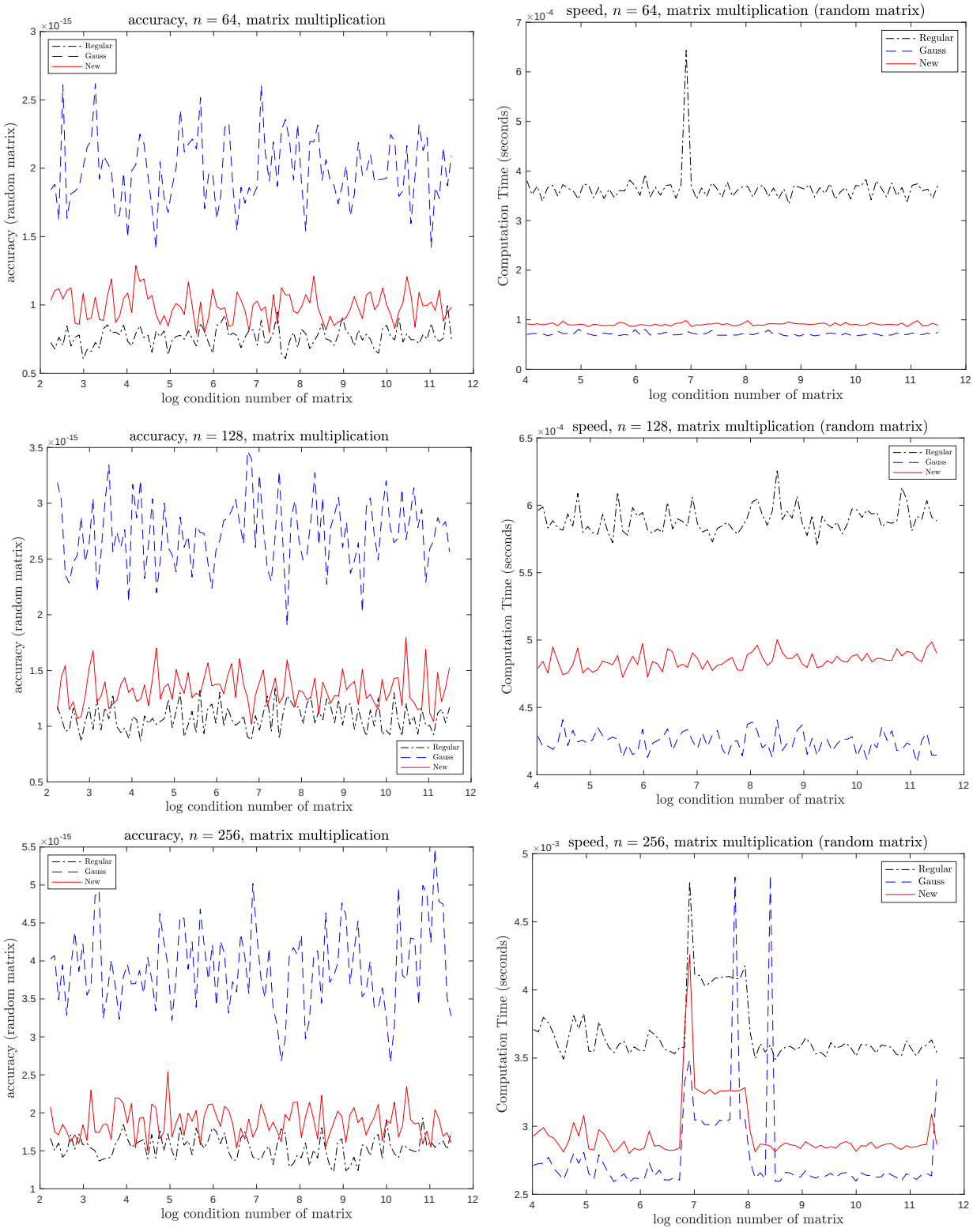


Figure 3.3: Accuracy and speed of algorithms for complex matrix multiplication.

3.6.2 Accuracy of the algorithms

We generate random $A + iB, C + iD \in \mathbb{C}^{n\times n}$ with $n = 64, 128, 256$ and with condition numbers ranging from 174 to 3×10^{11}. It is desirable to limit ourselves to matrices over Gaussian rationals, i.e., $\mathbb{Q}+\mathbb{Q}i$, as we will need to compute the exact values of their products later.

The way we generate such a matrix requires some elaboration. For an $X \in \mathbb{Z}^{n\times n}$ with a specified $\kappa_2(X) = \kappa \in \mathbb{Z}$, we form a diagonal $\Lambda \in \mathbb{R}^{n\times n}$ whose diagonal entries are 1 and κ toegether with $n-2$ other random integers between 1 and $\kappa-1$. We then form $X = H\Lambda H^{\mathsf{T}}$ with a random Hadamard matrix $H \in \mathbb{Z}^{n\times n}$. If A and B are generated in this manner, then they are dense matrices (important as we do not want sparsity to unduly influence arithmetic costs) and $\kappa_2(A + iB) = \kappa_2(A) = \kappa_2(B) = \kappa$ as $(\kappa + \kappa i)/(1 + i) = \kappa$.

We compute the exact value of $(A+iB)(C+iD)$ symbolically with MATLAB's symbolic toolbox. Given our relatively modest computational resources, this is the bottleneck for our experiments as this step becomes prohibitively expensive when $n > 256$. In generating the $n = 256$ plots in Figure 3.3, this step alone took 40 hours on our University's Research Computing Center servers.

For each pair of complex matrices $A+iB$ and $C+iD$, we compute their product $\widehat{E}$ using each of the three algorithms (3.5.4), (3.5.5), (3.5.6), and compare them against the exact result E via the max norm relative error

$$\frac{\|E - \widehat{E}\|_{\max}}{\|A + iB\|_{\max}\|C + iD\|_{\max}}.$$

As discussed in [55, 58], it is natural to measure error in matrix multiplication relative to the norms of the input matrices. We use the max norm in (3.5.18) to better capture entrywise accuracy.

The results are plotted in Figure 3.3: speed plots have wall time in seconds on the vertical

axes; accuracy plots have relative error on the vertical axes; all plots have $\log_{10}(\kappa)$ on the horizontal axes. We repeat each experiment ten times: every value on these plots comes from averaging across the results of ten pairs of random matrices with the same condition number.

Observations from Figure 3.3: The accuracy of our new algorithm is much higher than that of Gauss's algorithm and only slightly worse than that of the regular algorithm. Gauss's algorithm also shows a great deal more fluctuation across varying condition numbers than either our new algorithm or the regular one. When it comes to speed, our algorithm is closer to that of Gauss's than the regular algorithm. These accuracy results attest to Theorem 3.5.1 and the discussions around (3.5.17).

The relative errors and wall times for MATLAB's internal function for complex matrix multiplication are virtually indistinguishable from those of the regular algorithm (that we implemented ourselves) and thus omitted. In the next three sections, we will compare the accuracy and speed of the three complex matrix multiplication algorithms in more realistic scenarios.

3.6.3 Matrix polynomial evaluations

We evaluate a polynomial $p(x) = \sum_{k=0}^{d} a_k x^k$ with coefficients $a_0, \ldots, a_k \in \mathbb{R}$ at a $X \in \mathbb{C}^{n\times n}$. This is a problem that occurs in many tasks involving matrix functions [58, 59]. We limit ourselves to real coefficients as this is by far most common scenario [59]; but the complex coefficients case simply reduces to evaluating two real polynomials $\operatorname{Re} p(x)$ and $\operatorname{Im} p(x)$. The celebrated Horner's rule [59, Algorithm 4.3] reduces the problem to one of repeated matrix multiplications.

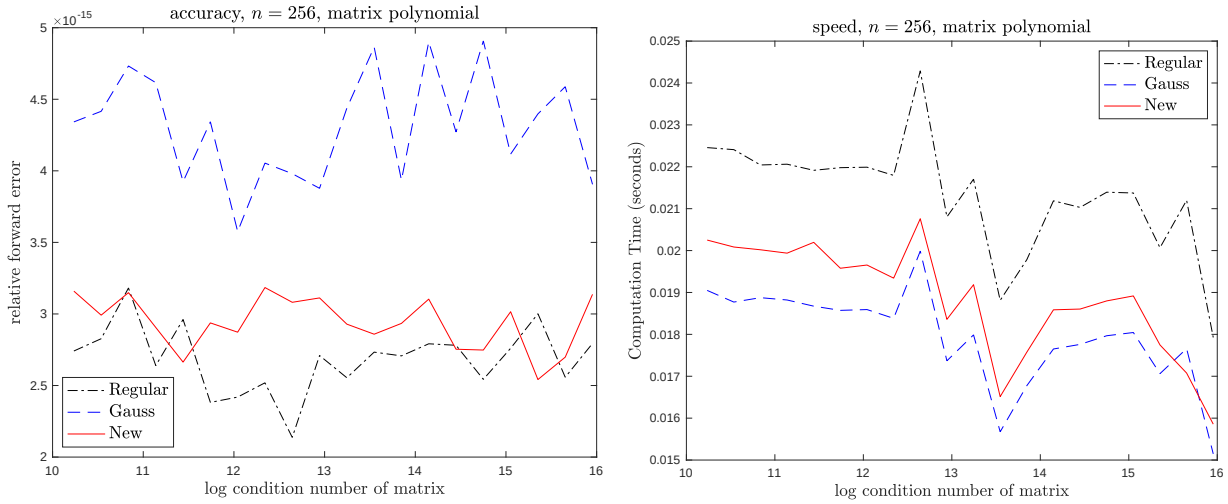


Figure 3.4: The three algorithms applied to matrix polynomial evaluations.

Algorithm 1 Compute $p(X)$ via Horner's rule

Input $a_0, a_1, \ldots, a_d \in \mathbb{R}$, $X \in \mathbb{C}^{n\times n}$

Output $a_0 I + a_1 X + \cdots + a_d X^d$

1: $P = X$;

2: $S = a_0 I + a_1 X$;

3: **for** $k = 2 : d$ **do**

4: $\quad P = PX$;

5: $\quad S = S + a_k P$;

6: **end for**

7: return S;

We generate random matrices $X \in \mathbb{C}^{256\times 256}$ with condition numbers from 2^{34} to 2^{53} as described in Section 3.6.2. We set $d = 5$ and choose random $b_0, \ldots, b_5 \in (0, 1)$ uniformly. We then evaluate $p(X)$ using Algorithm 1, with Step 4 computed via (3.5.4), (3.5.5), and (3.5.6). We measure accuracy in terms of the max norm relative forward error

$$\frac{\|p(X) - \widehat{p}(X)\|_{\max}}{\|p(X)\|_{\max}},$$

using MATLAB symbolic toolbox for the exact value of $p(X)$. The results presented in Figure 3.4 again show that our new algorithm is nearly as accurate as the regular algorithm and nearly as fast as Gauss's algorithm. While our accuracy tests are again limited by our capacity for symbolic computation ($n = 256$ is fine, $n = 512$ is beyond reach), our speed tests can go far beyond (to around $n = 4096$), and they show a profile much like Figure 3.2.

3.6.4 Unitary transforms

Given a unitary matrix $U \in \mathbb{C}^{n\times n}$ and a complex matrix $X \in \mathbb{C}^{n\times n}$, it may come as a surprise to the reader that unless U happens to be some special transforms like FFT, DCT, DWT, etc, or has already been factored into a product of Householder or Givens matrices, there is no known special algorithm for forming UX that would take advantage of the unitarity of U. Nevertheless, such unitary matrices with no additional special structure are not uncommon. For instance, the matrix U could come from polar decompositions or matrix sign functions [54, 56, 65], and computed via iterative methods [54, 56, 65] and thus not in Householder- or Givens-factored form. Here we will explore the use of algorithms (3.5.4), (3.5.5), (3.5.6) for unitary transforms $X \mapsto UX$.

We generate the unitary matrix $U \in \mathbb{C}^{256\times 256}$ by QR factoring complex random matrices with entries in $\mathcal{U}[0,1] + \mathcal{U}[0,1]i$. Note that a unitary matrix is always perfectly conditioned. The matrix $X \in \mathbb{C}^{256\times 256}$ is generated randomly with condition numbers from 2^{34} to 2^{53} as in Section 3.6.3. We compute the exact value $E \coloneqq UX$ symbolically as before and measure the accuracy of our computed value $\widehat{E}$ by

$$\frac{\|E - \widehat{E}\|_{\max}}{\|U\|_{\max}\|X\|_{\max}}.$$

The results, presented in Figure 3.5, allow us to draw the same conclusion as in the Section 3.6.3. Further speed tests up to $n = 4096$ again show a profile much like Figure 3.2.

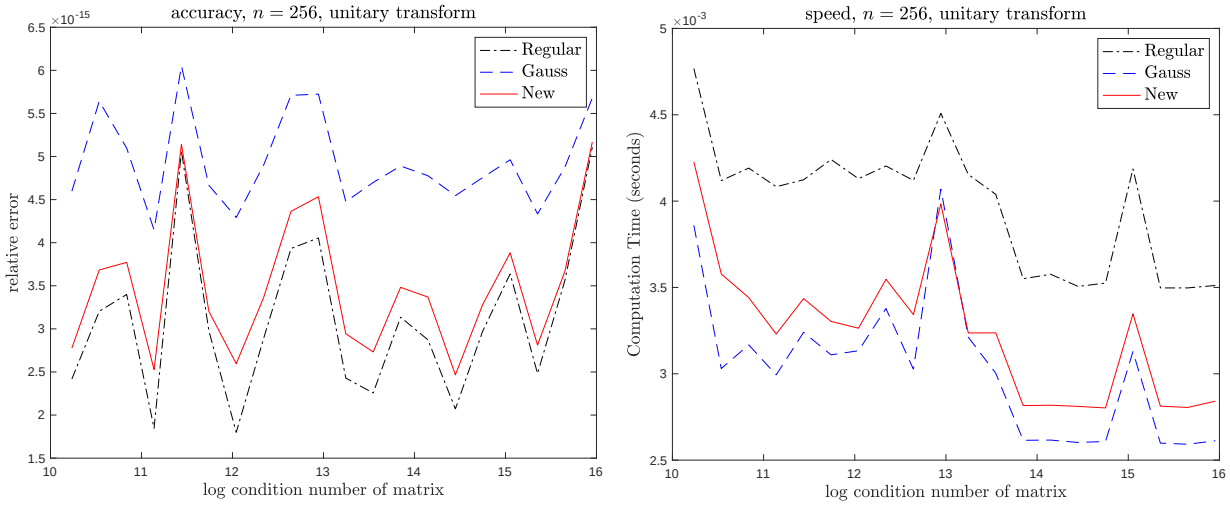


Figure 3.5: The three algorithms applied to unitary transforms.

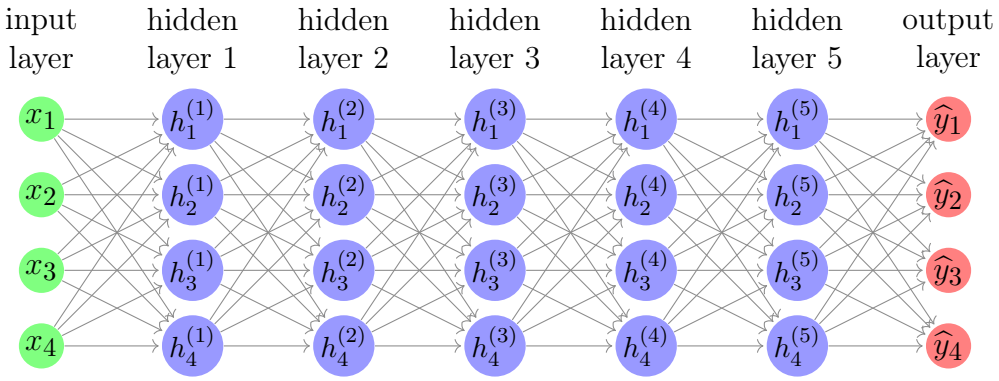


Figure 3.6: A constant width neural network with input dimension $n = 4$ and depth $d = 6$. The edges between adjacent layers are weighted with weight matrices.

3.6.5 Complex-valued neural networks

Given the amount of recent attention it has received, a feedforward neural network probably requires no introduction, and we will just limit ourselves to the obligatory Figure 3.6.

A complex-valued neural networks is simply a neural network with complex-valued weights and is activated by a complex function. It has become increasingly important and is widely used in signal processing and computer vision [1, 8, 26, 105, 117, 126]. For simplicity,

we consider a d-layer constant width version $f : \mathbb{C}^n \to \mathbb{C}^n$ given by

$$f(W_1, \ldots, W_d, \sigma)(x) \coloneqq W_d\sigma(W_{d-1}\sigma(\cdots W_2\sigma(W_1x)\cdots)),$$

with weight matrices $W_1, \ldots, W_d \in \mathbb{C}^{n\times n}$ and activation function $\sigma : \mathbb{C} \to \mathbb{C}$ applied coordinatewise on $\mathbb{C}^n$. Evaluating a trained complex-valued neural network on multiple inputs $x_1, \ldots, x_m \in \mathbb{C}^n$ is an indispensable task when we employ it to make new predictions. Here we will compare the performance of the three algorithms (3.5.4), (3.5.5), (3.5.6) when used for this purpose.

For concreteness, we choose a depth of $d = 6$ and use the complex ReLU activation [8, 117]

$$\sigma(a + bi) \coloneqq \max(a, 0) + \max(b, 0)i.$$

We generate random weight matrices $W_1, \ldots, W_6 \in \mathbb{C}^{n\times n}$ with $n = 64$ and 128, and with condition numbers ranging from 2^{34} to 2^{53}. We also generate random inputs $X = [x_1, \ldots, x_m] \in \mathbb{C}^{n\times m}$ with entries drawn from $\mathcal{U}[-\frac{1}{2}, \frac{1}{2}] + \mathcal{U}[-\frac{1}{2}, \frac{1}{2}]i$, and with $(m, n) = (25, 64)$ or $(50, 128)$. The task is then to evaluate

$$E \coloneqq f(W_1, \ldots, W_d, \sigma)(X) \coloneqq W_d\sigma(W_{d-1}\sigma(\cdots W_2\sigma(W_1X)\cdots)).$$

Again we compute its exact value E symbolically, apply the three algorithms to obtain $\widehat{E}$ numerically, and measure accuracy in terms of the relative forward error

$$\frac{\|E - \widehat{E}\|_{\max}}{\|E\|_{\max}}.$$

The results, shown in Figure 3.7, are fully consistent with those in Sections 3.6.3 and 3.6.4.

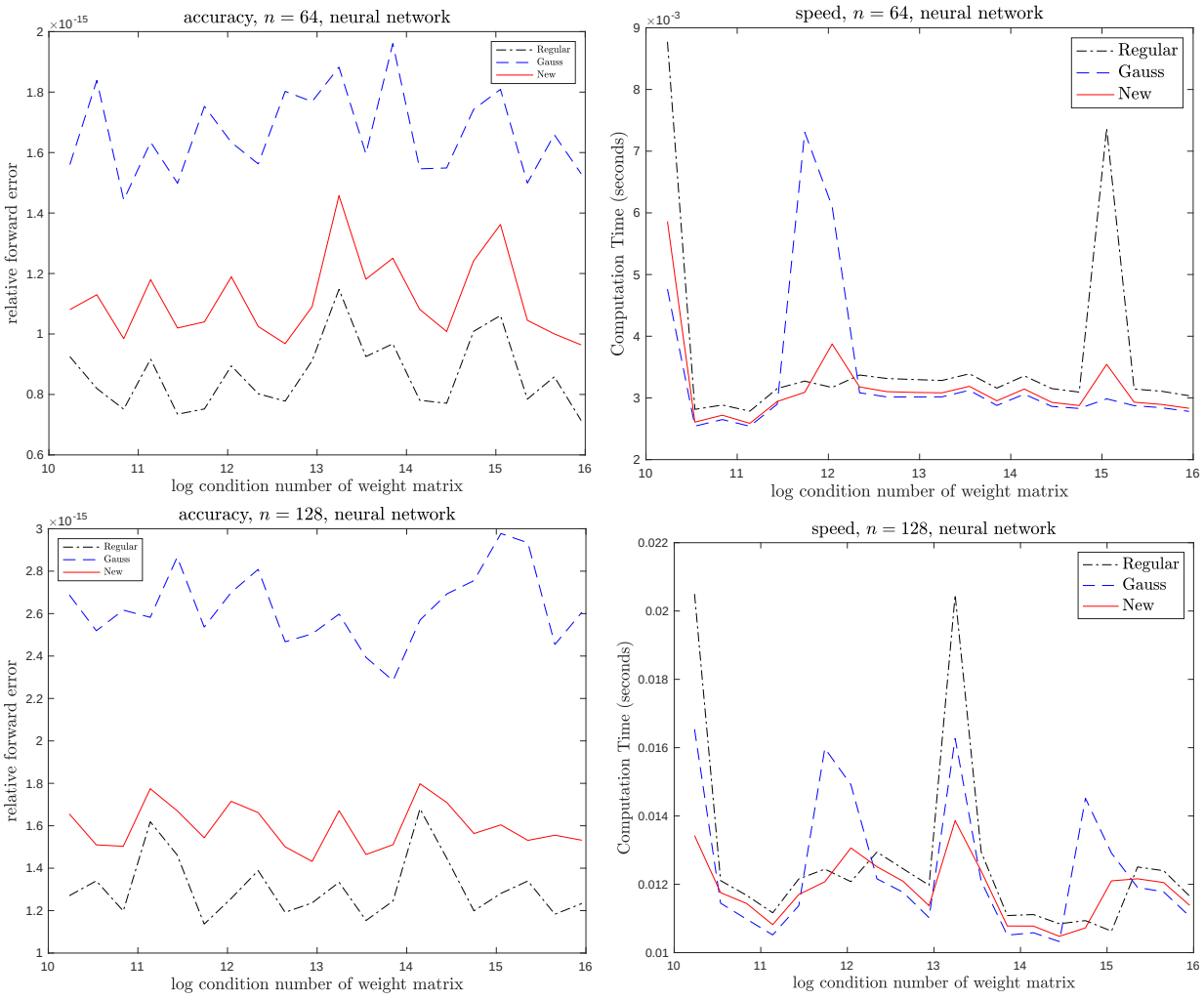


Figure 3.7: The three algorithms applied to 6-layer complex neural networks with complex ReLU activation and widths 64 and 128.

