PRINCIPAL ANGLES BETWEEN SUBSPACES AS RELATED TO RAYLEIGH QUOTIENT AND RAYLEIGH RITZ INEQUALITIES WITH APPLICATIONS TO EIGENVALUE ACCURACY AND AN EIGENVALUE SOLVER

by

Merico Edward Argentati

B.S., Worcester Polytechnic Institute, 1970

M.S., University of California at San Diego, 1972

M.S., Unversity of Colorado at Boulder, 1977

A thesis submitted to the

University of Colorado at Denver

in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Applied Mathematics

2003

This thesis for the Doctor of Philosophy

degree by

Merico Edward Argentati

has been approved

by

Andrew Knyazev

Lynn Bennethum

Jan Mandel

Tom Russell

Junping Wang

Date

Argentati, Merico Edward (Ph.D., Applied Mathematics)

Principal Angles Between Subspaces as Related to Rayleigh Quotient and Rayleigh Ritz Inequalities with Applications to Eigenvalue Accuracy and an Eigenvalue Solver

Thesis directed by Professor Andrew Knyazev

ABSTRACT

In this dissertation we focus on three related areas of research: 1) principal angles (sometimes denoted as canonical angles) between subspaces including theory and numerical results, 2) Rayleigh quotient and Rayleigh Ritz perturbations and eigenvalue accuracy, and 3) parallel software implementation and numerical results concerning the eigenvalue solver LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient Method) [35], using parallel software libraries and interface specifications based on the Lawrence Livermore National Laboratory, Center for Applied Scientific Computing (LLNL-CASC) High Performance Preconditioners (Hypre) project.

Concerning principal angles or canonical angles between subspaces, we provide some theoretical results and discuss how current codes compute the cosine of principal angles, thus making impossible, because of round-off errors, finding small angles accurately. We propose several MATLAB based algorithms that compute both small and large angles accurately, and illustrate their practical robustness with numerical examples. We prove basic perturbation theorems for absolute errors for sine and cosine of principal angles with improved constants. MATLAB release 13 has implemented our SVD–based algorithm for the sine.

Secondly, we discuss Rayleigh quotient and Rayleigh Ritz perturbations and eigenvalue accuracy. Several completely new results are presented. One of the interesting findings characterizes the perturbation of Ritz values for a symmetric positive definite matrix and two different subspaces, in terms of the spectral condition number and the largest principal angle between the subspaces.

The final area of research involves the parallel implementation of the LOBPCG algorithm using Hypre, which involves the computation of eigenvectors that are computed by optimizing Rayleigh quotients with the conjugate gradient method. We discuss a flexible "matrix–free" parallel algorithm and performance on several test problems. This LOBPCG Hypre software has been integrated into LLNL Hypre Beta Release Hypre–1.8.0b.

This abstract accurately represents the content of the candidate's thesis. I recommend its publication.

Signed ________________________________

Andrew Knyazev

DEDICATION

I dedicate this thesis to my dear wife Shannon and my shining, creative daughter Angela who have supported my research and long hours of study over the years, in spite of my strange obsession with mathematics.

I also dedicate this thesis to my mother and father for their love and support over the years, and especially for their encouragement that one can do anything that one wants to if you approach each task with enough confidence and optimism. I also dedicate this thesis to my brothers Jim and John who have always shown support and interest in my work, even though they were far removed from the details.

Finally, I also would like to dedicate this thesis to my uncle Jack Morrissey who inspired me toward science and mathematics, when I was ten years old, by showing me the moon through his 8 inch Newtonian reflecting telescope.

ACKNOWLEDGMENT

I would like to thank my advisor, Andrew Knyazev, for his support and guidance during all stages of my research and dissertation preparation.

This work was in part supported by NSF Award DMS 0208773, Preconditioned Algorithms for Large Eigenvalue Problems, PI Andrew Knyazev. A portion of this work, involving the scalable eigensolver, was supported by Lawrence Livermore National Laboratory, Center for Applied Scientific Computing (LLNL–CASC). I would like to thank Rob Falgout, Charles Tong, Panayot Vassilevski and other members of the LLNL–CASC team for their help.

I would also like to thank several CU-Denver graduate students, including Sean Jenson, who helped to test the SUBSPACEA.m code, and Chan-Chai Aniwathananon and Saulo Oliveira, who participated in testing the code ORTHA.m. Rapheal Bar-Or has been very helpful concerning the development of enhancements to algorithms for computing angles between subspaces. Tessa Weinstein has also been a great source of ideas and encouragement which have contributed to progress in my general research.

CONTENTS

Figures x

Tables xi

Algorithms xii

Chapter

1. Introduction 1

1.1 Overview, Motivation and Organization 1

1.2 Notation 6

2. Principal Angles Between Subspaces 8

2.1 Introduction to Principal Angles 8

2.2 Inaccuracy in the Cosine-Based Algorithm 17

2.3 Properties of Principal Angles and a Sine-Based Algorithm 18

2.4 Generalization to an A–Based Scalar Product 31

2.5 Perturbation of Principal Angles in the A–Based Scalar Product 39

2.6 Algorithm Implementation 51

2.7 Numerical Examples 57

2.8 Availability of the Software 72

2.9 Conclusions Concerning Angles Between Subspaces 73

3. Rayleigh Quotient Perturbations and Eigenvalue Accuracy 75

3.1 Introduction to Rayleigh Quotient Perturbations 75

3.2 Subspaces and the Rayleigh–Ritz Procedure 79

3.3 Analysis in 2–Dimensions 85

3.3.1 Eigenvalue Accuracy in 2–Dimensions 88

3.3.2 Rayleigh Quotient Perturbations in 2–Dimensions 89

3.4 Analysis in the General Case 90

3.4.1 Eigenvalue Accuracy 90

3.4.2 Rayleigh Quotient Perturbations 94

3.5 Bounds for Rayleigh–Ritz Estimates 99

3.5.1 Eigenvalue Approximation by Ritz Values 99

3.5.2 Perturbation of Ritz Values 100

3.6 Rayleigh Quotient Perturbations using a Matrix Inner Product 101

3.7 Conclusions Concerning Rayleigh Quotient Perturbations 106

4. Implementation of a Preconditioned Eigensolver Using Hypre 108

4.1 Introduction to the Preconditioned Eigenvalue Solver 108

4.2 Background Concerning the LOBPCG Algorithm 109

4.3 The Block LOBPCG Algorithm 109

4.4 LOBPCG Hypre Implementation Strategy 116

4.5 Software Implementation Details 118

4.6 Applications Program Interface (API) 120

4.7 Compiling and Testing 120

4.8 Numerical Results 121

4.9 Conclusions Concerning the Hypre LOBPCG Implementation 128

5. Overall Conclusions and Impact 130

5.1 Overall Conclusions 130

5.2 Impact 133

6. Future Work Directions 135

Appendices

A MATLAB Code for SUBSPACEA.M and ORTHA.M 136

B LOBPCG Hypre API Functions 141

C Primary LOBPCG Functions – lobpcg.c 143

References 161

FIGURES

Figure

2.1 Errors in principal angles as functions of $n/2$ 62

2.2 Errors in individual angles in algorithms 2.2 and 2.4 with $A = I$ . . 65

2.3 Errors in individual angles between F_2 and G_2 and F_3 and G_3 . . . 67

2.4 μ_k as functions of l and errors as functions of condition number . . 70

4.1 LOBPCG Hypre software modules 119

4.2 LOBPCG scalability as problem size increases 123

4.3 Execution time to converge as problem size increases 124

4.4 Performance versus quality of preconditioner 125

4.5 Convergence rates for different preconditioners 127

TABLES

Table

2.1 Accuracy of computed angles 18

2.2 Computed sine and cosine of principal angles of the example of [5] 59

4.1 LOBPCG test script 121

4.2 Timing results – scalability 122

4.3 Scalability data for 3–D Laplacian 124

ALGORITHMS

Algorithm

2.1 SUBSPACE.m 21

2.2 Modified and Improved SUBSPACE.m 24

2.3 SUBSPACEA.m 54

2.4 Modified and Improved SUBSPACEA.m 56

4.1 Block LOBPCG Algorithm 110

4.2 Block LOBPCG Algorithm in Hypre 114

1. Introduction

1.1 Overview, Motivation and Organization

In this dissertation we focus on three related areas of research: 1) principal angles (sometimes denoted as canonical angles) between subspaces including theory and numerical results, 2) Rayleigh quotient and Rayleigh Ritz perturbations and eigenvalue accuracy, and 3) parallel software implementation and numerical results concerning the eigenvalue solver LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient Method) [35], using parallel software libraries and interface specifications based on the Lawrence Livermore National Laboratory, Center for Applied Scientific Computing (LLNL-CASC) High Performance Preconditioners (Hypre) project.

The framework for this research embodies finite dimensional real vector spaces, vectors, subspaces, angles between vectors/subspaces, orthogonal projectors, symmetric and/or positive definite operators, eigenvalue problems, Rayleigh quotients and Rayleigh–Ritz approximations.

In Chapter 2 we discuss principal angles between subspaces. Computation of principal angles or canonical angles between subspaces is important in many applications including statistics [8], information retrieval [43], and analysis of algorithms [52]. This concept allows us to characterize or measure, in a natural way, how two subspaces differ, which is the main connection with perturbation theory. When applied to column-spaces of matrices, the principal angles describe canonical correlations of a matrix pair. We provide some theoretical results and discuss how current codes compute cosine of principal angles, thus making impossible, because of round-off errors, finding small angles accurately. We review a combination of sine and cosine based algorithms that provide accurate results for all angles, and include a generalization to an A-based scalar product for a symmetric and positive definite matrix A. We propose several MATLAB based algorithms that compute both small and large angles accurately, and illustrate their practical robustness with numerical examples. We prove basic perturbation theorems for absolute errors for sine and cosine of principal angles with improved constants. Numerical examples and a description of our code are given. MATLAB release 13 has implemented our SVD–based algorithm for the sine.

A portion of this research involving angles, was presented in several talks including a talk at the Seventh Copper Mountain Conference on Iterative Methods, March 24-29, 2002, Copper Mountain: Principal Angles Between Subspaces in an A-Based Scalar Product: Dense and Sparse Matrix Algorithms – joint with Andrew Knyazev, and another talk at the Sixth IMACS International Symposium on Iterative Methods in Scientific Computing March 27-30, 2003 University of Colorado at Denver: Principal Angles Between Subspaces in an A-Based Scalar Product: Interesting Properties, Algorithms and Numerical Examples – joint with Andrew Knyazev.

In Chapter 3 we discuss Rayleigh quotient and Rayleigh Ritz perturbations and eigenvalue accuracy. There are two reasons for studying this problem [52]: first, the results can be used in the design and analysis of algorithms, and second, a knowledge of the sensitivity of an eigenpair is of both theoretical and of practical interest. Rayleigh quotient and Rayleigh Ritz equalities and inequalities are central to determining eigenvalue accuracy and in analyzing many situations when individual vectors and/or subspaces are perturbed. We address the case of a perturbation of general vector, as well as perturbations involving a general vector and an eigenvector, and perturbations involving subspaces. The absolute magnitude of these perturbations is a function of 1) the characteristics of the matrix A (e.g., condition number), 2) the angles between vectors/subspaces, and 3) orthogonal projectors onto the vectors/subspaces that are involved. So the framework for angles, which is discussed in Chapter 2, is also important as a background for Chapter 3.

Several completely new results are presented. One of the interesting findings characterizes the perturbation of Ritz values for a symmetric positive definite matrix and two different subspaces, in terms of the spectral condition number and the largest principal angle between the subspaces. Thus, this is an area where angles between subspaces can be used to obtain some elegant results.

The Rayleigh quotient is defined by

$$\rho(x) = \rho(x; A) = (Ax, x)/(x, x),$$

and we are often interested in bounding the absolute difference $|\rho(x; A) - \rho(y; A)|$, when $A \in \mathbf{R}^{\mathbf{n}\times\mathbf{n}}$ is a symmetric and/or positive definite matrix and x and y are general vectors, or when one of them is an eigenvector. It is natural to characterize these perturbation as some function of angles between individual vectors and/or by the largest principal angle between the subspaces. In this investigation some relevant and interesting perturbations are discussed and proofs are included. For example we prove that

$$\frac{|\rho(y) - \rho(x)|}{\rho(y)} \leq (\kappa(A) - 1) \sin(\angle\{x, y\}),$$

where $\kappa(A)$ is the spectral condition number of A. Considering two subspaces $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R^{n\times n}}$ we obtain the new result

$$\frac{|\alpha_j - \beta_j|}{\alpha_j} \leq (\kappa(A) - 1) \sin(\angle\{\mathcal{X}, \mathcal{Y}\}) \quad j = 1, \ldots, m,$$

where α_j, β_j for $j = 1, \ldots, m$ are the Ritz values of a positive definite matrix A w.r.t to the subspaces $\mathcal{X}$ and $\mathcal{Y}$ and $\angle\{\mathcal{X}, \mathcal{Y}\}$ is the largest principal angle between the subspaces $\mathcal{X}$ and $\mathcal{Y}$.

In our analysis we first restrict the domain to a 2-D subspace. These 2-D results are then extended to larger dimensional spaces. In the literature, this is referred to as a "mini–dimensional" analysis [39]. This extension requires a careful analysis of the properties of subspaces and Rayleigh–Ritz approximations, which is provided. We also provide several alternative proofs, one of which uses a somewhat unique approach of expressing the Rayleigh quotient as a Frobenius inner product; $\rho(x; A) = (A, P_x) = \text{trace}(AP_x)$, where P_x is the orthogonal projector onto x.

In Chapter 4 we discuss the LOBPCG eigensolver algorithm, in detail, and the parallel software implementation of this algorithm. This final area of research involves a project to develop a Scalable Preconditioned Eigenvalue Solver for the solution of eigenvalue problems for large sparse symmetric matrices on massively parallel computers. This work involves the implementation of the LOBPCG algorithm, where eigenvectors are computed by optimizing the Rayleigh quotient with the Conjugate Gradient Method. This Rayleigh quotient optimization uses the Rayleigh–Ritz method, hence this is the connection with Chapter 3 of this work.

The code implementation takes advantage of advances in the Scalable Linear Solvers project, in particular in multigrid technology and in incomplete factorizations (ILU) developed under the Hypre project, at LLNL-CASC. The solver allows for the utilization of Hypre preconditioners for symmetric eigenvalue problems. We discuss a flexible "matrix-free" parallel algorithm, and the capabilities of the developed software. We gain a large amount of leverage through use of the Hypre parallel library functions that handle the construction of a flexible set of preconditioners, application of the preconditioned solve during execution of the algorithm and other linear algebraic operations such as matrix–vector multiplies, scalar products, etc. In the LOBPCG algorithm we implement *soft locking*, whereby certain vectors may be eliminated from the Rayleigh Ritz procedure at each iteration. This is done somewhat differently as compared to *locking* as it is discussed in [2]. This entire development amounts to approximately $4,000$ lines of non-commentary source code. We also discuss performance on several test problems. The LOBPCG Hypre software has been integrated into the Hypre software at LLNL and is part of the Hypre Beta Release Hypre–1.8.0b.

A portion of the work, concerning the implementation of a Hypre version of LOBPCG, was presented at Eleventh Copper Mountain Conference on Multigrid Methods, March 30 - April 4, 2003: Implementation of a Scalable Preconditioned

Eigenvalue Solver Using Hypre - joint with Andrew Knyazev.

Many of the insights in this investigation stem from concrete examples and problems, and special cases. For example, concerning Rayleigh quotient inequalities, the special case of 2–D analysis yields major results for the general case. In the case of angles between subspaces, the simple example of the inaccuracy in the current algorithm provided insights to remedy the problem. Finally the LOBPCG parallel implementation has benefited from code development of MATLAB and C-language programs, and through a number of numerical experiments using both of these earlier implementations.

1.2 Notation

$\mathbf{R^n}$	– Vector space of real n–tuples.
$\mathbf{R^{n\times m}}$	– Vector space of real $n \times m$ matrices.
$x, y, \ldots$	– Lower case Roman letters denote vectors.
$A, B, \ldots$	– Upper case Roman letters denote matrices.
$\mathcal{F}, \mathcal{G}, \ldots$	– Calligraphic letters denote subspaces of $\mathbf{R^n}$.
(x, y)	– Euclidean inner product$(x, y) = x^T y$ where $x, y \in \mathbf{R^n}$.
$\|x\|$	– Euclidean norm $\|x\|^2 = (x, x)$ where $x \in \mathbf{R^n}$.
$\|A\|$	– For a matrix this is the induced operator norm which is derived from the Euclidean norm and is also called the spectral norm.
$\|x\|_B$	– B–norm $\|x\|_B^2 = (Bx, x)$ where $x \in \mathbf{R^n}$ and $B \in \mathbf{R^{n\times n}}$ is a positive definite matrix.
$\|A\|_B$	– For a matrix this is the induced operator norm which is

	derived from the B–norm.
(A, B)	– For $A, B \in \mathbf{R^{n\times n}}$ this is the Frobenius inner product $(A, B) = \text{trace}(A^T B)$.
$\kappa(A)$	– Spectral condition number $\kappa(A) = \|A\|\|A^{-1}\| = \dfrac{\lambda_n}{\lambda_1}$ for A positive definite.
$\mathcal{F}^{\perp}$	– The orthogonal complement of the subspace $\mathcal{F}$.
$\mathcal{F} \ominus \mathcal{G}$	– Is the subspace$\mathcal{F}$ intersected with the orthogonal complement of $\mathcal{G}$, that is $\mathcal{F} \ominus \mathcal{G} = \mathcal{F} \cap \mathcal{G}^{\perp}$.
$\rho(x; A)$	– The Rayleigh quotient is defined for $x \in \mathbf{R^n}$ with $x \neq 0$ by $\rho(x; A) = (Ax, x)/(x, x)$. If it is obvious which matrix A is involved, the form $\rho(x)$ may be used.
$r(x; A)$	– The residual is defined for $x \in \mathbf{R^n}$ with $x \neq 0$ by $r(x; A) = (A - \rho(x; A)I)x$. If it is obvious which matrix A is involved, the form $r(x)$ may be used.
$\angle\{x, y\}$	– The acute angle between the vectors $x, y \in \mathbf{R^n}$ with $x, y \neq 0$ is given by $\angle\{x, y\} = \arccos \frac{\|(x,y)\|}{\|x\|\|y\|}$.
$\angle\{\mathcal{F}, \mathcal{G}\}$	– The largest principal angle between the subspaces $\mathcal{F}, \mathcal{G} \subseteq \mathbf{R^n}$.

2. Principal Angles Between Subspaces

2.1 Introduction to Principal Angles

In this chapter we present concepts and results concerning angles between subspaces. Given two non–zero vectors, the acute angle between the vectors x, y $\in \mathbf{R^n}$ with $x, y \neq 0$ is denoted by

$$\angle\{x, y\} = \arccos \frac{|(x, y)|}{\|x\|\|y\|}.$$

It is important to emphasize that $0 \leq \angle\{x, y\} \leq \frac{\pi}{2}$, by definition.

Considering two subspaces, we can recursively define a set of angles between them, which are denoted as principal or canonical angles. Let us consider two real-valued matrices F and G, each with n rows, and their corresponding column-spaces $\mathcal{F}$ and $\mathcal{G}$, which are subspaces in $\mathbf{R^n}$, assuming that

$$p = \dim\mathcal{F} \geq \dim\mathcal{G} = q \geq 1.$$

Then the *principal angles*

$$\theta_1, \ldots, \theta_q \in [0, \pi/2]$$

between $\mathcal{F}$ and $\mathcal{G}$ may be defined, e.g., [29, 21], recursively for $k = 1, \ldots, q$ by

$$\cos(\theta_k) = \max_{u \in \mathcal{F}} \max_{v \in \mathcal{G}} u^T v \ = \ u_k^T v_k$$

subject to

$$\|u\| = \|v\| = 1, \quad u^T u_i = 0, \quad v^T v_i = 0, \quad i = 1, \ldots, k-1.$$

The vectors $u_1, \ldots, u_q$ and $v_1, \ldots, v_q$ are called principal vectors. We first find θ_1 and the corresponding principal vectors u_1 and v_1. To obtain the second angle, we search in subspaces that are orthogonal, respectively to u_1 and v_1. Continuing in this manner, always searching in subspaces orthogonal to principal vectors that have already been found, we obtain the complete set of principal angles and principal vectors.

Here and below $\|\cdot\|$ denotes the standard Euclidean norm of a vector or, when applied to a matrix, the corresponding induced operator norm, also called the spectral norm, of the matrix.

We are often interested in the largest principal angle θ_q. In this investigation we will denote the largest principal angle by

$$\theta_q = \angle\{\mathcal{F}, \mathcal{G}\}.$$

Definition 4.2.1 of [52], provides an equivalent, slightly more intuitive characterization of the largest principal angle when $p = q$, which is given by

$$\theta_q = \max_{\substack{u \in \mathcal{F} \\ u \neq 0}} \min_{\substack{v \in \mathcal{G} \\ v \neq 0}} \angle\{u, v\}.$$

According to [51, 47], the notion of canonical angles between subspaces goes back to Jordan (1875). Principal angles between subspaces, and particularly the smallest and the largest angles, serve as important tools in functional analysis (see books [1, 20, 30] and a survey [14]) and in perturbation theory of invariant subspaces, e.g., [9, 51, 48, 33, 42].

Computation of principal angles between subspaces is needed in many applications. For example, in statistics, the angles are closely related to measures of dependency and covariance of random variables; see a canonical analysis of [8]. When applied to column-spaces $\mathcal{F}$ and $\mathcal{G}$ of matrices F and G, the principal angles describe canonical correlations $\sigma_k(F, G)$ of a matrix pair, e.g., [29, 23], which is important in applications such as system identification and information retrieval. Principal angles between subspaces also appear naturally in computations of eigenspaces, e.g., [34, 35], where angles provide information about solution quality and need to be computed with high accuracy.

In such large-scale applications, it is typical that $n \gg p$; in other words, informally speaking, we are dealing with a small number of vectors having a large number of components. Because of this, we are interested in "matrix-free" methods; i.e., no n-by-n matrices need to be stored in memory in our algorithms.

A singular value decomposition (SVD)–based algorithm [18, 5, 7, 21, 23] for computing cosines of principal angles can be formulated as follows. Let columns of matrices $Q_F \in \mathbf{R^{n\times p}}$ and $Q_G \in \mathbf{R^{n\times q}}$ form orthonormal bases for the subspaces $\mathcal{F}$ and $\mathcal{G}$, respectively. The reduced SVD of $Q_F^T Q_G$ is

$$Y^T Q_F^T Q_G Z \;=\; \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_q), \qquad 1 \geq \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q \geq 0, \tag{2.1}$$

where $Y \in \mathbf{R^{p\times q}}$, $Z \in \mathbf{R^{q\times q}}$ both have orthonormal columns. Then the principal angles can be computed as

$$\theta_k = \arccos(\sigma_k), \qquad k = 1, \ldots, q, \tag{2.2}$$

where

$$0 \leq \theta_1 \leq \cdots \leq \theta_q \leq \frac{\pi}{2},$$

while principal vectors are given by

$$u_k = Q_F y_k, \quad v_k = Q_G z_k, \quad k = 1, \ldots, q.$$

The equivalence [5, 21] of the original geometric definition of principal angles and the SVD–based approach follows from the next general theorem on an equivalent representation of singular values.

Theorem 2.1 *If $M \in \mathbf{R^{m\times n}}$, then the singular values of M are defined recursively by*

$$\sigma_k = \max_{y\, \in \mathbf{R^m}} \max_{z\, \in \mathbf{R^n}} y^T M z \;=\; y_k^T M z_k, \qquad k = 1, \ldots, \min\{m, n\}, \tag{2.3}$$

subject to

$$\|y\| = \|z\| = 1, \quad y^T y_i = 0,\; z^T z_i = 0, \quad i = 1, \ldots, k-1. \tag{2.4}$$

The vectors y_i and z_i are, respectively, left and right singular vectors.