3.7 Conclusion

The notion of bilinear complexity started by Strassen has been a great motivator for more than five decades of exciting developments in numerical linear algebra. Its success illustrates the adage that "less is more". Bilinear complexity does not capture every operation that underlies the speed of an algorithm; but by focusing on a single operation (variable multiplications) and disregarding the rest (e.g., scalar multiplications, additions), it allows speed to be measured by the number of terms in a decomposition of a 3-tensor and the fastest algorithm to be given by a rank decomposition. This opens a door to other areas of mathematics like algebraic geometry where such decompositions are studied independent of their computational relevance.

We hope the notion of bilinear stability proposed in this article would do for the study of numerical stability what bilinear complexity did for the study of time complexity. By focusing on a single factor (growth) and disregarding other factors (e.g., cancellation errors) that play a role in numerical stability, it allows stability to be measured by the growth factor in a decomposition of a 3-tensor and the stablest algorithm to be given by a nuclear decomposition. Just as tensor rank connects to algebraic geometry, tensor nuclear norm connects to functional analysis [27, 29, 104]; thus bilinear stability could potentially open a door to this rich area of mathematics.

A very recent development in bilinear complexity is the automated discovery of fast algorithms using deep reinforcement learning. In [40], AlphaTensor found more than 14000 inequivalent 49-term decompositions for 4×4 matrix product. This is impressive. But when one has that many different algorithms the question becomes which one to pick? From the perspective of numerical linear algebra, numerical stability would be the most natural secondary criteria. Since the 14000 algorithms are all given in the form of 49-term decompositions, their growth factors are trivial to calculate and all one needs to do is to pick the decomposition with the smallest growth factor.

CHAPTER 4

INVERTING A COMPLEX MATRIX

4.1 Introduction

We analyze a complex matrix inversion algorithm mentioned in [15, 32, 35, 72, 82, 106, 109, 116, 127]. Given a complex matrix $A + iB \in \mathbb{C}^{n\times n}$ with $A, B \in \mathbb{R}^{n\times n}$, we compute its inverse by

$$Z^{-1} = \begin{cases} (A + BA^{-1}B)^{-1} - iA^{-1}B(A + BA^{-1}B)^{-1} & \text{if } A \text{ is invertible,} \\ B^{-1}A(AB^{-1}A + B)^{-1} - i(AB^{-1}A + B)^{-1} & \text{if } B \text{ is invertible.} \end{cases}$$

We call this algorithm "Frobenius inversion". Although this algorithm is known, its numerical properties have not been studied, as far as we know. In this paper, we give a thorough analysis of Frobenius inversion. We prove that Frobenius inversion is optimal in the sense of least number of matrix multiplications and inversions over $\mathbb{R}$. Then, we show that Frobenius inversion is faster than MATLAB's inversion algorithm. In addition, we provide several applications where Frobenius inversion is more efficient than MATLAB's inversion algorithm.

Matrix inversion is not preferred in most numerical linear algebra problems. For instance, when solving a system of linear equations $Ax = b$, where A is an n by n invertible matrix, we should not compute A^{-1} first and then multiplying it to b. Instead, we could use a LU factorization approach. First, we compute a LU factorization of $A = LU$. Then, we solve the systems $Ly = b$ and $Ux = y$ by forward substitution and backward substitution. This approach is both faster and more accurate than the one using matrix inversion [58]. In terms of speed, the LU factorization approach takes $2n^3/3$ flops while the matrix inversion approach takes $2n^3$ flops [58]. To compare the accuracy, let $\widehat{x}_{LU}$ and $\widehat{x}_{\text{inv}}$ be the computed values of x using LU factorization and matrix inversion respectively. Let $\widehat{L}$ and $\widehat{U}$ be the

computed LU factors of A. As shown in [58],

$$|b - A\hat{x}_{\text{inv}}| \leq n|A||A^{-1}||b|\mathsf{u} + O(\mathsf{u}^2) \quad \text{and} \quad |b - A\hat{x}_{LU}| \leq 3n|\widehat{L}||\widehat{U}||\hat{x}_{LU}|\mathsf{u} + O(\mathsf{u}^2),$$

where u is the unit roundoff, n is the dimension of b, and $|\cdot|$ is applied componentwise. Usually, $\||\widehat{L}||\widehat{U}|\|_\infty \approx \|A\|_\infty$ [58]. Thus, $\hat{x}_{LU}$ is likely to be much more accurate than $\hat{x}_{\text{inv}}$ when $\|x\|_\infty \ll \||A^{-1}||b|\|_\infty$ [58].

Although matrix inversion is generally not preferred in numerical linear algebra, it is necessary to do matrix inversion in certain circumstances. In some eigenvalue-related problems, matrix inversion is necessary in iterations of certain algorithms [3, 22, 60, 102]. For instance, Newton's iteration of solving Sylvester equation has the following update rule [102]:

$$X_{t+1} = \frac{1}{2}(X_t + X_t^{-1}), \quad t \in \mathbb{N},$$

where X_t is some matrix related to the inputs of Sylvester equation. We will discuss this in detail in section 4.7. In superconductivity computations, matrix inversion is needed in some numerical integrations [52]. Specifically, in the KKR-CPA algorithm, we need to compute the KKR matrix inverse and then integrate it over the first Brillouin zone [52]. In some MIMO radios, matrix inverse must be applied in hardware [31, 34, 115]. In statistics, sometimes matrix inverse reveals important statistical properties [6, 85, 87]. For instance, in linear regression, we want to learn a model $Y = X\hat{\beta}$, where X is the design matrix, Y are the observed values of the dependent variable, and $\hat{\beta}$ are the coefficients of least square regression [87]. Then, the covariance matrix of $\hat{\beta}$ is $\sigma^2 \cdot (X^\mathsf{T}X)^{-1}$, where σ^2 is the variance of the dependent variable [86]. Thus, we need to compute the inverse of $X^\mathsf{T}X$ in order to understand the statistical properties of $\hat{\beta}$. Moreover, if we want to estimate the accuracy of the prediction at a new point x, the variance of the predicted value $x^T\hat{\beta}$ is $\sigma^2 x^\mathsf{T}(X^\mathsf{T}X)^{-1}x$. In order to build a confidence interval for this new prediction, we need

to compute $\sigma^2 x^\mathsf{T}(X^\mathsf{T}X)^{-1}x$. Since we are interested in predicting the value of Y at many different points x, it is preferable to compute the inverse of $X^\mathsf{T}X$. In addition, by Cramer-Rao lower bound [25, 99], the inverse of the Fisher information matrix is an asymptotic lower bound for the covariance matrix of an unbiased estimator. In some Gaussian process, this lower bound could be attained [67]. Thus, we need to compute the inverse of Fisher information matrix in order to understand certain statistical problems. In this case, matrix inversion is unavoidable since the final result is the inverse of some matrix. In graph theory, the inverses of the adjacency matrix and forward adjacency matrix of a graph G reveal important combinatorial properties of G [89, 92, 95, 124]. For instance, the inverse of the adjacency matrix of G is the adjacency matrix of the inverse graph of G.

Furthermore, matrix inversion could even be beneficial in certain numerical linear algebra tasks. We first reproduce an example in [30]. Consider the scenario where we want to solve a sequence of linear systems $Ax^{(k)} = b^{(k)}$, for $k = 1, \ldots, K$, with the same coefficient matrix A. Suppose that $b^{(k+1)}$ depends on $b^{(k)}$ so that we have to solve the linear systems one by one. Then there are two natural approaches to solve this problem. One approach is to compute A^{-1} first and then multiply it to $b^{(k)}$ one at a time. Another approach is to compute a LU factorization of $A = LU$ and then solve $LUx^{(k)} = b^{(k)}$ by forward substitution and backward substitution for each $k = 1, \ldots, K$. When $K \gg n$, both algorithms takes roughly the same number of flops. However, the matrix inversion approach is faster than the LU factorization approach in practice due to some data structure considerations [30]. In terms of accuracy, the forward errors $\|\hat{x}^{(k)}_{\text{inv}} - x^{(k)}\|$ and $\|\hat{x}^{(k)}_{LU} - x^{(k)}\|$ are close to each other under mild conditions [31], where $\hat{x}^{(k)}_{\text{inv}}$ and $\hat{x}^{(k)}_{LU}$ are solutions of $Ax^{(k)} = b^{(k)}$ computed by matrix inversion and LU factorization respectively for each $k = 1, \ldots, K$. In addition, matrix inversion could also be beneficial for rank-one updates. Suppose that we have computed B^{-1} for some matrix B. Then, for any vectors u and v, we can compute the inverse of $A = B + uv^\mathsf{T}$ by the

reinforcement method [37]:

$$A^{-1} = B^{-1} - \frac{1}{1 + v^{\intercal} B^{-1} u} B^{-1} u v^{\intercal} B^{-1},$$

assuming that $1 + v^{\intercal} B^{-1} u \neq 0$. Besides this example, computing the inverse of a matrix A could also be beneficial if A has special structures [42, 66, 70]. For instance, if A is block circulant, then A is diagonalizable by some FFT type matrices, which makes it fast to compute the inverse of A [42].

4.1.1 Related work

Frobenius inversion was first discovered by Frobenius and Schur [116]. It was then derived from Frobenius-Schur's relation in [35, pp. 217–219]. Frobenius inversion was also derived in [82, pp. 137–138]. In [15], the authors proposed a way to invert $A + iB$ by inverting the real matrix $\begin{bmatrix} A & B \\ -B & A \end{bmatrix}$. In [32], the authors compared several ways to invert a complex matrix using real matrix inversions with directly inverting a complex matrix in complex arithmetic. They proposed that inverting a complex matrix in complex arithmetic is better than inverting a complex matrix using real inversions. The methods they considered include Frobenius inversion and the one proposed in [15]. However, in this paper, we show that Frobenius inversion could actually be advantageous in certain circumstances. In [72], the authors proposed a way to invert $A + iB$ using real matrix inversions when both A and B are singular. Their algorithm uses Frobenius inversion as an intermediate step. In [109], the authors proposed another way to invert $A + iB$ using real matrix inversions without assuming A, B, $A + B$ or $A - B$ is invertible. In [106, 127], Frobenius inversion was derived again.

4.2 Multiplications in Quadratic Field Extensions

Let $\Bbbk$ be a field and let $\mathbb{F}$ be a quadratic field extension of $\Bbbk$. Thus there exists some monic irreducible quadratic polynomial $f(x) \in \Bbbk[x]$ such that

$$\mathbb{F} \simeq \Bbbk[x]/(f(x)).$$

We explicitly write $f(x) = x^2 + \beta x + \tau$ for some $\beta, c \in \Bbbk$. Up to an isomorphism, we are able to write f in a normal form:

- $\mathrm{char}(\Bbbk) \neq 2$: $\beta = 0$ and $-\tau$ is not a complete square in $\Bbbk$.
- $\mathrm{char}(\Bbbk) = 2$: either $\beta = 0$ and $-\tau$ is not a complete square in $\Bbbk$, or $\beta = 1$ and $x^2+x+\tau$ has no solution in $\Bbbk$.

Let ξ be a root of $f(x)$ in $\overline{\Bbbk}$. We also have $\mathbb{F} \simeq \Bbbk[\xi]$. In particular, an element in $\mathbb{F}$ can be written as $a_1 + a_2\xi$. The multiplicaiton on $\mathbb{F}$ is given by the lemma that follows.

Lemma 4.2.1. Let $\Bbbk, f, \xi$ be as above. Given two elements $x_1 + x_2\xi, y_1 + y_2\xi$ in $\Bbbk[\xi]$, their product is

$$(x_1 + x_2\xi)(y_1 + y_2\xi) = \begin{cases} (x_1y_1 - \tau x_2y_2) + (x_1y_2 + x_2y_1)\xi, \text{ if } f(x) = x^2 + \tau, \\ (x_1y_1 - cx_2y_2) + (x_1y_2 + x_2y_1 - x_2y_2)\xi, \text{ if } f(x) = x^2 + x + \tau. \end{cases} \tag{4.2.1}$$

Proposition 4.2.2. Let $\Bbbk, f, \tau, \xi$ be as above. There exists an algorithm for the multiplication in $\Bbbk[\xi]$, which costs three multiplications in $\Bbbk$. Moreover, the algorithm is optimal in the sense of bilinear complexity, i.e., the minimum number of multiplications in $\Bbbk$.

Proof. We first assume that $f(x) = x^2+\tau$. In this case, the multiplication $(x_1+x_2\xi)(y_1+y_2\xi)$ can be computed by three multiplications: $M_1 = (x_1-x_2)(y_1+\tau y_2), M_2 = x_1y_2, M_3 = x_2y_1$,

since

$$x_1y_1 - \tau x_2y_2 = M_1 - \tau M_2 + M_3, \quad x_1y_2 + x_2y_1 = M_2 + M_3. \tag{4.2.2}$$

Next we assume that $f(x) = x^2 + x + \tau$. We consider: $M_1 = x_1y_1$, $M_2 = x_2y_2$ and $M_3 = (x_1 - x_2)(y_1 - y_2)$. It is straightforward to verify that

$$x_1y_1 - cx_2y_2 = M_1 - cM_2, \quad x_1y_2 + x_2y_1 - x_2y_2 = M_1 - M_3. \tag{4.2.3}$$

Now we assume that in both cases there exists an algorithm for the multiplication in $\Bbbk[\xi]$, which costs two multiplications M_1' and M_2'. Then we have

$$x_1y_1 - \tau x_2y_2, x_1y_2 + x_2y_1 - \delta x_2y_2 \in \operatorname{span}\{M_1', M_2'\},$$

where $\delta = 0$ if $f(x) = x^2 + \tau$ and $\delta = 1$ if $f(x) = x^2 + x + \tau$. Clearly $x_1y_1 - \tau x_2y_2$ and $x_1y_2 + x_2y_1 - \delta x_2y_2$ are not collinear, thus

$$M_1', M_2' \in \operatorname{span}\{x_1y_1 - \tau x_2y_2, x_1y_2 + x_2y_1 - \delta x_2y_2\}.$$

Therefore, we may find constants $a, b, d, e \in \Bbbk$ such that

$$\begin{aligned} M_1' &= a(x_1y_1 - \tau x_2y_2) + b(x_1y_2 + x_2y_1 - \delta x_2y_2) \\ &= ax_1y_1 + bx_1y_2 + bx_2y_1 + (-\tau a - \delta b)x_2y_2, \\ M_2' &= c(x_1y_1 - \tau x_2y_2) + d(x_1y_2 + x_2y_1 - \delta x_2y_2) \\ &= cx_1y_1 + dx_1y_2 + dx_2y_1 + (-\tau c - \delta d)x_2y_2. \end{aligned}$$

Clearly, $ad - bc \neq 0$ and in particular, at least one of a, b, c, d is nonzero. We observe that

M_1' is a multiplication, thus it can be written as $(rx_1 + sx_2)(uy_1 + vy_2)$. This implies that

$$a(-\tau a - \delta b) = b^2, \tag{4.2.4}$$

$$c(-\tau c - \delta d) = d^2. \tag{4.2.5}$$

If $f(x) = x^2 + \tau$, then (4.2.4) reduces to $\tau a^2 + b^2 = 0$. This implies that $a = b = 0$ since $-\tau$ is not a complete square in $\Bbbk$. Similarly, we also have $c = d = 0$. This contradicts the assumption that at least one of a, b, c, d is nonzero.

If $f(x) = x^2 + x + \tau$ then (4.2.4) is $\tau a^2 + ab + b^2 = 0$. We remark that $a \neq 0$. Otherwise, we may derive $b = 0$ which contradicts the fact that $ad - bc \neq 0$. By the substitution $b' = b/a$, we have $b'^2 + b' + \tau = 0$. This contradicts to the assumption that the equation $x^2 + x + \tau = 0$ has no solution in $\Bbbk$. □

It is worthy to remark that we may regard the multiplication in $\Bbbk[\xi]$ as the bilinear map

$$m : \Bbbk[\xi] \times \Bbbk[\xi] \to \Bbbk[\xi], \quad (x_1 + x_2\xi, y_1 + y_2\xi) \mapsto (x_1 + x_2\xi)(y_1 + y_2\xi).$$

A multiplication in $\Bbbk$ means the multiplication between two indeterminates from different inputs. For example, we count x_1y_1 as one multiplication while we do not count τx_1 as a multiplication since τ is a constant. Moreover, we do not allow multiplications between indeterminates from the same input, such as x_1x_2. In terms of tensor rank, Proposition 4.2.2 means the rank of the structure tensor of m is exactly three. According to [46, 74], it is known that every tensor in $\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2$ has rank at most three. As a comparison, Proposition 4.2.2 deals with a particular tensor over an arbitrary field $\Bbbk$. More importantly, it determines the exact value of the rank of that tensor, together with a rank decomposition. An important special case of Proposition 4.2.2 is $\Bbbk = \mathbb{R}$ and $f(x) = x^2 + 1$. In this case, $\xi = i$ and $\Bbbk[\xi] = \mathbb{C}$. The algorithm presented in (4.2.2) is the celebrated Gauss algorithm [69] for the multiplication of complex numbers, whose optimality is proved in [90, 119].

4.3 Gauss Matrix Multiplication

Let n be a positive integer and let $\mathbb{L}$ be a field. We denote by $M_n(\mathbb{L})$ the $\mathbb{L}$-algebra consisting of $n \times n$ matrices over $\mathbb{L}$. We denote by $\mathrm{GL}_n(\mathbb{L})$ the group of invertible $n \times n$ matrices over $\mathbb{L}$ and we consider the multiplication map

$$m_{n,\mathbb{L}} : \mathrm{M}_n(\mathbb{L}) \times \mathrm{M}_n(\mathbb{L}) \to \mathrm{M}_n(\mathbb{L}), \quad m_{n,\mathbb{L}}(Z, W) = ZW$$

and the inversion map

$$\mathrm{inv}_{n,\mathbb{L}} : \mathrm{GL}_n(\mathbb{L}) \to \mathrm{GL}_n(\mathbb{L}), \quad \mathrm{inv}_{n,\mathbb{L}}(Z) = Z^{-1}.$$

Assume that $\mathbb{F} \simeq \Bbbk[x]/(f(x)) \simeq \Bbbk[\xi]$ is a quadratic field extension of $\Bbbk$, where ξ is a root of $f \in \Bbbk[x]$ in $\overline{\Bbbk}$. Here $f(x)$ is either $x^2+\tau$ or $x^2+x+\tau$. We notice that $M_n(\mathbb{F}) = M_n(\Bbbk) \otimes_{\Bbbk} \mathbb{F}$. Thus an element in $Z \in M_n(\mathbb{F})$ can be written as $Z = A + \xi B$ where $A, B \in M_n(\Bbbk)$. As a consequence of Proposition 4.2.2, we have the following.

Proposition 4.3.1 (Gauss matrix multiplication)**.** Let $\Bbbk, \mathbb{F}, n, f, \tau, \xi$ be as above. Suppose that $Z = A + \xi B$ and $W = C + \xi D$ are two elements in $M_n(\mathbb{F})$ where $A, B, C, D \in M_n(\Bbbk)$. If $f(x) = x^2 + \tau$, then one can compute the product ZW by

$$\begin{aligned} M_1 &= (A - B)(C + \tau D), \quad M_2 = AD, \quad , M_3 = BC, \\ N_1 &= M_1 - \tau M_2 + M_3, \quad N_2 = M_2 + M_3, \\ ZW &= N_1 + \xi N_2. \end{aligned} \tag{4.3.1}$$

If $f(x) = x^2 + x + \tau$, then one can compute the product ZW by

$$\begin{aligned} M_1 = AC, \quad M_2 = BD, \quad M_3 = (A-B)(C-D), \\ N_1 = M_1 - \tau M_2, \quad N_2 = M_1 - M_3, \\ ZW = N_1 + \xi N_2. \end{aligned} \tag{4.3.2}$$

Moreover, (4.3.1) and (4.3.2) are optimal in the sense of minimum number of multiplications in $M_n(\Bbbk)$.

We remind the readers that if we denote by $\mathcal{E}$ the collection of algorithms for $m_{n,\mathbb{F}}$ using multiplications and additions of indeterminates, together with scalar multiplications in $M_n(\Bbbk)$, then the algorithm presented in Proposition 4.3.1 is an minimal element with respect to the partial ordering on $\mathcal{E}$ induced by the number of multiplications of indeterminates in $M_n(\Bbbk)$.

Proof. It is straightforward to verify that algorithms in (4.3.1) and (4.3.2) indeed compute ZW. To see the optimality, one may repeat the argument in the proof of Proposition 4.2.2. □

4.4 Frobenius Matrix Inversion

In this section, we discuss the matrix inversion on $M_n(\mathbb{F})$. To do that, we observe that for $Z = A + \xi B, W = C + \xi D \in \mathrm{GL}_n(\mathbb{F})$ where $A, B, C, D \in M_n(\Bbbk)$, W is the inverse of Z if and only if

$$(A + \xi B)(C + \xi D) = ZW = I_n. \tag{4.4.1}$$

Thus we may have two equations for C and D. According to Lemma 4.2.1, the multiplication formula in $\mathbb{F}$ thus in $M_n(\mathbb{F})$ depends on the form of f. In the following, we separate our discussions with respect to the two normal forms of f.

4.4.1 *First case:* $f(x) = x^2 + \tau$

We define

$$\mathcal{S}_1 \coloneqq \{Z = A + \xi B \in \mathrm{GL}_n(\mathbb{F}) : A, A + \tau BA^{-1}B \in \mathrm{GL}_n(\Bbbk)\},$$
$$\mathcal{S}_2 \coloneqq \{Z = A + \xi B \in \mathrm{GL}_n(\mathbb{F}) : B, \tau B + AB^{-1}A \in \mathrm{GL}_n(\Bbbk)\}.$$

It is worth mentioning that the complement of $\mathcal{S}_1 \cup \mathcal{S}_2$ consists of $n \times n$ $\mathbb{F}$-matrices $Z = A + \xi B$ where $\det\Big(A(A + \tau BA^{-1}B)\Big) = \det\Big(B(\tau B + AB^{-1}A)\Big) = 0$. Therefore, $\mathcal{S}_1 \cup \mathcal{S}_2$ is an open dense (in Zariski topology) subset of $\mathbb{F}^{n\times n} \simeq \Bbbk^{2\times n\times n}$. Thus with respect to any reasonable probability measure on $\Bbbk^{2\times n\times n}$, a random complex matrix Z lies in $\mathcal{S}_1 \cup \mathcal{S}_2$ with probability one. The restriction of $\mathrm{inv}_{n,\mathbb{F}}$ on $\mathcal{S}_1$ and $\mathcal{S}_2$ can be easily expressed in terms of $\mathrm{inv}_{n,\Bbbk}$ and $m_{n,\Bbbk}$. Indeed, according to (4.4.1), we have

$$AC - \tau BD = I_n, \quad AD + BC = 0.$$

Solving the equation for C and D, we obtain the lemma that follows.

Lemma 4.4.1 (Frobenius inversion I)**.** Let $\mathbb{F} = \Bbbk[x]/(f(x))$ where $f(x) = x^2 + \tau$ is an irreducible polynomial over $\Bbbk$. For each $Z = A + \xi B \in \mathrm{GL}_n(\mathbb{F})$, where $A, B \in \mathrm{M}_n(\Bbbk)$, we have

(a) If $Z \in \mathcal{S}_1$, then

$$Z^{-1} = (A + \tau BA^{-1}B)^{-1} - \xi A^{-1}B(A + \tau BA^{-1}B)^{-1}. \tag{4.4.2}$$

(b) If $Z \in \mathcal{S}_2$, then

$$Z^{-1} = B^{-1}A(AB^{-1}A + \tau B)^{-1} - \xi(AB^{-1}A + \tau B)^{-1}. \tag{4.4.3}$$

In particular, $\mathrm{inv}_{n,\mathbb{F}}$ on $\mathcal{S}_1 \cup \mathcal{S}_2$ can be evaluated by performing $\mathrm{inv}_{n,\Bbbk}$ twice and $m_{n,\Bbbk}$ thrice.

We remark that if $\Bbbk = \mathbb{R}, \mathbb{F} = \mathbb{C}$, then (4.4.2) and (4.4.3) are the well-known inversion formulae for complex matrices first discovered by Frobenius [15, 32, 35, 72, 82, 106, 109, 116, 127]. It is easy to turn (4.4.2) and (4.4.3) into Algorithm 2. Note that Algorithm 2 reduces

Algorithm 2 Frobenius inversion I

Input $Z = A + \xi B \in \mathcal{S}_1 \cup \mathcal{S}_2$
Output inverse of Z
if $Z \in \mathcal{S}_1$ **then**
    set $X = A, Y = B, \tau_1 = 1, \tau_2 = \tau$;
else if $Z \in \mathcal{S}_2$ **then**
    set $X = B, Y = A, \tau_1 = \tau, \tau_2 = 1$;
end if
compute X^{-1};
compute $X^{-1}Y$;
compute $YX^{-1}Y$;
compute $\tau_1 X + \tau_2 YX^{-1}Y$;
compute $J = (\tau_1 X + \tau_2 YX^{-1}Y)^{-1}$;
compute $K = X^{-1}Y(\tau_1 X + \tau_2 YX^{-1}Y)^{-1}$;
if $Z \in \mathcal{S}_1$ **then return** $Z^{-1} = J - \xi K$;
else if $Z \in \mathcal{S}_2$ **then return** $Z^{-1} = K - \xi J$;
end if

to the algorithm in [21, p. 15] and [77] when $n = 1$, $\Bbbk = \mathbb{R}$ and $\mathbb{F} = \mathbb{C}$.

One may expect that (4.4.2) and (4.4.3) are consequences of any of the Sherman–Morrison–Woodbury (SMW) identities [49, 51, 53, 58, 101, 122, 123]:

$$\begin{aligned}(A+B)^{-1} &= A^{-1} - A^{-1}(B^{-1}+A^{-1})^{-1}A^{-1} \\ &= A^{-1} - A^{-1}\left(AB^{-1}+I_n\right)^{-1} \\ &= A^{-1} - \left(A+AB^{-1}A\right)^{-1} \\ &= A^{-1} - A^{-1}B\left(A+B\right)^{-1}.\end{aligned}$$

However, this is not the case since SMW identities all involve the inversion of a matrix over $\mathbb{F}$ while (4.4.2) and (4.4.3) only require inversions of matrices over $\Bbbk$.

From the algebraic perspective, $\mathrm{inv}_{n,\mathbb{F}}$ is an operation on $M_n(\mathbb{F})$, which is an $M_n(\Bbbk)$-bimodule. Hence it is natural to discuss the computational complexity of $\mathrm{inv}_{n,\mathbb{F}}$ over $M_n(\Bbbk)$. We observe that Algorithm 2 performs $m_{n,\Bbbk}$ three times, $\mathrm{inv}_{n,\Bbbk}$ twice and real matrix addition once. In the following we prove the optimality of Algorithm 2.

Theorem 4.4.2 (optimality I)**.** Let $n \geq 2$ be a positive integer. Algorithm 2 is optimal in the sense of least number of multiplications, inversions and additions in $M_n(\Bbbk)$.

Proof. It is clear that one matrix addition over $\Bbbk$ is necessary to compute $\mathrm{inv}_{n,\mathbb{F}}$. Let Γ be a straight-line program computing Z^{-1} for $Z \in \mathcal{S}_1 \cup \mathcal{S}_2$ over the algebra $\mathrm{M}_n(\Bbbk)$. We denote by $L(\Gamma)$ the total number of inversions and multiplications in $\mathrm{M}_n(\Bbbk)$ performed by Γ. If we denote by Γ_0 the straight-line program for Algorithm 2, then it is clear that $L(\Gamma_0) = 5$. We assume that there exists some straight-line program Γ_1 with $L(\Gamma_1) = 4$ computing Z^{-1}.

We first prove that Γ_1 performs $\mathrm{inv}_{n,\Bbbk}$ at least twice. Otherwise, Γ_1 can compute Z^{-1} by only one $\Bbbk$-matrix inversion. Without loss of generality, we may assume that $Z \in \mathcal{S}_1$ so that $(A + \tau BA^{-1}B)^{-1}$ can be computed by at most one $\Bbbk$-matrix inversion. Furthermore, we may also assume that B is invertible. Now we have

$$A(A + \tau BA^{-1}B)^{-1} = (I_n + \tau(BA^{-1})^2)^{-1}.$$

We consider the case where $B = \Lambda A$ where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ is a diagonal matrix. Then $(I_n + \tau(BA^{-1})^2)^{-1}$ is the diagonal matrix whose diagonal elements are $1/(1 + \tau\lambda_j^2), j = 1, \ldots, n$. This indicates that Γ_1 requires at least two inversions in $M_n(\Bbbk)$ to compute $(I_n + \tau(BA^{-1})^2)^{-1}$, since A, B are not necessarily commuting.

Next, we prove that Γ_1 requires at least three $\Bbbk$-matrix multiplications. Again, we assume that $Z = A + \xi B \in \mathcal{S}_1$. Then the output of Γ_1 is the pair

$$\left((A + \tau BA^{-1}B)^{-1}, \quad A^{-1}B(A + \tau BA^{-1}B)^{-1}\right).$$

We proceed by contradiction. Suppose that Γ_1 costs at most two $\Bbbk$-matrix multiplications. Since Γ_1 is a straight-line program, we may assume without loss of generality, that Γ_1 first computes $(A+\tau BA^{-1}B)^{-1}$ and then it computes $A^{-1}B(A+\tau BA^{-1}B)^{-1}$. We claim that Γ_1 already requires two $\Bbbk$-matrix multiplications to compute $(A+\tau BA^{-1}B)^{-1}$. To that end, we notice that

$$BA^{-1}B = \tau^{-1}\left(\left((A+\tau BA^{-1}B)^{-1}\right)^{-1} - A\right).$$

This implies that one can compute $BA^{-1}B$ using the same number of $\Bbbk$-matrix multiplications as $(A+\tau BA^{-1}B)^{-1}$. However, it is clear that computing $BA^{-1}B$ by one multiplication is impossible and this proves our claim. Thus Γ_1 computes $A^{-1}B(A+\tau BA^{-1}B)^{-1}$ as a linear combination of $(A+\tau BA^{-1}B)^{-1}$ and intermediate outputs. However, this is not possible. Indeed, we may again consider the case where $B=\Lambda A$ and $\Lambda = \mathrm{diag}(\lambda_1,\ldots,\lambda_n)$ is a diagonal matrix. Therefore, we have

$$\Big((A+\tau BA^{-1}B)^{-1},\quad A^{-1}B(A+\tau BA^{-1}B)^{-1}\Big) = (A^{-1}(I_n+\tau\Lambda^2)^{-1}, A^{-1}(I_n+\tau\Lambda^2)^{-1}\Lambda).$$

Moreover, $A^{-1}(I_n+\tau\Lambda^2)^{-1}(I_n-\Lambda)$ can be computed as a linear combination of intermediate outputs. In particular, one can compute both $A^{-1}(I_n+\tau\Lambda^2)^{-1}$ and $A^{-1}(I_n+\tau\Lambda^2)^{-1}\Lambda$ by two multiplications in $M_n(\Bbbk)$, which is ridiculous. □

4.4.2 Second case: $f(x) = x^2+x+\tau$

We define

$$\mathcal{T}_1 \coloneqq \left\{Z = A+\xi B \in M_n(\mathbb{F}) : A,B\in\Bbbk, A-B, A+\tau B(A-B)^{-1}B \in \mathrm{GL}_n(\Bbbk)\right\},$$
$$\mathcal{T}_2 \coloneqq \left\{Z = A+\xi B \in M_n(\mathbb{F}) : A,B\in\Bbbk, B, AB^{-1}A-A+\tau B \in \mathrm{GL}_n(\Bbbk)\right\}.$$

Again, $\mathcal{S}_1 \cup \mathcal{S}_2$ is an open dense subset of $\mathbb{F}^{n\times n} \simeq \Bbbk^{2\times n\times n}$ in Zariski topology. According to (4.4.1), we have

$$AC - \tau BD = I_n, \quad AD + BC - BD = 0.$$

Solving the equation for C and D, we have the following

Lemma 4.4.3 (Frobenius inversion II)**.** Let $\Bbbk$ be a field of characteristic 2 and let $\mathbb{F} = \Bbbk[x]/(f(x))$ where $f(x) = x^2 + x + \tau$ is an irreducible polynomial over $\Bbbk$. For each $Z = A + \xi B \in \mathrm{GL}_n(\mathbb{F})$ where $A, B \in M_n(\Bbbk)$, we have

(a) If $Z \in \mathcal{T}_1$, then

$$Z^{-1} = (A + \tau B(A-B)^{-1}B)^{-1} - \xi(A-B)^{-1}B(A + \tau B(A-B)^{-1}B)^{-1}. \tag{4.4.4}$$

(b) If $Z \in \mathcal{T}_2$, then

$$Z^{-1} = (B^{-1}A - I_n)(AB^{-1}A - A + \tau B)^{-1} - \xi(AB^{-1}A - A + \tau B)^{-1}. \tag{4.4.5}$$

As a direct consequence of (4.4.4) and (4.4.5), we obtain Algorithm 3. It is straightforward to verify that Algorithm 3 costs two matrix inversions and three matrix multiplications over $\Bbbk$. In fact, by a similar argument for Theorem 4.4.2, we can prove that Algorithm 3 is optimal.

Theorem 4.4.4 (optimality II)**.** Let $n \geq 2$ be a positive integer. Algorithm 3 is optimal in the sense of least number of multiplications, inversions and additions in $M_n(\Bbbk)$.

Algorithm 3 Frobenius inversion II

Input $Z = A + \xi B \in \mathcal{T}_1 \cup \mathcal{T}_2$
Output inverse of Z
1: **if** $Z \in \mathcal{T}_1$ **then**
2: compute $X_1 = (A - B)^{-1}$;
3: compute $X_2 = X_1 B$;
4: compute $X_3 = A + \tau B X_2$;
5: compute $X_4 = X_3^{-1}$;
6: compute $X_5 = X_2 X_4$; **return** $Z^{-1} = X_4 - \xi X_5$
7: **else if** $Z \in \mathcal{T}_2$ **then**
8: compute $X_1 = B^{-1}$;
9: compute $X_2 = X_1 A$;
10: compute $X_3 = A X_2 - A + \tau B$;
11: compute $X_4 = X_3^{-1}$;
12: compute $X_5 = (X_2 - I_n) X_4$; **return** $Z^{-1} = X_5 - \xi X_4$
13: **end if**

4.4.3 An application

To conclude this section, we discuss an application of Algorithms 2 and 3. We consider a tower of quadratic field extensions:

$$\Bbbk = \mathbb{K}_0 \subsetneq \mathbb{K}_1 \subsetneq \cdots \subsetneq \mathbb{K}_m, \tag{4.4.6}$$

where $[\mathbb{K}_j : \mathbb{K}_{j-1}] = 2, j = 1, \ldots, m$. For each $j = 1, \ldots, m$ there exists some $\xi_j \in \mathbb{K}_j$ such that $\mathbb{K}_j = \mathbb{K}_{j-1}[\xi_j]$. We denote by $f_j \in \Bbbk[x]$ the minimal polynomial of ξ_j. Then f_j is a monic irreducible quadratic polynomial. We may also assume that f_j is in the normal form:

$$f_j(x) = x^2 + \tau_j \text{ or } f_j(x) = x^2 + x + \tau_j, \quad j = 1, \ldots, m.$$

It is obvious that $[\mathbb{K}_m : \Bbbk] = 2^m$ and an element x in $\mathbb{K}_m$ can be written as

$$x = \sum_{\alpha \in \{0,1\}^m} c_\alpha \xi^\alpha,$$

where for each $\alpha = (\alpha_1, \ldots, \alpha_m) \in \{0,1\}^m$, $\xi^\alpha = \xi_1^{\alpha_1} \cdots \xi_m^{\alpha_m}$ and $c_\alpha \in \Bbbk$. Moreover, we may regard $\mathbb{K}_m$ as a quotient ring of the polynomial ring in m variables, i.e.,

$$\mathbb{K}_m \simeq \Bbbk[x_1, \ldots, x_m]/(f_1, \ldots, f_m) = \bigotimes_{j=1}^{m} \Big(\Bbbk[x]/(f_j) \Big). \tag{4.4.7}$$

A particularly interesting example of $\mathbb{K}_m$ is $\mathbb{Q}[S]$ where $S = \{\sqrt{a_j} : a_j \in \mathbb{Q}, 1 \leq j \leq m\}$. It is proved in [12] that

$$\mathbb{Q} \subsetneq \mathbb{Q}[\sqrt{a_1}] \subsetneq \mathbb{Q}[\sqrt{a_1}, \sqrt{a_2}] \subsetneq \cdots \subsetneq \mathbb{Q}[S]$$

is a tower of quadratic field extensions if the product of any nonempty subset of S is not in $\mathbb{Q}$. In this case, we have $\mathbb{K}_j = \mathbb{Q}[\sqrt{a}_1, \ldots, \sqrt{a}_j]$ and $f_j(x) = x^2 - a_j, j = 1, \ldots, m$. Another commonly seen example of the tower (4.4.6) is

$$\mathbb{Q} \subsetneq \mathbb{Q}[a^{\frac{1}{2}}] \subsetneq \cdots \subsetneq \mathbb{Q}[a^{\frac{1}{2^m}}],$$

where $a \in \mathbb{Q}$ is not a complete square.

We also remark that (4.4.6) is analogous to the tower of multicomplex number systems [98]:

$$\mathbb{R} \subsetneq \mathbb{C}_1 \subsetneq \cdots \subsetneq \mathbb{C}_m,$$

where $\mathbb{C}_j, 1 \leq j \leq m$ is defined inductively as

$$\begin{aligned} \mathbb{C}_1 &= \mathbb{C}, \\ \mathbb{C}_{k+1} &= \{a + b i_{k+1} : a, b \in \mathbb{C}_k\}, \quad k \geq 1 \end{aligned}$$

where $i_{k+1}^2 = -1$ and $i_k i_l = i_l i_k$ for any $k, l \geq 1$. However, since $\mathbb{C}_1 = \mathbb{C}$ is algebraic closed, $\mathbb{C}_j$ is not even a field extension of $\mathbb{C}_1$ for $j \geq 2$. We refer interested readers to [107, 98] for

detailed discussions of analytic properties of $\mathbb{C}_m$.

We observe that $M_n(\mathbb{K}_m) = M_n(\Bbbk) \otimes_{\Bbbk} \mathbb{K}_m$, which implies that a matrix $Z \in M_n(\mathbb{K}_m)$ can be written as

$$Z = \sum_{\alpha \in \{0,1\}^m} C_\alpha \xi^\alpha. \tag{4.4.8}$$

On the other hand, we also have

$$M_n(\Bbbk) \otimes_{\Bbbk} \mathbb{K}_m = M_n(\Bbbk) \otimes_{\Bbbk} \mathbb{K}_1 \otimes_{\mathbb{K}_1} \cdots \otimes_{\mathbb{K}_{m-1}} \mathbb{K}_m.$$

Thus $Z \in \mathbb{K}_m$ can also be inductively constructed as follows:

$$\begin{aligned} Z &= A_0 + \xi_m A_1, \\ A_\beta &= A_{0,\beta} + \xi_{m-j} A_{1,\beta}, \ \ \beta \in \{0,1\}^j, \ \ j = 1, \ldots, m-1, \end{aligned} \tag{4.4.9}$$

where $A_\beta \in \mathrm{M}_n(\mathbb{K}_{m-|\beta|})$. We record in the next lemma the relation between the two expressions of Z in (4.4.8) and (4.4.9).

Lemma 4.4.5. Let C_α and A_β be as in (4.4.8) and (4.4.9) respectively, where $\alpha \in \{0,1\}^m$ and $\beta \in \bigcup_{j=1}^m \{0,1\}^j$. For each $1 \le j \le m$,

$$Z = \sum_{\beta \in \{0,1\}^j} A_\beta \xi_{m-j+1}^{\beta_1} \cdots \xi_m^{\beta_j}.$$

Moreover, for each $\beta \in \{0,1\}^j$, we have

$$A_\beta = \sum_{\beta' \in \{0,1\}^{m-j}} C_{\beta',\beta} \xi_1^{\beta'_1} \cdots \xi_{m-j}^{\beta'_{m-j}}, \quad \beta' \in \{0,1\}^{m-j}.$$

In particular, $C_\alpha = A_\alpha$.

Proof. We proceed by induction on j. Clearly the formula holds for $j = 1$ by (4.4.9). Assume

that the formula holds for $j = s$, i.e.,

$$Z = \sum_{\beta \in \{0,1\}^s} A_\beta \xi_{m-s+1}^{\beta_1} \cdots \xi_m^{\beta_s}.$$

To prove the formula for $j = s+1$, we notice that $A_\beta = A_{0,\beta} + \xi_{m-s} A_{1,\beta}$, thus

$$Z = \sum_{\beta \in \{0,1\}^s} (A_{0,\beta} + \xi_{m-s} A_{1,\beta}) \xi_{m-s+1}^{\beta_1} \cdots \xi_m^{\beta_s} = \sum_{\gamma \in \{0,1\}^{s+1}} A_\gamma \xi_{m-s}^{\gamma_1} \cdots \xi_m^{\gamma_{s+1}}$$

and this completes the induction. The moreover part follows easily by comparing (4.4.9) with (4.4.8). □

Lemma 4.4.5 provides us a method to compute the multiplication and inversion in $M_n(\mathbb{K}_m)$.

Proposition 4.4.6 (multiplication in $M_n(\mathbb{K}_m)$)**.** There exists an algorithm for the multiplication in $M_n(\mathbb{K}_m)$, which costs 3^m multiplications in $M_n(\Bbbk)$.