Proof: The proof of the theorem is straightforward if based on Allakhverdiev's representation (see [20]) of singular numbers,

$$\sigma_k = \left\| M - \sum_{i=1}^{k-1} v_i u_i^T \sigma_i \right\|,$$

and using the well-known formula of the induced Euclidean norm of a matrix as the norm of the corresponding bilinear form. ■

To apply the theorem to principal angles, one takes $M = Q_F^T Q_G$.

In the most recent publication on the subject, [16], the SVD–based algorithm for cosine is proved to be stable, but QR factorizations with complete pivoting are recommended for computing Q_F and Q_G to improve stability.

The SVD–based algorithm for cosine was considered as standard and is implemented in software packages, e.g., in MATLAB, version 5.3, 2000, code SUBSPACE.m, revision 5.5,[1] where $Q_F \in \mathbf{R}^{\mathbf{n}\times\mathbf{p}}$ and $Q_G \in \mathbf{R}^{\mathbf{n}\times\mathbf{q}}$ are computed using the QR factorization.

However, this algorithm cannot provide accurate results for small angles because of the presence of round-off errors. Namely, when using the standard double-precision arithmetic with $EPS \approx 10^{-16}$ the algorithm fails to accurately compute angles smaller than 10^{-8} (see section 2.2). The problem has been highlighted in the classical paper [5], as well as a cure has been suggested (see also publications on cosine-sine (CS) decomposition methods [50, 54, 51, 47]), but apparently it did not attract enough attention.

In statistics, most software packages include a code for computing $\sigma_k = cos(\theta_k)$, which are called *canonical correlations*; see, e.g., CANCOR Fortran code in FIRST MDS Package of AT&T, CANCR (DCANCR) Fortran subroutine in IMSL STAT/LIBRARY, G03ADF Fortran code in NAG package, CANCOR subroutine in Splus, and CANCORR procedure in SAS/STAT Software. While accurately computing the cosine of principal angles in corresponding precision, these codes do not compute the sine. However, the cosine simply equals one in

[1]Revision 5.8 of SUBSPACE.m in MATLAB release 12.1, version 6.1.0.450, May 18, 2001, is still identical to revision 5.5, which we have used for numerical tests in this investigation. MATLAB release 13 has implemented the SVD–based algorithm for sine.

double precision for all angles smaller than 10^{-8} (see next section). Therefore, it is impossible in principal to observe an improvement in canonical correlations for angles smaller than 10^{-8} in double precision. It might not be typically important when processing experimental statistical data because the expected measurement error may be so great that a statistician would deem the highly correlated variable essentially redundant and therefore not useful as a further explanatory variable in their model. Statistical computer experiments are different, however, as there is no measurement error, so accurate computation of very high correlations may be important in such applications.

The largest principal angle is related to the notion of distance, or a gap, between equidimensional subspaces. If $p = q$, the distance is defined [1, 20, 21, 30] as

$$\text{gap}(\mathcal{F}, \mathcal{G}) = \|P_F - P_G\| = \sin(\theta_q) = \sqrt{1 - (\cos(\theta_q))^2}, \tag{2.5}$$

where P_F and P_G are orthogonal projectors onto $\mathcal{F}$ and $\mathcal{G}$, respectively.

This formulation provides insight into a possible alternative algorithm for computing the sine of principal angles. The corresponding algorithm, described in [5], while being mathematically equivalent to the previous one in exact arithmetic, is accurate for small angles in computer arithmetic as it computes the sine of principal angles directly, without using SVD (2.1) leading to the cosine. We review the algorithm of [5] based on a general form of (2.5) in section 2.3 and suggest an improved version, similar to the CS decomposition algorithm of [54], with the second SVD of the reduced size.

The CS decomposition methods, e.g., [50, 54, 51, 47], one of which we just mentioned, provide a well-known and popular alternative approach for computing principal angles between subspaces given by selected p (q) columns of orthogonal matrices of the size n. For example, if the matrix $Q_{F^\perp}$, with orthonormal columns that span the subspace $\mathcal{F}^\perp$, the orthogonal complement of $\mathcal{F}$, is available to us, the CS decomposition methods compute the SVD of $(Q_F)^T Q_G$ together with the SVD of $(Q_{F^\perp})^T Q_G$, thus providing cosine and sine of principal angles. When p is of the same order as $n/2$, matrix $Q_{F^\perp}$ is about of the same size as matrix Q_F, and the CS decomposition methods are effective and are recommended for practical computations. However, when $n \gg p$, the CS decomposition methods, explicitly using matrix $Q_{F^\perp}$ of the size n-by-$n-p$, will be less efficient compared to "matrix-free" methods we consider in this investigation. Let us highlight that the cosine- and sine–based methods of [5] that we investigate here in section 2.3, while different algorithmically from the CS decomposition methods, are very close mathematically to them.

A different sine–based approach, using eigenvalues of $P_F - P_G$, is described in [7, 47]; see a similar statement of Theorem 2.5. It is also not attractive numerically, when $n \gg p$, as it requires computing an n-by-n matrix and finding all its nonzero eigenvalues.

In some applications, e.g., when solving symmetric generalized eigenvalue problems [34], the default scalar product $u^T v$ cannot be used and needs to be replaced with an A–based scalar product $(u, v)_A = u^T A v$, where A is a symmetric positive definite matrix. In statistics, a general scalar product for computing canonical correlations gives a user an opportunity, for example, to take into account a priori information that some vector components are more meaningful than others. In a purely mathematical setting, generalization to A–based scalar products brings nothing really new. In practical computations, however, it carries numerous algorithmic and numerical problems, especially for ill-conditioned cases, which are important in applications.

In section 2.4, we propose extension of the algorithms to an A–based scalar product and provide the corresponding theoretical justification.

In section 2.5, we turn our attention to perturbation estimates, which generalize the following trigonometric inequalities: if an angle $\theta \in [0, \pi/2]$ is perturbed by $\epsilon \in [0, \pi/2]$ such that $\theta + \epsilon \in [0, \pi/2]$, then

$$0 \le \cos(\theta) - \cos(\theta + \epsilon) \le \sin(\theta + \epsilon)\sin(\epsilon) \le \sin(\epsilon),$$

$$0 \le \sin(\theta + \epsilon) - \sin(\theta) \le \cos(\theta)\sin(\epsilon) \le \sin(\epsilon).$$

We prove new absolute perturbation estimates for the sine and cosine of principal angles computed in the A–based scalar product. When $A = I$, our estimates are similar to those of [5, 55, 53, 23, 22], but the technique we use is different. More importantly, our constants are somewhat better, in fact, in the same way the constants in the middle terms of the trigonometric inequalities above are less than one.

We consider particular implementation of algorithms used in our MATLAB code SUBSPACEA.m in section 2.6, with emphasis on the large-scale case, $n \gg p$, and sparse ill-conditioned matrix A, which may be specified only as a function that multiplies A by a given vector. When matrices F and G are sparse, our code can still be used even though it performs orthogonalization of columns of matrices F and G that increases the fill-in; cf. [23]. Also, we do not even touch here upon a practically important issue of the possibility of recomputing the correlations with an increase in the data; see again [23].

Finally, numerical results, presented in section 2.7, demonstrate the practical robustness of our code.

For simplicity, we discuss only real spaces and real scalar products; however, all results can be trivially generalized to cover complex spaces as well. In fact, our code SUBSPACEA.m is written for the general complex case.

As was pointed out in [40], there are several natural questions that are of interest, but are not considered here.

- Our algorithms are based on SVD. How does SVD accuracy (cf. [4, 12, 11, 16]), especially for small singular values, or in ill-conditioned cases, affect the results?

- In [16], a formal stability analysis is done for the SVD–based algorithm for cosine, which is proved to be mixed stable. In our numerical tests, practical robustness of our algorithms is encouraging. Are our methods accurate and stable theoretically, e.g., see [26]?

- For A–based scalar products, how does the increase of the condition number of A influence the accuracy? Which parts of our algorithm are responsible for the main error growth?

We feel, however, that investigating these matters is not within the scope of this work. They may rather serve as interesting directions for future research.

2.2 Inaccuracy in the Cosine-Based Algorithm

Let d be a constant and

$$\mathcal{F} = \text{span}\left\{(1\ 0)^T\right\}, \qquad \mathcal{G} = \text{span}\left\{(1\ d)^T\right\}.$$

Then the angle between the one-dimensional subspaces $\mathcal{F}$ and $\mathcal{G}$ can be computed as

$$\theta = \arcsin\left(\frac{d}{\sqrt{1+d^2}}\right). \tag{2.6}$$

In the table below d varies from one to small values. Formula (2.6) is accurate for small angles, so we use it as an "exact" answer in the second column of the table. We use the MATLAB built-in function SUBSPACE.m (revision 5.5) which implements (2.1) to compute values for the third column of the table.

It is apparent that SUBSPACE.m returns inaccurate results for $d \le 10^{-8}$, which is approximately $\sqrt{EPS}$ for double precision. In this simple one-dimensional example the algorithm of SUBSPACE.m is reduced to computing

$$\theta = \arccos\left(\frac{1}{\sqrt{1+d^2}}\right).$$

This formula clearly shows that the inability to compute accurately small angles is integrated in the standard algorithm and cannot be fixed without changing the algorithm itself. The cosine, that is, a canonical correlation, is computed accurately and simply equals to one for all positive $d \le 10^{-8}$. However, one cannot determine small angles from a cosine accurately in the presence of round-off errors. In statistical terms, it illustrates the problem we already mentioned

d	Formula (2.6)	SUBSPACE.m
1.0	7.853981633974483e-001	7.853981633974483e-001
1.0e-04	9.999999966666666e-005	9.999999986273192e-005
1.0e-06	9.999999999996666e-007	1.000044449242271e-006
1.0e-08	1.000000000000000e-008	-6.125742274543099e-017
1.0e-10	1.000000000000000e-010	-6.125742274543099e-017
1.0e-16	9.999999999999998e-017	-6.125742274543099e-017
1.0e-20	9.999999999999998e-021	-6.125742274543099e-017
1.0e-30	1.000000000000000e-030	-6.125742274543099e-017

Table 2.1: Accuracy of computed angles

above that the canonical correlation itself does not show any improvement in correlation when d is smaller than 10^{-8} in double precision.

In the next section, we consider a formula [5] that directly computes the sine of principal angles as in (2.6).

2.3 Properties of Principal Angles and a Sine-Based Algorithm

We first review known sine–based formulas for the largest principal angle. Results of [1, 30] concerning the aperture of two linear manifolds give

$$\|P_F - P_G\| = \max\left\{\max_{x\in\mathcal{G},\|x\|=1} \|(I-P_F)x\|,\ \max_{y\in\mathcal{F},\|y\|=1} \|(I-P_G)y\|\right\}. \tag{2.7}$$

Let columns of matrices $Q_F \in \mathbf{R^{n\times p}}$ and $Q_G \in \mathbf{R^{n\times q}}$ form orthonormal bases for the subspaces $\mathcal{F}$ and $\mathcal{G}$, respectively. Then orthogonal projectors on $\mathcal{F}$ and $\mathcal{G}$ are $P_F = Q_F Q_F^T$ and $P_G = Q_G Q_G^T$, respectively, and

$$\|P_F - P_G\| = \max\{\|(I - Q_F Q_F^T)Q_G\|, \|(I - Q_G Q_G^T)Q_F\|\}. \tag{2.8}$$

If $p \neq q$, then expression of (2.8) is always equal to one; e.g., if $p > q$, then the second term under the maximum is one. If $p = q$, then both terms are the same and yield $\sin(\theta_q)$ by (2.5). Therefore, under our assumption $p \geq q$, only the first term is interesting to analyze. We note that the first term is the largest singular value of $(I - Q_F Q_F^T)Q_G$. What is the meaning of other singular values of the matrix?

This provides an insight into how to find a sine–based formulation to obtain the principal angles, which is embodied in the following theorem [5].

Theorem 2.2 *Singular values* $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_q$ *of the matrix* $(I - Q_F Q_F^T)Q_G$ *are given by* $\mu_k = \sqrt{1 - \sigma_k^2}$, $k = 1, \ldots, q$, *where* σ_k *are defined in* (2.1). *Moreover, the principal angles satisfy the equalities* $\theta_k = \arcsin(\mu_k)$.

The right principal vectors can be computed as

$$v_k = Q_G z_k, \qquad k = 1, \ldots, q,$$

where z_k *are corresponding orthonormal right singular vectors of matrix* $(I - Q_F Q_F^T)Q_G$. *The left principal vectors are then computed by*

$$u_k = Q_F Q_F^T v_k / \sigma_k \quad \text{if } \sigma_k \neq 0, \quad k = 1, \ldots, q.$$

Proof: Our proof is essentially the same as that of [5]. We reproduce it here for completeness as we use a similar proof later for a general scalar product.

Let $B = (I - P_F)Q_G = (I - Q_F Q_F^T)Q_G$. Using the fact that $I - P_F$ is a projector and that $Q_G^T Q_G = I$, we have

$$\begin{aligned} B^T B &= Q_G^T (I - P_F)(I - P_F) Q_G = Q_G^T (I - P_F) Q_G \\ &= I - Q_G^T Q_F Q_F^T Q_G. \end{aligned}$$

Utilizing the SVD (2.1), we obtain $Q_F^T Q_G = Y \Sigma Z^T$, where $\Sigma = \text{diag}\,(\sigma_1, \sigma_2, \ldots, \sigma_q)$; then

$$Z^T B^T B Z = I - \Sigma^2 = \text{diag}\,(1 - \sigma_1^2, 1 - \sigma_2^2, \ldots, 1 - \sigma_q^2).$$

Thus, the singular values of B are given by $\mu_k = \sqrt{1 - \sigma_k^2},\ k = 1, \ldots, q$, and the formula for the principal angles $\theta_k = \arcsin(\mu_k)$ follows directly from (2.2).

■

We can now use the theorem to formulate an algorithm for computing all the principal angles. This approach meets our goal of a sine–based formulation, which should provide accurate computation of small angles. However, for large angles we keep the cosine–based algorithm.

```
ALGORITHM 2.1: SUBSPACE.m

Input: real matrices $F$ and $G$ with the same number of rows.
  1. Compute orthonormal bases $Q_F = \mathrm{orth}(F)$, $Q_G = \mathrm{orth}(G)$ of column-spaces
     of $F$ and $G$.
  2. Compute SVD for cosine: $[Y, \Sigma, Z] = \mathrm{svd}(Q_F^T Q_G)$, $\Sigma = \mathrm{diag}\,(\sigma_1, \ldots, \sigma_q)$.
  3. Compute matrices of left $U_{\cos} = Q_F Y$ and right $V_{\cos} = Q_G Z$ principal
     vectors.
  4. Compute matrix $B = \begin{cases} Q_G - Q_F(Q_F^T Q_G) & \textit{if } \mathrm{diag}\,(Q_F) \geq \mathrm{diag}\,(Q_G); \\ Q_F - Q_G(Q_G^T Q_F) & \textit{otherwise.} \end{cases}$
  5. Compute SVD for sine: $[Y, \mathrm{diag}\,(\mu_1, \ldots, \mu_q), Z] = \mathrm{svd}(B)$.
  6. Compute matrices $U_{\sin}$ and $V_{\sin}$ of left and right principal vectors:
       $V_{\sin} = Q_G Z$, $U_{\sin} = Q_F(Q_F^T V_{\sin})\Sigma^{-1}$   if $\mathrm{diag}\,(Q_F) \geq \mathrm{diag}\,(Q_G)$;
       $U_{\sin} = Q_F Z$, $V_{\sin} = Q_G(Q_G^T U_{\sin})\Sigma^{-1}$   otherwise.
  7. Compute the principal angles, for $k = 1, \ldots, q$:
       $\theta_k = \begin{cases} \arccos(\sigma_k) & \textit{if} \ \ \sigma_k^2 < 1/2; \\ \arcsin(\mu_k) & \textit{if} \ \ \mu_k^2 \leq 1/2. \end{cases}$
  8. Form matrices $U$ and $V$ by picking up corresponding columns of $U_{\sin}$, $V_{\sin}$
     and $U_{\cos}$, $V_{\cos}$, according to the choice for $\theta_k$ above.
Output: Principal angles $\theta_1, \ldots, \theta_q$ between column-spaces of matrices $F$ and $G$,
  and corresponding matrices $U$ and $V$ of left and right principal vectors,
  respectively.
```

Remark 2.1 *In step* 1 *of the algorithm, the orthogonalization can be performed using the QR method or the SVD. In our actual code, an SVD–based built-in MATLAB function ORTH.m is used for the orthogonalization.*

It is pointed out in [16] that errors in computing Q_F and Q_G, especially expected for ill-conditioned F and G, may lead to an irreparable damage in final answers. A proper column scaling of F and G could in some cases significantly reduce condition numbers of F and G. We highlight that an explicit columnwise normalization of matrices F and G is not required prior to orthogonalization if a particular orthogonalization algorithm used here is invariant under column scaling in finite precision arithmetic. Our numerical tests show that the explicit column scaling is not needed if we utilize a built-in MATLAB function QR.m for orthonormalization. However, the explicit column scaling apparently helps to improve the accuracy when the SVD–based built-in MATLAB function ORTH.m is used for orthonormalization. In [16], QR factorizations with complete pivoting are recommended for computing Q_F and Q_G.

Remark 2.2 *A check of* $\text{diag}\ (Q_F) \geq \text{diag}\ (Q_G)$ *in steps* 4 *and* 6 *of the algorithm removes the need for our assumption* $p = \text{diag}\ (Q_F) \geq \text{diag}\ (Q_G) = q$.

Remark 2.3 *We replace here in step* 4

$$(I - Q_F Q_F^T) Q_G = Q_G - Q_F (Q_F^T Q_G), \quad (I - Q_G Q_G^T) Q_F = Q_F - Q_G (Q_G^T Q_F)$$

to avoid storing in memory any n-by-n matrices in the algorithm, which allows us to compute principal angles efficiently for large $n \gg p$ as well.

If the matrix $Q_{F^\perp}$, with orthonormal columns that span the subspace $\mathcal{F}^\perp$, the orthogonal complement of $\mathcal{F}$, was available to us when $p \geq q$, we could take here

$$B = Q_{F^\perp} Q_G,$$

as in the CS decomposition methods; see [50, 54, 51, 47]. *Under our assumption $n \gg p$, however, the matrix $Q_{F^\perp}$ is essentially of the size n and thus shall be* *avoided.*

The 1/2 threshold used in Algorithm 2.1 in steps 7 and 8 to separate small and large principal angles and corresponding vectors seems to be a natural choice. However, such an artificial fixed threshold may cause troubles with orthogonality in the resulting choice of vectors in step 8 if there are several angles close to each other but on different sides of the threshold. The problem is that the corresponding principal vectors, picked up from two orthogonal sets computed by different algorithms, may not be orthogonal. A more accurate approach would be to identify such possible cluster of principal angles around the original threshold and to make sure that all principal vectors corresponding to the cluster are chosen according to either step 3, or step 6, but not both.

```
ALGORITHM 2.2: Modified and Improved SUBSPACE.m

Input: real matrices F and G with the same number of rows.
 1. Compute orthonormal bases $Q_F = \mathrm{orth}(F)$, $Q_G = \mathrm{orth}(G)$ of column-spaces
    of $F$ and $G$.
 2. Compute SVD for cosine: $[Y, \Sigma, Z] = \mathrm{svd}(Q_F^T Q_G)$, $\Sigma = \mathrm{diag}\ (\sigma_1, \ldots, \sigma_q)$.
 3. Compute matrices of left $U_{\cos} = Q_F Y$ and right $V_{\cos} = Q_G Z$ principal
    vectors.
 4. Compute large principal angles, for $k = 1, \ldots, q$:
        $\theta_k = \arccos(\sigma_k)$ if $\sigma_k^2 < 1/2$.
 5. Form parts of matrices $U$ and $V$ by picking up corresponding columns of
    $U_{\cos}$, $V_{\cos}$, according to the choice for $\theta_k$ above. Put columns of $U_{\cos}$, $V_{\cos}$,
    which are left, in matrices $R_F$ and $R_G$. Collect the corresponding $\sigma$'s in a
    diagonal matrix $\Sigma_R$.
 6. Compute the matrix $B = R_G - Q_F(Q_F^T R_G)$.
 7. Compute SVD for sine: $[Y, \mathrm{diag}\ (\mu_1, \ldots, \mu_q), Z] = \mathrm{svd}(B)$.
 8. Compute matrices $U_{\sin}$ and $V_{\sin}$ of left and right principal vectors:
        $V_{\sin} = R_G Z$, $U_{\sin} = R_F(R_F^T V_{\sin})\Sigma_R^{-1}$.
 9. Recompute the small principal angles, for $k = 1, \ldots, q$:
        $\theta_k = \arcsin(\mu_k)$ if $\mu_k^2 \le 1/2$.
10. Complete matrices $U$ and $V$ by adding columns of $U_{\sin}$, $V_{\sin}$.
Output: Principal angles $\theta_1, \ldots, \theta_q$ between column-spaces of matrices $F$ and
    $G$, and corresponding matrices $U$ and $V$ of left and right principal vectors,
    respectively.
```

Let us repeat that, in exact arithmetic, the sine and cosine based approaches give the same results; e.g., columns of $U_{\sin}$ and $V_{\sin}$ must be the same as those of $U_{\cos}$ and $V_{\cos}$. Why do we need to recompute essentially the same vectors a second time? What if we compute only $U_{\cos}$, $V_{\cos}$ and then recompute just small principal angles using, e.g., the obvious formula

$$\mu_k = \|u_k - \sigma_k v_k\|? \tag{2.9}$$

This approach was discussed in [40]. It was suggested that it would resolve the inaccuracy in the cosine–based algorithm illustrated in the previous section, without the need for the second SVD.

The answer is that the cosine–based algorithm fails to compute accurately not only the small principal angles but also the corresponding principal vectors. The reason for this is that singular values computed in step 2 of Algorithm 2.1 are the cosines of principal angles, while singular values of the matrix B in step 5 are the sines of principal angles. Thus, the distribution of singular values is different in steps 2 and 5; e.g., singular values corresponding to small angles are much better separated in step 5 than in step 2. For example, angles 10^{-10} and 10^{-12} will produce a multiple singular value 1 in step 2 in double precision but will produce two distinct small singular values in step 5. This means that singular vectors, corresponding to small principal angles, might not be computed accurately in computer arithmetic using only SVD in step 2, which will also lead to inaccurate computation of the small principal angles by formula (2.9). Our numerical tests support this conclusion.

There is some obvious redundancy in Algorithm 2.1. Indeed, we do not need to calculate columns of U_{sin} and V_{sin}, corresponding to large sines, and columns of U_{cos} and V_{cos}, corresponding to large cosines, as they are computed inaccurately in computer arithmetic and we just discard them later in the algorithm. However, first, for large-scale applications with $n \gg p$ that we are interested in, the redundancy is insignificant. Second, this allows us to perform steps 2–3 and steps 4–5 of Algorithm 2.1 independently in parallel. We note that Σ must be invertible in Algorithm 2.1.

For sequential computations, we now describe Algorithm 2.2. Here, we reduce computational costs of the second SVD by using already computed vectors $U_{\cos}$ and $V_{\cos}$ for the cosines. The cosine–based algorithm computes inaccurately individual principal vectors corresponding to small principal angles. However, it may find accurately the corresponding invariant subspaces spanned by all these vectors. Thus, the idea is that, using $U_{\cos}$ and $V_{\cos}$, we can identify invariant subspaces in $\mathcal{F}$ and $\mathcal{G}$, which correspond to all small principal angles. Then, we perform the second SVD only for these subspaces, computing only columns of $U_{\sin}$ and $V_{\sin}$ that we actually need, which may significantly reduce the size of the matrix in the second SVD. This idea is used in the CS decomposition algorithm of [54].

We keep steps 1–3 of Algorithm 2.1 unchanged but modify accordingly later steps to obtain Algorithm 2.2. Such changes may significantly reduce the size of matrix B and, thus, the costs, if large principal angles outnumber small ones; e.g., if there are no small principal angles at all, the algorithm simply stops at step 3. We note that matrix Σ_R is always invertible, unlike matrix Σ.