Proof. Let $Z, W \in M_n(\mathbb{K}_m)$. According to (4.4.9), we may write

$$Z = A_0 + \xi_m A_1, \quad W = B_0 + \xi_m B_1.$$

Thus one can compute ZW in terms of A_0, A_1, B_0, B_1 by three multiplications in $M_n(\mathbb{K}_{m-1})$ by Proposition 4.3.1. Each multiplication in $M_n(\mathbb{K}_{m-1})$ costs three multiplications in $M_n(\mathbb{K}_{m-2})$ again by Proposition 4.3.1. Repeat the above process until we arrive at multiplications in $M_n(\mathbb{K}_0) = M_n(\Bbbk)$. Thus the total cost of multiplications in $M_n(\Bbbk)$ is 3^m. □

According to (4.4.8), we can also write Z and W as $Z = \sum_{\alpha \in \{0,1\}^m} C_\alpha \xi^\alpha$ and $W = \sum_{\beta \in \{0,1\}^m} D_\beta \xi^\alpha$ respectively. Here $C_\alpha, D_\beta \in M_n(\Bbbk)$ for $\alpha, \beta \in \{0,1\}^m$. Thus one can

compute ZW via the formula

$$ZW = \sum_{\gamma \in \{0,2\}^m} \left(\sum_{\substack{\alpha+\beta=\gamma \\ \alpha,\beta \in \{0,1\}^m}} C_\alpha D_\beta \right) \xi^\gamma.$$

However, it is obvious that the above formula costs 4^m multiplications in $M_n(\Bbbk)$. Hence the algorithm presented in Proposition 4.4.6 reduces the complexity of evaluating $m_{n,\mathbb{K}_m}$ over $M_n(\Bbbk)$ from $O(N^2)$ to $O(N^{\log_2 3})$, where $N = 2^m$.

Due to Proposition 4.3.1, one can recognize the algorithm in Proposition 4.4.6 as an analogue of the Karatsuba algorithm [64] for fast integer multiplication. By (4.4.7), we may also regard the algorithm in Proposition 4.4.6 as an analogue of the multidimensional fast Fourier transform [108].

We also remark that the algorithm presented in Proposition 4.4.6 relies on the technique called divide and conquer, which is employed by Strassen to design the first algorithm [111] for $n \times n$ matrix multiplication whose complexity is smaller than $O(n^3)$. In our case, however, the size n of matrices is fixed, while the level m of the tower (4.4.6) varies.

Proposition 4.4.7 (inversion in $M_n(\mathbb{K}_m)$)**.** There exists an algorithm for the inversion of a generic element in $M_n(\mathbb{K}_m)$, which costs $3(3^m - 2^m)$ multiplications and 2^m inversions in $M_n(\Bbbk)$.

Proof. For $1 \leq j \leq m$ and a generic $Z \in M_n(\mathbb{K}_j)$. We write $Z = A_0 + \xi_m A_1$ where $A_0, A_1 \in M_n(\mathbb{K}_{j-1})$. Since Z is generic, Algorithm 2 (resp. Algorithm 3) is applicable for $f_j(x) = x^2 + \tau_j$ (resp. $f_j(x) = x^2 + x + \tau_j$). This costs three multiplications and two inversions in $M_n(\mathbb{K}_{j-1})$. Thus we have

$$\mathrm{inv}_{n,\mathbb{K}_j} = 3m_{n,\mathbb{K}_{j-1}} + 2\,\mathrm{inv}_{n,\mathbb{K}_{j-1}}, 1 \leq j \leq m.$$

Moreover, Proposition 4.3.1 implies $m_{n,\mathbb{K}_j} = 3m_{n,\mathbb{K}_{j-1}}$. Inductively, we may derive that

$$\mathrm{inv}_{n,\mathbb{K}_m} = 3(3^m - 2^m)m_{n,\mathbb{k}} + 2^m \,\mathrm{inv}_{n,\mathbb{k}}\,.$$

□

4.5 General Matrix Inversion

In Section 4.4, we investigate properties of the Frobenius inversion from the symbolic perspective. In practice, the most important quadratic field extension is $\mathbb{R} \subseteq \mathbb{C}$. The rest of this paper is devoted to the discussion of numerical properties of the Frobenius inversion. To this end, we consider the case where $\mathbb{k} = \mathbb{R}$ and $\mathbb{F} = \mathbb{C}$, which is probably the most important quadratic field extension.

In this section, we compare the computational complexity of Algorithm 2 with the usual complex matrix inversion algorithm based on LU decomposition, which is widely employed in various main stream platforms for numerical computing such as MATLAB, Maple, Julia and Python.

4.5.1 Frobenius inversion v.s. invertion via LU decomposition

To begin with, we first recall the algorithm for matrix inversion via LU decomposition [58], which we reproduce in Algorithm 4 for ease of reference. We notice that the main idea behind Algorithm 4 is that $B = A^{-1}$ if and only if $BA = I_n$. It is obviously true that we may replace $BA = I_n$ by $AB = I_n$ in the above. Accordingly, we obtain Algorithm 5 which is slightly different from Algorithm 4.

Before we proceed, we fix some notations. Let $\mathcal{A}$ be an algorithm for real matrix multiplication. We denote by $T^{\mathcal{A}}_{\mathrm{mult}}(n)$ the average running time of $\mathcal{A}$ on pairs of $n \times n$ real matrices. In addition, we let $T^{\mathcal{A}}_{\mathrm{inv}}(n)$ be the average running time of Algorithm 4 on invertible $n \times n$

Algorithm 4 matrix inversion via LU decomposition

Input $A \in \mathrm{GL}_n(\mathbb{k})$
Output inverse of A
1: compute LU factorization of $A = LU$;
2: compute U^{-1};
3: solve for X from $XL = U^{-1}$;
4: **return** X;

Algorithm 5 matrix inversion via LU decomposition

Input $A \in \mathrm{GL}_n(\mathbb{k})$
Output inverse of A
1: compute LU factorization of $A = LU$;
2: compute L^{-1};
3: solve for X from $UX = L^{-1}$;
4: **return** X;

real matrices, in which real matrix multiplications are computed by $\mathcal{A}$. By symmetry, the average running time of Algorithm 5 on invertible $n \times n$ real matrices is also $T^{\mathcal{A}}_{\mathrm{inv}}(n)$. Now with these notations, we are ready to present our threshold theorem.

Theorem 4.5.1 (threshold)**.** Let $\mathcal{A}$ be an algorithm for real matrix multiplication. Assume that the running time of $\mathcal{A}$ on pairs of $n \times n$ matrices of which at least one is upper or lower triangular is $\lambda T^{\mathcal{A}}_{\mathrm{mult}}(n)$ for some $0 < \lambda \leq 1$. Then Algorithm 2 is asymptotically faster than Algorithm 4 over $\mathbb{C}$ if and only if $\lim_{n\to\infty} \left(T^{\mathcal{A}}_{\mathrm{inv}}(n)/T^{\mathcal{A}}_{\mathrm{mult}}(n) \right) > 1 + \lambda/2$. In particular, if $\mathcal{A}$ is the usual matrix multiplication algorithm, then Algorithm 2 is asymptotically faster than Algorithm 4 over $\mathbb{C}$ if and only if $\lim_{n\to\infty} \left(T^{\mathcal{A}}_{\mathrm{inv}}(n)/T^{\mathcal{A}}_{\mathrm{mult}}(n) \right) > 5/4$.

Proof. We first show that the running time of Algorithm 2 is dominated by $2T^{\mathcal{A}}_{\mathrm{inv}}(n) + 5T^{\mathcal{A}}_{\mathrm{mult}}(n)/2$. In fact, we notice that the first two steps in Algorithm 2 (i.e., computing X^{-1} and $X^{-1}Y$) can be combined into one step by solving for F from $XF = Y$. To that end, we compute an LU- decomposition $X = LU$. After that, we compute L^{-1} and $L^{-1}Y$. Finally, we solve for F from $UF = L^{-1}Y$. As a comparison, we note that the only step that is not included in Algorithm 5 is the computation of $L^{-1}Y$.

Since L^{-1} is lower triangular, multiplying L^{-1} with Y takes $\lambda T^{\mathcal{A}}_{\mathrm{mult}}(n)$ operations. There-

fore, the first two steps in Algorithm 2 takes $T^{\mathcal{A}}_{\text{inv}}(n) + \lambda T^{\mathcal{A}}_{\text{mult}}(n)$ time. Then, computing $YX^{-1}Y$ requires one matrix multiplication, computing $X + YX^{-1}Y$ requires one matrix addition, computing $J = (X + YX^{-1}Y)^{-1}$ requires one matrix inversion, and computing $K = X^{-1}YJ$ requires one matrix multiplication. Since matrix addition takes $O(n^2)$ flops, it is not the dominant term in the computation time of Algorithm 2 and we can omit that in this analysis. To sum up, the running time of Algorithm 2 is dominated by $2T^{\mathcal{A}}_{\text{inv}}(n) + (2 + \lambda)T^{\mathcal{A}}_{\text{mult}}(n)$.

Next, we consider the running time of Algorithm 4 for complex matrices. We prove that the running time of Algorithm 4 is dominated by $4T^{\mathcal{A}}_{\text{inv}}(n)$. Note that the complex addition takes 2 real flops and the complex multiplication takes 6 real flops. In Algorithm 4, there are "roughly" the same number of additions and multiplications. Algorithm 4 contains roughly three operations: computing LU factorization, computing forward substitution, and computing the inverse of an upper triangular matrix. Computing LU factorization using Gaussian Elimination requires roughly the same number of additions and multiplications. Computing forward substitution also requires roughly the same number of additions and multiplications. Finally, according to Method 1 in page 263 of [58], inverting a triangular matrix can be done by a sequence of forward substitutions. Thus, inverting a triangular matrix also requires roughly the same number of additions and multiplications. Therefore, the running time of Algorithm 4 over $\mathbb{C}$ is dominated by $4T^{\mathcal{A}}_{\text{inv}}(n)$.

Now, Algorithm 2 is faster than Algorithm 4 if and only if for n sufficiently large,

$$4T^{\mathcal{A}}_{\text{inv}}(n) > 2T^{\mathcal{A}}_{\text{inv}}(n) + (2 + \lambda)T^{\mathcal{A}}_{\text{mult}}(n),$$

which is equivalent to

$$\lim_{n\to\infty} \left(\frac{T^{\mathcal{A}}_{\text{inv}}(n)}{T^{\mathcal{A}}_{\text{mult}}(n)} \right) > 1 + \frac{\lambda}{2}. \qquad \square$$

It is remarkable that the inequality $\lim_{n\to\infty} \left(T^{\mathcal{A}}_{\text{inv}}(n)/T^{\mathcal{A}}_{\text{mult}}(n) \right) > 5/4$ holds in MAT-

LAB. In Section 4.7, we shall see by numerical examples that Algorithm 2 is indeed faster than Algorithm 4 in MATLAB, which confirms Theorem 4.5.1. Moreover, according to Theorem 4.5.1, for any $\mathcal{A}$, Algorithm 2 is asymptotically faster than Algorithm 4 if

$$\lim_{n\to\infty}\left(T^{\mathcal{A}}_{\text{inv}}(n)/T^{\mathcal{A}}_{\text{mult}}(n)\right) > 3/2.$$

We conclude this subsection by a remark on solving systems of linear equations

$$(A+iB)(x+iy) = c+id \tag{4.5.1}$$

by the Frobenius inversion, where $A, B \in \mathbb{R}^{n\times n}$, and $x, y, c, d \in \mathbb{R}^n$. Namely, we can first compute $(A+iB)^{-1}$ by Algorithm 2 and then compute $(A+iB)^{-1}(c+id)$. For a single linear system (4.5.1), it is obviously more efficient to solve it by the LU decomposition together with backward and forward substitutions. However, as pointed out in [30], it is common in scientific computing that one needs to solve (4.5.1) repeatedly with the same $(A+iB)$ but different $(c+di)$. To be more precise, we have

$$(A+iB)(x^{(k)}+iy^{(k)}) = c^{(k)}+id^{(k)}, \quad k=1,\ldots,K,$$

where K is much larger than n. In this scenario, inverting $(A+iB)^{-1}$ by Algorithm 2 is more favourable in the sense of computational efficiency.

4.5.2 Rounding error analysis

In this subsection, we provide a rounding error analysis for Algorithm 2. To do that, we denote by u the unit roundoff. Given an $n\times n$ matrix X, we denote by L_X (resp. U_X) the computed lower (resp. upper) triangular factor in the LU decomposition of X, i.e., $X = L_X U_X$. We also denote by $|X|$ the matrix whose entries are absolute values of elements

of X.

Theorem 4.5.2. Let $Z = A + iB \in \mathcal{S}_1 \cup \mathcal{S}_2$ be the input of Algorithm 2 and W be the output of Algorithm 2. Let $X = A$ and $Y = B$ if $Z \in \mathcal{S}_1$ and let $X = B$ and $Y = A$ if $Z \in \mathcal{S}_2$. Then we have

$$\begin{aligned}|Z^{-1} - W| \leq O(n)\Big(&|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y| \cdot |J| \\ &+ \big(|X^{-1}Y| + I\big)|J|\big(|X| + |Y| \cdot |X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y| \\ &+ |L_P| \cdot |U_P|\big)|J|\Big)\mathsf{u},\end{aligned}$$

where $\leq$ between matrices holds componentwise and P is the computed value of $X+YX^{-1}Y$ by Algorithm 2 and $J = (X + YX^{-1}Y)^{-1}$.

The proof of Theorem 4.5.2 relies on the following lemma.

Lemma 4.5.3. Let $A \in \mathrm{GL}_n(\mathbb{R})$ and $B \in \mathbb{R}^{n\times n}$ be such that $A + \mathsf{u}B \in \mathrm{GL}_n(\mathbb{R})$, where u is the unit roundoff. Then,

$$(A + \mathsf{u}B)^{-1} = A^{-1} - \mathsf{u}A^{-1}BA^{-1} + O(\mathsf{u}^2).$$

Proof. Note that

$$\begin{aligned}(A + \mathsf{u}B)^{-1} &= (I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1} \\ &\overset{(a)}{=} (I_n - \mathsf{u}A^{-1}B + \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B)A^{-1} \\ &= A^{-1} - \mathsf{u}A^{-1}BA^{-1} + O(\mathsf{u}^2),\end{aligned}$$

where (a) follows from the fact that

$$
\begin{aligned}
&(I_n + \mathsf{u}A^{-1}B)(I_n - \mathsf{u}A^{-1}B + \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B) \\
&= I_n - \mathsf{u}A^{-1}B + \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B + \mathsf{u}A^{-1}B - \mathsf{u}^2A^{-1}BA^{-1}B \\
&\quad + \mathsf{u}^3A^{-1}BA^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B \\
&= I_n + \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B - \mathsf{u}^2A^{-1}BA^{-1}B \\
&\quad + \mathsf{u}^3A^{-1}BA^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B \\
&= I_n + \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B - \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B \\
&\quad + \mathsf{u}^3A^{-1}BA^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B \\
&= I_n + \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B - \mathsf{u}^2A^{-1}B(I_n + \mathsf{u}A^{-1}B)^{-1}A^{-1}B \\
&= I_n.
\end{aligned}
$$

□

The proof of Theorem 4.5.2 also requires the following facts in [58]:

- Let $A \in \mathrm{GL}_n(\mathbb{R})$ and let $\widehat{X}$ be the computed inverse of A by Algorithm 4. Then,

$$
\begin{aligned}
\widehat{X} &= A^{-1} + O(n)|\widehat{X}| \cdot |L| \cdot |U| \cdot |A^{-1}|\mathsf{u} + O(\mathsf{u}^2) \\
&= A^{-1} + O(n)|A^{-1}| \cdot |L| \cdot |U| \cdot |A^{-1}|\mathsf{u} + O(\mathsf{u}^2),
\end{aligned} \tag{4.5.2}
$$

  where $A = LU$ is the computed LU decomposition of A.

- Let $A, B \in \mathbb{R}^{n\times n}$ and let $\widehat{C}$ be the computed product of A and B. Then,

$$
\widehat{C} = AB + n|A| \cdot |B|\mathsf{u} + O(\mathsf{u}^2). \tag{4.5.3}
$$

- Let $a, b \in \mathbb{R}$ and let $\hat{c}$ be the computed sum of a and b. Then,

$$\hat{c} = (a + b) + (|a| + |b|)\mathsf{u} + O(\mathsf{u}^2). \tag{4.5.4}$$

- Let $a, b \in \mathbb{R}$ and let $\hat{c}$ be the computed value of $a - b$. Then,

$$\hat{c} = (a - b) + (|a| + |b|)\mathsf{u} + O(\mathsf{u}^2). \tag{4.5.5}$$

Now, we are able to prove Theorem 4.5.2. In the following, we denote by $\widehat{X}$ the the computed value of a matrix X.

Proof of Theorem 4.5.2. Let L_X and U_X be the computed LU factors of X. Then according to (4.5.2), we have

$$\widehat{X^{-1}} = X^{-1} + O(n)|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}|\mathsf{u} + O(\mathsf{u}^2).$$

Let $H_1 = X^{-1}Y$ and let $\widehat{H}_1$ be the computed value of H_1. By (4.5.2) and (4.5.3), we derive

$$\begin{aligned} \widehat{H}_1 &= X^{-1}Y + O(n)|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y|\mathsf{u} + O(n)|X^{-1}| \cdot |Y|\mathsf{u} + O(\mathsf{u}^2) \\ &= H_1 + O(n)|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y|\mathsf{u} + O(\mathsf{u}^2), \end{aligned} \tag{4.5.6}$$

since $|X^{-1}| \cdot |Y| = |I_n| \cdot |X^{-1}| \cdot |Y| = O(|X^{-1}L_X U_X| \cdot |X^{-1}||Y|) = O(|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y|)$.

Let $H_2 = YX^{-1}Y$ and $\widehat{H}_2$ be the computed value of H_2.Then (4.5.2) and (4.5.3) again

imply

$$\begin{aligned}\widehat{H}_2 &= YX^{-1}Y + O(n)|Y|\cdot|X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y|\mathsf{u} \\ &\quad + O(n)|Y|\cdot|X^{-1}|\cdot|Y|\mathsf{u} + O(\mathsf{u}^2) \\ &= H_2 + O(n)|Y|\cdot|X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y|\mathsf{u} + O(\mathsf{u}^2),\end{aligned}$$

since $|Y|\cdot|X^{-1}|\cdot|Y| = |Y|O(|X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y|) = O(|Y|\cdot|X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y|)$.

Let $H_3 = X + YX^{-1}Y$ and $\widehat{H_3}$ be the computed value of H_3. Then according to (4.5.2) and (4.5.3), we have

$$\widehat{H_3} = H_3 + O(n)(|X| + |Y||X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y|)\mathsf{u} + O(\mathsf{u}^2).$$

We recall that P is the computed value of $X + YX^{-1}Y$ and $J = (X + YX^{-1}Y)^{-1}$, thus Lemma 4.5.3 indicates that

$$\begin{aligned}\widehat{P^{-1}} &= H_3^{-1} - O(n)H_3^{-1}(|X| + |Y|\cdot|X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y|)H_3^{-1}\mathsf{u} \\ &\quad + O(n)H_3^{-1}|L_P|\cdot|U_P|\cdot|H_3^{-1}|\mathsf{u} + O(\mathsf{u}^2) \qquad (4.5.7) \\ &= J + O(n)|H_3^{-1}|(|X| + |Y|\cdot|X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y| \qquad (4.5.8) \\ &\quad + |L_P|\cdot|U_P|)|H_3^{-1}|\mathsf{u} + O(\mathsf{u}^2) \\ &= J + O(n)|J|(|X| + |Y|\cdot|X^{-1}|\cdot|L_X|\cdot|U_X|\cdot|X^{-1}|\cdot|Y| + |L_P|\cdot|U_P|)|J|\mathsf{u} + O(\mathsf{u}^2),\end{aligned}$$

where L_P and U_P are the computed LU factors of P.

Let $H_4 = X^{-1}YP^{-1}$ and let $\widehat{H}_4$ be the computed value of H_4. By (4.5.6) and (4.5.7),

we have

$$\begin{aligned}\widehat{H}_4 &= X^{-1}YJ + O(n)|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y| \cdot |J|\mathsf{u} \\ &\quad + O(n)|X^{-1}Y| \cdot |J|(|X| + |Y| \cdot |X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y| + |L_P| \cdot |U_P|)|J|\mathsf{u} \\ &\quad + O(n)|X^{-1}Y| \cdot |J|\mathsf{u} + O(\mathsf{u}^2) \\ &\overset{(b)}{=} K + O(n)\Big(|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y| \cdot |J| \\ &\quad + |X^{-1}Y| \cdot |J|(|X| + |Y| \cdot |X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y| + |L_P| \cdot |U_P|)|J|\Big)\,\mathsf{u} + O(\mathsf{u}^2),\end{aligned} \tag{4.5.9}$$

where $K = X^{-1}YP^{-1}$ and (b) follows from the fact that

$$|X^{-1}Y| \cdot |J| = O(|X^{-1}| \cdot |L_X| \cdot |U_X| \cdot |X^{-1}| \cdot |Y| \cdot |J|).$$

The proof is complete by rearranging the terms in (4.5.9) and adding the error terms in (4.5.9) and (4.5.7). □

4.5.3 Randomized Frobenius inversion

Lemma 4.5.4. Let $Z = A + iB \in \mathbb{C}^{n\times n}$ be an invertible complex matrix where $A, B \in \mathbb{R}^{n\times n}$. There exist at most n values of $\mu \in \mathbb{R}$ such that the real part of $(1 + \mu i)Z$ is invertible.

Proof. The real part of $(1 + \mu i)(A + iB)$ is $A - \mu B$. We consider the matrix pencil $D(t) \coloneqq A + tB$ where $t \in \mathbb{C}$. Since $\det\big(D(t)\big)$ is a polynomial in t of degree at most n and $\det\big(D(i)\big) = \det(Z) \neq 0$, $D(t)$ is singular for at most n values of $t \in \mathbb{C}$. In particular, $A - \mu B$ must be invertible for all but at most n values of $\mu \in \mathbb{R}$. □

We remark that for an arbitrary pair (A, B) of matrices, it is possible that the pencil $D(t) = A + tB$ is singular for all $t \in \mathbb{C}$. The essential ingredient in the proof of Lemma 4.5.4 is that A (resp. B) is the real (resp. imaginary) part of an invertible complex matrix. Based

on Lemma 4.5.4, we obtain Algorithm 6, a randomized version of the Frobenius inversion formula.

Algorithm 6 randomized Frobenius inversion

Input $Z = A + iB \in \mathrm{GL}_n(\mathbb{C})$
Output inverse of Z
1: randomly generate a real number $\mu \in [0, 1]$;
2: compute $X = A - \mu B$ and $Y = \mu A + B$;
3: compute $W = (X + iY)^{-1}$; ▷ by Algorithm 2
4: set $W_1 = \mathrm{Re}(W)$ and $W_2 = \mathrm{Im}(W)$;
5: **return** $Z^{-1} = (W_1 - \mu W_2) + i(\mu W_1 + W_2)$;

Proposition 4.5.5. The running time of Algorithm 6 is $T_n + 8n^2$ where T_n denotes the running time of the Frobenius inversion in Algorithm 2. If we sample $\mu \in [0, 1]$ with respect to a non-atomic probability measure, then Algorithm 6 outputs Z^{-1} correctly, with probability one.

Proof. The statement for the running time of Algorithm 6 is obvious since the running time of Steps 2 and 4 is $4n^2$ respectively. It suffices to prove the almost sure correctness of Algorithm 6. Since μ is randomly picked from $[0, 1]$ with a non-atomic probability measure, any finite subset of $[0, 1]$ is a null set. According to Lemma 4.5.4, $X = A - \mu B$ in Step 2 is invertible with probability one. Thus Algorithm 2 is applicable to $X + iY$ and we have

$$(W_1-\mu W_2)+i(\mu W_1+W_2) = (1+\mu i)W = (1+\mu i)(X+iY)^{-1} = (1+\mu i)\left(Z(1+\mu i)\right)^{-1} = Z^{-1}.$$

□

It is clear that $Z^{-1} = (MZ)^{-1}M$ for any invertible matrix $M \in \mathbb{C}^{n\times n}$. By suitably choosing M, we obtain variants of the Frobenius inversion formula [127, 36]. Unfortunately, these variants are more computationally expensive. Indeed, if we multiply real matrices by the usual method. Then the one given in [36] costs $6n^3$ multiplications and the one in [127] costs $7n^3 + 2n^2$ multiplications. As a comparison, the Frobenius formula only costs $5n^3$

multiplications. On the other side, the Frobenius inversion formula (and its variants) is not applicable to Z in a subvariety of $\mathrm{GL}_n(\mathbb{C})$. To resolve the issue, we may randomly generate M so that the inversion formula is applicable to MZ with probability one. However, if M is a dense matrix then computing MZ and $(MZ)^{-1}M$ costs n^3 multiplications, respectively. Thus the resulting randomized algorithm would cost $7n^3$ multiplications [127]. According to Proposition 4.5.5, it is sufficient to take $M = (1 + \mu i)I_n$ where μ is randomly picked from $[0, 1]$. More importantly, such a choice of M reduces the number of multiplications from $7n^3$ to $5n^3 + 8n^2$.

4.6 Hermitian Positive Definite Matrix Inversion

In this section, we consider a special class of matrices that occur frequently in practice. We assume that $Z = A+iB$ is Hermitian positive definite and $Z \in \mathcal{S}_1$ According to Algorithm 2, we need to compute $A^{-1}B$, $BA^{-1}B$, $(A + BA^{-1}B)^{-1}$ and $A^{-1}B(A + BA^{-1}B)^{-1}$. Since Z is Hermitian positive definite, we claim that both A and $A + BA^{-1}B$ are symmetric positive definite. Moreover, B is skew-symmetric. Therefore we may compute $A^{-1}B$ and $(A + BA^{-1}B)^{-1}$ by Cholesky decomposition. Suppose that $A = U^{\mathsf{T}}U$ is the Cholesky decomposition of A. Then we have

$$\begin{aligned}
&A^{-1}B = U^{-1}(U^{-1})^{\mathsf{T}}B, \\
&BA^{-1}B = BU^{-1}(U^{-1})^{\mathsf{T}}B = -\left[(U^{-1})^{\mathsf{T}}B\right]^{\mathsf{T}}\left[(U^{-1})^{\mathsf{T}}B\right], \\
&(A + BA^{-1}B)^{-1} = \left(A - \left[(U^{-1})^{\mathsf{T}}B\right]^{\mathsf{T}}\left[(U^{-1})^{\mathsf{T}}B\right]\right)^{-1} = V^{-1}(V^{-1})^{\mathsf{T}}, \\
&A^{-1}B(A + BA^{-1}B)^{-1} = A^{-1}BV^{-1}(V^{-1})^{\mathsf{T}},
\end{aligned} \tag{4.6.1}$$

where $A + BA^{-1}B = V^{\mathsf{T}}V$ is the Cholesky decomposition of $A - \left[(U^{-1})^{\mathsf{T}}B\right]^{\mathsf{T}}\left[(U^{-1})^{\mathsf{T}}B\right]$.

Lemma 4.6.1. Let $Z = A + iB$ be an $n \times n$ Hermitian positive definite matrix where

$A, B \in \mathbb{R}^{n\times n}$. Then the following properties hold:

(a) A is symmetric positive definite and B is skew-symmetric.

(b) $A + BA^{-1}B$ is positive definite.

In particular, if we denote by $\mathbb{H}_n^{++}$ the set of all $n \times n$ Hermitian positive definite matrix. Then we must have

$$\mathbb{H}_n^{++} \subseteq \mathcal{S}_1 \coloneqq \{Z = A + iB \in \mathrm{GL}_n(\mathbb{C}) : A, A + BA^{-1}B \in \mathrm{GL}_n(\mathbb{R})\}.$$

Proof. We notice that $A = (Z + \overline{Z})/2$ and $B = (Z - \overline{Z})/2i$. Thus A (resp. B) is symmetric (resp. skew-symmetric) since Z is Hermitian. For any $x \in \mathbb{R}^n$, we have

$$x^{\mathsf{T}}Ax = \frac{x^*(Z + \overline{Z})x}{2} = \frac{x^*Zx}{2} + \overline{\left(\frac{x^*Zx}{2}\right)} = x^*Zx \geq 0$$

and the equality holds if and only if $x = 0$. This proves the positive definiteness of A.

For each $z \in \mathbb{C}^n$, we have

$$z^*\overline{Z}z = \overline{\overline{z}^*Z\overline{z}} \geq 0,$$

and the equality holds if and only if $z = 0$ since Z is positive definite. This implies that $\overline{Z}$ is Hermitian positive definite. We also observe that $A^{-\frac{1}{2}}ZA^{-\frac{1}{2}} = I_n + iA^{-\frac{1}{2}}BA^{-\frac{1}{2}} \succ 0$ and moreover

$$A + BA^{-1}B = A^{\frac{1}{2}}\left(I_n + (A^{-\frac{1}{2}}BA^{-\frac{1}{2}})(A^{-\frac{1}{2}}BA^{-\frac{1}{2}})\right)A^{\frac{1}{2}}.$$

Therefore, it is sufficient to prove (b) for $A = I_n$. In this case, we have

$$I_n + iB \succ 0, \quad I_n - iB \succ 0,$$

from which we may conclude that $I_n + B^2 \succ 0$ since

$I_n + B^2 = (I_n - iB)^{\frac{1}{2}}(I_n + iB)(I_n - iB)^{\frac{1}{2}}$. □

We remark that there is another way to see the positive definiteness of $A + BA^{-1}B$ if we assume a priori it is invertible. Indeed, Lemma 4.4.1 impies that $(A + BA^{-1}B)^{-1}$ is the real part of Z^{-1}. By assumption, Z is positive definite, so are Z^{-1} and its real part $(A + BA^{-1}B)^{-1}$. This implies that $A + BA^{-1}B$ must be positive definite as well.

We give two implementations of equation (4.6.1) which have the same time complexity.

Algorithm 7 First Variant of Frobenius inversion I

Input $Z = A + iB \in \mathcal{S}_1 \cup \mathcal{S}_2$
Output inverse of Z
1: **if** $Z \in \mathcal{S}_1$ **then**
2: set $X = A, Y = B$;
3: **else if** $Z \in \mathcal{S}_2$ **then**
4: set $X = B, Y = A$;
5: **end if**
6: compute Cholesky decomposition of $X = U^{\mathsf{T}}U$;
7: compute $K_1 = (U^{\mathsf{T}})^{-1}Y$;
8: compute $K_2 = U^{-1}K_1$;
9: compute $K_3 = K_1^{\mathsf{T}}K_1$;
10: compute $K_4 = X - K_3$;
11: compute Cholesky decomposition of $K_4 = V^{\mathsf{T}}V$;
12: compute $K_5 = V^{-1}$;
13: compute $K_6 = K_5K_5^{\mathsf{T}}$;
14: compute $K_7 = K_2K_6$;
15: **if** $Z \in \mathcal{S}_1$ **then return** $Z^{-1} = K_6 - iK_7$;
16: **else if** $Z \in \mathcal{S}_2$ **then return** $Z^{-1} = K_7 - iK_6$;
17: **end if**

Note that the only differences between these two implementations are line 12 and line 13. In the first implementation, computing $K_5 = V^{-1}$ takes $\Theta(n^3)$ flops since V is upper triangular and computing $K_6 = K_5K_5^{\mathsf{T}}$ takes $\Theta(n^3)$ flops since K_5 is upper triangular. In the second implementation, computing $K_5 = (V^{\mathsf{T}})^{-1}$ takes $\Theta(n^3)$ flops since V^{T} is lower triangular and computing $K_6 = V^{-1}K_5$ takes $\Theta(n^3)$ flops since solving the triangular system $VK_6 = K_5$ takes n back substitutions, each of which takes $\Theta(n^2)$ flops. Thus, Algorithm 7 and Algorithm 8 have the same flop count. In practice, Algorithm 7 is more efficient than Algorithm 8 when implemented on Matlab.

Algorithm 8 Second Variant of Frobenius inversion I

Input $Z = A + iB \in \mathcal{S}_1 \cup \mathcal{S}_2$
Output inverse of Z
if $Z \in \mathcal{S}_1$ **then**
    set $X = A, Y = B$;
else if $Z \in \mathcal{S}_2$ **then**
    set $X = B, Y = A$;
end if
compute Cholesky decomposition of $X = U^{\mathsf{T}}U$;
compute $K_1 = (U^{\mathsf{T}})^{-1}Y$;
compute $K_2 = U^{-1}K_1$;
compute $K_3 = K_1^{\mathsf{T}}K_1$;
compute $K_4 = X - K_3$;
compute Cholesky decomposition of $K_4 = V^{\mathsf{T}}V$;
compute $K_5 = (V^{\mathsf{T}})^{-1}$;
compute $K_6 = V^{-1}K_5$;
compute $K_7 = K_2K_6$;
if $Z \in \mathcal{S}_1$ **then return** $Z^{-1} = K_6 - iK_7$;
else if $Z \in \mathcal{S}_2$ **then return** $Z^{-1} = K_7 - iK_6$;
end if

4.6.1 Variant of Frobenius inversion vs matrix inversion via Cholesky decomposition

In this section, we compare the variant of Frobenius inversion to matrix inversion algorithm using Cholesky decomposition. When the input matrix Z is hermitian positive definite, Matlab's inversion function exploits Cholesky decomposition to compute Z^{-1}. Similar to what we did to the variant of Frobenius inversion, we give two implementations of Cholesky decompositions that have the same time complexity.

Algorithm 9 First matrix inversion via Cholesky decomposition

Input $A \in \mathrm{GL}_n(\Bbbk)$
Output inverse of A
compute Cholesky decomposition of $A = U^{\mathsf{T}}U$;
compute $K = U^{-1}$;
compute $X = KK^{\mathsf{T}}$;
return X;

Algorithm 10 Second matrix inversion via Cholesky decomposition

Input $A \in \mathrm{GL}_n(\mathbb{k})$
Output inverse of A
1: compute Cholesky decomposition of $A = U^\mathsf{T}U$;
2: compute $(U^\mathsf{T})^{-1}$;
3: solve for X from $UX = (U^\mathsf{T})^{-1}$;
4: **return** X;

These two implementations have the same flop counts by the same reasoning as we did in the variant of Frobenius inversion case. In practice, Algorithm 9 runs faster than Algorithm 10.

For the sake of speed analysis, we will compare Algorithm 8 to Algorithm 10. However, our results hold for Algorithm 7 versus Algorithm 9 as well since Algorithm 8 has the same flop count as Algorithm 7 and Algorithm 10 has the same flop count as Algorithm 9. In order to analyze the speed of Algorithm 8, we present an algorithm for Cholesky decomposition in Algorithm 11 [110].

Algorithm 11 Cholesky decomposition

Input $Z \in \mathbb{C}^n$ is hermitian positive definite
Output inverse of Z
1: let Z be the upper triangular part of Z;
2: **for** $k = 1$ to n **do**
3: $\quad Z[1:k,k] = (Z[1:k-1,1:k-1]^*)^{-1}Z[1:k-1,k]$;
4: $\quad Z[k,k] = \sqrt{Z[k,k] - Z[1:k-1,k]^*Z[1:k-1,k]}$;
5: **end for**
6: **return** Z;

Note that Algorithm 11 uses roughly the same number of additions and multiplications since its main operation is forward substitution: solve for X in $Z[1:k-1,1:k-1]^*X = Z[1:k-1,k]$, where $Z[1:k-1,1:k-1]$ is upper triangular since we take Z to be the upper triangular part of Z in the first line of Algorithm 11.

Before we proceed, we fix some notations. Let $\mathcal{A}$ be an algorithm for real matrix multiplication. We denote by $T^{\mathcal{A}}_{\mathrm{mult}}(n)$ the average running time of $\mathcal{A}$ on pairs of $n \times n$ real matrices.

In addition, we let $T^{\mathcal{A}}_{\text{pinv}}(n)$ be the average running time of Algorithm 10 on invertible $n \times n$ real symmetric positive definite matrices, in which real matrix multiplications are computed by $\mathcal{A}$. Now with these notations, we are ready to present our threshold theorem.

Theorem 4.6.2 (threshold)**.** Let $\mathcal{A}$ be the usual algorithm for real matrix multiplication. Then Algorithm 8 is asymptotically faster than Algorithm 10 over $\mathbb{C}$ if and only if

$$\lim_{n\to\infty} \left(T^{\mathcal{A}}_{\text{pinv}}(n)/T^{\mathcal{A}}_{\text{mult}}(n) \right) > 7/6.$$

Proof. We first show that the running time of Algorithm 8 is dominated by $2T^{\mathcal{A}}_{\text{pinv}}(n) + 7T^{\mathcal{A}}_{\text{mult}}(n)/3$.

First, we show that the first three steps in Algorithm 8 take $T^{\mathcal{A}}_{\text{pinv}}(n) + T^{\mathcal{A}}_{\text{mult}}(n)/3$ time. Recall that the first three steps in Algorithm 8 are computing the Cholesky decomposition of $X = U^{\mathsf{T}}U$, computing $K_1 = (U^{\mathsf{T}})^{-1}Y$, and computing $K_2 = U^{-1}K_1$. In terms of speed, the only difference between these three steps and Algorithm 10 is in step two, where Algorithm 10 solve for X in $U^{\mathsf{T}}X = I$ whereas Algorithm 8 solve for X in $U^{\mathsf{T}}X = Y$. Solving for X in $U^{\mathsf{T}}X = I$ only takes $\Theta(n^3/3)$ flops whereas solving for X in $U^{\mathsf{T}}X = Y$ takes $\Theta(n^3)$ flops if Y does not have special structures. Thus, Algorithm 8 takes $2\Theta(n^3/3)$ more flops which is $T^{\mathcal{A}}_{\text{mult}}(n)/3$ since the usual matrix multiplication algorithm takes $\Theta(2n^3)$ flops.

Then, note that the running time of the rest of Algorithm 8 is dominated by two real matrix multiplication and one real matrix inversion. Thus, the running time of Algorithm 8 is dominated by $2T^{\mathcal{A}}_{\text{pinv}}(n) + 7T^{\mathcal{A}}_{\text{mult}}(n)/3$.

Next, we consider the running time of Algorithm 10 for complex matrices. We prove that the running time of Algorithm 10 is dominated by $4T^{\mathcal{A}}_{\text{pinv}}(n)$. Note that the complex addition takes 2 real flops and the complex multiplication takes 6 real flops. In Algorithm 10, there are "roughly" the same number of additions and multiplications. Algorithm 10 contains roughly three operations: computing Cholesky factorization, computing backward substitution, and computing the inverse of an lower triangular matrix. As we have observed in

Algorithm 11, computing Cholesky factorization requires roughly the same number of additions and multiplications. Computing backward (or forward) substitution also requires roughly the same number of additions and multiplications. Finally, according to Method 1 in page 263 of [58], inverting a triangular matrix can be done by a sequence of forward substitutions. Thus, inverting a triangular matrix also requires roughly the same number of additions and multiplications. Therefore, the running time of Algorithm 10 over $\mathbb{C}$ is dominated by $4T^{\mathcal{A}}_{\text{pinv}}(n)$.

Now, Algorithm 8 is faster than Algorithm 10 if and only if for n sufficiently large,

$$4T^{\mathcal{A}}_{\text{inv}}(n) > 2T^{\mathcal{A}}_{\text{inv}}(n) + 7T^{\mathcal{A}}_{\text{mult}}(n)/3,$$

which is equivalent to

$$\lim_{n\to\infty} \left(\frac{T^{\mathcal{A}}_{\text{inv}}(n)}{T^{\mathcal{A}}_{\text{mult}}(n)} \right) > 7/6. \qquad \square$$

4.6.2 Rounding error analysis

In this section, we give a rounding error analysis of Algorithm 7. Before giving the result, we first introduce some notations. For each matrix X, we use $\widehat{X}$ to denote the computed

value of X. Let

$$\begin{aligned}
\Delta_1' &= n|(\widehat{U}^{\mathsf{T}})^{-1}| \cdot |\widehat{U}^{\mathsf{T}}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} \\
\Delta_2' &= n|\widehat{U}^{-1}| \cdot |\widehat{U}| \cdot |\widehat{U}^{-1}(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} \\
\Delta_3' &= |Y^{\mathsf{T}}\widehat{U}^{-1}|\Delta_1' + \Delta_1'^{\mathsf{T}}(\widehat{U}^{\mathsf{T}})^{-1}Y + n|Y^{\mathsf{T}}\widehat{U}^{-1}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} \\
\Delta_4' &= |Y^{\mathsf{T}}X^{-1}\Delta X X^{-1}Y| + \Delta_3' \\
\Delta_5' &= \Delta_4' + (|X| + |Y^{\mathsf{T}}X^{-1}Y|)\mathsf{u} \\
\Delta_6' &= O(n)|\widehat{V}^{\mathsf{T}}||\widehat{V}|\mathsf{u} \\
\Delta_7' &= O(n)|\widehat{V}^{-1}| \cdot (|(\widehat{V}^{-1})^{\mathsf{T}}| \cdot |\widehat{V}^{\mathsf{T}}| + |\widehat{V}| \cdot |\widehat{V}^{-1}| + I_n) \cdot |(\widehat{V}^{-1})^{\mathsf{T}}|\mathsf{u} \\
\Delta_8' &= \Delta_5' + \Delta_6' \\
\Delta_9' &= |(X - Y^{\mathsf{T}}X^{-1}Y)^{-1}|\Delta_8'|(X - Y^{\mathsf{T}}X^{-1}Y)^{-1}| + \Delta_7'.
\end{aligned}$$

Theorem 4.6.3. Let $Z = A + iB \in \mathcal{S}_1 \cup \mathcal{S}_2$ be a hermitian positive definite matrix and W be the computed inverse of Z using Algorithm 7. Let $X = A$ and $Y = B$ if $Z \in \mathcal{S}_1$ and let $X = B$ and $Y = A$ if $Z \in \mathcal{S}_2$. Then we have

$$|Z^{-1} - W| \leq (|K_2| + 1)\Delta_9' + \Delta_2'|K_6| + n|K_2||K_6|\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.2}$$

The proof of Theorem 4.6.3 relies on the following facts in [58]:

- Let $A \in \mathbb{R}^{n\times n}$ be a symmetric positive definite matrix. Let $\widehat{R}$ be the computed Cholesky factor of A. Then

$$\widehat{R}^{\mathsf{T}}\widehat{R} = A + \Delta A, \tag{4.6.3}$$

  where $|\Delta A| \leq (n+1)|\widehat{R}^{\mathsf{T}}| \cdot |\widehat{R}|\mathsf{u} + O(\mathsf{u}^2)$.

- Let $T \in \mathbb{R}^{n\times n}$ be a triangular matrix and $b \in \mathbb{R}^n$. Let $\hat{x}$ be the solution to $Tx = b$

obtained by forward/backward substitution. Then

$$(T + \Delta T)\hat{x} = b, \tag{4.6.4}$$

where $|\Delta T| \leq n|T|\mathsf{u} + O(u^2)$.