Remark 2.4 *By construction, matrices R_F and R_G have the same number of already orthogonal columns, which removes the need for orthogonalization and for comparing, in step* 6*, their ranks.*

Remark 2.5 *We have three, equivalent in exact arithmetic, possibilities to com-*

pute matrix B:

$$B = (I - R_F R_F^T) R_G = R_G - R_F(R_F^T R_G) = R_G - Q_F(Q_F^T R_G).$$

The first formula is ruled out to avoid storing in memory any n-by-n matrices in the algorithm. Our numerical tests show that the third expression, though somewhat more expensive than the second one, often provides more accurate results in the presence of round-off errors.

To summarize, Algorithms 2.1 and 2.2 use the cosine–based formulation (2.1), (2.2) for large angles and the sine–based formulation of Theorem 2.2 for small angles, which allows accurate computation of all angles. The algorithms are reasonably efficient for large-scale applications with $n \gg p$ and are more robust than the original cosine–based only version.

In the rest of the section, we describe some useful properties of principal angles not yet mentioned. In this investigation, we follow [33] and make use of an orthogonal projectors technique [25]. For an alternative approach, popular in matrix theory, which is based on representation of subspaces in a canonical CS form, we refer to [51].

Theorem 2.2 characterizes singular values of the product $(I - P_F)Q_G$, which are the sine of the principal angles. What are singular values of the matrix $P_F Q_G$? A trivial modification of the previous proof leads to the following not really surprising result that these are the cosine of the principal angles.

Theorem 2.3 *Singular values of the matrix $Q_F Q_F^T Q_G$ are exactly the same as σ_k, defined in* (2.1).

We conclude this section with other simple and known (e.g., [55, 51, 56]) sine and cosine representations of principal angles and principal vectors, this time using orthogonal projectors P_F and P_G on subspaces $\mathcal{F}$ and $\mathcal{G}$, respectively.

Theorem 2.4 *Let assumptions of Theorem* 2.2 *be satisfied. Then $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q$ are the q largest singular values of the matrix $P_F P_G$; in particular,*

$$\sigma_1 = \|P_F P_G\|.$$

Other $n-q$ singular values are all equal to zero.

Remark 2.6 *As singular values of $P_F P_G$ are the same as those of $P_G P_F$, subspaces $\mathcal{F}$ and $\mathcal{G}$ play symmetric roles in Theorem* 2.4*; thus, our assumption that $p = \dim\mathcal{F} \geq \dim\mathcal{G} = q$ is irrelevant here.*

Theorem 2.5 *Let assumptions of Theorem* 2.2 *be satisfied. Then $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_q$ are the q largest singular values of the matrix $(I-P_F)P_G$; in particular,*

$$\mu_q = \|(I-P_F)P_G\|.$$

Other $n-q$ singular values are all equal to zero.

Remark 2.7 *Comparing Theorems* 2.4 *and* 2.5 *shows trivially that sine of principal angles between $\mathcal{F}$ and $\mathcal{G}$ are the same as cosine of principal angles between $\mathcal{F}^\perp$ and $\mathcal{G}$ because $I - P_F$ is an orthogonal projector on $\mathcal{F}^\perp$. If $p > n/2 > q$, it may be cheaper to compute principal angles between $\mathcal{F}^\perp$ and $\mathcal{G}$ instead of principal angles between $\mathcal{F}$ and $\mathcal{G}$.*

What can we say about singular values of the matrix $(I - P_G)P_F$? In other words, how do cosine of principal angles between subspaces $\mathcal{F}^\perp$ and $\mathcal{G}$ compare to cosine of principal angles between their orthogonal complements $\mathcal{F}$ and $\mathcal{G}^\perp$? If $p = q$, they are absolutely the same; in particular, the minimal angle between subspaces $\mathcal{F}^\perp$ and $\mathcal{G}$ is in this case the same as the minimal angle between their orthogonal complements $\mathcal{F}$ and $\mathcal{G}^\perp$, e.g., in [14], and, in fact, is equal to $\text{gap}(\mathcal{F}, \mathcal{G}) = \|P_F - P_G\|$ as we already discussed. When $p > q$, subspaces $\mathcal{F}$ and $\mathcal{G}^\perp$ must have a nontrivial intersection because the sum of their dimensions is too big; thus, the minimal angle between subspaces $\mathcal{F}$ and $\mathcal{G}^\perp$ must be zero in this case, which corresponds to $\|(I - P_G)P_F\| = 1$, while $\|(I - P_F)P_G\|$ may be less than one. To be more specific, $\dim(\mathcal{F} \cap \mathcal{G}^\perp) \geq p - q$; thus, at least $p - q$ singular values of the matrix $(I - P_G)P_F$ are equal to one. Then, we have the following statement, which completely clarifies the issue of principal angles between orthogonal complements; cf. Ex. 1.2.6 of [7].

Theorem 2.6 *The set of singular values of $(I - P_G)P_F$, when $p > q$, consists of $p - q$ ones, q singular values of $(I - P_F)P_G$, and $n - p$ zeros.*

In particular, this shows that the smallest positive sine of principal angles between $\mathcal{F}$ and $\mathcal{G}$, called the *minimum gap*, is the same as that between $\mathcal{F}^\perp$ and $\mathcal{G}^\perp$; see [30]. This theorem can also be used to reduce the costs of computing the principal angles between subspaces $\mathcal{F}$ and $\mathcal{G}$, when their dimensions p and q are greater than $n/2$, by replacing $\mathcal{F}$ and $\mathcal{G}$ with their orthogonal complements.

Let us finally mention a simple property of principal vectors, emphasized in [55], which helps us to understand a geometric meaning of pairs of corresponding principal vectors from different subspaces.

Theorem 2.7 *We have*

$$P_F v_k = \sigma_k u_k, \quad P_G u_k = \sigma_k v_k, \quad k = 1, \ldots, q,$$

and

$$u_i^T v_j = (P_F u_i)^T v_j = u_i^T P_F v_j = \sigma_j u_i^T u_j = \sigma_j \delta_{ij}, \qquad i, j = 1, \ldots, q.$$

In other words, a chosen pair u_k, v_k spans a subspace, invariant with respect to orthoprojectors P_F and P_G and orthogonal to all other such subspaces. The kth principal angle θ_k is simply the angle between u_k and v_k; see (2.9)*.*

Moreover, the subspace $\operatorname{span}\{u_k, v_k\}$ *is also invariant with respect to orthoprojectors $I - P_F$ and $I - P_G$. Let us define two other unit vectors in this subspace:*

$$u_k^\perp = (v_k - \sigma_k u_k)/\mu_k \in \mathcal{F}^\perp, \quad v_k^\perp = (u_k - \sigma_k v_k)/\mu_k \in \mathcal{G}^\perp$$

such that $u_k^T u_k^\perp = v_k^T v_k^\perp = 0$. Then

- *u_k, v_k are principal vectors for subspaces $\mathcal{F}$ and $\mathcal{G}$;*
- *$u_k^\perp, v_k$ are principal vectors for subspaces $\mathcal{F}^\perp$ and $\mathcal{G}$;*
- *$u_k, v_k^\perp$ are principal vectors for subspaces $\mathcal{F}$ and $\mathcal{G}^\perp$;*
- *$u_k^\perp, -v_k^\perp$ are principal vectors for subspaces $\mathcal{F}^\perp$ and $\mathcal{G}^\perp$,*

which concludes the description of all cases.

In the next section, we deal with an arbitrary scalar product.

2.4 Generalization to an A–Based Scalar Product

Let $A \in \mathbf{R}^{\mathbf{n}\times\mathbf{n}}$ be a fixed symmetric positive definite matrix. Let $(x, y)_A = (x, Ay) = y^T Ax$ be an A–based scalar product, $x, y \in \mathbf{R}^{\mathbf{n}}$. Let $\|x\|_A = \sqrt{(x,x)_A}$ be the corresponding vector norm and let $\|B\|_A$ be the corresponding induced matrix norm of a matrix $B \in \mathbf{R}^{\mathbf{n}\times\mathbf{n}}$. We note that $\|x\|_A = \|A^{1/2}x\|$ and $\|B\|_A = \|A^{1/2}BA^{-1/2}\|$.

In order to define principal angles based on this scalar product, we will follow arguments of [5, 21] but in an A–based scalar product instead of the standard Euclidean scalar product. Again, we will assume for simplicity of notation that $p \geq q$.

Principal angles

$$\theta_1, \ldots, \theta_q \in [0, \pi/2]$$

between subspaces $\mathcal{F}$ and $\mathcal{G}$ in the A–based scalar product $(\cdot,\cdot)_A$ are defined recursively for $k = 1, \ldots, q$ by analogy with the previous definition for $A = I$ as

$$\cos(\theta_k) = \max_{u\ \in\mathcal{F}} \max_{v\ \in\mathcal{G}}\ (u, v)_A\ = (u_k, v_k)_A \tag{2.10}$$

subject to

$$\|u\|_A = \|v\|_A = 1, \quad (u, u_i)_A = 0, \quad (v, v_i)_A = 0, \quad i = 1, \ldots, k-1. \tag{2.11}$$

The vectors $u_1, \ldots, u_q$ and $v_1, \ldots, v_q$ are called principal vectors relative to the A–based scalar product.

The following theorem justifies the consistency of the definition above and provides a cosine–based algorithm for computing the principal angles in the

A–based scalar product. It is a direct generalization of the cosine–based approach of [5, 21].

Theorem 2.8 *Let columns of* $Q_F \in \mathbf{R^{n\times p}}$ *and* $Q_G \in \mathbf{R^{n\times q}}$ *now be A-orthonormal bases for the subspaces* F *and* G*, respectively. Let* $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q$ *be singular values of* $Q_F^T A Q_G$ *with corresponding left and right singular vectors* y_k *and* z_k*,* $k = 1, \ldots, q$*. Then the principal angles relative to the scalar product* $(\cdot,\cdot)_A$ *as defined in* (2.10) *and* (2.11) *are computed as*

$$\theta_k = \arccos(\sigma_k), \qquad k = 1, \ldots, q, \tag{2.12}$$

where

$$0 \leq \theta_1 \leq \cdots \leq \theta_q \leq \frac{\pi}{2},$$

while the principal vectors are given by

$$u_k = Q_F y_k, \quad v_k = Q_G z_k, \quad k = 1, \ldots, q.$$

Proof: We first rewrite definition (2.10) and (2.11) of principal angles in the following equivalent form. For $k = 1, \ldots, q$,

$$\cos(\theta_k) = \max_{y\,\in \mathbf{R^p}} \max_{z\,\in \mathbf{R^q}} y^T Q_F^T A Q_G z \;=\; y_k^T Q_F^T A Q_G z_k$$

subject to

$$\|y\| = \|z\| = 1, \quad y^T y_i = 0, \quad z^T z_i = 0, \quad i = 1, \ldots, k-1,$$

where $u = Q_F y \in \mathcal{F}$, $v = Q_G z \in \mathcal{G}$ and $u_k = Q_F y_k \in \mathcal{F}$, $v_k = Q_G z_k \in \mathcal{G}$.

Since Q_F and Q_G have A-orthonormal columns, $Q_F^T A Q_F = I$ and $Q_G^T A Q_G = I$. This implies

$$\|u\|_A^2 = y^T Q_F^T A Q_F y = y^T y = \|y\|^2 = 1$$

and

$$\|v\|_A^2 = z^T Q_G^T A Q_G z = z^T z = \|z\|^2 = 1.$$

For $i \neq j$, we derive

$$(u_i, u_j)_A = y_i^T Q_F^T A Q_F y_j = y_i^T y_j = 0$$

and

$$(v_i, v_j)_A = z_i^T Q_G^T A Q_G z_j = z_i^T z_j = 0.$$

Now, let the reduced SVD of $Q_F^T A Q_G$ be

$$Y^T Q_F^T A Q_G Z \;=\; \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_q), \tag{2.13}$$

where $Y \in \mathbf{R}^{\mathbf{p}\times\mathbf{q}}$, $Z \in \mathbf{R}^{\mathbf{q}\times\mathbf{q}}$ both have orthonormal columns.

Then, by Theorem 2.1 with $M = Q_F^T A Q_G$, the equality $\cos(\theta_k) = \sigma_k,\ k = 1, \ldots, q$, just provides two equivalent representations of the singular values of $Q_F^T A Q_G$, and y_z and z_k can be chosen as columns of matrices Y and Z, respectively. The statement of the theorem follows. ■

Let us now make a trivial but important observation that links principal angles in the A–based scalar product with principal angles in the original standard scalar product, when a factorization of $A = K^T K$, e.g., $K = A^{1/2}$, is available. We formulate it as the following theorem.

Theorem 2.9 *Let $A = K^T K$. Under assumptions of Theorem 2.8 the principal angles between subspaces $\mathcal{F}$ and $\mathcal{G}$ relative to the scalar product $(\cdot,\cdot)_A$ coincide with the principal angles between subspaces $K\mathcal{F}$ and $K\mathcal{G}$ relative to the original scalar product $(\cdot,\cdot)$.*

Proof: One way to prove this is to notice that our definition of the principal angles between subspaces $\mathcal{F}$ and $\mathcal{G}$ relative to the scalar product $(\cdot,\cdot)_A$ turns into a definition of the principal angles between subspaces $K\mathcal{F}$ and $K\mathcal{G}$ relative to the original scalar product $(\cdot,\cdot)$ if we make substitutions $Ku \mapsto u$ and $Kv \mapsto v$.

Another proof is to use the representation

$$Q_F^T A Q_G = (KQ_F)^T KQ_G,$$

where columns of matrices KQ_F and KQ_G are orthonormal with respect to the original scalar product $(\cdot,\cdot)$ and span subspaces $K\mathcal{F}$ and $K\mathcal{G}$, respectively. Now Theorem 2.8 is equivalent to the traditional SVD theorem on cosine of principal angles between subspaces $K\mathcal{F}$ and $K\mathcal{G}$ relative to the original scalar product $(\cdot,\cdot)$, formulated in the introduction. ■

The A-orthogonal projectors on subspaces $\mathcal{F}$ and $\mathcal{G}$ are now defined by formulas

$$P_F = Q_F Q_F^{*_A} = Q_F Q_F^T A \text{ and } P_G = Q_G Q_G^{*_A} = Q_G Q_G^T A,$$

where $*_A$ denotes the A-adjoint.

To obtain a sine–based formulation in the A–based scalar product that is accurate for small angles, we first adjust (2.5) and (2.7) to the new A–based scalar product:

$$\begin{aligned}\operatorname{gap}_A(\mathcal{F},\mathcal{G}) &= \|P_F - P_G\|_A \\ &= \max\left\{\max_{x\in\mathcal{G},\|x\|_A=1} \|(I-P_F)x\|_A,\ \max_{y\in\mathcal{F},\|y\|_A=1} \|(I-P_G)y\|_A\right\}. \quad (2.14)\end{aligned}$$

If $p = q$, this equation will yield $\sin(\theta_q)$, consistent with Theorem 2.8. Similar to the previous case $A = I$, only the first term under the maximum is of interest under our assumption that $p \geq q$. Using the fact that

$$\|x\|_A = \|Q_G z\|_A = \|z\| \ \forall x \in \mathcal{G}, \qquad x = Q_G z, \, z \in \mathbf{R^q},$$

the term of interest can be rewritten as

$$\max_{x \in \mathcal{G}, \|x\|_A = 1} \|(I - P_F)x\|_A = \|A^{1/2}(I - Q_F Q_F^T A) Q_G\|. \tag{2.15}$$

Here we use the standard induced Euclidean norm $\|\cdot\|$ for computational purposes. Similar to our arguments in the previous section, we obtain a more general formula for all principal angles in the following.

Theorem 2.10 *Let $S = (I - Q_F Q_F^T A) Q_G$. Singular values $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_q$ of matrix $A^{1/2}S$ are given by $\mu_k = \sqrt{1 - \sigma_k^2}$, $k = 1, \ldots, q$, where σ_k are defined in* (2.13)*. Moreover, the principal angles satisfy the equalities $\theta_k = \arcsin(\mu_k)$. The right principal vectors can be computed as*

$$v_k = Q_G z_k, \qquad k = 1, \ldots, q,$$

where z_k are corresponding orthonormal right singular vectors of matrix $A^{1/2}S$. The left principal vectors are then computed by

$$u_k = Q_F Q_F^T A v_k / \sigma_k \quad \text{if } \sigma_k \neq 0, \quad k = 1, \ldots, q.$$

Proof: We first notice that squares of the singular values μ_k of the matrix $A^{1/2}S$, which appear in (2.15), coincide with eigenvalues $\nu_k = \mu_k^2$ of the product

S^TAS. Using the fact that $Q_F^TAQ_F = I$ and $Q_G^TAQ_G = I$, we have

$$\begin{aligned} S^TAS &= Q_G^T(I - AQ_FQ_F^T)A(I - Q_FQ_F^TA)Q_G \\ &= I - Q_G^TAQ_FQ_F^TAQ_G. \end{aligned}$$

Utilizing the SVD (2.13), we obtain $Q_F^TAQ_G = Y\Sigma Z^T$, where $\Sigma = \text{diag}\,(\sigma_1, \sigma_2, \ldots, \sigma_q)$; then

$$Z^TS^TASZ = I - \Sigma^2 = \text{diag}\,(1-\sigma_1^2, 1-\sigma_2^2, \ldots, 1-\sigma_q^2).$$

Thus, the eigenvalues of S^TAS are given by $\nu_k = 1 - \sigma_k^2,\ k = 1, \ldots, q$, and the formula for the principal angles follows directly from (2.12). ■

For computational reasons, when n is large, we need to avoid dealing with the square root $A^{1/2}$ explicitly. Also, A may not be available as a matrix but only as a function performing the multiplication of A by a given vector. Fortunately, the previous theorem can be trivially reformulated as follows to resolve this issue.

Theorem 2.11 *Eigenvalues* $\nu_1 \leq \nu_2 \leq \cdots \leq \nu_q$ *of matrix* S^TAS, *where* $S = (I - Q_FQ_F^TA)Q_G$, *are equal to* $\nu_k = 1 - \sigma_k^2,\ k = 1, \ldots, q$, *where* σ_k *are defined in* (2.13). *Moreover, the principal angles satisfy the equalities* $\theta_k = \arcsin\left(\sqrt{\nu_k}\right),\ k = 1, \ldots, q$. *The right principal vectors can be computed as*

$$v_k = Q_Gz_k, \qquad k = 1, \ldots, q,$$

where z_k *are corresponding orthonormal right eigenvectors of matrix* S^TAS. *The left principal vectors are then computed by*

$$u_k = Q_FQ_F^TAv_k/\sigma_k \quad \text{if } \sigma_k \neq 0, \quad k = 1, \ldots, q.$$

We can easily modify the previous proof to obtain the following analogue of Theorem 2.3.

Theorem 2.12 *Singular values of the matrix* $A^{1/2}Q_F Q_F^T A Q_G = A^{1/2}P_F Q_G$ *coincide with* σ_k*, defined in* (2.13).

It is also useful to represent principal angles using exclusively A-orthogonal projectors P_F and P_G on subspaces $\mathcal{F}$ and $\mathcal{G}$, respectively, similarly to Theorems 2.4 and 2.5.

Theorem 2.13 *Under assumptions of Theorem* 2.8*,* $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q$ *are the* q *largest singular values of the matrix* $A^{1/2}P_F P_G A^{-1/2}$*; in particular,*

$$\sigma_1 = \|P_F P_G\|_A.$$

Other $n-q$ *singular values are all equal to zero.*

Proof: First, we rewrite

$$\begin{aligned} A^{1/2}P_F P_G A^{-1/2} &= A^{1/2}Q_F Q_F^T A Q_G Q_G^T A A^{-1/2} \\ &= A^{1/2}Q_F \left(A^{1/2}Q_F\right)^T A^{1/2}Q_G \left(A^{1/2}Q_G\right)^T. \end{aligned}$$

As columns of matrices $A^{1/2}Q_F$ and $A^{1/2}Q_G$ are orthonormal with respect to the original scalar product $(\cdot,\cdot)$ bases of subspaces $A^{1/2}\mathcal{F}$ and $A^{1/2}\mathcal{G}$, respectively, the last product is equal to the product of orthogonal (not A-orthogonal!) projectors $P_{A^{1/2}\mathcal{F}}$ and $P_{A^{1/2}\mathcal{G}}$ on subspaces $A^{1/2}\mathcal{F}$ and $A^{1/2}\mathcal{G}$.

Second, we can now use Theorem 2.4 to state that cosine of principal angles between subspaces $A^{1/2}\mathcal{F}$ and $A^{1/2}\mathcal{G}$ with respect to the original scalar product

$(\cdot,\cdot)$ are given by the q largest singular values of the product $P_{A^{1/2}\mathcal{F}}P_{A^{1/2}\mathcal{G}} = A^{1/2}P_F P_G A^{-1/2}$.

Finally, we use Theorem 2.9 to conclude that these singular values are, in fact, $\sigma_k,\ k = 1,\ldots,q$, i.e., the cosine of principal angles between subspaces $\mathcal{F}$ and $\mathcal{G}$ with respect to the A–based scalar product. ■

Theorem 2.14 *Let assumptions of Theorem* 2.11 *be satisfied. Then* $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_q$ *are the* q *largest singular values of the matrix* $A^{1/2}(I - P_F)P_G A^{-1/2}$*; in particular,*

$$\mu_q = \|(I - P_F)P_G\|_A.$$

The other $n - q$ *singular values are all equal to zero.*

Proof: We rewrite

$$\begin{aligned} A^{1/2}(I - P_F)P_G A^{-1/2} &= \left(I - A^{1/2}Q_F\left(A^{1/2}Q_F\right)^T\right)A^{1/2}Q_G\left(A^{1/2}Q_G\right)^T \\ &= \left(I - P_{A^{1/2}\mathcal{F}}\right)P_{A^{1/2}\mathcal{G}} \end{aligned}$$

and then follow arguments similar to those of the previous proof, but now using Theorem 2.5 instead of Theorem 2.4. ■

Remarks 2.6–2.7 and Theorems 2.6–2.7 for the case $A = I$ hold in the general case, too, with obvious modifications.

Our final theoretical results are perturbation theorems in the next section.

2.5 Perturbation of Principal Angles in the A–Based Scalar Product

In the present section, for simplicity, we *always assume* that matrices F, G and their perturbations $\tilde{F}$, $\tilde{G}$ have the same rank; thus, in particular, $p = q$.

We notice that F and G appear symmetrically in the definition of the principal angles, under our assumption that they and their perturbations have the same rank. This means that we do not have to analyze the perturbation of F and G together at the same time. Instead, we first study only a perturbation in G.

Before we start with an estimate for cosine, let us introduce a new notation $\ominus$ using an example:

$$(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G} = (\mathcal{G} + \tilde{\mathcal{G}}) \cap \mathcal{G}^{\perp},$$

where $\ominus$ and the orthogonal complement to $\mathcal{G}$ are understood in the A–based scalar product.

Lemma 2.1 *Let $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q$ and $\hat{\sigma}_1 \geq \hat{\sigma}_2 \geq \cdots \geq \hat{\sigma}_q$ be cosine of principal angles between subspaces $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{F}$, $\tilde{\mathcal{G}}$, respectively, computed in the A–based scalar product. Then, for $k = 1, \ldots, q$,*

$$\begin{aligned} |\sigma_k - \hat{\sigma}_k| \;\leq\; & \max\bigl[\cos(\theta_{\min}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G},\, \mathcal{F}\}); \\ & \cos(\theta_{\min}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \tilde{\mathcal{G}},\, \mathcal{F}\})\bigr] \operatorname{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}), \end{aligned} \tag{2.16}$$

where $\theta_{\min}$ is the smallest angle between corresponding subspaces, measured in the A–based scalar product.