- Let T be a triangular matrix. Let $\widehat{X}$ be the computed inverse of T. Then

$$|\widehat{X} - T^{-1}| \leq O(n)|T^{-1}| \cdot |T| \cdot |T^{-1}|\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.5}$$

Now, we are able to prove Theorem 4.6.3. In the following, we denote by $\widehat{X}$ the the computed value of a matrix X.

proof of Theorem 4.6.3. We introduce some error terms Δ_i in the proof such that $|\Delta_i| \leq \Delta_i' + O(\mathsf{u}^2)$ for all i. Let $\widehat{U}$ be the computed Cholesky factor of X. By equation (4.6.3),

$$\widehat{U}^{\mathsf{T}}\widehat{U} = X + \Delta X, \qquad |\Delta X| \leq O(n)|\widehat{U}^{\mathsf{T}}| \cdot |\widehat{U}|\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.6}$$

Then, by equation (4.6.4),

$$(\widehat{U}^{\mathsf{T}} + \Delta\widehat{U}^{\mathsf{T}})\widehat{K}_1 = Y, \qquad |\Delta\widehat{U}^{\mathsf{T}}| \leq n|\widehat{U}^{\mathsf{T}}|\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.7}$$

This implies that

$$\begin{aligned} \widehat{K}_1 &= (\widehat{U}^{\mathsf{T}} + \Delta\widehat{U}^{\mathsf{T}})^{-1}Y \\ &\overset{(a)}{=} (\widehat{U}^{\mathsf{T}})^{-1}Y - (\widehat{U}^{\mathsf{T}})^{-1}\Delta\widehat{U}^{\mathsf{T}}(\widehat{U}^{\mathsf{T}})^{-1}Y + O(\mathsf{u}^2) \\ &= (\widehat{U}^{\mathsf{T}})^{-1}Y + \Delta_1, \end{aligned} \tag{4.6.8}$$

where (a) follows from Lemma 4.5.3 and $\Delta_1 := -(\widehat{U}^\mathsf{T})^{-1}\Delta\widehat{U}^\mathsf{T}(\widehat{U}^\mathsf{T})^{-1}Y + O(\mathsf{u}^2)$. Note that

$$|\Delta_1| \leq n|(\widehat{U}^\mathsf{T})^{-1}| \cdot |\widehat{U}^\mathsf{T}| \cdot |(\widehat{U}^\mathsf{T})^{-1}Y|\mathsf{u} + O(\mathsf{u}^2), \tag{4.6.9}$$

by equation (4.6.7). Similarly,

$$(\widehat{U} + \Delta\widehat{U})\widehat{K}_2 = \widehat{K}_1, \qquad |\Delta\widehat{U}| \leq n|\widehat{U}|\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.10}$$

This implies that

$$\begin{aligned}
\widehat{K}_2 &= (\widehat{U} + \Delta\widehat{U})^{-1}\widehat{K}_1 \\
&\overset{(b)}{=} \widehat{U}^{-1}\widehat{K}_1 - \widehat{U}^{-1}\Delta\widehat{U}\widehat{U}^{-1}\widehat{K}_1 + O(\mathsf{u}^2) \\
&\overset{(c)}{=} \widehat{U}^{-1}\widehat{K}_1 - \widehat{U}^{-1}\Delta\widehat{U}\widehat{U}^{-1}(\widehat{U}^\mathsf{T})^{-1}Y + O(\mathsf{u}^2) \\
&= \widehat{U}^{-1}\widehat{K}_1 + \Delta_2,
\end{aligned} \tag{4.6.11}$$

where (b) follows from Lemma 4.5.3, (c) follows from equation (4.6.8), and

$$\Delta_2 := -\widehat{U}^{-1}\Delta\widehat{U}\widehat{U}^{-1}(\widehat{U}^\mathsf{T})^{-1}Y + O(\mathsf{u}^2).$$

Note that

$$|\Delta_2| \leq n|\widehat{U}^{-1}| \cdot |\widehat{U}| \cdot |\widehat{U}^{-1}(\widehat{U}^\mathsf{T})^{-1}Y|\mathsf{u} + O(\mathsf{u}^2) \tag{4.6.12}$$

by equation (4.6.10). Then, by equation (4.5.3),

$$\begin{aligned}
\widehat{K}_3 &= \widehat{K}_1^{\mathsf{T}}\widehat{K}_1 + n|\widehat{K}_1^{\mathsf{T}}| \cdot |\widehat{K}_1|\mathsf{u} + O(\mathsf{u}^2) \\
&\overset{(d)}{=} Y^{\mathsf{T}}\widehat{U}^{-1}(\widehat{U}^{\mathsf{T}})^{-1}Y + Y^{\mathsf{T}}\widehat{U}^{-1}\Delta_1 + \Delta_1^{\mathsf{T}}(\widehat{U}^{\mathsf{T}})^{-1}Y + n|Y^{\mathsf{T}}\widehat{U}^{-1}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} + O(\mathsf{u}^2) \\
&= Y^{\mathsf{T}}\widehat{U}^{-1}(\widehat{U}^{\mathsf{T}})^{-1}Y + \Delta_3 \\
&\overset{(e)}{=} Y^{\mathsf{T}}(X + \Delta X)^{-1}Y + \Delta_3 \\
&\overset{(f)}{=} Y^{\mathsf{T}}X^{-1}Y - Y^{\mathsf{T}}X^{-1}\Delta X X^{-1}Y + \Delta_3 \\
&= Y^{\mathsf{T}}X^{-1}Y + \Delta_4,
\end{aligned} \tag{4.6.13}$$

where (d) follows from equation (4.6.8), (e) follows from equation (4.6.6), (f) follows from Lemma 4.5.3,

$$\Delta_3 := Y^{\mathsf{T}}\widehat{U}^{-1}\Delta_1 + \Delta_1^{\mathsf{T}}(\widehat{U}^{\mathsf{T}})^{-1}Y + n|Y^{\mathsf{T}}\widehat{U}^{-1}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} + O(\mathsf{u}^2)$$

and

$$\Delta_4 := -Y^{\mathsf{T}}X^{-1}\Delta X X^{-1}Y + \Delta_3.$$

Note that

$$\begin{aligned}
|\Delta_4| &\overset{(a)}{\leq} O(n)|Y^{\mathsf{T}}X^{-1}| \cdot |\widehat{U}^{\mathsf{T}}| \cdot |\widehat{U}| \cdot |X^{-1}Y|\mathsf{u} + n|Y^{\mathsf{T}}\widehat{U}^{-1}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}| \cdot |\widehat{U}^{\mathsf{T}}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} \\
&\quad + n|Y^{\mathsf{T}}\widehat{U}^{-1}| \cdot |\widehat{U}| \cdot |\widehat{U}^{-1}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} + n|Y^{\mathsf{T}}\widehat{U}^{-1}| \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|\mathsf{u} + O(\mathsf{u}^2) \\
&= O(n)(|Y^{\mathsf{T}}X^{-1}| \cdot |\widehat{U}^{\mathsf{T}}| \cdot |\widehat{U}| \cdot |X^{-1}Y| \\
&\quad + |Y^{\mathsf{T}}\widehat{U}^{-1}| \cdot (|(\widehat{U}^{\mathsf{T}})^{-1}| \cdot |\widehat{U}^{\mathsf{T}}| + |\widehat{U}| \cdot |\widehat{U}^{-1}| + I_n) \cdot |(\widehat{U}^{\mathsf{T}})^{-1}Y|)\mathsf{u} + O(\mathsf{u}^2),
\end{aligned} \tag{4.6.14}$$

where (a) follows from equation (4.6.6) and equation (4.6.9). Then, by equation (4.5.5),

$$\begin{aligned}\widehat{K}_4 &= X - Y^\mathsf{T} X^{-1} Y + \Delta_4 + (|X| + |Y^\mathsf{T} X^{-1} Y|)\mathsf{u} + O(\mathsf{u}^2) \\ &= X - Y^\mathsf{T} X^{-1} Y + \Delta_5,\end{aligned} \tag{4.6.15}$$

where

$$\Delta_5 := \Delta_4 + (|X| + |Y^\mathsf{T} X^{-1} Y|)\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.16}$$

Let $\widehat{V}$ be the computed Cholesky factor of $\widehat{K}_4$. By equation (4.6.3),

$$\widehat{V}^\mathsf{T}\widehat{V} = \widehat{K}_4 + \Delta_6, \qquad |\Delta_6| \le O(n)|\widehat{V}^\mathsf{T}||\widehat{V}|\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.17}$$

Then, by equation (4.6.5),

$$\widehat{K}_5 = \widehat{V}^{-1} + O(n)|\widehat{V}^{-1}| \cdot |\widehat{V}| \cdot |\widehat{V}^{-1}|\mathsf{u} + O(\mathsf{u}^2). \tag{4.6.18}$$

Then, by equation (4.5.3),

$$\begin{aligned}\widehat{K}_6 &= \widehat{V}^{-1}(\widehat{V}^\mathsf{T})^{-1} + O(n)|\widehat{V}^{-1}| \cdot |(\widehat{V}^{-1})^\mathsf{T}| \cdot |\widehat{V}^\mathsf{T}| \cdot |(\widehat{V}^{-1})^\mathsf{T}|\mathsf{u} \\ &\quad + O(n)|\widehat{V}^{-1}| \cdot |\widehat{V}| \cdot |\widehat{V}^{-1}||(\widehat{V}^{-1})^\mathsf{T}|\mathsf{u} + n|\widehat{V}^{-1}||(\widehat{V}^{-1})^\mathsf{T}|\mathsf{u} + O(\mathsf{u}^2) \\ &= \widehat{V}^{-1}(\widehat{V}^\mathsf{T})^{-1} + O(n)|\widehat{V}^{-1}| \cdot (|(\widehat{V}^{-1})^\mathsf{T}| \cdot |\widehat{V}^\mathsf{T}| + |\widehat{V}| \cdot |\widehat{V}^{-1}| + I_n) \cdot |(\widehat{V}^{-1})^\mathsf{T}|\mathsf{u} + O(\mathsf{u}^2) \\ &= \widehat{V}^{-1}(\widehat{V}^\mathsf{T})^{-1} + \Delta_7 \\ &\overset{(a)}{=} (\widehat{K}_4 + \Delta_6)^{-1} + \Delta_7 \\ &\overset{(b)}{=} (X - Y^\mathsf{T} X^{-1} Y + \Delta_8)^{-1} + \Delta_7 \\ &\overset{(c)}{=} (X - Y^\mathsf{T} X^{-1} Y)^{-1} + (X - Y^\mathsf{T} X^{-1} Y)^{-1} \Delta_8 (X - Y^\mathsf{T} X^{-1} Y)^{-1} + \Delta_7 + O(\mathsf{u}^2) \\ &= K_6 + \Delta_9,\end{aligned} \tag{4.6.19}$$

where (a) follows from equation (4.6.17), (b) follows from equation (4.6.15), (c) follows from Lemma 4.5.3, and

$$\begin{aligned}
\Delta_7 &:= O(n)|\widehat{V}^{-1}| \cdot (|(\widehat{V}^{-1})^{\mathsf{T}}| \cdot |\widehat{V}^{\mathsf{T}}| + |\widehat{V}| \cdot |\widehat{V}^{-1}| + I_n) \cdot |(\widehat{V}^{-1})^{\mathsf{T}}|\mathsf{u} + O(\mathsf{u}^2) \\
\Delta_8 &:= \Delta_5 + \Delta_6 \\
\Delta_9 &:= (X - Y^{\mathsf{T}}X^{-1}Y)^{-1}\Delta_8(X - Y^{\mathsf{T}}X^{-1}Y)^{-1} + \Delta_7 + O(\mathsf{u}^2)
\end{aligned}$$

Note that

$$\begin{aligned}
|\Delta_9| &\leq |K_6| \cdot |\Delta_8| \cdot |K_6| + |\Delta_7| + O(\mathsf{u}^2) \\
&\leq |K_6| \cdot (|\Delta_5| + |\Delta_6|) \cdot |K_6| + |\Delta_7| + O(\mathsf{u}^2) \\
&\overset{(a)}{\leq} |K_6| \cdot (|\Delta_4| + (|X| + |Y^{\mathsf{T}}X^{-1}Y|)\mathsf{u} + O(n)|\widehat{V}^{\mathsf{T}}| \cdot |\widehat{V}|\mathsf{u}) \cdot |K_6| + |\Delta_7| + O(\mathsf{u}^2),
\end{aligned} \tag{4.6.20}$$

where (a) follows from equation (4.6.17) and equation (4.6.16). Then, by equation (4.5.3),

$$\begin{aligned}
\widehat{K}_7 &= \widehat{K}_2\widehat{K}_6 + n|\widehat{K}_2| \cdot |\widehat{K}_6|\mathsf{u} + O(\mathsf{u}^2) \\
&= (K_2 + \Delta_2)(K_6 + \Delta_9) + n|K_2||K_6|\mathsf{u} + O(\mathsf{u}^2) \\
&= K_7 + |K_2||\Delta_9| + |\Delta_2||K_6| + n|K_2||K_6|\mathsf{u} + O(\mathsf{u}^2).
\end{aligned} \tag{4.6.21}$$

The result then follows from equation (4.6.21) and equation (4.6.19). □

4.7 Experiments

In this section, we conduct some experiments to compare the Frobenius inversion (Algorithm 2) with the inversion via LU decomposition (Algorithm 4). We compare both the speed and accuracy of the two methods by randomly generated examples. Moreover, we test Algorithms 2 and 4 on matrix sign function, Sylvester equations, Lyapunov equations and

polar decomposition. These experiments show that the Frobenius inversion is more efficient than the inversion via LU decomposition, confirming the threshold Theorem 4.5.1. It is also clear from these experiments that the Frobenius inversion is a bit less accuracy than the inversion via LU decomposition. However, comparing with the increase of efficiency, such a decrease of accuracy is ignorable. Finally, we compare the variant of Frobenius inversion (Algorithm 7) to matrix inversion using Cholesky decomposition (Algorithm 9) on hermitian positive matrices. We show that the variant of Frobenius inversion is more efficient than matrix inversion using Cholesky decomposition with little loss in accuracy. This in turn leads to a more efficient procedure to process MIMO radios.

4.7.1 Efficiency

Let $3600 \leq n \leq 6000$ be a positive integer. We first generate two real matrices $A, B \in \mathbb{R}^{n\times n}$ where elements of A and B are generated uniformly and randomly from $(0, 1)$. Then we take $X = A + iB$. For each dimension n, we generate 10 random matrices by the above procedure.

For each randomly generated matrix X, we compute the inverse of X by Algorithm 2 and Algorithm 4[1] respectively. For each fixed n, we average the computation time for the 10 random instances. In Figure 4.1, we exhibit the computation time of the two algorithms versus the logarithmic dimension of matrices. It is clear from Figure 4.1 that Frobenius inversion is faster than the inversion algorithm via LU decomposition.

4.7.2 Accuracy

Next we compare the accuracy of the two algorithms. For any complex invertible matrix $X = A + iB$, where $A, B \in \mathbb{R}^{n\times n}$, we let $\widehat{X}$ be the computed inverse of X. We assess the

1. In MATLAB, this is simply the command $X \backslash \text{eye}(n)$.

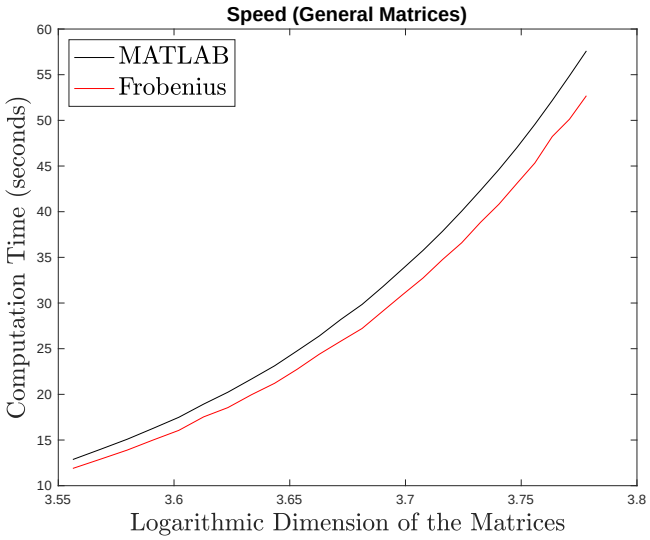


Figure 4.1: comparison of efficiency

.

accuracy of $\widehat{X}$ in terms of its left and right relative residual defined as

$$\mathrm{res}_L(X,\widehat{X}) := \frac{\|\widehat{X}X - I\|_{\max}}{\|X\|_{\max}\|\widehat{X}\|_{\max}} \quad \text{and} \quad \mathrm{res}_R(X,\widehat{X}) := \frac{\|X\widehat{X} - I\|_{\max}}{\|X\|_{\max}\|\widehat{X}\|_{\max}}, \tag{4.7.1}$$

where $\|\cdot\|_{\max}$ is defined as

$$\|A + iB\|_{\max} := \max(\|A\|_{\max}, \|B\|_{\max}) := \max\left(\max_{i,j\in[n]} |a_{ij}|, \max_{i,j\in[n]} |b_{ij}|\right), \tag{4.7.2}$$

where a_{ij} and b_{ij} are the i,jth entries of A and B respectively. To avoid numerical issues, we only test the two algorithms on well-conditioned matrices (i.e., matrices whose condition number is 10) whose dimension goes from 2 to 4096. We generate a random matrices whose condition number is κ by the procedure that follows. We first generate $(n-2)$ integers from 1 to $(\kappa - 1)$ uniformly and randomly. Then, we consider the n by n diagonal matrix D whose diagonal elements are the $(n-2)$ integers we just generate together with 1 and κ. We next multiply the Hadamard matrix H of order n to the left of D and the transpose of H to the right of D to obtain HDH^{T}. Lastly, we scale this matrix by some constant so that

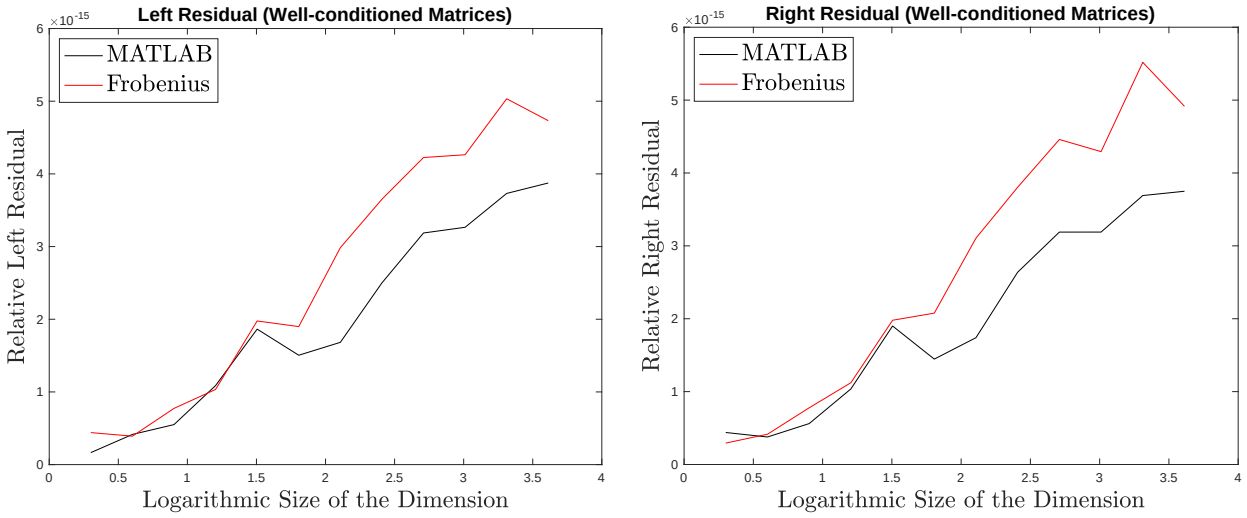


Figure 4.2: comparison of relative residuals

its Frobenius norm is one. Since the smallest and largest singular values of this matrix are c and κc for some $c \in \mathbb{R}$, the condition number of this matrix is κ. In our experiment, we take $\kappa = 10$. We note that there is no specific sparsity patterns on the matrices we generate so that our experiment is not biased toward a certain class of matrices with special sparsity patterns.

In Figure 4.2, we compare the left and right relative residuals of $\widehat{X}$ computed by the two algorithms. It is obvious from the right plot in Figure 4.2 that the Frobenius inversion is slightly less accurate than the inversion via LU decomposition. However, the difference is so small that one can ignore it for practical purposes.

4.7.3 Matrix sign function

In this subsection, we apply matrix inversion algorithms to compute the matrix sign function. The matrix sign function can be used to solve a wide range of problems such as algebraic Riccati equation [102], Sylvester equation [56, 102], polar decomposition [56], and spectral decomposition [3, 4, 10, 61, 81]. We first recall the definition of the matrix sign function.

Given a matrix $A \in \mathbb{C}^{n\times n}$, we write $A = ZJZ^{-1}$ where $J = \begin{bmatrix} J_1 & 0 \\ 0 & J_2 \end{bmatrix}$ is the Jordan canonical form of A such that eigenvalues of A in the diagonal of J_1 (resp. J_2) have negative (resp. positive) real parts. The matrix sign function is defined to be

$$\operatorname{sign}(A) = Z \begin{bmatrix} -I & 0 \\ 0 & I \end{bmatrix} Z^{-1}.$$

The matrix sign function $\operatorname{sign}(A)$ can be computed by Newton iterations [59, 102]

$$X_{t+1} = \frac{1}{2}(X_t + X_t^{-1}), \quad t \in \mathbb{N},$$

with $X_0 = A$. In each iteration, we compute X_t^{-1} either by Frobenius inversion or by inversion via LU decomposition. This gives us two computed value of $\operatorname{sign}(A)$ for each matrix A. We measure the progress in each Newton iteration by the relative change in X_t:

$$\delta_t := \frac{\|X_t - X_{t-1}\|_{\max}}{\|X_t\|_{\max}},$$

where $\|\cdot\|_{\max}$ is defined in (4.7.2). We stop the Newton iterations when either $\delta_t \le 10^{-3}$ or $t \ge 100$ is achieved.