Proof: The proof is based on the following identity:

$$A^{1/2}Q_FQ_F^TAQ_{\tilde{G}} = A^{1/2}Q_FQ_F^TAQ_GQ_G^TAQ_{\tilde{G}} + A^{1/2}Q_FQ_F^TA(I - Q_GQ_G^TA)Q_{\tilde{G}}, \tag{2.17}$$

which is a multidimensional analogue of the trigonometric formula for the cosine of the sum of two angles. Now we use two classical theorems on perturbation of singular values with respect to addition:

$$s_k(T + S) \leq s_k(T) + \|S\|, \tag{2.18}$$

and with respect to multiplication:

$$s_k(TS^T) \leq s_k(T)\|S^T\|, \tag{2.19}$$

where T and S are matrices of corresponding sizes. We first need to take $T = A^{1/2}Q_FQ_F^TAQ_GQ_G^TAQ_{\tilde{G}}$ and $S = A^{1/2}Q_FQ_F^TA(I - Q_GQ_G^TA)Q_{\tilde{G}}$ in (2.18) to get

$$\begin{aligned}\hat{\sigma}_k = s_k(A^{1/2}Q_FQ_F^TAQ_{\tilde{G}}) \ &\leq \ s_k(A^{1/2}Q_FQ_F^TAQ_GQ_G^TAQ_{\tilde{G}}) \\ &\quad + \|A^{1/2}Q_FQ_F^TA(I - Q_GQ_G^TA)Q_{\tilde{G}}\|,\end{aligned}$$

where the first equality follows from Theorem 2.12. In the second term in the sum on the right, we need to estimate a product, similar to a product of *three* orthoprojectors. We notice that column vectors of $(I - Q_GQ_G^TA)Q_{\tilde{G}}$ belong to the subspace $(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G}$. Let $P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}$ be an A-orthogonal projector on the subspace. Then the second term can be rewritten, also using the projector $Q_FQ_F^TA = P_F$, as

$$A^{1/2}Q_FQ_F^TA(I - Q_GQ_G^TA)Q_{\tilde{G}} = A^{1/2}Q_FQ_F^TAP_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}(I - Q_GQ_G^TA)Q_{\tilde{G}}$$

$$= \left(A^{1/2} P_F P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}} A^{-1/2}\right) A^{1/2}(I - Q_G Q_G^T A) Q_{\tilde{G}};$$

therefore, it can be estimated as

$$\|A^{1/2} Q_F Q_F^T A(I - Q_G Q_G^T A) Q_{\tilde{G}}\| \leq \\ \|A^{1/2} P_F P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}} A^{-1/2}\| \|A^{1/2}(I - Q_G Q_G^T A) Q_{\tilde{G}}\|.$$

The first multiplier in the last product equals

$$\|A^{1/2} P_F P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}} A^{-1/2}\| = \|P_F P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}\|_A = \cos(\theta_{\min}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G},\, \mathcal{F}\}),$$

similar to (2.15) and using Theorem 2.13 for subspaces $(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}$ and $\mathcal{F}$; while the second multiplier is $\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}})$, because of our assumption $\dim\mathcal{F} = \dim\mathcal{G} = \dim\tilde{\mathcal{G}}$. To estimate the first term in the sum, we apply (2.19) with $T = A^{1/2} Q_F Q_F^T A Q_G$ and $S^T = Q_G^T A Q_{\tilde{G}}$:

$$\begin{aligned} s_k(A^{1/2} Q_F Q_F^T A Q_G Q_G^T A Q_{\tilde{G}}) &\leq s_k(A^{1/2} Q_F Q_F^T A Q_G) \|Q_G^T A Q_{\tilde{G}}\| \\ &\leq s_k(A^{1/2} Q_F Q_F^T A Q_G) = \sigma_k, \end{aligned}$$

simply because the second multiplier here is the cosine of an angle between $\mathcal{G}$ and $\tilde{\mathcal{G}}$ in the A–based scalar product, which is, of course, bounded by one from above. Thus, we just proved

$$\hat{\sigma}_k \leq \sigma_k + \cos(\theta_{\min}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G},\, \mathcal{F}\})\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}}).$$

Changing places of $Q_{\tilde{G}}$ and Q_G, we obtain

$$\sigma_k \leq \hat{\sigma}_k + \cos(\theta_{\min}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\tilde{\mathcal{G}},\, \mathcal{F}\})\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}})$$

and come to the statement of the lemma. ■

Remark 2.8 *Let us try to clarify the meaning of constants appearing in the statement of Lemma* 2.1*. Let us consider, e.g.,* $\cos(\theta_{\min}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G},\,\mathcal{F}\})$*. The cosine takes its maximal value, one, when at least one direction of the perturbation of* $\mathcal{G}$ *is* A*-orthogonal to* $\mathcal{G}$ *and parallel to* $\mathcal{F}$ *at the same time. It is small, on the contrary, when a part of the perturbation,* A*-orthogonal to* $\mathcal{G}$*, is also* A*-orthogonal to* $\mathcal{F}$*. As* $(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}\subseteq\mathcal{G}^{\perp}$*, we have*

$$\begin{aligned}\cos(\theta_{\min}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G},\,\mathcal{F}\}) &\le \cos\left(\theta_{\min}\{\mathcal{G}^{\perp},\,\mathcal{F}\}\right) \qquad (2.20)\\ &= \sin\left(\theta_{\max}\{\mathcal{G},\,\mathcal{F}\}\right)\\ &= \operatorname{gap}_A(\mathcal{G},\mathcal{F}),\end{aligned}$$

which is the constant of the asymptotic perturbation estimate of [5] *(where* $A = I$*). The latter constant is small if subspaces* $\mathcal{G}$ *and* $\mathcal{F}$ *are close to each other, which can be considered more as a fortunate cancellation, as in this case cosine of all principal angles is almost one, and a perturbation estimate for the cosine does not help much because of the cancellation effect.*

Remark 2.9 *A natural approach similar to that of* [23] *with* $A = I$ *involves a simpler identity:*

$$Q_F^T A Q_{\tilde{G}} = Q_F^T A Q_G + Q_F^T A(Q_{\tilde{G}} - Q_G),$$

where a norm of the second term is then estimated. Then (2.18) *gives an estimate of singular values using* $\|A^{1/2}(Q_{\tilde{G}}-Q_G)\|$*. As singular values are invariant with respect to particular choices of matrices* $Q_{\tilde{G}}$ *and* Q_G *with* A*-orthonormal columns, as far as they provide ranges* $\tilde{\mathcal{G}}$ *and* $\mathcal{G}$*, respectively, we can choose them* *to minimize the norm of the difference, which gives*

$$\inf_Q \|A^{1/2}(Q_G - Q_{\tilde{G}}Q)\|, \tag{2.21}$$

where Q is an arbitrary q-by-q orthogonal matrix. This quantity appears in [23] *with $A = I$ as a special type of the Procrustes problem. In* [23]*, it is estimated in terms of the gap between subspaces $\tilde{\mathcal{G}}$ and $\mathcal{G}$ (using an extra assumption that $2q \le n$). Repeating similar arguments, we derive*

$$|\sigma_k - \hat{\sigma}_k| \le \inf_Q \|A^{1/2}(Q_G - Q_{\tilde{G}}Q)\|_A \le \sqrt{2}\,\mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}), \qquad k = 1, \ldots, q. \tag{2.22}$$

Lemma 2.1 *furnishes estimates of the perturbation of singular values in terms of the gap directly, which gives a much better constant, consistent with that of the asymptotic estimate of* [5] *for $A = I$; see the previous remark.*

Now we prove a separate estimate for sine.

Lemma 2.2 *Let $\mu_1 \le \mu_2 \le \cdots \le \mu_q$ and $\hat{\mu}_1 \le \hat{\mu}_2 \le \cdots \le \hat{\mu}_q$ be sine of principal angles between subspaces $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{F}$, $\tilde{\mathcal{G}}$, respectively, computed in the A–based scalar product. Then, for $k = 1, \ldots, q$,*

$$\begin{aligned} |\mu_k - \hat{\mu}_k| \;\le\; & \max\big[\sin(\theta_{\max}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G},\, \mathcal{F}\}); \\ & \sin(\theta_{\max}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \tilde{\mathcal{G}},\, \mathcal{F}\})\big]\,\mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}), \end{aligned} \tag{2.23}$$

where $\theta_{\max}$ is the largest angle between corresponding subspaces, measured in the A–based scalar product.

Proof: The proof is based on the following identity:

$$\begin{aligned} A^{1/2}(I - Q_F Q_F^T A)Q_{\tilde{G}} = {} & A^{1/2}(I - Q_F Q_F^T A)Q_G Q_G^T A Q_{\tilde{G}} \\ & + A^{1/2}(I - Q_F Q_F^T A)(I - Q_G Q_G^T A)Q_{\tilde{G}}, \end{aligned}$$

which is a multidimensional analogue of the trigonometric formula for the sine of the sum of two angles. The rest of the proof is similar to that of Lemma 2.1.

We first need to take $T = A^{1/2}(I - Q_F Q_F^T A)Q_G Q_G^T A Q_{\tilde{G}}$ and $S = A^{1/2}(I - Q_F Q_F^T A)(I - Q_G Q_G^T A)Q_{\tilde{G}}$ and use (2.18) to get

$$\begin{aligned} s_k(A^{1/2}(I - Q_F Q_F^T A)Q_{\tilde{G}}) &\le s_k(A^{1/2}(I - Q_F Q_F^T A)Q_G Q_G^T A Q_{\tilde{G}}) \\ &\quad + \|A^{1/2}(I - Q_F Q_F^T A)(I - Q_G Q_G^T A)Q_{\tilde{G}}\|. \end{aligned}$$

In the second term in the sum on the right, $Q_F Q_F^T A = P_F$ and we deduce

$$\begin{aligned} A^{1/2}(I - Q_F Q_F^T A)(I - Q_G Q_G^T A)Q_{\tilde{G}} &= A^{1/2}(I - P_F)P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}(I - Q_G Q_G^T A)Q_{\tilde{G}} \\ &= \left(A^{1/2}(I - P_F)P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}A^{-1/2}\right)\left(A^{1/2}(I - Q_G Q_G^T A)Q_{\tilde{G}}\right), \end{aligned}$$

using notation $P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}$ for the A-orthogonal projector on the subspace $(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G}$, introduced in the proof of Lemma 2.1. Therefore, the second term can be estimated as

$$\begin{aligned} \|A^{1/2}(I - Q_F Q_F^T A)(I &- Q_G Q_G^T A)Q_{\tilde{G}}\| \\ &\le \|A^{1/2}(I - P_F)P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}A^{-1/2}\|\|A^{1/2}(I - Q_G Q_G^T A)Q_{\tilde{G}}\|. \end{aligned}$$

The first multiplier is

$$\|A^{1/2}(I-P_F)P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}A^{-1/2}\| = \|(I-P_F)P_{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G}}\|_A = \sin(\theta_{\max}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G},\, \mathcal{F}\})$$

by Theorem 2.14 as $\dim\mathcal{F} \ge \dim((\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G})$, while the second multiplier is simply $\mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}})$ because of our assumption $\dim\mathcal{G} = \dim\tilde{\mathcal{G}}$.

To estimate the first term in the sum, we take with $T = A^{1/2}(I - Q_F Q_F^T A)Q_G$ and $S^T = Q_G^T A Q_{\tilde{G}}$ and apply (2.19):

$$\begin{aligned} s_k(A^{1/2}(I - Q_F Q_F^T A)Q_G Q_G^T A Q_{\tilde{G}}) &\leq s_k(A^{1/2}(I - Q_F Q_F^T A)Q_G)\|Q_G^T A Q_{\tilde{G}}\| \\ &\leq s_k(A^{1/2}(I - Q_F Q_F^T A)Q_G), \end{aligned}$$

using exactly the same arguments as in the proof of Lemma 2.1.

Thus, we have proved

$$\hat{\mu}_k \leq \mu_k + \sin(\theta_{\max}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G}, \mathcal{F}\})\mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}).$$

Changing the places of $Q_{\tilde{G}}$ and Q_G, we get

$$\mu_k \leq \hat{\mu}_k + \sin(\theta_{\max}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \tilde{\mathcal{G}}, \mathcal{F}\})\mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}).$$

The statement of the lemma follows. ■

Remark 2.10 *Let us also highlight that simpler estimates,*

$$|\mu_k - \hat{\mu}_k| \leq \mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}), \quad |\sigma_k - \hat{\sigma}_k| \leq \mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}), \quad k = 1, \ldots, q,$$

which are not as sharp as those we prove in Lemmas 2.1 *and* 2.2, *can be derived almost trivially using orthoprojectors (see* [55, 53, 22]*), where this approach is used for the case $A = I$. Indeed, we start with identities*

$$A^{1/2} P_F P_{\tilde{G}} A^{-1/2} = A^{1/2} P_F P_G A^{-1/2} + \left(A^{1/2} P_F A^{-1/2}\right)\left(A^{1/2}(P_{\tilde{G}} - P_G)A^{-1/2}\right)$$

for the cosine and

$$\begin{aligned} A^{1/2}(I - P_F)P_{\tilde{G}} A^{-1/2} &= A^{1/2}(I - P_F)P_G A^{-1/2} \\ &\quad + \left(A^{1/2}(I - P_F)A^{-1/2}\right)\left(A^{1/2}(P_{\tilde{G}} - P_G)A^{-1/2}\right) \end{aligned}$$

for the sine, and use (2.18) *and Theorems* 2.13 *and* 2.14*. A norm of the second term is then estimated from above by* $\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}})$*, using the fact that for an* A*-orthoprojector* P_F *we have* $\|P_F\|_A = \|I - P_F\|_A = 1$.

Instead of the latter, we can use a bit more sophisticated approach, as in [22]*, if we introduce the* A*-orthogonal projector* $P_{\mathcal{G}+\tilde{\mathcal{G}}}$ *on the subspace* $\mathcal{G}+\tilde{\mathcal{G}}$*. Then the norm of second term is bounded by* $\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}})$ *times* $\|P_F P_{\mathcal{G}+\tilde{\mathcal{G}}}\|_A$ *for the cosine and times* $\|(I-P_F)P_{\mathcal{G}+\tilde{\mathcal{G}}}\|_A$ *for the sine, where we can now use Theorem* 2.13 *to provide a geometric interpretation of these two constants. This leads to estimates similar to those of* [22] *for* $A = I$*:*

$$|\sigma_k - \hat{\sigma}_k| \le \cos(\theta_{\min}\{\mathcal{F},\, \mathcal{G}+\tilde{\mathcal{G}}\})\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}}), \qquad k = 1,\ldots,q,$$

and

$$|\mu_k - \hat{\mu}_k| \le \cos(\theta_{\min}\{\mathcal{F}^{\perp},\, \mathcal{G}+\tilde{\mathcal{G}}\})\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}}), \qquad k = 1,\ldots,q.$$

However, the apparent constant "improvement" in the second estimate, for the sine, is truly misleading as

$$\cos(\theta_{\min}\{\mathcal{F}^{\perp},\, \mathcal{G}+\tilde{\mathcal{G}}\}) = 1$$

simply because $\dim\mathcal{F} < \dim(\mathcal{G}+\tilde{\mathcal{G}})$ *in all cases except for the trivial possibility* $\mathcal{G} = \tilde{\mathcal{G}}$*, so subspaces* $\mathcal{F}^{\perp}$ *and* $\mathcal{G}+\tilde{\mathcal{G}}$ *must have a nontrivial intersection.*

The first estimate, for the cosine, does give a better constant (compare to one), but our constant is sharper; e.g.,

$$\cos(\theta_{\min}\{\mathcal{F},\, (\mathcal{G}+\tilde{\mathcal{G}}) \ominus G\}) \le \cos(\theta_{\min}\{\mathcal{F},\, \mathcal{G}+\tilde{\mathcal{G}}\}).$$

Our more complex identities used to derive perturbation bounds provide an extra projector in the error term, which allows us to obtain more refined constants.

We can now establish an estimate of absolute sensitivity of cosine and sine of principal angles with respect to absolute *perturbations of subspaces.*

Theorem 2.15 *Let $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q$ and $\tilde{\sigma}_1 \geq \tilde{\sigma}_2 \geq \cdots \geq \tilde{\sigma}_q$ be cosine of principal angles between subspaces $\mathcal{F}$, $\mathcal{G}$, and $\tilde{\mathcal{F}}$, $\tilde{\mathcal{G}}$, respectively, computed in the A–based scalar product. Then*

$$|\sigma_k - \tilde{\sigma}_k| \leq c_1 \mathrm{gap}_A(\mathcal{F}, \tilde{\mathcal{F}}) + c_2 \mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}), \qquad k = 1, \ldots, q, \tag{2.24}$$

where

$$c_1 = \max\{\cos(\theta_{\min}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G},\, \mathcal{F}\});\, \cos(\theta_{\min}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \tilde{\mathcal{G}},\, \mathcal{F}\})\},$$

$$c_2 = \max\{\cos(\theta_{\min}\{(\mathcal{F} + \tilde{\mathcal{F}}) \ominus \mathcal{F},\, \tilde{\mathcal{G}}\});\, \cos(\theta_{\min}\{(\mathcal{F} + \tilde{\mathcal{F}}) \ominus \tilde{\mathcal{F}},\, \tilde{\mathcal{G}}\})\},$$

where $\theta_{\min}$ is the smallest angle between corresponding subspaces in the A–based scalar product.

Proof: First, by Lemma 2.1, for $k = 1, \ldots, q$,

$$\begin{aligned} |\sigma_k - \hat{\sigma}_k| \ \leq\ & \max\{\cos(\theta_{\min}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \mathcal{G},\, \mathcal{F}\}); \\ & \qquad \cos(\theta_{\min}\{(\mathcal{G} + \tilde{\mathcal{G}}) \ominus \tilde{\mathcal{G}},\, \mathcal{F}\})\}\mathrm{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}). \end{aligned}$$

Second, we apply a similar statement to cosine of principal angles between subspaces $\mathcal{F}$, $\tilde{\mathcal{G}}$ and $\tilde{\mathcal{F}}$, $\tilde{\mathcal{G}}$, respectively, computed in the A–based scalar product:

$$\begin{aligned}|\tilde{\sigma}_k - \hat{\sigma}_k| \;&\le\; \max\{\cos(\theta_{\min}\{(\mathcal{F}+\tilde{\mathcal{F}})\ominus\mathcal{F},\,\tilde{\mathcal{G}}\}); \\ &\qquad\qquad \cos(\theta_{\min}\{(\mathcal{F}+\tilde{\mathcal{F}})\ominus\tilde{\mathcal{F}},\,\tilde{\mathcal{G}}\})\}\mathrm{gap}_A(\mathcal{F},\tilde{\mathcal{F}}).\end{aligned}$$

The statement of the theorem now follows from the triangle inequality. ■

Theorem 2.16 *Let $\mu_1 \le \mu_2 \le \cdots \le \mu_q$ and $\tilde{\mu}_1 \le \tilde{\mu}_2 \le \cdots \le \tilde{\mu}_q$ be sine of principal angles between subspaces $\mathcal{F}$, $\mathcal{G}$, and $\tilde{\mathcal{F}}$, $\tilde{\mathcal{G}}$, respectively, computed in the A–based scalar product. Then*

$$|\mu_k - \tilde{\mu}_k| \le c_3\mathrm{gap}_A(\mathcal{F},\tilde{\mathcal{F}}) + c_4\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}}), \qquad k = 1,\ldots,q, \tag{2.25}$$

where

$$c_3 = \max\{\sin(\theta_{\max}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G},\,\mathcal{F}\}); \sin(\theta_{\max}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\tilde{\mathcal{G}},\,\mathcal{F}\})\},$$

$$c_4 = \max\{\sin(\theta_{\max}\{(\mathcal{F}+\tilde{\mathcal{F}})\ominus\mathcal{F},\,\tilde{\mathcal{G}}\}); \sin(\theta_{\max}\{(\mathcal{F}+\tilde{\mathcal{F}})\ominus\tilde{\mathcal{F}},\,\tilde{\mathcal{G}}\})\},$$

where $\theta_{\max}$ is the largest angle between corresponding subspaces in the A–based scalar product.

Proof: First, by Lemma 2.2, for $k = 1,\ldots,q$,

$$\begin{aligned}|\mu_k - \hat{\mu}_k| \;&\le\; \max\{\sin(\theta_{\max}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\mathcal{G},\,\mathcal{F}\}); \\ &\qquad\qquad \sin(\theta_{\max}\{(\mathcal{G}+\tilde{\mathcal{G}})\ominus\tilde{\mathcal{G}},\,\mathcal{F}\})\}\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}}).\end{aligned}$$

Second, we apply a similar statement to sine of principal angles between subspaces $\mathcal{F}$, $\tilde{\mathcal{G}}$ and $\tilde{\mathcal{F}}$, $\tilde{\mathcal{G}}$, respectively, computed in the A–based scalar product:

$$\begin{aligned}|\tilde{\mu}_k - \hat{\mu}_k| \;\leq\; & \max\{\sin(\theta_{\max}\{(\mathcal{F}+\tilde{\mathcal{F}}) \ominus \mathcal{F},\, \tilde{\mathcal{G}}\}); \\ & \qquad \sin(\theta_{\max}\{(\mathcal{F}+\tilde{\mathcal{F}}) \ominus \tilde{\mathcal{F}},\, \tilde{\mathcal{G}}\})\}\mathrm{gap}_A(\mathcal{F},\tilde{\mathcal{F}}).\end{aligned}$$

The statement of the theorem now follows from the triangle inequality. ■

Finally, we want a perturbation analysis in terms of matrices F and G that generate subspaces $\mathcal{F}$ and $\mathcal{G}$. For that, we have to estimate the sensitivity of a column-space of a matrix, for example, matrix G.

Lemma 2.3 *Let*

$$\kappa_A(G) = \frac{s_{\max}(A^{1/2}G)}{s_{\min}(A^{1/2}G)}$$

denote the corresponding A–based condition number of G, where $s_{\max}$ and $s_{\min}$ are, respectively, largest and smallest singular values of the matrix $A^{1/2}G$. Let $\mathcal{G}$ and $\tilde{\mathcal{G}}$ be column-spaces of matrices G and $\tilde{G}$, respectively. Then

$$\mathrm{gap}_A(\mathcal{G},\tilde{\mathcal{G}}) \leq \kappa_A(G)\frac{\|A^{1/2}(G-\tilde{G})\|}{\|A^{1/2}G\|}. \tag{2.26}$$

Proof: Here, we essentially just adopt the corresponding proof of [55] for the A–based scalar product using the same approach as in Theorem 2.9.

Let us consider the polar decompositions

$$A^{1/2}G = A^{1/2}Q_GT_G \text{ and } A^{1/2}\tilde{G} = A^{1/2}Q_{\tilde{G}}T_{\tilde{G}},$$

where matrices $A^{1/2}Q_G$ and $A^{1/2}Q_{\tilde{G}}$ have orthonormal columns and matrices T_G and $T_{\tilde{G}}$ are q-by-q symmetric positive definite; e.g., $T_G = (Q_GQ_G^T)^{1/2}$. Singular values of T_G and $T_{\tilde{G}}$ are, therefore, the same as singular values of $A^{1/2}G$ and $A^{1/2}\tilde{G}$, respectively. Then,

$$(I - P_{\tilde{G}})(G - \tilde{G}) = (I - P_{\tilde{G}})Q_G T_G.$$

Therefore,

$$A^{1/2}(I - P_{\tilde{G}})Q_G = \left(A^{1/2}(I - P_{\tilde{G}})A^{-1/2}\right) A^{1/2}(G - \tilde{G})T_G^{-1},$$

and

$$\text{gap}_A(\mathcal{G}, \tilde{\mathcal{G}}) \le \|A^{1/2}(G - \tilde{G})\| \|T_G^{-1}\| = \frac{\|A^{1/2}(G - \tilde{G})\|}{s_{\min}(A^{1/2}G)},$$

as $\|A^{1/2}(I - P_{\tilde{G}})A^{-1/2}\| = \|I - P_{\tilde{G}}\|_A \le 1$. The statement of the lemma follows.

■

Remark 2.11 *Some matrices allow improvement of their condition numbers by column scaling, which trivially does not change the column range. Our simple Lemma* 2.3 *does not capture this property. A more sophisticated variant can be easily obtained using a technique developed in* [23, 22].

Our cosine theorem follows next. It generalizes results of [53, 22, 23] to A–based scalar products and somewhat improves the constant.