We compute sign function for random square matrices whose dimension n is an even number between 2100 and 4000. To generate such a random matrix, we first generate an $n \times n$ diagonal matrix J whose first $n/2$ diagonal elements have negative real parts and the rest have positive real parts. Next we construct an $n \times n$ matrix Z such that imaginary and real parts of elements of Z are uniformly and randomly taken from $(0, 1)$. Then we take $A = ZJZ^{-1}$. Let $\widehat{S}$ be the computed $\operatorname{sign}(A)$. We measure the accuracy by relative forward

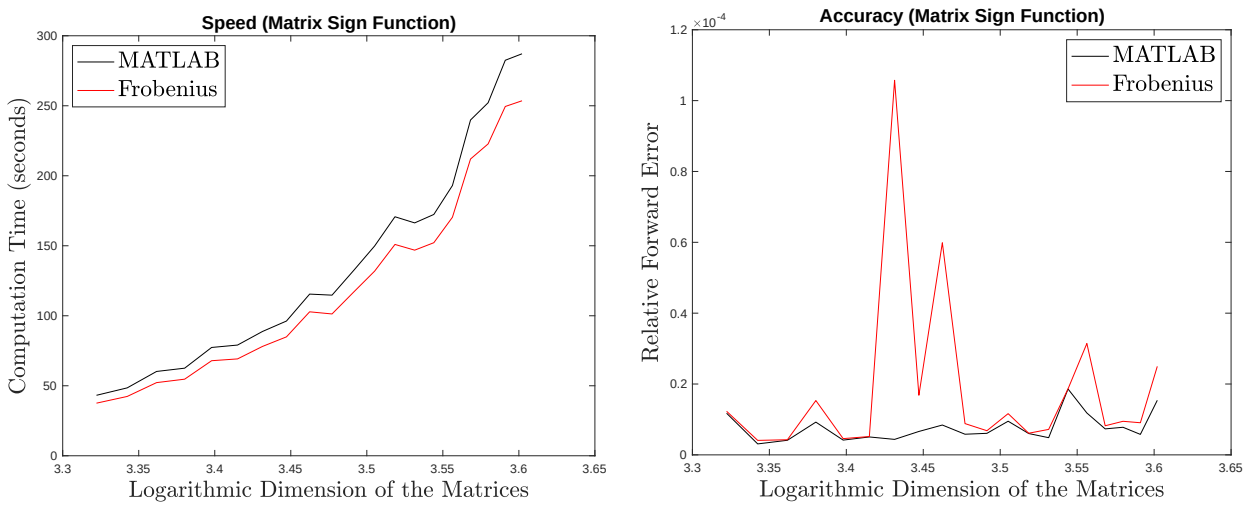


Figure 4.3: comparison on the matrix sign function

error

$$\mathrm{Err}_{\mathrm{relative}}(\mathrm{sign}(A), \widehat{S}) := \frac{\|\mathrm{sign}(A) - \widehat{S}\|_{\max}}{\|\mathrm{sign}(A)\|_{\max}},$$

where $\|\cdot\|_{\max}$ is defined in (4.7.2).

In Figure 4.3, we compare the efficiency and accuracy of Frobenius inversion and inversion via LU decomposition. From the left plot in Figure 4.3, it is clear that the Frobenius inversion is faster than inversion via LU decomposition. The right plot in Figure 4.3 indicates that the Frobenius inversion is a bit less accurate than inversion via LU decomposition. However, in practice, such a small loss of accuracy can be ignored.

4.7.4 Sylvester equation

In this subsection, we apply Algorithms 2 and 4 to solve the Sylvester equation:

$$AX + XB = C, \tag{4.7.3}$$

where $A \in \mathbb{C}^{m\times m}, B \in \mathbb{C}^{n\times n}$, and $C \in \mathbb{C}^{m\times n}$ are given. The goal is to solve for X. As noted in [56, 102], given that $\operatorname{sign}(A) = I_m$ and $\operatorname{sign}(B) = I_n$, we have

$$\operatorname{sign}\left(\begin{bmatrix} A & -C \\ 0 & -B \end{bmatrix}\right) = \begin{bmatrix} I & -2X \\ 0 & -I \end{bmatrix}.$$

This implies that the Sylvester equation can be solved by computing the matrix sign function. Thus the Sylvester equation can be solved by Newton iterations as well. For simplicity, we denote by X_t the variable in Newton iterations:

$$X_{t+1} = \frac{1}{2}(X_t + X_t^{-1}), \quad t \in \mathbb{N},$$

with

$$X_0 = \begin{bmatrix} A & -C \\ 0 & -B \end{bmatrix}.$$

It is noticeable that X_t does not converge to X. Instead, X_t converges to $\begin{bmatrix} I & -2X \\ 0 & -I \end{bmatrix}$. In each iteration, we compute X_t^{-1} either by Frobenius inversion or by inversion via LU decomposition. This gives us two computed values of X for each triple (A, B, C) of parameters in the Sylvester (4.7.3). We measure the progress in Newton iteration by the relative change in X_t:

$$\delta_t := \frac{\|X_t - X_{t-1}\|_{\max}}{\|X_t\|_{\max}},$$

where $\|\cdot\|_{\max}$ is defined in (4.7.2). We stop the Newton iteration when either $\delta_t \le 10^{-1}$ or $t \ge 100$ is achieved. Let $\widehat{X}$ be the computed X. We measure the accuracy by relative forward error

$$\operatorname{Err}_{\text{relative}}(X, \widehat{X}) := \frac{\|X - \widehat{X}\|_{\max}}{\|X\|_{\max}},$$

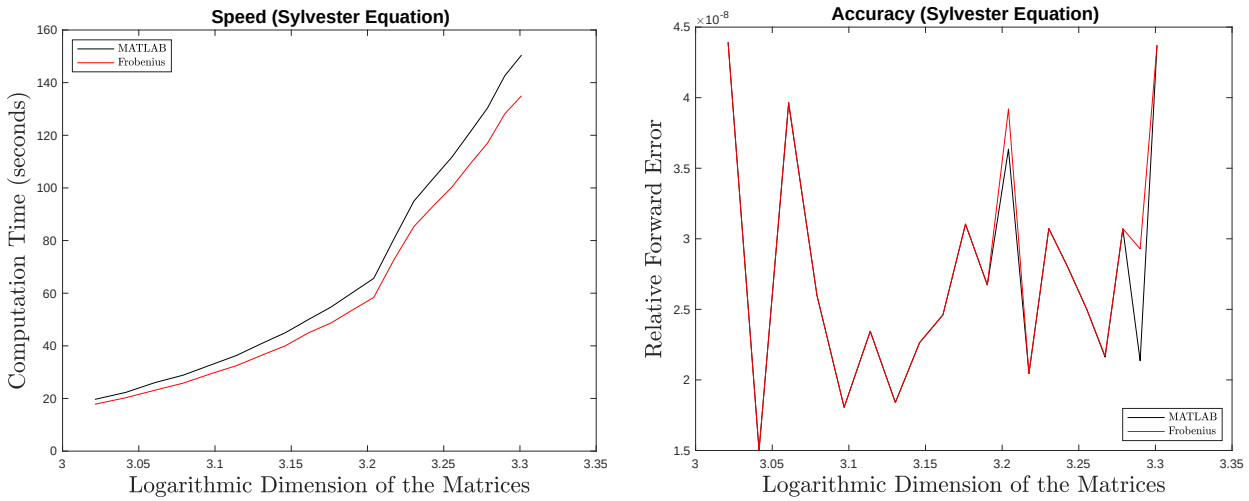


Figure 4.4: comparison on the Sylvester equation

where $\|\cdot\|_{\max}$ is defined in (4.7.2).

We test and compare Algorithms 2 and 4 on the Sylvester equation (4.7.3) whose matrix parameters A, B, C have dimension n between 1050 and 2000. We first generate an $n \times n$ matrix Z such that the real and imaginary parts of its elements are taken uniformly and randomly taken from $(0, 1)$. Next we generate an $n \times n$ diagonal matrix J such that the real and imaginary parts of its diagonal elements are taken uniformly and randomly from the interval $(9, 10)$. Here we do not use the interval $(0, 1)$ since it generates matrices that are close to singular, which makes the algorithms inaccurate. We take $A = ZJZ^{-1}$ and we generate B in the same way. Lastly, we generate a random complex $n \times n$ matrix X such that the real and imaginary parts of each element of X are chosen uniformly and randomly from $(0, 1)$ and take $C = AX + XB$.

In Figure 4.4, we compare the efficiency and accuracy of Frobenius inversion and inversion via LU decomposition on Sylvester equations whose parameters A, B, C are randomly generated by the above procedure. One may easily see from Figure 4.4 that on Sylvester equations, the Frobenius inversion is faster than inversion by LU decomposition and the loss of accuracy is ignorable.

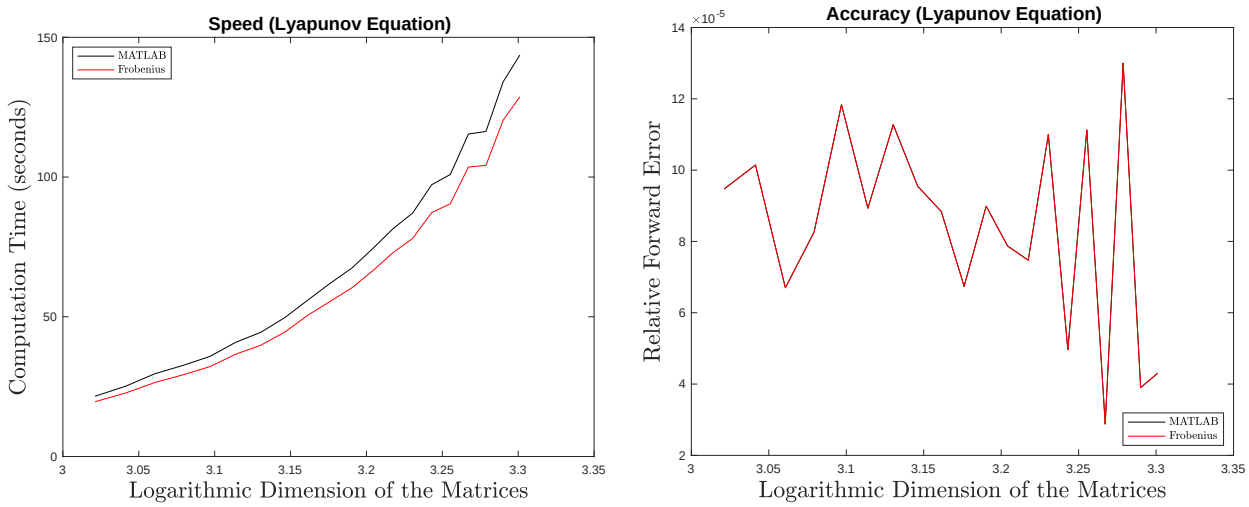


Figure 4.5: comparison on the Lyapunov equation

Lyapunov equation

As an important special case of the Sylvester equation, the Lyapunov equation occurs in control and system theory [58]. To obtain Lyapunov equation, we simply take $B = A^*$ in (4.7.3):

$$AX + XA^* = C, \tag{4.7.4}$$

where $A, C \in \mathbb{C}^{n\times n}$ are given. We apply Algorithms 2 and 4 to solve (4.7.4) in the same way as what we do for Sylvester equations, except that we take $B = A^*$ after we generate A.

From Figure 4.5, we see again that Frobenius inversion is faster than inversion via LU decomposition with almost no loss of accuracy.

4.7.5 Polar decomposition

In this subsection, we apply Algorithms 2 and 4 to compute the polar decomposition. Given a complex matrix $A \in \mathbb{C}^{n\times n}$, its polar decomposition is a matrix decomposition $A = UH$, where U is unitary and H is hermitian positive semidefinite. The Polar decomposition of A

can be computed by the sign function of a block matrix [54, 56, 65]:

$$\operatorname{sign}\left(\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}\right) = \begin{bmatrix} 0 & U \\ U^* & 0 \end{bmatrix},$$

where U is the unitary factor of A in its polar decomposition. Once we obtain U, we can compute H by $H = U^*A$. Therefore, we are able to compute the polar decomposition of a matrix by Newton iterations [54, 56, 65]:

$$X_{t+1} = \frac{1}{2}(X_t + X_t^{-*}), \quad t \in \mathbb{N},$$

with $X_0 = A$. Here X_t^{-*} denotes the conjugate transpose of the inverse of X_t. In each iteration, we compute X_t^{-*} either by Frobenius inversion or by inversion via LU decomposition. This gives us two computed polar decompositions of A. We measure the progress in Newton iterations by the relative change in X_t:

$$\delta_t := \frac{\|X_t - X_{t-1}\|_{\max}}{\|X_t\|_{\max}},$$

where $\|\cdot\|_{\max}$ is defined in (4.7.2). We stop the Newton iterations when either $\delta_t \leq 10^{-3}$ or $t \geq 100$ is achieved.

We test Algorithms 2 and 4 on Polar decompositions of matrices whose dimension n is between 2100 and 4000. We generate a random complex $n \times n$ matrix A as follows. We first generate two random complex matrices B, C such that the real and imaginary parts of elements of B, C are uniformly and randomly taken from $(0, 1)$. Then we compute the QR-decomposition of B and let U be its unitary factor. Lastly, we compute $H = C^*C$ and let $A = UH$.

Let $\widehat{U}$ and $\widehat{H}$ be the factors in the computed polar decomposition of A. We measure

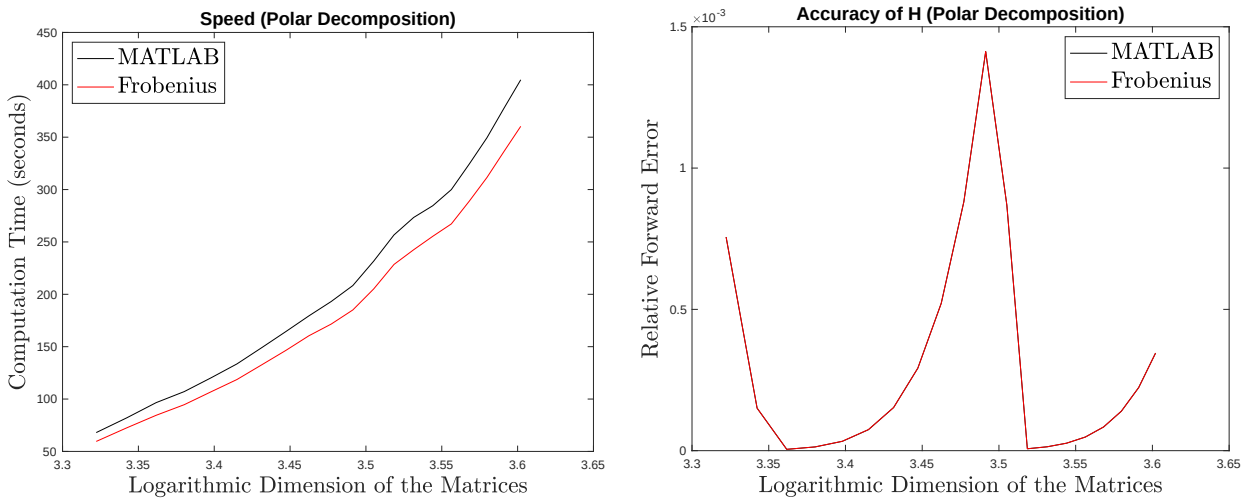


Figure 4.6: comparison on the polar decomposition

their accuracy by the relative forward error:

$$\mathrm{Err}_{\mathrm{relative}}(U,\widehat{U}) := \frac{\|U-\widehat{U}\|_{\max}}{\|U\|_{\max}}, \quad \mathrm{Err}_{\mathrm{relative}}(H,\widehat{H}) := \frac{\|H-\widehat{H}\|_{\max}}{\|H\|_{\max}},$$

where $\|\cdot\|_{\max}$ is defined in (4.7.2).

From the left plot in Figure 4.6, we see that Frobenius inversion is faster than inversion via LU decomposition. Moreover, the right plot in Figure 4.6 indicates that the two algorithms on Polar decompositions have the same accuracy.

4.7.6 Hermitian positive matrices

In this section, we apply Algorithm 7, 9, 4, and 2 to compute the inverse of a hermitian positive matrix X. Inverting hermitian positive matrices is an important task in practice. For instance, in some MIMO radios, inversion of some hermitian positive matrices are needed [115]. Thus, a faster algorithm for inverting hermitian positive matrices leads to a more efficient procedure in processing MIMO radios.

Let $3600 \leq n \leq 6000$ be a positive integer. We first generate two real matrices $A, B \in \mathbb{R}^{n\times n}$ where elements of A and B are generated uniformly and randomly from $(0,1)$. Then

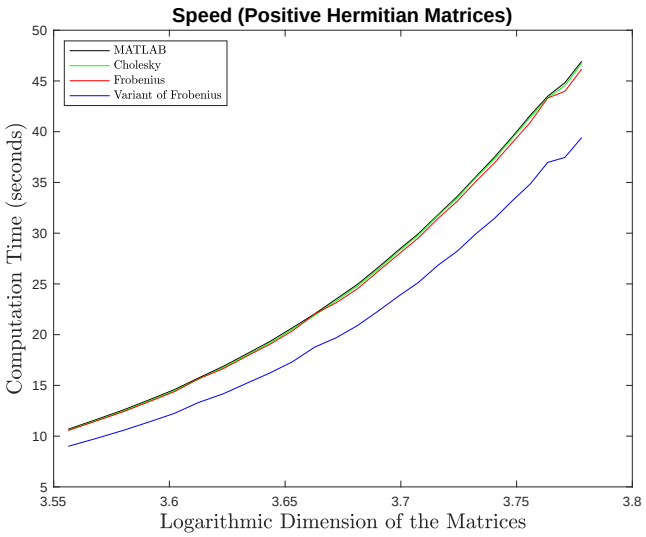


Figure 4.7: comparison of efficiency for positive matrices

.

we take $X = (A + iB)(A + iB)^* + 0.01I$. For each dimension n, we generate 10 random matrices by the above procedure.

For each randomly generated matrix X, we compute the inverse of X by Algorithm 7, 9, 4, and 2 respectively. For each fixed n, we average the computation time for the 10 random instances. In Figure 4.7, we exhibit the computation time of the four algorithms versus the logarithmic dimension of matrices. It is clear from Figure 4.7 that the variant of Frobenius inversion is the fastest one.

Next, we compare the accuracy of the four algorithms. For any hermitian positive matrix X, let $\widehat{X}$ be the computed inverse of X. We assess the accuracy of $\widehat{X}$ in terms of its left and right relative residual defined in equation (4.7.1).

To avoid numerical issues, we only test the four algorithms on well-conditioned matrices (i.e., matrices whose condition number is 10) whose dimension goes from 2 to 4096. We generate a random matrix whose condition number is κ by the procedure that follows. We first generate $(n-2)$ integers from 1 to $(\kappa-1)$ uniformly and randomly. Then, we consider the n by n diagonal matrix D whose diagonal elements are the $(n-2)$ integers we just generate together with 1 and κ. Then, we generate a random unitary matrix U as follows.

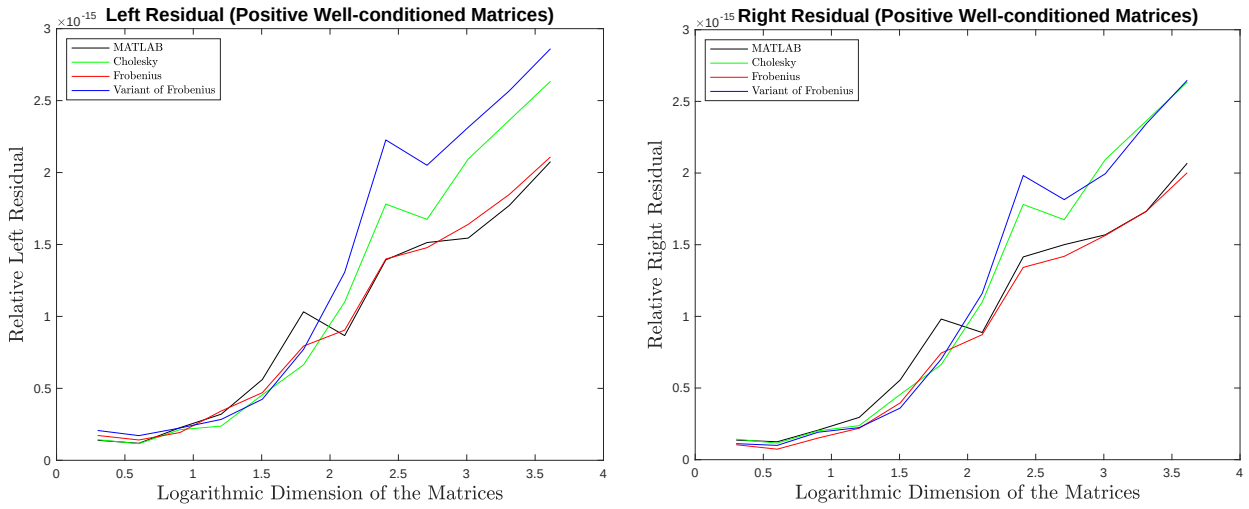


Figure 4.8: comparison of relative residuals for positive matrices

We first generate a random matrix B such that the real and imaginary parts of elements of B are uniformly and randomly taken from $(0, 1)$. Then we compute the QR-decomposition of B and let U be its unitary factor. We next multiply U to the left of D and the conjugate transpose of U to the right of D to obtain UDU^*. Since the smallest and largest singular values of this matrix are c and κc for some $c \in \mathbb{R}$, the condition number of this matrix is κ. In our experiment, we take $\kappa = 10$. We note that there is no specific sparsity patterns on the matrices we generate so that our experiment is not biased toward a certain class of matrices with special sparsity patterns.

In Figure 4.8, we compare the left and right relative residuals of $\widehat{X}$ computed by the four algorithms. It is obvious from the left plot in Figure 4.8 that the variant of Frobenius inversion is slightly less accurate than the other algorithms. However, the difference is so small that one can ignore it for practical purposes.