Theorem 2.17 *Under assumptions of Theorem* 2.15*,*

$$|\sigma_k - \tilde{\sigma}_k| \le c_1 \kappa_A(F) \frac{\|A^{1/2}(F - \tilde{F})\|}{\|A^{1/2}F\|} + c_2 \kappa_A(G) \frac{\|A^{1/2}(G - \tilde{G})\|}{\|A^{1/2}G\|}, \ k = 1, \ldots, q. \tag{2.27}$$

The theorem above does not provide an accurate estimate for small angles. To fill the gap, we suggest the following perturbation theorem in terms of sine of principal angles; cf. [53, 22] for $A = I$.

Theorem 2.18 *Under assumptions of Theorem* 2.16*,*

$$|\mu_k - \tilde{\mu}_k| \leq c_3 \kappa_A(F) \frac{\|A^{1/2}(F - \tilde{F})\|}{\|A^{1/2}F\|} + c_4 \kappa_A(G) \frac{\|A^{1/2}(G - \tilde{G})\|}{\|A^{1/2}G\|}, \quad k = 1, \ldots, q. \tag{2.28}$$

Finally, let us underscore that all sensitivity results in this chapter are for *absolute* errors. Golub and Zha in [22] observe that relative errors of sine and cosine of principal angles are not, in general, bounded by the perturbation.

We consider algorithms of computing principal angles with respect to an A–based scalar product in the next section.

2.6 Algorithm Implementation

In this section, we provide a detailed description of our code SUBSPACEA.m and discuss the algorithm implementation.

Theorem 2.11 is our main theoretical foundation for a sine–based algorithm for computing principal angles with respect to an A–based scalar product. However, a naive implementation, using the SVD of the matrix $S^T AS$, may not produce small angles accurately in computer arithmetic. We now try to explain informally this fact, which is actually observed in numerical tests.

Let, for simplicity of notation, all principal angles be small.

Let us consider a particular case, where columns of matrices Q_F and Q_G are already principal vectors in exact arithmetic. In reality, this is the situation we will face in Algorithm 2.4. Then, in exact arithmetic, columns of S are A-orthogonal and their A-norms are exactly the sine of principal angles. Thus, if there are several small angles different in orders of magnitude, the columns of S are badly scaled. When we take the norms squared, by explicitly computing the product S^TAS, we make the scaling even worse, as the diagonal entries of this diagonal matrix are now the sine of the principal angles squared, in exact arithmetic. In the presence of round-off errors, the matrix S^TAS is usually not diagonal; thus, principal angles smaller than 10^{-8} will lead to an underflow effect in double precision, which cannot be cured by taking square roots of its singular values later in the algorithm.

To resolve this, we want to be able to compute the SVD of the matrix S^TAS without using either S^TAS itself or $A^{1/2}S$. One possibility is suggested in the following lemma.

Lemma 2.4 *The SVD of the matrix $A^{1/2}S$ coincides with the SVD of the matrix Q_S^TAS, where Q_S is a matrix with A-orthonormal columns, which span the same column-space as columns of matrix S.*

Proof: We have

$$(Q_S^TAS)^TQ_S^TAS = S^TAQ_SQ_S^TAS = S^TAP_SS = S^TAS,$$

where $P_S = Q_SQ_S^TA$ is the A-orthogonal projector on the column-space of Q_S, which is the same, by definition, as the column-space of S, so that $P_SS = S$. ■

This contributes to the accuracy of our next Algorithm 2.3, based on Lemma 2.4, to be more reliable in the presence of round-off errors, when several principal angles are small.

By analogy with Algorithms 2.1 and (2.2), we can remove the restriction that matrix Σ is invertible and somewhat improve the costs in Algorithm 2.3 by reducing the size of the matrix S in step 4, which leads to Algorithm 2.4.

```
ALGORITHM 2.3: SUBSPACEA.m

Input: real matrices $F$ and $G$ with the same number of rows, and a symmetric
    positive definite matrix $A$ for the scalar product, or a device to compute $Ax$
    for a given vector $x$.

 1. Compute $A$-orthonormal bases $Q_F = \mathrm{ortha}(F),\ Q_G = \mathrm{ortha}(G)$ of
    column-spaces of $F$ and $G$.
 2. Compute SVD for cosine $[Y, \Sigma, Z] = \mathrm{svd}(Q_F^T A\, Q_G)$, $\Sigma = \mathrm{diag}\ (\sigma_1, \ldots, \sigma_q)$.
 3. Compute matrices of left $U_{\cos} = Q_F Y$ and right $V_{\cos} = Q_G Z$ principal
    vectors.
 4. Compute the matrix $S = \begin{cases} Q_G - Q_F(Q_F^T A\, Q_G) & \mathrm{diag}\ (Q_F) \geq \mathrm{diag}\ (Q_G), \\ Q_F - Q_G(Q_G^T A\, Q_F) & \textit{otherwise.} \end{cases}$
 5. Compute $A$-orthonormal basis $Q_S = \mathrm{ortha}(S)$ of the column-space of $S$.
 6. Compute SVD for sine: $[Y, \mathrm{diag}\ (\mu_1, \ldots, \mu_q), Z] = \mathrm{svd}(Q_S^T AS)$.
 7. Compute matrices $U_{\sin}$ and $V_{\sin}$ of left and right principal vectors:
        $V_{\sin} = Q_G Z$, $U_{\sin} = Q_F(Q_F^T A V_{\sin})\Sigma^{-1}$   $\mathrm{diag}\ (Q_F) \geq \mathrm{diag}\ (Q_G)$;
        $U_{\sin} = Q_F Z$, $V_{\sin} = Q_G(Q_G^T A U_{\sin})\Sigma^{-1}$   otherwise.
 8. Compute the principal angles, for $k = 1, \ldots, q$:
        $\theta_k = \begin{cases} \arccos(\sigma_k) & \textit{if}\ \ \sigma_k^2 < 1/2, \\ \arcsin(\mu_k) & \textit{if}\ \ \mu_k^2 \leq 1/2. \end{cases}$
 9. Form matrices $U$ and $V$ by picking up corresponding columns of $U_{\sin}$, $V_{\sin}$
    and $U_{\cos}$, $V_{\cos}$, according to the choice for $\theta_k$ above.

Output: Principal angles $\theta_1, \ldots, \theta_q$ between column-spaces of matrices $F$ and $G$
    in the $A$–based scalar product, and corresponding matrices of left, $U$, and
    right, $V$, principal vectors.
```

Previous remarks of section 2.3 for the algorithms with $A = I$ are applicable to the present algorithms with self-evident changes. A few additional A-specific remarks follow.

Remark 2.12 *In step* 1 *of Algorithms* 2.3 *and* 2.4, *we use our SVD–based* *function ORTHA.m for the A-orthogonalization. Specifically, computing an A-orthogonal basis Q of the column-space of a matrix X is done in three steps in ORTHA.m. First, we orthonormalize X, using SVD–based built-in MATLAB code ORTH.m with a preceding explicit column scaling; see Remark* 2.1 *on whether the scaling is actually needed. Second, we compute* $[U, S, V] = \mathrm{svd}(X^T AX)$, *using MATLAB's built-in SVD code. Finally, we take* $Q = XUS^{-1/2}$. *If A is ill-conditioned, an extra cycle may be performed to improve the accuracy. While formal stability and accuracy analysis of this method has not been done, ORTHA.m demonstrates practical robustness in our numerical tests.*

```
ALGORITHM 2.4: **Modified and Improved SUBSPACEA.m**

Input: real matrices $F$ and $G$ with the same number of rows, and a symmetric
    positive definite matrix $A$ for the scalar product, or a device to compute
    $Ax$ for a given vector $x$.

 1. Compute A-orthogonal bases $Q_F = \mathrm{ortha}(F)$, $Q_G = \mathrm{ortha}(G)$ of
    column-spaces of $F$ and $G$.
 2. Compute SVD for cosine $[Y, \Sigma, Z] = \mathrm{svd}(Q_F^T\, A\, Q_G)$, $\Sigma = \mathrm{diag}\ (\sigma_1, \dots, \sigma_q)$.
 3. Compute matrices of left $U_{\cos} = Q_F Y$ and right $V_{\cos} = Q_G Z$ principal
    vectors.
 4. Compute large principal angles for $k = 1, \dots, q$:
        $\theta_k = \arccos(\sigma_k)$ if $\sigma_k^2 < 1/2$.
 5. Form parts of matrices $U$ and $V$ by picking up corresponding columns
    according to the choice for $\theta_k$ above. Put columns of $U_{\cos}$, $V_{\cos}$, which are
    left,of $U_{\cos}$, $V_{\cos}$, in matrices $R_F$ and $R_G$. Collect the corresponding $\sigma$'s
    in a diagonal matrix $\Sigma_R$.
 6. Compute the matrix $S = R_G - Q_F(Q_F^T A R_G)$.
 7. Compute A-orthogonal basis $Q_S = \mathrm{ortha}(S)$ of the column-space of $S$.
 8. Compute SVD for sine: $[Y, \mathrm{diag}\ (\mu_1, \dots, \mu_q), Z] = \mathrm{svd}(Q_S^T A S)$.
 9. Compute matrices $U_{\sin}$ and $V_{\sin}$ of left and right principal vectors:
        $V_{\sin} = R_G Z$, $U_{\sin} = R_F(R_F^T A V_{\sin})\Sigma_R^{-1}$.
10. Compute the missing principal angles, for $k = 1, \dots, q$:
        $\theta_k = \arcsin(\mu_k)$ if $\mu_k^2 \le 1/2$.
11. Complete matrices $U$ and $V$ by adding columns of $U_{\sin}$, $V_{\sin}$.

Output: Principal angles $\theta_1, \dots, \theta_q$ between column-spaces of matrices $F$ and
    $G$, and corresponding matrices $U$ and $V$ of left and right principal vectors,
    respectively.
```

Remark 2.13 *We note that, when $n \gg p$, the computational costs of SVDs of p-by-p matrices are negligible; it is multiplication by A, which may be very computationally expensive. Therefore, we want to minimize the number of multiplications by A. In the present version 4.0 of our code SUBSPACEA.m, based on Algorithm 2.4, we multiply matrix A by a vector $2p+q$ times in the worst-case scenario of all angles being small, in steps 1 and 7. We can avoid multiplying by A on steps 2, 8, and 9 by using appropriate linear combinations of earlier computed vectors instead.*

Remark 2.14 *Our actual code is written for a more general complex case, where we require matrix A to be Hermitian.*

Let us finally underline that in a situation when a matrix K from the factorization $A = K^T K$ is given rather than the matrix A itself, we do not advise using Algorithms 2.3 and 2.4. Instead, we recommend multiplying matrices F and G by K on the right and using simpler Algorithms 2.1 and 2.2, according to Theorem 2.9.

2.7 Numerical Examples

Numerical tests in the present section were performed using version 4.0 of our SUBSPACEA.m code, based on Algorithms 2.2 and 2.4. Numerical results presented were obtained, unless indicated otherwise, on Red Hat 6.1 LINUX Dual Pentium-III 500, running MATLAB release 12, version 6.0.0.18. Tests were also made on Compaq Dual Alpha server DS 20, running MATLAB release 12, version 6.0.0.18 and on several Microsoft Windows Pentium-III systems, running

MATLAB version 5.1–5.3. In our experience, Intel PIII–based LINUX systems typically provided more accurate results, apparently utilizing the extended 80 bit precision of FPU registers of PIII; see the discussion at the end of the section.

The main goal of our numerical tests is to check a practical robustness of our code, using the following argument. According to our analysis of section 2.5 and similar results of [5, 53, 22, 23], an absolute change in cosine and sine of principal angles is bounded by perturbations in matrices F and G, with the constant, proportional to their condition numbers taken after a proper column scaling; see Theorems 2.17 and 2.28 and Remark 2.11. Assuming a perturbation of entries of matrices F and G at the level of double precision, $EPS \approx 10^{-16}$, we expect a similar perturbation in cosine and sine of principal angles, when matrices F and G are well-conditioned after column scaling. We want to check if our code achieves this accuracy in practice.

We concentrate here on testing our sine–based algorithms, i.e., for principal angles smaller than $\pi/4$. The cosine–based algorithm with $A = I$ is recently studied in [16].

Our first example is taken from [5] with $p = 13$ and $m = 26$. Matrices F and G were called A and B in [5]. F was orthogonal, while G was an m-by-p Vandermonde matrix with $\text{cond}(G) \approx 10^4$. Matrix G was generated in double precision and then rounded to single precision.

According to our theory and a perturbation analysis of [5, 53, 22, 23], in this example an absolute change in principal angles is bounded by a perturbation in matrix G times its condition number. Thus, we should expect sine and cosine of principal angles computed in [5] to be accurate with approximately four decimal digits.

In our code, all computations are performed in double precision; therefore, answers in Table 2.2 should be accurate up to twelve decimal digits. We observe, as expected, that our results are consistent with those of [5] within four digits.

k	$\sin(\theta_k)$	$\cos(\theta_k)$
1	0.00000000000	1.00000000000
2	0.05942261363	0.99823291519
3	0.06089682091	0.99814406635
4	0.13875176720	0.99032719194
5	0.14184708183	0.98988858230
6	0.21569434797	0.97646093022
7	0.27005046021	0.96284617096
8	0.33704307148	0.94148922881
9	0.39753678833	0.91758623677
10	0.49280942462	0.87013727135
11	0.64562133627	0.76365770483
12	0.99815068733	0.06078820101
13	0.99987854229	0.01558527040

Table 2.2: Computed sine and cosine of principal angles of the example of [5]

In our next series of tests, we assume n to be even and $p = q \leq n/2$. Let D be a diagonal matrix of the size p:

$$D = \text{diag}\ (d_1, \ldots, d_p), \quad d_k > 0, \quad k = 1, \ldots, p.$$

We first define n-by-p matrices

$$F_1 = [I\ 0]^T, \quad G_1 = [I\ D\ 0]^T, \tag{2.29}$$

where I is the identity matrix of the size p and 0 are zero matrices of appropriate sizes. We notice that condition numbers of F_1 and G_1 are, respectively, one and

$$\operatorname{cond} G_1 = \sqrt{\frac{1 + (\max\{\text{diag }(D)\})^2}{1 + (\min\{\text{diag }(D)\})^2}}.$$

Thus, the condition number may be large only when large diagonal entries in D are present. Yet in this case, the condition number of G_1 can be significantly reduced by column scaling; see Remark 2.11.

The exact values of sine and cosine of principal angles between column-spaces of matrices F_1 and G_1 are obviously given by

$$\mu_k = \frac{d_k}{\sqrt{1 + d_k^2}}, \quad \sigma_k = \frac{1}{\sqrt{1 + d_k^2}}, \quad k = 1, \ldots, p, \tag{2.30}$$

respectively, assuming that d_k's are sorted in the increasing order. The collective error in principal angles is measured as the following sum:

$$\sqrt{(\mu_1 - \tilde{\mu}_1)^2 + \cdots + (\mu_p - \tilde{\mu}_p)^2} + \sqrt{(\sigma_1 - \tilde{\sigma}_1)^2 + \cdots + (\sigma_p - \tilde{\sigma}_p)^2}, \tag{2.31}$$

where μ's are the sine and σ's are the cosine of principal angles, and the tilde sign $\tilde{}$ is used for actual computed values.

We multiply matrices F_1 and G_1 by a random orthogonal matrix U of the size n on the left to get

$$F_2 = U * F_1, \quad G_2 = U * G_1. \tag{2.32}$$

This transformation does not change angles and condition numbers. It still allows for improving the condition number of G_2 by column scaling.

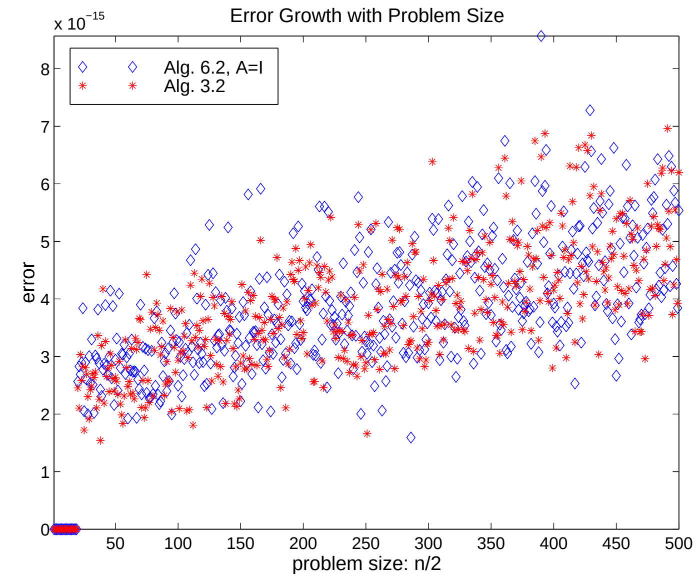
Finally, we multiply matrices by random orthogonal p-by-p matrices T_F and T_G, respectively, on the right:

$$F_3 = F_2 * T_F, \quad G_3 = G_2 * T_G. \tag{2.33}$$

This transformation does not change angles or condition numbers. It is likely, however, to remove the possibility of improving the condition number of G_3 by column scaling. Thus, if G_3 is ill-conditioned, we could expect a loss of accuracy; see Theorems 2.17 and 2.28 and Remark 2.11.

We start by checking scalability of the code for well-conditioned cases. We increase the size of the problem n and plot the collective error, given by 2.31, for the principal angles between F_3 and G_3, against the value $n/2$. We solve the same problem two times, using Algorithm 2.2 and Algorithm 2.4 with $A = I$.

In the first two tests, diagonal entries of D are chosen as uniformly distributed random numbers *rand* on the interval $(0, 1)$. On Figure 2.1 (top) we fix $p = q = 20$. We observe that the average error grows approximately two times with a ten times increase in the problem size. On Figure 2.1 (bottom), we also raise $p = q = n/2$. This time, the error grows with the same pace as the problem size. Please note different scales used for errors on Figure 2.1.



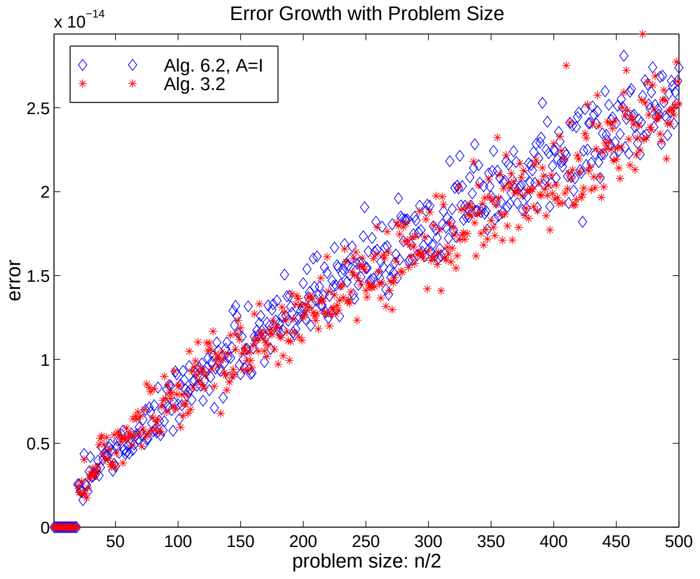


Figure 2.1: Errors in principal angles as functions of $n/2$

To test our methods for very small angles with $p = q = 20$, we first choose

$$p = 20, \quad d_k = 10^{-k}, \quad k = 1, \ldots, p.$$

We observe a similar pattern of the absolute error as that of Figure 2.1 (top), but do not reproduce this figure here.

In our second test for small angles, we set $p = q = n/2$ as on Figure 2.1 (bottom) and select every diagonal element of D in the form $10^{-17 \cdot rand}$, where $rand$ is again a uniformly distributed random number on the interval $(0, 1)$. The corresponding figure, not shown here, looks the same as Figure 2.1 (bottom), except that the error grows a bit faster and reaches the level $\approx 4 \cdot 10^{-14}$ (compare to $\approx 3 \cdot 10^{-14}$ value on Figure 2.1 (bottom).

In all these and our other analogous tests, Algorithm 2.2 and Algorithm 2.4 with $A = I$ behave very similarly, so that Figure 2.1 provides a typical example.

In our next series of experiments, we fix a small $p = q$ and $n = 100$, and compute angles between F_2 and G_2 and between F_3 and G_3 500 times, changing only the random matrices used in the construction of our F_i and G_i. Instead of the collective error, given by 2.31, we now compute errors for individual principal angles as

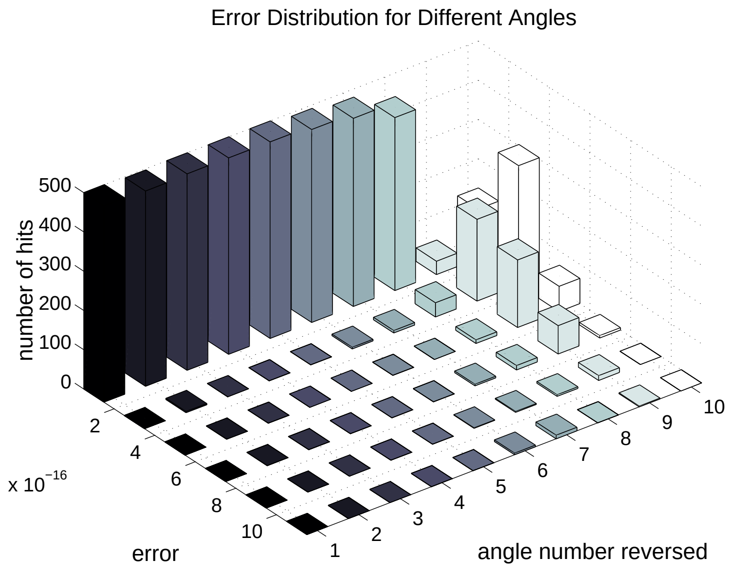
$$|\mu_k - \tilde{\mu}_k| + |\sigma_k - \tilde{\sigma}_k|, \qquad k = 1, \ldots, p.$$

We tested several different combinations of angles less than $\pi/4$. In most cases, the error was only insignificantly different from $EPS \approx 10^{-16}$. The worst-case scenario found numerically corresponds to

$$D = \text{diag}\ \{1, 0.5, 10^{-11}, 10^{-12}, 10^{-13}, 5 \cdot 10^{-15}, 2 \cdot 10^{-15}, 10^{-15}, 10^{-16}, 0\}$$

and is presented on Figure 2.2. Figure 2.2 (top) shows the distribution of the error for individual angles between F_3 and G_3 in Algorithm 2.2 in 500 runs. Figure 2.2 (bottom) demonstrates the performance of Algorithm 2.4 with $A = I$, for the same problem. The numeration of angles is reversed for technical reasons; i.e., smaller angles are further away from the viewer. Also the computed distribution of the error for individual angles between F_2 and G_2 are very similar and, because of that, are not shown here.

We detect that for such small values of p and n most of the angles are computed essentially within the double precision accuracy. Only multiple small angles present a slight challenge, more noticeable on Figure 2.2 (bottom), which uses a somewhat different scale for the error to accommodate a larger error. Nevertheless, all observed errors in all 500 runs are bounded from above by $\approx 6 \cdot 10^{-15}$ on Figure 2.2, which seems to be a reasonable level of accuracy for accumulation of round-off errors appearing in computations with 200-by-10 matrices in double precision.



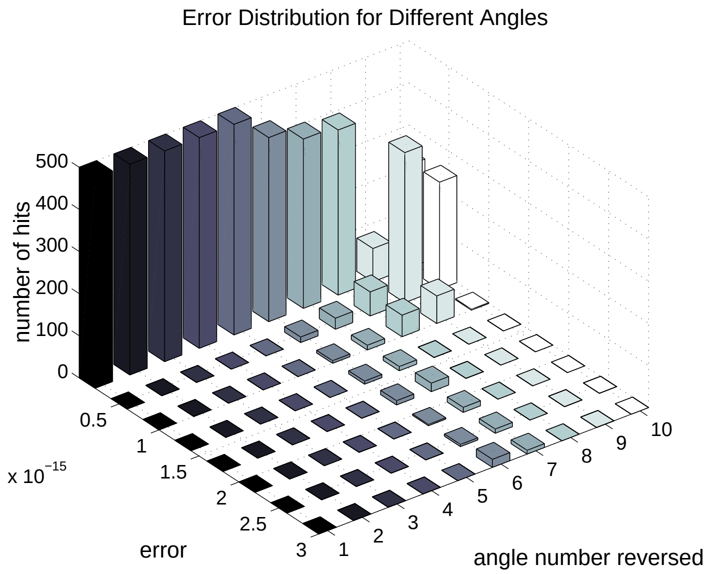


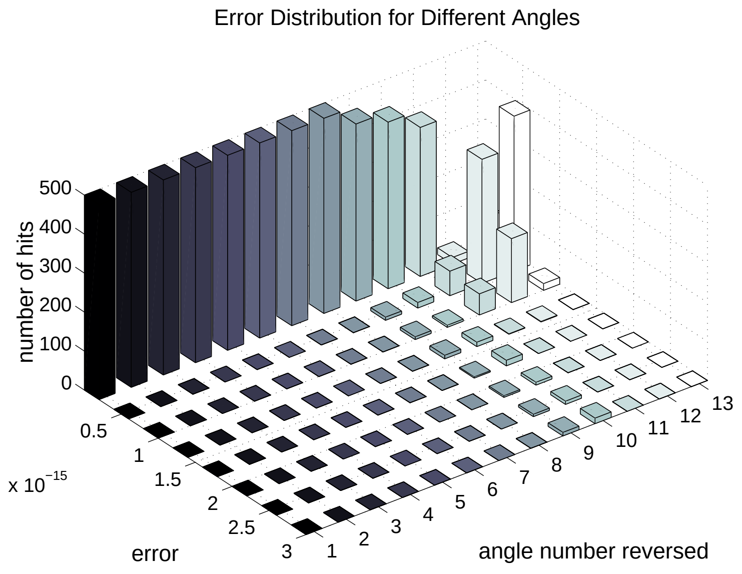
Figure 2.2: Errors in individual angles in algorithms 2.2 and 2.4 with $A = I$

To test our code for an ill-conditioned case, we add two large values to the previous choice of D to obtain

$$D = \operatorname{diag}\{10^{10}, 10^{8}, 1, 0.5, 10^{-11}, 10^{-12}, 10^{-13}, 5 \cdot 10^{-15}, 2 \cdot 10^{-15}, 10^{-15}, 10^{-16}, 0\},$$

which leads to $\operatorname{cond} G_1 \approx 10^{10}$.

Figure 2.3 (top) shows the distribution of the error for individual angles between F_2 and G_2 in Algorithm 2.4 with $A = I$ after 500 runs. The numeration of angles is again reversed; i.e., smaller angles are further away from the viewer. There is no visible difference between the new Figure 2.3 (top) and the previous Figure 2.2 for the well-conditioned case, which confirms results of [23, 22] on column scaling of ill-conditioned matrices; see Remark 2.11. Namely, our ill-conditioned matrix G_2 can be made well-conditioned by column scaling. Thus, perturbations in the angles should be small. Figure 2.3 (top) therefore demonstrates that our code SUBSPACEA.m is able to take advantage of this.



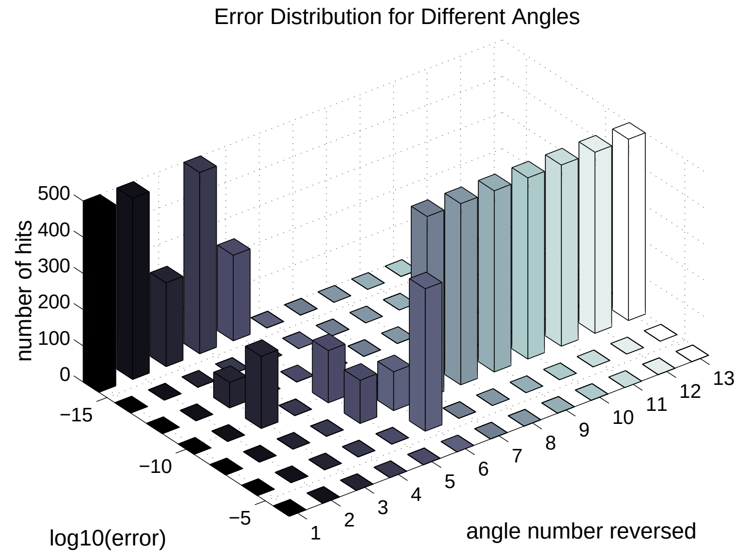


Figure 2.3: Errors in individual angles between F_2 and G_2 and F_3 and G_3

As we move to computing angles between F_3 and G_3 in this example (see Figure 2.3 (bottom)), where the distribution of the error for individual angles in Algorithm 2.4 with $A = I$ after 500 runs is shown, the situation changes dramatically. In general, it is not possible to improve the condition number $\operatorname{cond}(G_3) \approx 10^{10}$ by column scaling. Thus, according to [5, 23] and our perturbation analysis of Theorems 2.17 and 2.28 and Remark 2.11, we should expect the absolute errors in angles to grow $\approx 10^{10}$ times (compare to the errors on Figure 2.3 (bottom)), i.e., up to the level 10^{-5}. On Figure 2.3 (bottom), which shows actual errors obtained during 500 runs of the code, we indeed observe absolute errors in angles at the level up to 10^{-5}, as just predicted. Surprisingly, the absolute error for angles with small cosines is much better. We do not have an explanation of such good behavior, as the present theory does not provide individual perturbation bounds for different angles.

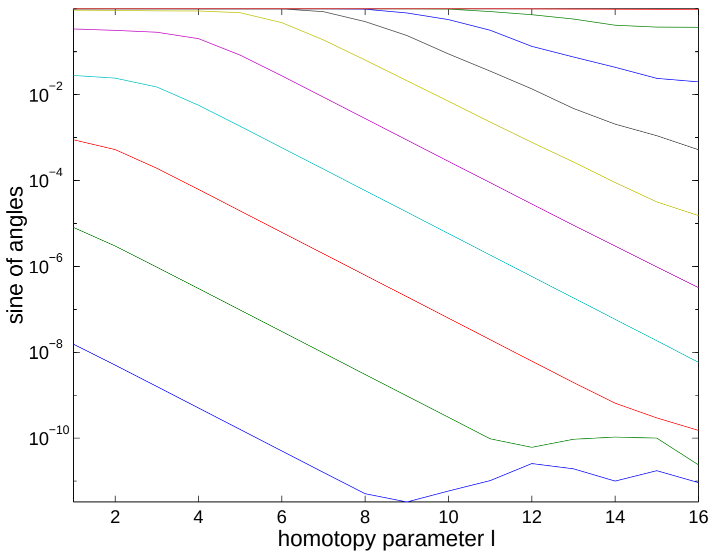
Our concluding numerical results illustrate performance of our code for ill-conditioned scalar products.

We take G to be the first ten columns of the identity matrix of size twenty, and F to be the last ten columns of the Vandermonde matrix of size twenty with elements $v_{i,j} = i^{20-j}$, $i, j = 1, \ldots, 20$. Matrix F is ill-conditioned, $\operatorname{cond} F \approx 10^{13}$. We compute principal angles and vectors between F and G in an A–based scalar product for the following family of matrices:

$$A = A_l = 10^{-l} I + H, \qquad l = 1, \ldots, 16,$$

where I is identity and H is the Hilbert matrix of the order twenty, whose elements are given by $h_{i,j} = 1/(i+j-1)$, $i, j = 1, \ldots, 20$. Our subspaces $\mathcal{F}$ and

$\mathcal{G}$ do not change with l; only the scalar product that describes the geometry of the space changes. When $l = 1$, we observe three angles with cosine less than 10^{-3} and three angles with sine less than 10^{-3}. When l increases, we are getting closer to the Hilbert matrix, which emphasizes first rows in matrices F and G, effectively ignoring last rows. By construction of F, its large elements, which make subspace $\mathcal{F}$ to be further away from subspace $\mathcal{G}$, are all in last rows. Thus, we should expect large principal angles between $\mathcal{F}$ and $\mathcal{G}$ to decrease monotonically as l grows. We observe this in our numerical tests (see Figure 2.4 (top)), which plots in logarithmic scale sine of all ten principal angles as functions of l.



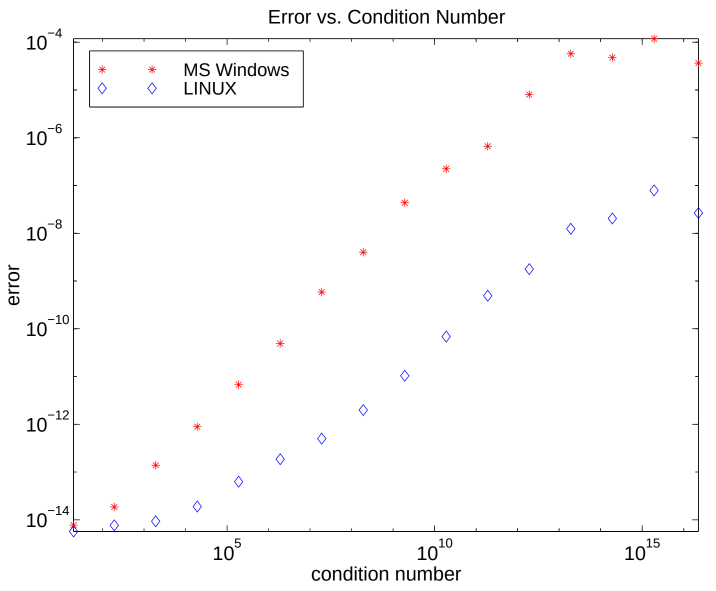


Figure 2.4: μ_k as functions of l and errors as functions of condition number

Of course, such change in geometry that makes sine of an angle to decrease 10^4 times, means that matrix A_l, describing the scalar product, gets more and more ill-conditioned, as it gets closer to Hilbert matrix H, namely, $\mathrm{cond}(A) \approx 10^l$ in our case. It is known that ill-conditioned problems usually lead to a significant increase of the resulting error, as ill-conditioning amplifies round-off errors. To investigate this effect for our code, we introduce the error as the following sum:

$$\text{error} = \|V^T AV - I\| + \|U^T AU - I\| + \|\Sigma - U^T AV\|,$$

where the first two terms control orthogonality of principal vectors and the last term measures the accuracy of cosine of principal angles. We observe in our experiments that different terms in the sum are close to each other, and none dominates. The accuracy of sine of principal angles is not crucial in this example as angles are not small enough to cause concerns. As U and V are constructed directly from columns of F and G, they span the same subspaces with high accuracy independently of condition number of A, as we observe in the tests.

We plot the error on the y-axis of Figure 2.4 (bottom) for a Pentium III 500 PC running two different operating systems: Microsoft Windows NT 4.0 SP6 (red stars) and RedHat LINUX 6.1 (blue diamonds), where the x-axis presents condition number of A; both axes are in logarithmic scale. The MATLAB Version 5.3.1.29215a (R11.1) is the same on both operating systems.

We see, as expected, that the error grows, apparently linearly, with the condition number. We also observe, now with quite a surprise, that the error on LINUX is much smaller than the error on MS Windows!

As the same MATLAB's version and the same code SUBSPACEA.m are run on the same hardware, this fact deserves an explanation. As a result of a discussion with Nabeel and Lawrence Kirby at the News Group *sci.math.num-analysis*, it has been found that MATLAB was apparently compiled on LINUX to take advantage of extended 80 bit precision of FPU registers of PIII, while Microsoft compilers apparently set the FPU to 64 bit operations. To demonstrate this, Nabeel suggested the following elegant example: compute scalar product

$$(1\ 10^{-19}\ \ -1)^T\ (1\ 1\ 1).$$

On MS Windows, the result is zero, as it should be in double precision, while on LINUX the result is $1.0842 10^{-19}$.

Figure 2.4 (bottom) shows that our algorithm turns this difference into a significant error improvement for an ill-conditioned problem.

Finally, our code SUBSPACEA.m has been used in the code LOBPCG.m (see [34, 35]) to control accuracy of invariant subspaces of large symmetric generalized eigenvalue problems, and thus has been tested for a variety of large-scale practical problems.

2.8 Availability of the Software

A listing for SUBSPACEA.m and the function ORTHA.m is provided in Appendix A. Also see http://www.mathworks.com/support/ftp/linalgv5.shtml for the code SUBSPACEA.m and the function ORTHA.m it uses, as well as the fix for the MATLAB code SUBSPACE.m, as submitted to MathWorks. They are also publicly available at the Web page: http://www-math.cudenver.edu/~aknyazev/software.

2.9 Conclusions Concerning Angles Between Subspaces

Let us formulate here the main points of this section:

- A bug in the cosine–based algorithm for computing principal angles between subspaces, which prevents one from computing small angles accurately in computer arithmetic, is illustrated.

- An algorithm is presented that computes all principal angles accurately in computer arithmetic and is proved to be equivalent to the standard algorithm in exact arithmetic.

- A generalization of the algorithm to an arbitrary scalar product given by a symmetric positive definite matrix is suggested and justified theoretically.

- Perturbation estimates for absolute errors in cosine and sine of principal angles, with improved constants and for an arbitrary scalar product, are derived.

- A description of the code is given as well as results of numerical tests. The code is robust in practice and provides accurate angles for large-scale and ill-conditioned cases we tested numerically. It is also reasonably efficient for large-scale applications with $n \gg p$.

- Our algorithms are "matrix-free"; i.e., they do not require storing in memory any matrices of the size n and are capable of dealing with A, which may be specified only as a function that multiplies A by a given vector.

- MATLAB release 13 has implemented the SVD–based sine version of our algorithm.

3. Rayleigh Quotient Perturbations and Eigenvalue Accuracy

In this chapter we discuss Rayleigh quotient and Rayleigh Ritz perturbations and eigenvalue accuracy. There are two reasons for studying this problem [52]: first, the results can be used in the design and analysis of algorithms, and second, a knowledge of the sensitivity of an eigenpair is of both theoretical and of practical interest. Rayleigh quotient and Rayleigh Ritz equalities and inequalities are central to determining eigenvalue accuracy and in analyzing many situations when individual vectors and/or subspaces are perturbed.

3.1 Introduction to Rayleigh Quotient Perturbations

Let $A \in \mathbf{R^{n\times n}}$ be a matrix and x be a vector in $\mathbf{R^n}$. In this chapter, for the majority of results, we will assume that the matrix A is symmetric and that we are dealing with finite dimensional real vector spaces. Let $\lambda_1 \le \cdots \le \lambda_n$ be the eigenvalues of A with corresponding eigenvectors, respectively, $u_1, \ldots, u_n$. These eigenvectors may be chosen to be orthonormal. In some cases, we will assume that A is positive definite, with spectral condition number $\kappa(A) = \|A\|\|A^{-1}\| = \frac{\lambda_n}{\lambda_1}$.

In this chapter we will avoid introducing a basis to the extent possible. This makes these concepts and results more general, and possibly applicable to infinite dimensional vector spaces (e.g., Hilbert spaces). Investigation into this area is beyond the scope of this work, but promises interesting results for future work.

The Rayleigh quotient $\rho(x;A)$, is defined for $x \in \mathbf{R^n}$ with $x \neq 0$ as:

$$\rho(x) = \rho(x;A) = (Ax,x)/(x,x), \tag{3.1}$$

where $(x,x) = x^T x = \|x\|^2$ as in [49]. If it is obvious which matrix A is involved, the form $\rho(x)$ will be used.

A significant application of the Rayleigh quotient involves the Courant–Fischer Theorem. We restate here the Courant–Fischer Theorem (Theorem 10.2.1 of [49]), which is known as the minimax characterization of eigenvalues. It is used in this investigation to obtain several important results.

Theorem 3.1 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix with eigenvalues $\lambda_1 \leq \cdots \leq \lambda_n$. Then*

$$\lambda_j = \min_{\substack{\mathcal{G}^j \subseteq \mathbf{R^n} \\ dim\ \mathcal{G}^j = j}} \max_{g \in \mathcal{G}^j} \ \rho(g;A), \ j = 1, \ldots, n. \tag{3.2}$$

Given a vector $x \in \mathbf{R^n}$ with $x \neq 0$, the residual is defined as:

$$r(x) = r(x;A) = (A - \rho(x;A)I)x, \tag{3.3}$$

where I is the identity matrix. If it is obvious which matrix A is involved, the form $r(x)$ will be used. It is worth noting that the gradient of $\rho(x;A)$ is given by

$$\nabla\rho(x,A) = \frac{2}{(x,x)}\ r(x;A). \tag{3.4}$$

We need to define some concepts concerning angles between vectors and angles between subspaces which are used in this chapter. The acute angle between two non–zero vectors x, $y \in \mathbf{R^n}$ is denoted by

$$\angle\{x,y\} = \arccos\frac{|(x,y)|}{\|x\|\|y\|}.$$

It is important to emphasize that $0 \leq \angle\{x, y\} \leq \frac{\pi}{2}$, by definition. The largest principal angle between the subspaces $\mathcal{F}, \mathcal{G} \subseteq \mathbf{R^n}$ is denoted by $\angle\{\mathcal{F}, \mathcal{G}\}$. The definition and concepts involving principal angles between subspaces is discussed in detail in Chapter 2.

The following result is used in [33], although this reference does not include a proof.

Lemma 3.1 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix and let $x \in \mathbf{R^n}$ with $x \neq 0$. Let P_x be the orthogonal projector onto the subspace spanned by the vector x. Then*

$$\frac{\|r(x;A)\|}{\|x\|} = \|(I - P_x)AP_x\|.$$

Proof: We have

$$\|(I - P_x)AP_x\| = \sup_{y\neq 0} \frac{\|(I - P_x)AP_x y\|}{\|y\|}.$$

First we prove that

$$\rho(x;A)P_x = P_x A P_x. \tag{3.5}$$

Let $y_1, y_2 \in \mathbf{R^n}$ with $y_1 = \alpha_1 x + w_1$ and $y_2 = \alpha_2 x + w_2$, where w_1, w_2 are orthogonal to x. Then

$$\begin{aligned}((\rho(x;A)P_x - P_x A P_x)y_1, y_2) &= ((\rho(x;A) - A)P_x y_1, P_x y_2) \\ &= \alpha_1\alpha_2((\rho(x;A) - A)x, x) = 0.\end{aligned}$$

Since y_1 and y_2 are arbitrary, $\rho(x;A)P_x - P_x A P_x = 0$.

For any vector y, we have $y = \alpha x + w$ for some real α and vector w orthogonal to x. Then

$$\begin{aligned}
\frac{\|(I-P_x)AP_xy\|}{\|y\|} &= \frac{|\alpha|\|(I-P_x)AP_xx\|}{\|y\|} = \frac{|\alpha|\|Ax - P_xAP_xx\|}{\|y\|} \\
&= \frac{|\alpha|\|Ax - \rho(x;A)x\|}{\|y\|} = \frac{|\alpha|\|r(x;A)\|}{\sqrt{|\alpha|^2\|x\|^2 + \|w\|^2}} \leq \frac{\|r(x;A)\|}{\|x\|},
\end{aligned}$$

and for $y = x$ this bound is achieved, which proves the result. ■

There is another interesting result which relates $\rho(x;A)$ and $r(x;A)$ to the angle between x and Ax.

Lemma 3.2 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix and let $x \in \mathbf{R^n}$ with $x \neq 0$ and assume that $Ax \neq 0$. Then*

$$\frac{\|r(x;A)\|}{\|Ax\|} = \sin(\angle\{x, Ax\}), \tag{3.6}$$

and assuming $\rho(x;A) \neq 0$, we have

$$\frac{\|r(x;A)\|}{\|x\|} = |\rho(x;A)| \tan(\angle\{x, Ax\}). \tag{3.7}$$

Proof: We have

$$\cos(\angle\{x, Ax\}) = \frac{|(x, Ax)|}{\|x\|\|Ax\|} = \frac{|\rho(x;A)|\ \|x\|}{\|Ax\|},$$

and

$$\sin^2(\angle\{x, Ax\}) = 1 - \cos^2(\angle\{x, Ax\}) = \frac{\|Ax\|^2 - \rho(x;A)^2\|x\|^2}{\|Ax\|^2} = \frac{\|r(x;A)\|^2}{\|Ax\|^2}.$$

This proves (3.6). Combining the results for cosine and sine, we obtain (3.7). ■

Another important concept involves what is regarded as an invariant subspace. A subspace $\mathcal{S} \subseteq \mathbf{R^n}$ is invariant w.r.t to the matrix A, if it satisfies the condition that if $x \in \mathcal{S}$, then $Ax \in \mathcal{S}$, i.e. $A\mathcal{S} \subseteq \mathcal{S}$.

3.2 Subspaces and the Rayleigh–Ritz Procedure

Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix and $\mathcal{S} \subseteq \mathbf{R^n}$ be an m–dimensional subspace, $0 < m \leq n$. We can define an operator $\tilde{A} = P_{\mathcal{S}}A|_{\mathcal{S}}$ on $\mathcal{S}$, where $P_{\mathcal{S}}$ is the orthogonal projection onto $\mathcal{S}$ and $P_{\mathcal{S}}A|_{\mathcal{S}}$ denotes the restriction of $P_{\mathcal{S}}A$ to $\mathcal{S}$. Eigenvalues of $\tilde{A}$ are called Ritz values, $\widetilde{\lambda}_1 \leq \cdots \leq \widetilde{\lambda}_m$, and the corresponding eigenvectors, $\tilde{u}_1 \leq \cdots \leq \tilde{u}_m$, are called Ritz vectors. Ritz values and Ritz vectors can be characterized in several ways as is discussed in [13, 49]. The operator $\tilde{A}$ is self–adjoint, since if $x, y \in \mathcal{S}$, then

$$(\tilde{A}x, y) = (P_{\mathcal{S}}Ax, y) = (x, AP_{\mathcal{S}}y) = (P_{\mathcal{S}}x, Ay) = (x, P_{\mathcal{S}}Ay) = (x, \tilde{A}y).$$

Utilizing a setup similar to the Courant–Fischer Theorem (Theorem 3.1), Ritz values are characterized, as in equation (11.5) of [49], by

$$\widetilde{\lambda}_j = \min_{\substack{\mathcal{G}^j \subseteq \mathcal{S} \\ \dim\ \mathcal{G}^j = j}} \max_{g \in \mathcal{G}^j}\ \rho(g; A),\ \ j = 1, \ldots, m. \tag{3.8}$$

Here, $\mathcal{G}^j \subseteq \mathcal{S}$ instead of $\mathcal{G}^j \subseteq \mathbf{R^n}$ as in (3.2).

Equivalently, let the columns of $Q \in \mathbf{R^{n\times m}}$ form an orthonormal basis for $\mathcal{S}$ (Theorem 11.4.1 of [49]), then $\hat{A} = Q^T A Q = Q^T \tilde{A} Q$ is a matrix representation of the operator $\tilde{A} = P_{\mathcal{S}}A|_{\mathcal{S}}$ in this basis, and $\widetilde{\lambda}_j,\ j = 1, \ldots, m$ are the eigenvalues of $\hat{A}$.

Assuming $w \in \mathbf{R^m}$, a linear map $\varphi : \mathbf{R^m} \to \mathcal{S}$ can be defined by $\varphi(w) = Qw$. The inner product (and hence the norm) is preserved under this map, since if

$w, v \in \mathbf{R^m}$ then

$$(w, v) = (Q^T Qw, v) = (Qw, Qv) = (\varphi(w), \varphi(v)). \tag{3.9}$$

Thus φ is an isometry. This also implies that angles between vectors are preserved under φ.