4.8 Conclusion

In this work, we analyze the Frobenius inversion. We show that Frobenius inversion uses the least number of real matrix multiplications and real matrix inversions among all complex

matrix inversion algorithms. Then, we show that it runs faster than the widely employed inversion algorithm based on the LU decomposition, both theoretically and empirically. Next, we give three applications of the Frobenius inversion: matrix sign function, Sylvester equation, and polar decomposition. Finally, we give a variant of Frobenius inversion on hermitian positive matrices. We show that this algorithm is faster than the widely used inversion algorithm based on Cholesky decomposition, both theoretically and empirically. This leads to a more efficient procedure to process MIMO radios.

CHAPTER 5

CONCLUSION

In this thesis, we discussed three problems in numerical linear algebra and nonconvex optimization. We studied computational complexity and numerical stability of these problems. In rank-constrained hyperbolic programming, we gave conditions under which rank-constrained problems are NP-hard and efficient algorithms that give low rank solutions. In inverting a complex matrix, we studied algorithms that are optimally fast in the sense of algebraic complexity. We showed that the largely forgotten Frobenius inversion is faster than the well-known LU based inversion algorithm and gave engineering applications of Frobenius inversion. In numerical stability and tensor nuclear norm, we proposed a new measure of accuracy on bilinear algorithms: growth factor. We showed that algorithms with smaller growth factor tend to be more accurate. Using this fact, we designed an algorithm for complex matrix multiplication that is both fast and accurate. A natural next step is to understand the tradeoff between bilinear complexity and growth factor in matrix multiplication. It is known that the tensor nuclear norm of two by two matrix multiplication tensor is 8 and the tensor rank of it is 7. However, it is not known whether there is an algorithm for two by two matrix multiplication that simultaneously attains the tensor nuclear norm and the tensor rank. So a natural question to ask is: "what is the smallest growth factor of any bilinear algorithm for two by two matrix multiplication whose bilinear complexity is 7?".

REFERENCES

[1] I. Aizenberg. *Complex-valued neural networks with multi-valued neurons*, volume 353 of *Studies in Computational Intelligence*. Springer-Verlag, Berlin, 2011.

[2] M. F. Anjos and H. Wolkowicz. Geometry of semidefinite max-cut relaxations via matrix ranks. volume 6, pages 237–270. 2002. New approaches for hard discrete optimization (Waterloo, ON,2001).

[3] Z. Bai, J. Demmel, J. Dongarra, A. Petitet, H. Robinson, and K. Stanley. The spectral decomposition of nonsymmetric matrices on distributed memory parallel computers. *SIAM J. Sci. Comput.*, 18(5):1446–1461, 1997.

[4] Z. Bai and J. W. Demmel. *Design of a parallel nonsymmetric eigenroutine toolbox*. University of Kentucky, Department of Mathematics, 1992.

[5] G. Ballard, A. R. Benson, A. Druinsky, B. Lipshitz, and O. Schwartz. Improving the numerical stability of fast matrix multiplication. *SIAM J. Matrix Anal. Appl.*, 37(4):1382–1418, 2016.

[6] Y. Bard. *Nonlinear parameter estimation*. Academic Press [A subsidiary of Harcourt Brace Jovanovich, Publishers], New York-London, 1974.

[7] A. I. Barvinok. Problems of distance geometry and convex properties of quadratic maps. *Discrete Comput. Geom.*, 13(2):189–202, 1995.

[8] J. Bassey, L. Qian, and X. Li. A survey of complex-valued neural networks. *arXiv preprint arXiv:2101.12249*, 2021.

[9] H. H. Bauschke, O. Güler, A. S. Lewis, and H. S. Sendov. Hyperbolic polynomials and convex analysis. *Canad. J. Math.*, 53(3):470–488, 2001.

[10] A. N. Beavers, Jr. and E. D. Denman. A computational method for eigenvalues and eigenvectors of a matrix with real eigenvalues. *Numer. Math.*, 21:389–396, 1973.

[11] M. Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs and non-approximability—towards tight results. In *36th Annual Symposium on Foundations of Computer Science (Milwaukee, WI, 1995)*, pages 422–431. IEEE Comput. Soc. Press, Los Alamitos, CA, 1995.

[12] A. S. Besicovitch. On the linear independence of fractional powers of integers. *J. London Math. Soc.*, 15:3–6, 1940.

[13] D. Bini and G. Lotti. Stability of fast algorithms for matrix multiplication. *Numer. Math.*, 36(1):63–72, 1980/81.

[14] D. Bini, G. Lotti, and F. Romani. Approximate solutions for the bilinear form computational problem. *SIAM J. Comput.*, 9(4):692–697, 1980.

[15] E. Bodewig. *Matrix calculus*. North-Holland Publishing Co., Amsterdam; Interscience Publishers, Inc., New York, enlarged edition, 1959.

[16] A. Borodin and I. Munro. *The computational complexity of algebraic and numeric problems*. Elsevier Computer Science Library: Theory of Computation Series, No. 1. American Elsevier Publishing Co., Inc., New York-London-Amsterdam, 1975.

[17] P. Brändén. Obstructions to determinantal representability. *Adv. Math.*, 226(2):1202–1212, 2011.

[18] R. P. Brent. Algorithms for matrix multiplication. March 1970. Report Stan-CS-70-157, Stanford University.

[19] R. P. Brent. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Numer. Math.*, 16:145–156, 1970.

[20] R. P. Brent and P. Zimmermann. *Modern computer arithmetic*, volume 18 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 2011.

[21] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 1997. With the collaboration of Thomas Lickteig.

[22] R. Byers. Solving the algebraic Riccati equation with the matrix sign function. *Linear Algebra Appl.*, 85:267–279, 1987.

[23] C. Carathéodory. Über den Variabilitätsbereich der Koeffizienten von Potenzreihen, die gegebene Werte nicht annehmen. *Math. Ann.*, 64(1):95–115, 1907.

[24] S. H. Cheng, N. J. Higham, C. S. Kenney, and A. J. Laub. Approximating the logarithm of a matrix to specified accuracy. *SIAM J. Matrix Anal. Appl.*, 22(4):1112–1125, 2001.

[25] H. Cramér. *Mathematical Methods of Statistics*. Princeton Mathematical Series, vol. 9. Princeton University Press, Princeton, N. J., 1946.

[26] B. N. Dash and N. Khare. Deep complex neural network applications in remote sensing: an introductory review. In K. I. Ranney and A. M. Raynal, editors, *Radar Sensor Technology XXV*, volume 11742, pages 34 – 44. International Society for Optics and Photonics, SPIE, 2021.

[27] A. Defant and K. Floret. *Tensor norms and operator ideals*, volume 176 of *North-Holland Mathematics Studies*. North-Holland Publishing Co., Amsterdam, 1993.

[28] H. Derksen. On the nuclear norm and the singular value decomposition of tensors. *Found. Comput. Math.*, 16(3):779–811, 2016.

[29] J. Diestel, J. H. Fourie, and J. Swart. *The metric theory of tensor products.* American Mathematical Society, Providence, RI, 2008. Grothendieck's résumé revisited.

[30] A. Ditkowski, G. Fibich, and N. Gavish. Efficient solution of $Ax^{(k)} = b^{(k)}$ using A^{-1}. *J. Sci. Comput.*, 32(1):29–44, 2007.

[31] A. Druinsky and S. Toledo. How accurate is inv$(A)*b$? *arXiv preprint arXiv:1201.6035*, 2012.

[32] K. Dudeck. Solving complex systems using spreadsheets: A matrix decomposition approach. In *2005 ASEE Annual Conference and Exposition, Conference Proceedings*, pages 12875–12880, 2005. 2005 ASEE Annual Conference and Exposition: The Changing Landscape of Engineering and Technology Education in a Global World ; Conference date: 12-06-2005 Through 15-06-2005.

[33] M. E.-Nagy, M. Laurent, and A. Varvitsiotis. Complexity of the positive semidefinite matrix completion problem with a rank constraint. In *Discrete geometry and optimization*, volume 69 of *Fields Inst. Commun.*, pages 105–120. Springer, New York, 2013.

[34] S. Eberli, D. Cescato, and W. Fichtner. Divide-and-conquer matrix inversion for linear mmse detection in sdr mimo receivers. In *2008 NORCHIP*, pages 162–167. IEEE, 2008.

[35] L. W. Ehrlich. Complex matrix inversion versus real. *Comm. ACM*, 13:561–562, 1970.

[36] M. El-Hawary. Further comments on "a note on the inversion of complex matrices". *IEEE Transactions on Automatic Control*, 20(2):279–280, 1975.

[37] D. K. Faddeev and V. N. Faddeeva. *Computational methods of linear algebra.* W. H. Freeman and Co., San Francisco-London, 1963. Translated by Robert C. Williams.

[38] A. T. Fam. Efficient complex matrix multiplication. *IEEE Trans. Comput.*, 37(7):877–879, 1988.

[39] J. Faraut and A. Korányi. *Analysis on symmetric cones.* Oxford Mathematical Monographs. The Clarendon Press, Oxford University Press, New York, 1994. Oxford Science Publications.

[40] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[41] L. Faybusovich. Jordan-algebraic approach to convexity theorems for quadratic mappings. *SIAM J. Optim.*, 17(2):558–576, 2006.

[42] B. Fischer and J. Modersitzki. Fast inversion of matrices arising in image processing. *Numer. Algorithms*, 22(1):1–11, 1999.

[43] S. Friedland and L.-H. Lim. Nuclear norm of higher-order tensors. *Math. Comp.*, 87(311):1255–1281, 2018.

[44] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[45] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoret. Comput. Sci.*, 1(3):237–267, 1976.

[46] D. Y. Grigoriev. Multiplicative complexity of a pair of bilinear forms and of the polynomial multiplication. In J. Winkowski, editor, *Mathematical Foundations of Computer Science 1978*, pages 250–256, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[47] H. Grötzsch. Ein dreifarbensatz fur dreikreisfreie netze auf der kugal. *Math.- Naturwiss.*, (8):109–120, 1958.

[48] L. Gurvits. Combinatorial and algorithmic aspects of hyperbolic polynomials. *arXiv preprint math/0404474*, 2004.

[49] L. Guttman. Enlargement methods for computing the inverse matrix. *Ann. Math. Statistics*, 17:336–343, 1946.

[50] J. Hå stad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001.

[51] W. W. Hager. Updating the inverse of a matrix. *SIAM Rev.*, 31(2):221–239, 1989.

[52] M. T. Heath, G. A. Geist, and J. B. Drake. Early experience with the intel ipsc/860 at oak ridge national laboratory. *The International Journal of Supercomputing Applications*, 5(2):10–26, 1991.

[53] H. V. Henderson and S. R. Searle. On deriving the inverse of a sum of matrices. *SIAM Rev.*, 23(1):53–60, 1981.

[54] N. J. Higham. Computing the polar decomposition—with applications. *SIAM J. Sci. Statist. Comput.*, 7(4):1160–1174, 1986.

[55] N. J. Higham. Stability of a method for multiplying complex matrices with three real matrix multiplications. *SIAM J. Matrix Anal. Appl.*, 13(3):681–687, 1992.

[56] N. J. Higham. The matrix sign decomposition and its relation to the polar decomposition. In *Proceedings of the 3rd ILAS Conference (Pensacola, FL, 1993)*, volume 212/213, pages 3–20, 1994.

[57] N. J. Higham. Stable iterations for the matrix square root. *Numer. Algorithms*, 15(2):227–242, 1997.

[58] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[59] N. J. Higham. *Functions of matrices*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2008. Theory and computation.

[60] N. J. Higham and P. Papadimitriou. A parallel algorithm for computing the polar decomposition. *Parallel Comput.*, 20(8):1161–1173, 1994.

[61] J. L. Howland. The sign matrix and the separation of matrix eigenvalues. *Linear Algebra Appl.*, 49:221–232, 1983.

[62] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4):413–423, 1978.

[63] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Dokl. Akad. Nauk SSSR*, 14(145):293–294, 1962.

[64] A. A. Karatsuba. The complexity of computations. *Trudy Mat. Inst. Steklov.*, 211(Optim. Upr. i Differ. Uravn.):186–202, 1995.

[65] C. Kenney and A. J. Laub. On scaling Newton's method for polar decomposition and the matrix sign function. *SIAM J. Matrix Anal. Appl.*, 13(3):698–706, 1992.

[66] J. Kim and F. L. Teixeira. Parallel and explicit finite-element time-domain method for Maxwell's equations. *IEEE Trans. Antennas and Propagation*, 59(6, part 2):2350–2356, 2011.

[67] A. Klein and G. Mélard. Computation of the Fisher information matrix for time series models. *J. Comput. Appl. Math.*, 64(1-2):57–68, 1995.

[68] V. V. Kljuev and N. I. Kokovkin-Ščerbak. On the minimization of the number of arithmetic operations for solving linear algebraic systems of equations. *Ž. Vyčisl. Mat i Mat. Fiz.*, 5:21–33, 1965.

[69] D. E. Knuth. *The art of computer programming. Vol. 2*. Addison-Wesley, Reading, MA, 1998. Seminumerical algorithms, Third edition.

[70] A. Kochnev and N. Savelov. Symmetric matrix inversion using modified gaussian elimination. *arXiv preprint arXiv:1504.06734*, 2015.

[71] J.-C. Lafon. Base tensorielle des matrices de Hankel (ou de Toeplitz) applications. *Numer. Math.*, 23:349–361, 1975.

[72] C. Lanczos. *Applied analysis*. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1956.

[73] J. M. Landsberg. The border rank of the multiplication of 2×2 matrices is seven. *J. Amer. Math. Soc.*, 19(2):447–459, 2006.

[74] J. M. Landsberg. *Tensors: geometry and applications*, volume 128 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 2012.

[75] J. M. Landsberg. *Geometry and complexity theory*, volume 169 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 2017.

[76] A. Lemon, A. M.-C. So, and Y. Ye. Low-rank Semidefinite Programming: Theory and Applications. *Foundations and Trends in Optimization*, 2(1-2):1–156, 2015.

[77] T. Lickteig. The computational complexity of division in quadratic extension fields. *SIAM J. Comput.*, 16(2):278–311, 1987.

[78] D. Lijun and L. Lek-Heng. Higher-Order Cone Programming. *arXiv:1811.05461*, 2018.

[79] L.-H. Lim. Self-concordance is NP-hard. *J. Global Optim.*, 68(2):357–366, 2017.

[80] L.-H. Lim. Tensors in computations. *Acta Numer.*, 30:555–764, 2021.

[81] C.-c. Lin and E. Zmijewski. *A parallel algorithm for computing the eigenvalues of an unsymmetric matrix on an SIMD mesh of processors*. University of California, Santa Barbara, College of Engineering, Department . . . , 1991.

[82] K. Lo. Several numerical methods for matrix inversion. *International Journal of Electrical Engineering Education*, 15(2):131–141, 1978.

[83] D. Luenberger and Y. Ye. Linear and nonlinear programming. *Springer*, 3rd edition, 2008.

[84] Z. Luo, W. Ma, A. M. So, Y. Ye, and S. Zhang. Semidefinite relaxation of quadratic optimization problems. *IEEE Signal Processing Magazine*, 27(3):20–34, 2010.

[85] J. H. Maindonald. *Statistical computation*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons, Inc., New York, 1984.

[86] P. McCullagh. Generalized linear models. *European Journal of Operational Research*, 16(3):285–292, 1984.

[87] P. McCullagh and J. A. Nelder. *Generalized linear models*. Monographs on Statistics and Applied Probability. Chapman & Hall, London, 1989. Second edition [of MR0727836].

[88] C. Moore and S. Mertens. *The nature of computation*. Oxford University Press, Oxford, 2011.

[89] P. Mukherjee and L. Satish. On the inverse of forward adjacency matrix. *arXiv preprint arXiv:1711.09216*, 2017.

[90] I. Munro. Some results concerning efficient and optimal algorithms. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 40–44, New York, NY, USA, 1971. Association for Computing Machinery.

[91] M. Muramatsu and T. Suzuki. A new second-order cone programming relaxation for MAX-CUT problems. *J. Oper. Res. Soc. Japan*, 46(2):164–177, 2003.

[92] S. K. Panda. Inverses of bicyclic graphs. *Electron. J. Linear Algebra*, 32:217–231, 2017.

[93] G. Pataki. On the rank of extreme matrices in semidefinite programs and the multiplicity of optimal eigenvalues. *Math. Oper. Res.*, 23(2):339–358, 1998.

[94] G. Pataki. The geometry of semidefinite programming. In *Handbook of semidefinite programming*, pages 29–65. Springer, 2000.

[95] S. Pavlíková. A note on inverses of labeled graphs. *Australas. J. Combin.*, 67:222–234, 2017.

[96] R. Peeters. Ranks and Structure of Graphs. *dissertation*, Tilburg University, 1995.

[97] R. Peeters. Orthogonal representations over finite fields and the chromatic number of graphs. *Combinatorica*, 16(3):417–431, 1996.

[98] G. B. Price. *An introduction to multicomplex spaces and functions*, volume 140 of *Monographs and Textbooks in Pure and Applied Mathematics*. Marcel Dekker, Inc., New York, 1991. With a foreword by Olga Taussky Todd.

[99] C. Radhakrishna Rao. Information and the accuracy attainable in the estimation of statistical parameters. *Bull. Calcutta Math. Soc.*, 37:81–91, 1945.

[100] J. Renegar. Hyperbolic programs, and their derivative relaxations. *Found. Comput. Math.*, 6(1):59–79, 2006.

[101] K. S. Riedel. A Sherman-Morrison-Woodbury identity for rank augmenting matrices with application to centering. *SIAM J. Matrix Anal. Appl.*, 13(2):659–662, 1992.

[102] J. D. Roberts. Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. *Internat. J. Control*, 32(4):677–687, 1980.

[103] S. Rudich. Complexity theory: from Gödel to Feynman. In *Computational complexity theory*, volume 10 of *IAS/Park City Math. Ser.*, pages 5–87. Amer. Math. Soc., Providence, RI, 2004.

[104] R. A. Ryan. *Introduction to tensor products of Banach spaces*. Springer Monographs in Mathematics. Springer-Verlag London, Ltd., London, 2002.

[105] S. Scardapane, S. Van Vaerenbergh, A. Hussain, and A. Uncini. Complex-valued neural networks with nonparametric activation functions. *IEEE Trans. Emerg. Topics Comput.*, 4(2):140–150, 2018.

[106] J. Schur. Über potenzreihen, die im innern des einheitskreises beschränkt sind. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1918:122 – 145, 1918.

[107] C. Segre. Le rappresentazioni reali delle forme complesse e gli enti iperalgebrici. *Math. Ann.*, 40(3):413–467, 1892.

[108] W. W. Smith and J. Smith. *Handbook of real-time fast Fourier transforms*. IEEE New York, 1995.

[109] W. W. Smith, Jr. and S. Erdman. A note on the inversion of complex matrices. *IEEE Trans. Automatic Contol*, AC-19:64, 1974.

[110] G. W. Stewart. *Matrix algorithms. Vol. I.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998. Basic decompositions.

[111] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[112] V. Strassen. Vermeidung von Divisionen. *J. Reine Angew. Math.*, 264:184–202, 1973.

[113] V. Strassen. Relative bilinear complexity and matrix multiplication. *J. Reine Angew. Math.*, 375/376:406–443, 1987.

[114] V. Strassen. Algebraic complexity theory. In *Handbook of theoretical computer science, Vol. A*, pages 633–672. Elsevier, Amsterdam, 1990.

[115] C. Studer, S. Fateh, and D. Seethaler. Asic implementation of soft-input soft-output mimo detection using mmse parallel interference cancellation. *IEEE Journal of Solid-State Circuits*, 46(7):1754–1765, 2011.

[116] L. Tornheim. Inversion of a complex matrix. *Comm. ACM*, 4:398, 1961.

[117] C. Trabelsi, O. Bilaniuk, Y. Zhang, D. Serdyuk, S. Subramanian, J. F. Santos, S. Mehri, N. Rostamzadeh, Y. Bengio, and C. J. Pal. Deep complex networks. In *International Conference on Learning Representations*, 2018.

[118] L. Trevisan, G. B. Sorkin, M. Sudan, and D. P. Williamson. Gadgets, approximation, and linear programming. *SIAM J. Comput.*, 29(6):2074–2097, 2000.

[119] S. Winograd. On the number of multiplications necessary to compute certain functions. *Comm. Pure Appl. Math.*, 23:165–179, 1970.

[120] S. Winograd. On multiplication of 2×2 matrices. *Linear Algebra Appl.*, 4:381–388, 1971.

[121] H. Wolkowicz, R. Saigal, and L. Vandenberghe, editors. *Handbook of semidefinite programming*, volume 27 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, Boston, MA, 2000. Theory, algorithms, and applications.

[122] M. A. Woodbury. *The Stability of Out-Input Matrices*. Chicago, Ill., 1949.

[123] M. A. Woodbury. *Inverting modified matrices.* Princeton University, Princeton, N. J., 1950. Statistical Research Group, Memo. Rep. no. 42,.

[124] D. Ye, Y. Yang, B. Mandal, and D. J. Klein. Graph invertibility and median eigenvalues. *Linear Algebra Appl.*, 513:304–323, 2017.

[125] K. Ye and L.-H. Lim. Fast structured matrix computations: tensor rank and Cohn-Umans method. *Found. Comput. Math.*, 18(1):45–95, 2018.

[126] H. Zhang, M. Gu, X. Jiang, J. Thompson, H. Cai, S. Paesani, R. Santagati, A. Laing, Y. Zhang, M. Yung, et al. An optical neural chip for implementing complex-valued neural network. *Nat. Commun.*, 12(1):1–11, 2021.

[127] A. Zielinski. On inversion of complex matrices. *Internat. J. Numer. Methods Engrg.*, 14(10):1563–1566, 1979.