Now we define the term residual matrix which is discussed in Section 11.4 of [49]. Given a subspace $\mathcal{S} \subseteq \mathbf{R^n}$, Q as defined above and $B \in \mathbf{R^{m\times m}}$, the residual matrix is given by

$$R(B) = AQ - QB.$$

We have

$$\begin{aligned} \|R(\hat{A})\| &= \|R(Q^T AQ)\| = \|AQ - Q(Q^T AQ)\| \\ &= \|(I - QQ^T)AQ\| = \|(I - QQ^T)AQQ^T\| \\ &= \|(I - P_{\mathcal{S}})AP_{\mathcal{S}}\|. \end{aligned} \tag{3.10}$$

The residual matrix has a minimal property defined by Theorem 11.4.2 of [49] which is repeated below:

Theorem 3.2 *For a given matrix $Q \in \mathbf{R^{n\times m}}$ with orthonormal columns and $\hat{A} = Q^T AQ$,*

$$\|R(\hat{A})\| \leq \|R(B)\| \quad \textit{for all} \quad B \in \mathbf{R^{m\times m}}.$$

Several important properties related to Rayleigh–Ritz approximations are given in the following Lemma:

Lemma 3.3 *Let the columns of $Q \in \mathbf{R^{n\times m}}$ form an orthonormal basis for $\mathcal{S}$ and let $\varphi(\cdot)$ be defined as above. Then*

1. *Considering the Rayleigh quotient,*

$$\rho(w, \hat{A}) = \rho(\varphi(w), \tilde{A}), \quad \forall w \in \mathbf{R^m}.$$

2. *Considering the residual,*

$$\|r(w, \hat{A})\| = \|r(\varphi(w), \tilde{A})\|, \quad \forall w \in \mathbf{R^m}.$$

3. *The eigenvectors $w_1, w_2, \ldots, w_m$ of $\hat{A}$ and the corresponding Ritz vectors $\tilde{u}_1, \ldots, \tilde{u}_m \in \mathcal{S}$, $\tilde{u}_i = \varphi(w_i)$, $i = 1, \ldots, m$, can be chosen to be orthonormal.*

Proof: Let $\hat{A} \in \mathbf{R^{m\times m}}$ be the matrix representation of $\tilde{A}$ in the basis of orthonormal columns of Q, as defined above. Then $\tilde{A}Q = Q\hat{A}$, and therefore $Q^T\tilde{A}Q = \hat{A}$.

1. Let $w \in \mathbf{R^m}$ and let $\varphi(w) = x \in \mathcal{S}$, then considering the Rayleigh quotient, we have

$$\begin{aligned} \rho(w, \hat{A}) &= \frac{(Q^T\tilde{A}Qw, w)}{(w, w)} = \frac{(\tilde{A}Qw, Qw)}{(Qw, Qw)} \\ &= \frac{(\tilde{A}x, x)}{(x, x)} = \rho(x, \tilde{A}) = \rho(\varphi(w), \tilde{A}). \end{aligned}$$

2. Let $w \in \mathbf{R^m}$ and let $\varphi(w) = x \in \mathcal{S}$, then considering the residual, we have

$$\begin{aligned} \|r(w, \hat{A})\| &= \|\hat{A}w - \rho(w, \hat{A})w\| = \|Q^T\tilde{A}Qw - \rho(x, \tilde{A})w\| \\ &= \|Q^T(\tilde{A}x - \rho(x, \tilde{A})x\| = \|QQ^T(\tilde{A}x - \rho(x, \tilde{A})x\| \\ &= \|P_{\mathcal{S}}(\tilde{A}x - \rho(x, \tilde{A})x\| = |P_{\mathcal{S}}\tilde{A}x - \rho(x, \tilde{A})P_{\mathcal{S}}x\| \\ &= \|\tilde{A}x - \rho(x, \tilde{A})x\| = \|r(x, \tilde{A})\| = \|r(\varphi(w), \tilde{A})\|. \end{aligned}$$

3. Let $w, v \in \mathbf{R^m}$. Since $\tilde{A}$ is self-adjoint we have

$$(\hat{A}w, v) = (Q^T\tilde{A}Qw, v) = (\tilde{A}Qw, Qv) = (w, Q^T\tilde{A}Qv) = (w, \hat{A}v),$$

and $\hat{A}$ is self–adjoint (i.e. symmetric), thus $\hat{A}$ has an orthonormal set of eigenvectors $w_1, w_2, \ldots, w_m$. Since $\varphi(\cdot)$ is an isometry, the Ritz vectors $\tilde{u}_i = \varphi(w_i),\ i = 1, \ldots, m$ are also orthonormal in $\mathcal{S} \subseteq \mathbf{R^n}$.

■

Some important properties of Ritz values and Ritz vectors are given below:

Lemma 3.4 *Let $\mathcal{S} \subseteq \mathbf{R^n}$ be an m–dimensional subspace with $0 < m \leq n$, and $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix. Let $\tilde{A} = P_{\mathcal{S}}A|_{\mathcal{S}}$, where $P_{\mathcal{S}}$ is an orthogonal projection onto $\mathcal{S}$. Let $\widetilde{\lambda}_1 \leq \cdots \leq \widetilde{\lambda}_m$ be the Ritz values for A w.r.t to the subspace $\mathcal{S}$, and $\lambda_1 \leq \cdots \leq \lambda_n$ be the eigenvalues of A. Then the following properties hold:*

1. *Considering the Rayleigh Quotient, $\rho(x; \tilde{A}) = \rho(x; A), \forall x \in \mathcal{S}$.*
2. *The norm of the residual is dominated as follows: $\|r(x; \tilde{A})\| \leq \|r(x; A)\|$, $\forall x \in \mathcal{S}$. If the subspace $\mathcal{S}$ in invariant w.r.t A, then we have equality.*

3. *A has a Ritz–pair* $(\widetilde{\lambda}, \tilde{u})$ *(i.e. Ritz value and Ritz vector), if and only if* $A\tilde{u} - \widetilde{\lambda}\tilde{u} \in \mathcal{S}^{\perp}$.

4. *The Ritz values satisfy:* $\lambda_i \leq \widetilde{\lambda}_i \leq \lambda_n, \; i = 1, \ldots, m$.

5. *Assuming that A is positive definite,* $\kappa(\tilde{A}) \leq \kappa(A)$*, where* $\kappa(\cdot)$ *is the spectral condition number.*

Proof: Let $x \in \mathcal{S}$ in each of the following statements:

1. Based on direct calculation,

$$\rho(x, \tilde{A}) = \frac{(P_{\mathcal{S}}Ax, x)}{(x, x)} = \frac{(Ax, P_{\mathcal{S}}x)}{(x, x)} = \frac{(Ax, x)}{(x, x)} = \rho(x, A).$$

2. Invoking part 1 for the second equation, we have

$$\begin{aligned} \|r(x; \tilde{A})\| &= \|\tilde{A}x - \rho(x, \tilde{A})x\| = \|P_{\mathcal{S}}Ax - \rho(x, \tilde{A})x\| \\ &= \|P_{\mathcal{S}}Ax - \rho(x, A)x\| = \|P_{\mathcal{S}}(Ax - \rho(x, A)x)\| \\ &\leq \|P_{\mathcal{S}}\|\|Ax - \rho(x, A)x\| \\ &= \|Ax - \rho(x, A)x\| = \|r(x; A)\|. \end{aligned}$$

Assuming that $\mathcal{S}$ is invariant w.r.t A, starting with the second equation, we have

$$\|r(x; \tilde{A})\| = \|P_{\mathcal{S}}Ax - \rho(x, A)x\| = \|Ax - \rho(x, A)x\| = \|r(x; A)\|.$$

3. If A has a Ritz–pair $(\widetilde{\lambda}, \tilde{u})$, then $\tilde{A}\tilde{u} - \widetilde{\lambda}\tilde{u} = 0$. If $y \in \mathcal{S}$, then

$$\begin{aligned} (A\tilde{u} - \widetilde{\lambda}\tilde{u}, y) &= (A\tilde{u} - \widetilde{\lambda}\tilde{u}, P_{\mathcal{S}}y) = (P_{\mathcal{S}}(A\tilde{u} - \widetilde{\lambda}\tilde{u}), y) \\ &= ((P_{\mathcal{S}}A\tilde{u} - \widetilde{\lambda}\tilde{u}), y) = ((\tilde{A}\tilde{u} - \widetilde{\lambda}\tilde{u}), y) = 0. \end{aligned}$$

and $(A\tilde{u} - \widetilde{\lambda}\tilde{u}) \in \mathcal{S}^{\perp}$.

On the other hand, if $(A\tilde{u} - \widetilde{\lambda}\tilde{u}) \in \mathcal{S}^{\perp}$, then $P_{\mathcal{S}}(A\tilde{u} - \widetilde{\lambda}\tilde{u}) = 0$, therefore

$$\begin{aligned}\|\tilde{A}\tilde{u} - \widetilde{\lambda}\tilde{u}\|^2 &= (\tilde{A}\tilde{u} - \widetilde{\lambda}\tilde{u}, \tilde{A}\tilde{u} - \widetilde{\lambda}\tilde{u}) \\ &= (P_{\mathcal{S}}(A\tilde{u} - \widetilde{\lambda}\tilde{u}), P_{\mathcal{S}}(A\tilde{u} - \widetilde{\lambda}\tilde{u})) = 0.\end{aligned}$$

Thus, $\tilde{A}\tilde{u} - \widetilde{\lambda}\tilde{u} = 0$ and $(\widetilde{\lambda}, \tilde{u})$ is a Ritz-pair of A.

4. Let $\tilde{u}_1, \ldots, \tilde{u}_m$ be an orthonormal set of Ritz vectors. For $1 \leq k \leq m$, let $\widetilde{\mathcal{G}}^k = \langle \tilde{u}_1, \ldots, \tilde{u}_k \rangle$. Then

$$\widetilde{\lambda}_k = \max{}_{g \in \widetilde{\mathcal{G}}^k}\ \rho(g; A).$$

This follows, since if $g = \sum_{i=1}^k \alpha_i \tilde{u}_i$, with $\sum_{i=1}^k \alpha_i^2 = 1$, then

$$\rho(g; A) = (A(\sum_{i=1}^k \alpha_i \tilde{u}_i), \sum_{i=1}^k \alpha_i \tilde{u}_i) = \sum_{i=1}^k \widetilde{\lambda}_i \alpha_i^2. \tag{3.11}$$

Orthonormality of Ritz vectors and the fact that $A\tilde{u}_i - \widetilde{\lambda}_i \tilde{u}_i \in \mathcal{S}^{\perp}$ imply

$$(A\tilde{u}_i, \tilde{u}_j) = \begin{cases} \widetilde{\lambda}_i & \text{if } i = j; \\ 0 & \text{if } i \neq j; \end{cases}$$

which is needed in (3.11).

Then by Theorem 3.2

$$\lambda_k = \min{}_{\substack{\mathcal{G}^k \subseteq \mathbf{R^n} \\ \dim\ \mathcal{G}^k = k}}\ \max{}_{g \in \mathcal{G}^k}\ \rho(g; A) \leq \max{}_{g \in \widetilde{\mathcal{G}}^k}\ \rho(g; A) = \widetilde{\lambda}_k.$$

Clearly

$$\widetilde{\lambda}_k = \max{}_{g \in \widetilde{\mathcal{G}}^k}\ \rho(g; A) \leq \max{}_{g \in \mathbf{R^n}}\ \rho(g; A) = \lambda_n.$$

Thus $\lambda_k \leq \widetilde{\lambda}_k \leq \lambda_n$.

5. Invoking part 4, the spectral condition number satisfies

$$\kappa(\tilde{A}) = \frac{\widetilde{\lambda}_m}{\widetilde{\lambda}_1} \leq \frac{\lambda_n}{\lambda_1} = \kappa(A).$$

■

In our analysis concerning the Rayleigh Quotient, we will consider some subspaces that are 2–dimensional. Restricting the analysis to a 2-D subspace is a powerful approach that can yield results, which can then be extended to larger dimensional spaces, as in Lemma 3.4. In the literature, this is referred to as a "mini–dimensional" analysis, e.g., [31, 32, 39, 44, 45, 46]. Another interesting application of this concept is to prove that the field of values (numerical range) of an operator is convex [19].

3.3 Analysis in 2–Dimensions

Assume that a matrix $A \in \mathbf{R}^{2\times 2}$ is symmetric and has eigenvalues $\lambda_1 \leq \lambda_2$, with orthonormal eigenvectors, respectively, u_1 and u_2. Let $x = \alpha_1 u_1 + \alpha_2 u_2$, and $y = \beta_1 u_1 + \beta_2 u_2$, with $\|x\| = 1$ and $\|y\| = 1$. We have that $|\alpha_1| = \cos(\angle\{x, u_1\})$, $|\alpha_2| = \cos(\angle\{x, u_2\})$, $|\beta_1| = \cos(\angle\{y, u_1\})$ and $|\beta_2| = \cos(\angle\{y, u_2\})$. Then

$$\rho(y) - \rho(x) = \lambda_1(\beta_1^2 - \alpha_1^2) + \lambda_2(\beta_2^2 - \alpha_2^2),$$

and utilizing the assumptions that $\alpha_1^2 + \alpha_2^2 = 1$ and $\beta_1^2 + \beta_2^2 = 1$, we obtain

$$\begin{aligned} \rho(y) - \rho(x) &= (\lambda_2 - \lambda_1)(\beta_1^2 - \alpha_1^2) \\ &= (\lambda_2 - \lambda_1)(\alpha_2^2 - \beta_2^2). \end{aligned}$$

The first equation yields

$$\rho(y) - \rho(x) = (\lambda_2 - \lambda_1)(\alpha_1\beta_2 + \alpha_2\beta_1)\ (\alpha_1\beta_2 - \alpha_2\beta_1),$$

therefore

$$|\rho(y) - \rho(x)| = (\lambda_2 - \lambda_1)|(\alpha_1\beta_2 + \alpha_2\beta_1)|\ |(\alpha_1\beta_2 - \alpha_2\beta_1)|. \tag{3.12}$$

We know

$$\cos(\angle\{x, y\}) = |\alpha_1\beta_1 + \alpha_2\beta_2|,$$

and expressing the angle between x and y as

$$\begin{aligned} \sin^2(\angle\{x, y\}) &= 1 - (x, y)^2 \\ &= 1 - (\alpha_1\beta_1 + \alpha_2\beta_2)^2 \\ &= (\alpha_1\beta_2 - \alpha_2\beta_1)^2, \end{aligned}$$

we obtain

$$\sin(\angle\{x, y\}) = |\alpha_1\beta_2 - \alpha_2\beta_1|.$$

There are two very useful formulations involving sine and tangent. First

$$\begin{aligned} |\rho(y) - \rho(x)| &= (\lambda_2 - \lambda_1)|(\alpha_1\beta_2 + \alpha_2\beta_1)|\ |(\alpha_1\beta_2 - \alpha_2\beta_1)| \\ &= (\lambda_2 - \lambda_1)|\alpha_1\beta_2 + \alpha_2\beta_1| \sin(\angle\{x, y\}) \\ &\leq (\lambda_2 - \lambda_1)(|\alpha_1||\beta_2| + |\alpha_2||\beta_1|) \sin(\angle\{x, y\}) \\ &= (\lambda_2 - \lambda_1)G(x, y) \sin(\angle\{x, y\}), \end{aligned} \tag{3.13}$$

where

$$\begin{aligned} G(x,y) &= |\alpha_1||\beta_2| + |\alpha_2||\beta_1| \\ &= \cos(\angle\{x,u_1\})\ \sin(\angle\{y,u_1\}) \\ &\quad + \sin(\angle\{x,u_1\})\ \cos(\angle\{y,u_1\}). \end{aligned} \tag{3.14}$$

To compute the formula involving the tangent, assume $\cos(\angle\{x,y\}) \neq 0$, then

$$\begin{aligned} |\rho(y) - \rho(x)| &= (\lambda_2 - \lambda_1)|(\alpha_1\beta_2 + \alpha_2\beta_1)|\ |(\alpha_1\beta_2 - \alpha_2\beta_1)| \\ &= (\lambda_2 - \lambda_1)|(\alpha_1\alpha_2 + \beta_1\beta_2)|\frac{|(\alpha_1\beta_2 - \alpha_2\beta_1)|}{|(\alpha_1\beta_1 + \alpha_2\beta_2)|} \\ &= (\lambda_2 - \lambda_1)|(\alpha_1\alpha_2 + \beta_1\beta_2)|\tan(\angle\{x,y\}) \\ &\leq (\lambda_2 - \lambda_1)(|\alpha_1||\alpha_2| + |\beta_1||\beta_2|)\tan(\angle\{x,y\}). \end{aligned} \tag{3.15}$$

We can express the residual $r(x;A)$ in terms of α_1 and α_2 as

$$\begin{aligned} \|r(x;A)\|^2 &= (\lambda_1 - \rho(x;A))^2\alpha_1^2 + (\lambda_2 - \rho(x;A))^2\alpha_2^2 \\ &= (\lambda_1 - (\lambda_1\alpha_1^2 + \lambda_2\alpha_2^2))^2\alpha_1^2 + (\lambda_2 - (\lambda_1\alpha_1^2 + \lambda_2\alpha_2^2))^2\alpha_2^2 \\ &= (\lambda_2 - \lambda_1)^2(\alpha_2^4\alpha_1^2 + \alpha_1^4\alpha_2^2) \\ &= (\lambda_2 - \lambda_1)^2(\alpha_1^2\alpha_2^2)(\alpha_1^2 + \alpha_2^2) \\ &= (\lambda_2 - \lambda_1)^2(\alpha_1^2\alpha_2^2). \end{aligned}$$

Thus

$$\|r(x;A)\| = (\lambda_2 - \lambda_1)|\alpha_1|\ |\alpha_2|. \tag{3.16}$$

Similarly

$$\|r(y;A)\| = (\lambda_2 - \lambda_1)|\beta_1|\ |\beta_2|. \tag{3.17}$$

We note that

$$\frac{2}{\lambda_2 - \lambda_1}\sqrt{\frac{\|r(x)\|\|r(y)\|}{\|x\|\|y\|}} \leq G(x, y) \leq 1. \tag{3.18}$$

The upper bound follows from either the Cauchy–Schwarz inequality or a double angle formula. Considering the lower bound, we have

$$G(x, y) = \frac{1}{\lambda_2 - \lambda_1}\Big[t\frac{\|r(x)\|}{\|x\|} + \frac{1}{t}\frac{\|r(y)\|}{\|y\|}\Big], \tag{3.19}$$

where $t = \dfrac{|\beta_2|}{|\alpha_2|}$. Taking a minimum of this equation as a function of t, with $t > 0$, we obtain the lower bound in (3.18).

3.3.1 Eigenvalue Accuracy in 2–Dimensions

We first present results in 2–dimensions for eigenvalue errors.

Lemma 3.5 *Let* $A \in \mathbf{R}^{\mathbf{2\times 2}}$ *be a symmetric matrix with eigenvalues* $\lambda_1 < \lambda_2$, *and with corresponding eigenvectors, respectively,* u_1 *and* u_2, *and let* $x \in \mathbf{R}^{\mathbf{2}}$ *with* $x \neq 0$. *Then*

$$\frac{\rho(x) - \lambda_1}{\lambda_2 - \lambda_1} = \sin^2(\angle\{x, u_1\}). \tag{3.20}$$

Furthermore, if $\cos(\angle\{x, u_1\}) \neq 0$ *then*

$$\frac{\rho(x) - \lambda_1}{\lambda_2 - \rho(x)} = \tan^2(\angle\{x, u_1\}). \tag{3.21}$$

Proof: Let $y = u_1$ in (3.12) and without loss of generality assume $\|x\| = 1$, then since $\sin^2(\angle\{x, u_1\}) = 1 - \alpha_1^2 = \alpha_2^2$, we have

$$\frac{\rho(x) - \lambda_1}{\lambda_2 - \lambda_1} = \alpha_2^2 = \sin^2(\angle\{x, u_1\}).$$

Considering the second equation and rearranging (3.20), we have

$$\frac{\lambda_2 - \lambda_1}{\lambda_2 - \rho(x)} = \frac{1}{\cos^2(\angle\{x, u_1\})}$$

and combining this again with (3.20) we obtain the second equation. ■

Lemma 3.6 *Let $A \in \mathbf{R}^{\mathbf{2}\times\mathbf{2}}$ be a symmetric matrix and let λ be an eigenvalue of A with corresponding eigenvector u. Let $x \in \mathbf{R}^{\mathbf{2}}$ with $x \neq 0$ and assume $\cos(\angle\{x, u\}) \neq 0$, then*

$$\begin{aligned} |\rho(x) - \lambda| &\leq \frac{\|r(x)\|}{\|x\|}\tan(\angle\{x, u\}) \\ &= \|(I - P_x)AP_x\|\tan(\angle\{x, u\}). \end{aligned}$$

Proof: Considering (3.15), since $y = u$ is an eigenvector, either $\beta_1 = 0$ or $\beta_2 = 0$. Combining this result with (3.16) we obtain the first equation. The second equation then follows from Lemma 3.1. ■

This result is equivalent to Theorem 1.1 of [33], with the trial space being one dimensional and equal to the span of the vector x.

3.3.2 Rayleigh Quotient Perturbations in 2–Dimensions

In this section we present results concerning Rayleigh quotient perturbations in 2–dimensions.

Lemma 3.7 *Let $A \in \mathbf{R}^{\mathbf{2}\times\mathbf{2}}$ be a symmetric matrix and let $x, y \in \mathbf{R}^{\mathbf{2}}$ with $x, y \neq 0$. Then*

$$|\rho(y) - \rho(x)| \leq (\lambda_2 - \lambda_1)\sin(\angle\{x, y\}).$$

This inequality is sharp, in a sense that equality is achieved for all x and y such that $|\alpha_1\beta_2 - \alpha_2\beta_1| = 1$ in (3.13).

Proof: This follows from (3.13) and (3.18). Obviously equality is achieved for $|\alpha_1\beta_2 - \alpha_2\beta_1| = 1$ in (3.13). ■

Lemma 3.8 *Let $A \in \mathbf{R}^{\mathbf{2\times 2}}$ be a symmetric matrix and let $x, y \in \mathbf{R}^{\mathbf{2}}$ with $x, y \neq 0$, and assume $\cos(\angle\{x,y\}) \neq 0$. Then*

$$\begin{aligned} |\rho(x) - \rho(y)| &\leq \left[\frac{\|r(x)\|}{\|x\|} + \frac{\|r(y)\|}{\|y\|}\right] \tan(\angle\{x,y\}) \\ &= \left[\|(I-P_x)AP_x\| + \|(I-P_y)AP_y\|\right] \tan(\angle\{x,y\}). \end{aligned}$$

This inequality is sharp, in a sense that equality is achieved for all x and y such that $\alpha_1\alpha_2\beta_1\beta_2 \geq 0$ in (3.15).

Proof: The first equation follows from (3.15), (3.16) and (3.17), and the second equation follows from Lemma 3.1. Considering (3.15), since

$$|\alpha_1\alpha_2 + \beta_1\beta_2| = |\alpha_1||\alpha_2| + |\beta_1||\beta_2|,$$

if and only if $\alpha_1\alpha_2\beta_1\beta_2 \geq 0$, the statement concerning sharpness follows. ■

In the next section we perform a more general analysis.

3.4 Analysis in the General Case

3.4.1 Eigenvalue Accuracy

We first present results for eigenvalue errors. The following result is discussed in [31] and [33], and appears in equation (1.3) of [32].

Theorem 3.3 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix. Let the eigenvalues of A be $\lambda_1 < \lambda_2 \le \cdots \le \lambda_n$, with corresponding eigenvectors, respectively, $u_1, u_2, \ldots, u_n$. Let $x \in \mathbf{R^n}$ with $x \neq 0$, then*

$$\sin^2(\angle\{x,u_1\}) \le \frac{\rho(x)-\lambda_1}{\lambda_2-\lambda_1} \le \frac{\lambda_n-\lambda_1}{\lambda_2-\lambda_1}\sin^2(\angle\{x,u_1\}). \tag{3.22}$$

Proof: We will perform a "mini–dimensional" 2-D analysis on the subspace $\mathcal{S} = \langle x, u_1\rangle \subseteq \mathbf{R^n}$. Assume that x and u_1 are linearly independent, otherwise $\rho(x) = \rho(u_1)$ and we are done. Let $\tilde{A} = P_{\mathcal{S}}A|_{\mathcal{S}}$. The subspace $\mathcal{S}$ is 2–dimensional and A has Ritz values $\widetilde{\lambda}_1 \le \widetilde{\lambda}_2$ with $\widetilde{\lambda}_1 = \lambda_1$. We have

$$\begin{aligned} \rho(x;A) - \lambda_1 &= \rho(x;\tilde{A}) - \lambda_1 && \text{(Lemma 3.4, part 1)} \\ &= (\widetilde{\lambda}_2 - \lambda_1)\sin^2(\angle\{x,u_1\}) && \text{(Lemma 3.5)} \\ &\le (\lambda_n - \lambda_1)\sin^2(\angle\{x,u_1\}) && \text{(Lemma 3.4, part 4).} \end{aligned} \tag{3.23}$$

Dividing both sides of (3.23) by $\lambda_2 - \lambda_1$, we obtain the upper bound in (3.22). To obtain the lower bound in (3.22), using the second equality in (3.23), we have

$$\sin^2(\angle\{x,u_1\}) = \frac{\rho(x)-\lambda_1}{\widetilde{\lambda}_2-\lambda_1} \le \frac{\rho(x)-\lambda_1}{\lambda_2-\lambda_1}.$$

To obtain the upper bound here, we use the fact that $\widetilde{\lambda}_2 - \lambda_1 \ge \lambda_2 - \lambda_1$, since $\lambda_2 \le \widetilde{\lambda}_2$ by Lemma 3.4, part 4. ■

Corollary 3.1 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix. Let the eigenvalues of A be $\lambda_1 \le \cdots \le \lambda_n$, with corresponding eigenvectors, respectively, $u_1, \ldots, u_n$. Let $x \in \mathbf{R^n}$ with $x \neq 0$ and assume $\cos(\angle\{x,u_1\}) \neq 0$, then*

$$\frac{\rho(x)-\lambda_1}{\lambda_n-\rho(x)} \le \tan^2(\angle\{x,u_1\}). \tag{3.24}$$

Proof: By Theorem 3.3,

$$\frac{\rho(x) - \lambda_1}{\lambda_n - \lambda_1} \le \sin^2(\angle\{x, u_1\}).$$

Then using the fact that $\dfrac{x}{1-x}$ is a non-decreasing function on $[0, 1)$, we have

$$\frac{\dfrac{\rho(x) - \lambda_1}{\lambda_n - \lambda_1}}{1 - \dfrac{\rho(x) - \lambda_1}{\lambda_n - \lambda_1}} = \frac{\rho(x) - \lambda_1}{\lambda_n - \rho(x)} \le \frac{\sin^2(\angle\{x, u_1\})}{1 - \sin^2(\angle\{x, u_1\})} = \tan^2(\angle\{x, u_1\}).$$

■

Theorem 3.4 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix. Let λ be any eigenvalue of A with corresponding eigenvector u. Then for $x \neq 0$ and $\cos(\angle\{x, u\}) \neq 0$ we have*

$$\begin{aligned} |\rho(x) - \lambda| &\le \frac{\|r(x)\|}{\|x\|} \tan(\angle\{x, u\}) && (3.25) \\ &= \|(I - P_x)AP_x\| \tan(\angle\{x, u\}), && (3.26) \end{aligned}$$

where P_x is the orthogonal projector onto the one–dimensional subspace spanned by the vector x.

Proof: We will perform a "mini–dimensional" 2–D analysis on the subspace $\mathcal{S} = \langle x, u\rangle \subseteq \mathbf{R^n}$. Assume that x and u are linearly independent, otherwise $\rho(x) = \rho(u)$ and we are done. Therefore we will assume that the subspace $\mathcal{S}$ is

2–dimensional. Let $\tilde{A} = P_{\mathcal{S}} A|_{\mathcal{S}}$. To prove (3.25) we have

$$\begin{aligned}
|\rho(x) - \rho(u)| &= |\rho(x; A) - \rho(u; A)| \\
&= |\rho(x; \tilde{A}) - \rho(u; \tilde{A})| && \text{(Lemma 3.4, part 1)} \\
&\leq \frac{\|r(x; \tilde{A})\|}{\|x\|} \tan(\angle\{x, u\}) && \text{(Lemma 3.6)} \\
&\leq \frac{\|r(x; A)\|}{\|x\|} \tan(\angle\{x, u\}) && \text{(Lemma 3.4, part 2)} \\
&= \frac{\|r(x)\|}{\|x\|} \tan(\angle\{x, u\})
\end{aligned}$$

And equation (3.26) follows from Lemma 3.1. ■

The result of Theorem 3.4 is equivalent to Theorem 1.1 of [33], with the trial space being one dimensional and equal to the span of the vector x. Also see Theorem 3.2 of [33].

The following appears to be a new result.

Corollary 3.2 *Let $A \in \mathbf{R^{n \times n}}$ be a symmetric matrix. Let λ be an eigenvalue of A with corresponding eigenvector u. Assume that none of the quantities $x, Ax, \rho(x), \cos(\angle\{x, u\})$ are zero, then*

$$\frac{|\rho(x) - \lambda|}{|\rho(x)|} \leq \tan(\angle\{x, Ax\}) \tan(\angle\{x, u\}).$$

Proof: This result follows from Lemma 3.2 and Theorem 3.4. ■

In the next section we present results concerning Rayleigh quotient perturbations.

3.4.2 Rayleigh Quotient Perturbations

Theorem 3.5 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix. Let $\lambda_1 \le \cdots \le \lambda_n$ be the eigenvalues of A, and let $x, y \in \mathbf{R^n}$ with $x, y \neq 0$. Then*

$$|\rho(y) - \rho(x)| \le (\lambda_n - \lambda_1)\sin(\angle\{x,y\}). \tag{3.27}$$

Proof: We will perform a "mini–dimensional" 2–D analysis on the subspace $\mathcal{S} = \langle x, y\rangle \subseteq \mathbf{R^n}$. Assume that x and y are linearly independent, otherwise $\rho(x) = \rho(y)$ and we are done. Therefore assume that the subspace $\mathcal{S}$ is 2–dimensional. As usual we use the following notation: A has Ritz values $\widetilde{\lambda}_1 \le \widetilde{\lambda}_2$, and $\tilde{A} = P_{\mathcal{S}}A|_{\mathcal{S}}$. Then

$$\begin{aligned}
|\rho(y) - \rho(x)| &= |\rho(y; A) - \rho(x; A)| \\
&= |\rho(y; \tilde{A}) - \rho(x; \tilde{A})| \qquad \text{(Lemma 3.4, part 1)} \\
&\le (\widetilde{\lambda}_2 - \widetilde{\lambda}_1)\sin(\angle\{x,y\}) \quad \text{(Lemma 3.7)} \\
&\le (\lambda_n - \lambda_1)\sin(\angle\{x,y\}) \quad \text{(Lemma 3.4, part 4)}
\end{aligned}$$

■

We now provide an alternative independent proof of Theorem 3.5.

Proof: Without loss of generality, assume that $\|x\| = \|y\| = 1$. Then using the fact that $|\rho(y; A) - \rho(x; A)|$ is independent of a shift to the matrix A by any constant times the identity, we have

$$\begin{aligned}
|\rho(y) - \rho(x)| &= |\rho(y; A) - \rho(x; A)| = |(Ax, x) - (Ay, y)| \\
&= |(A(x-y), x+y)| = |((A - (\frac{\lambda_1 + \lambda_n}{2})I)(x-y), x+y)|.
\end{aligned}$$

Then by the Cauchy–Schwarz inequality

$$\begin{aligned}
&|((A - (\frac{\lambda_1+\lambda_n}{2})I)(x-y), x+y)| \\
&\qquad\qquad \leq\ 2\|A - (\frac{\lambda_1+\lambda_n}{2})I\|\Big[\frac{\|x+y\|\|x-y\|}{2}\Big] \\
&\qquad\qquad =\ (\lambda_n - \lambda_1)\sin(\angle\{x,y\}).
\end{aligned}$$

■

The following result is discussed in [31].

Corollary 3.3 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric positive definite matrix. And let $x, y \in \mathbf{R^n}$ with $x, y \neq 0$, then*

$$\frac{|\rho(y)-\rho(x)|}{\rho(x)} \leq (\kappa(A)-1)\sin(\angle\{x,y\}), \tag{3.28}$$

where $\kappa(A)$ is the spectral condition number of A.

Proof: Since $\rho(x) \geq \lambda_1$, this result follows from Theorem 3.5. ■

It appears that the following is a new result.

Theorem 3.6 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix, and let $x, y \in \mathbf{R^n}$ with $x, y \neq 0$, then assuming that* $\cos(\angle\{x,y\}) \neq 0$ *we have*

$$\begin{aligned}
&|\rho(y)-\rho(x)| \\
&\quad \leq\ \Big[\frac{r(y)}{\|y\|} + \frac{r(x)}{\|x\|}\Big]\tan(\angle\{x,y\}) && (3.29) \\
&\quad =\ \big[\|(I-P_y)AP_y\| + \|(I-P_x)AP_x\|\big]\tan(\angle\{x,y\}). && (3.30)
\end{aligned}$$

Furthermore, if none of the quantities $Ax, Ay, \rho(x), \rho(y)$ *are zero, then*

$$\begin{aligned} & |\rho(y) - \rho(x)| \\ & \quad \leq \big[|\rho(x)| \tan(\angle\{x, Ax\}) + |\rho(y)| \tan(\angle\{y, Ay\})\big] \tan(\angle\{x, y\}). \end{aligned} \tag{3.31}$$

Proof: Similar to the proof of Theorem 3.5, we will perform a "mini–dimensional" 2–D analysis on the subspace $\mathcal{S} = \langle x, y \rangle \subseteq \mathbf{R^n}$. Assume that x and y are linearly independent, otherwise $\rho(x) = \rho(y)$ and we are done. Therefore we will assume that the subspace $\mathcal{S}$ is 2–dimensional. Let $\tilde{A} = P_{\mathcal{S}} A|_{\mathcal{S}}$. To prove (3.29) we have

$$\begin{aligned} |\rho(y) - \rho(x)| &= |\rho(y; A) - \rho(x; A)| \\ &= |\rho(y; \tilde{A}) - \rho(x; \tilde{A})| && \text{(Lemma 3.4, part 1)} \\ &\leq \Big[\frac{\|r(y; \tilde{A})\|}{\|y\|} + \frac{\|r(x; \tilde{A})\|}{\|x\|}\Big] \tan(\angle\{x, y\}) && \text{(Lemma 3.8)} \\ &\leq \Big[\frac{\|r(y; A)\|}{\|y\|} + \frac{\|r(x; A)\|}{\|x\|}\Big] \tan(\angle\{x, y\}) && \text{(Lemma 3.4, part 2)} \\ &= \Big[\frac{\|r(y)\|}{\|y\|} + \frac{\|r(x)\|}{\|x\|}\Big] \tan(\angle\{x, y\}). \end{aligned}$$

Equation (3.30) follows from Lemma 3.1, and equation (3.31) follows from Lemma 3.2. ■

We now provide an alternative independent proof of Theorem 3.6, by adopting the arguments used in the proof of Theorem 3.2 of [33].

Proof: Let P_x and P_y be orthogonal projectors onto the one–dimensional subspaces spanned respectively, by x and y. We have

$$|\rho(y) - \rho(x)| \|P_x P_y\| = \|(\rho(y) - \rho(x)) P_x P_y\|.$$

Now

$$
\begin{aligned}
&(\rho(y)-\rho(x))P_xP_y \\
&\quad = P_x(\rho(y)P_y) - (\rho(x)P_x)P_y \\
&\quad = P_xP_yAP_y - P_xAP_xP_y \\
&\quad = P_x(P_yA - AP_x)P_y \\
&\quad = P_x(P_yA - A + A - AP_x)P_y \\
&\quad = P_x(P_y - I)AP_y + P_xA(I-P_x)P_y \\
&\quad = P_x(P_y - I)(P_y - I)AP_y + P_xA(I-P_x)(I-P_x)P_y.
\end{aligned}
$$

In the second equality above, we use the result in (3.5) that implies $\rho(x)P_x = P_xAP_x$ and $\rho(y)P_y = P_yAP_y$. We have

$$\|(\rho(y)-\rho(x))P_xP_y\| \le \|(I-P_y)AP_y\|\|P_x(I-P_y)\| + \|P_xA(I-P_x)\|\|(I-P_x)P_y\|.$$

Thus,

$$
\begin{aligned}
&|\rho(y)-\rho(x)| \\
&\quad \le \|(I-P_y)AP_y\|\frac{\|P_x(I-P_y)\|}{\|P_xP_y\|} + \|(I-P_x)AP_x\|\frac{\|(I-P_x)P_y\|}{\|P_xP_y\|}.
\end{aligned}
$$

According to Theorem 2.4, $\|P_xP_y\| = \cos(\angle\{x,y\})$, and

$$\frac{\|P_x(I-P_y)\|}{\|P_xP_y\|} = \frac{\|(I-P_x)P_y\|}{\|P_xP_y\|} = \tan(\angle\{x,y\}).$$

Therefore,

$$|\rho(y)-\rho(x)| \le [\|(I-P_y)AP_y\| + \|(I-P_x)AP_x\|]\tan(\angle\{x,y\}).$$

Again, equation (3.30) follows from Lemma 3.1, and equation (3.31) follows from Lemma 3.2. ■

We claim that the inequality in Theorem 3.6 is sharp in a sense that there are values of $x, y \in \mathbf{R^n}$ where equality is achieved. Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix and let $\lambda_1 \leq \lambda_2$ be any two eigenvalues of A with corresponding eigenvectors u_1 and u_2. Suppose that $x = \alpha_1 u_1 + \alpha_2 u_2$ and $y = \beta_1 u_1 + \beta_2 u_2$. If $\mathcal{S}$ is 1–dimensional, then $\rho(x) = \rho(y)$ and $\tan(\angle\{x, y\}) = 0$. Thus, we have equality. So now assume $\mathcal{S}$ is 2–dimensional (i.e. x and y are linearly independent). Then the subspace $\mathcal{S} = \langle x, y\rangle$ is invariant w.r.t the matrix A. So by Lemma (3.4), part 2, $\|r(x; \tilde{A})\| = \|r(x; A)\|$ and $\|r(y; \tilde{A})\| = \|r(y; A)\|$. Since $u_1, u_2 \in \mathcal{S}$, they are Ritz vectors of A w.r.t. $\mathcal{S}$ (as well as being eigenvectors of A). Working in the subspace $\mathcal{S}$, Lemma (3.8) applies directly, so we achieve equality if and only if $\alpha_1\alpha_2\beta_1\beta_2 \geq 0$. Thus, for any symmetric matrix A, it is easy to exhibit vectors x, y, where equality is achieved in Theorem 3.6. It appears that we can find examples of equality in the case when the subspace $\mathcal{S}$ is invariant w.r.t A. It is unknown whether we can achieve equality in the case that $\mathcal{S}$ is not invariant w.r.t. A.

In general, the inequalities (3.27) and (3.30) are sharp in the following sense. We have for $s \in (0, 1]$

$$\sup_{\sin(\angle\{x,y\})=s} \frac{|\rho(y) - \rho(x)|}{\sin(\angle\{x, y\})} = \lambda_n - \lambda_1, \tag{3.32}$$

and for $t \in (0, \infty)$

$$\sup_{\tan(\angle\{x,y\})=t} \frac{|\rho(y) - \rho(x)|}{\left[\|(I - P_x)AP_x\| + \|(I - P_y)AP_y\|\right] \tan(\angle\{x, y\})} = 1. \tag{3.33}$$

Thus, these constants cannot be improved. The first equation is proved in [37] and the second equation has a similar proof.

3.5 Bounds for Rayleigh–Ritz Estimates

Suppose instead of individual vectors x and y, we consider two subspaces $\mathcal{X}$, $\mathcal{Y} \subseteq \mathbf{R^n}$, that are of the same dimension. The result in this section illustrates an application of Rayleigh quotient inequalities and concepts involving principal angles between subspaces, e.g., [40].

3.5.1 Eigenvalue Approximation by Ritz Values

In this section we present results concerning eigenvalue accuracy and the Rayleigh–Ritz procedure. We state a known result [6] for the situation when one of the subspaces is spanned by eigenvectors corresponding to the first m eigenvalues.

Theorem 3.7 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix, and let the eigenvalues of A be $\lambda_1 \leq \cdots \leq \lambda_n$ with corresponding eigenvectors $u_1, \ldots, u_n$. Let $\mathcal{X}$ be an m–dimensional subspace of $\mathbf{R^n}$, and suppose $\widetilde{\lambda}_1 \leq \cdots \leq \widetilde{\lambda}_m$ are the Ritz values of A w.r.t $\mathcal{X}$. Let $\mathcal{U} = \langle u_1, \ldots u_m \rangle$ be the subspace spanned by the first m eigenvectors. Then*

$$0 \leq \widetilde{\lambda}_j - \lambda_j \leq (\lambda_n - \lambda_1)\sin(\angle\{\mathcal{X},\mathcal{U}\})^2, \qquad j = 1, \ldots, m. \tag{3.34}$$

The largest principal angle $\angle\{\mathcal{X},\mathcal{U}\}$, is defined in (2.2).

In the next section we present results concerning perturbation bounds and the Rayleigh–Ritz procedure.

3.5.2 Perturbation of Ritz Values

It appears that the following is a new result.

Theorem 3.8 *Let $A \in \mathbf{R^{n\times n}}$ be a symmetric matrix and let the eigenvalues of A be $\lambda_1 \le \cdots \le \lambda_n$, Let $\mathcal{X}$ and $\mathcal{Y}$ both be $m-$dimensional subspaces of $\mathbf{R^n}$, and suppose $\alpha_1 \le \cdots \le \alpha_m$ are the Ritz values for A w.r.t $\mathcal{X}$, and $\beta_1 \le \cdots \le \beta_m$ are the Ritz values for A w.r.t $\mathcal{Y}$. Then*

$$|\alpha_j - \beta_j| \le (\lambda_n - \lambda_1)\sin(\angle\{\mathcal{X},\mathcal{Y}\}) \quad j = 1,\ldots,m, \tag{3.35}$$

and if in addition, A is positive definite, then the relative error is bounded by

$$\frac{|\alpha_j - \beta_j|}{\alpha_j} \le (\kappa(A) - 1)\sin(\angle\{\mathcal{X},\mathcal{Y}\}) \quad j = 1,\ldots,m, \tag{3.36}$$

where $\kappa(A)$ is the spectral condition number of A. The largest principal angle $\angle\{\mathcal{X},\mathcal{Y}\}$, is defined in (2.2).

Proof: To prove (3.35), let $Q_\mathcal{X},\ Q_\mathcal{Y} \in \mathbf{R^{n\times m}}$ be matrices who's columns are orthonormal bases, respectively of the subspaces $\mathcal{X}$ and $\mathcal{Y}$. We can choose these such that $Q_\mathcal{X}^T Q_\mathcal{Y} = \text{diag}\ (\sigma_1, \ldots, \sigma_m) = \Lambda$ by virtue of (2.1). Here $\sigma_1 \ge \cdots \ge \sigma_m$ and σ_m is the cosine of the largest principal angle. By Weyl's Theorem (Theorem 5.1 of [13]), we have

$$|\alpha_j - \beta_j| \le \|Q_\mathcal{X}^T A Q_\mathcal{X} - Q_\mathcal{Y}^T A Q_\mathcal{Y}\| \quad j = 1,\ldots,m.$$

By the definition of the 2–norm, there exists a vector $v \in \mathbf{R^m}$, which is an eigenvector of $Q_{\mathcal{X}}^T A Q_{\mathcal{X}} - Q_{\mathcal{Y}}^T A Q_{\mathcal{Y}}$, such that

$$\begin{aligned} \|Q_{\mathcal{X}}^T A Q_{\mathcal{X}} - Q_{\mathcal{Y}}^T A Q_{\mathcal{Y}}\| &= |\rho(v, Q_{\mathcal{X}}^T A Q_{\mathcal{X}} - Q_{\mathcal{Y}}^T A Q_{\mathcal{Y}})| \\ &= |\rho(v, Q_{\mathcal{X}}^T A Q_{\mathcal{X}}) - \rho(v, Q_{\mathcal{Y}}^T A Q_{\mathcal{Y}})| \\ &= |\rho(Q_{\mathcal{X}} v, A) - \rho(Q_{\mathcal{Y}} v, A)|. \qquad (3.37) \end{aligned}$$

Let $x = Q_{\mathcal{X}} v \in \mathcal{X}$ and $y = Q_{\mathcal{Y}} v \in \mathcal{Y}$, then by Theorem 3.5,

$$\begin{aligned} \|Q_{\mathcal{X}}^T A Q_{\mathcal{X}} - Q_{\mathcal{Y}}^T A Q_{\mathcal{Y}}\| &= |\rho(x; A) - \rho(y; A)| \\ &\leq (\lambda_n - \lambda_1) \sin(\angle\{x, y\}) \\ &\leq (\lambda_n - \lambda_1) \sin(\angle\{\mathcal{X}, \mathcal{Y}\}). \qquad (3.38) \end{aligned}$$

This last inequality holds since

$$\begin{aligned} \sin^2(\angle\{x, y\}) &= 1 - \cos^2(\angle\{x, y\}) = 1 - \left[\frac{(Q_{\mathcal{X}}^T Q_{\mathcal{Y}} v, v)}{\|Q_{\mathcal{X}} v\| \|Q_{\mathcal{Y}} v\|}\right]^2 \\ &= 1 - \rho(v, \Lambda)^2 \\ &\leq 1 - \sigma_m^2 = 1 - \cos^2(\theta_m) \\ &= \sin^2(\theta_m) = \sin^2(\angle\{\mathcal{X}, \mathcal{Y}\}. \qquad (3.39) \end{aligned}$$

Equation (3.36) follows, since $\alpha_j \geq \lambda_1, \quad j = 1, \ldots, m$ by Lemma 3.4, part 4. ■

In the next section we provide an alternative approach.

3.6 Rayleigh Quotient Perturbations using a Matrix Inner Product

Consider $x \in \mathbf{R^n}$ and a matrix $A \in \mathbf{R^{n\times n}}$, then $\rho(x; A)$ is nonlinear in x and linear in A. Using a matrix inner product, we can express the Rayleigh quotient as a bilinear form. For fixed x, $l_x(A) = \rho(x; A)$ is a bounded linear functional on $\mathbf{R^{n\times n}}$. Now consider the inner product defined on the matrices $A, B \in \mathbf{R^{n\times n}}$ by

$$(A, B) = \text{trace}(A^T B). \tag{3.40}$$

This is called the Frobenius inner product e.g., Section 5.1 of [28], and $\|A\|_F = \sqrt{(A, A)}$ is the Frobenius matrix norm. Again assuming that x is fixed, according to the Riesz representation theorem (Theorem 3.10.1 of [10]), there exists a matrix $C_x \in \mathbf{R^{n\times n}}$, such that

$$l_x(A) = \rho(x; A) = (A, C_x),$$

which holds for all $A \in \mathbf{R^{n\times n}}$. Interestingly, it can be proved that $C_x = P_x$, where P_x is the orthogonal projector onto x.

Lemma 3.9 *For all $x \in \mathbf{R^n}$ and all $A \in \mathbf{R^{n\times n}}$, with the Rayleigh quotient $\rho(x; A)$ defined as in (3.1), we have*

$$\rho(x; A) = (A, P_x), \tag{3.41}$$

where $(\cdot, \cdot)$ is the inner product defined by (3.40) and P_x is the orthogonal projector onto x.

Proof: If $x \in \mathbf{R^n}$ with $x \neq 0$ and P_x is the orthogonal projection onto x, then trace(P_x)=1. This follows since P_x is a rank one operator with exactly one nonzero eigenvalue which is equal to one. We have

$$\begin{aligned}\rho(x;A) &= \rho(x) = \rho(x)\mathrm{trace}(P_x)\\ &= \mathrm{trace}(\rho(x)P_x) = \mathrm{trace}(P_xAP_x)\\ &= \mathrm{trace}(AP_x^2) = \mathrm{trace}(AP_x) = (A, P_x).\end{aligned}$$

In the second equality we use the result in (3.5) that implies $\rho(x)P_x = P_xAP_x$.

■

Although it is not as general as this development, a similar formulation is presented in Section 5.7 of [27], in relation to the numerical range of an operator.

In this investigation we are often interested in the difference in the Rayleigh quotient for two vectors $x, y \in \mathbf{R^n}$ with $x, y \neq 0$. Based on Lemma 3.9,

$$\rho(y;A) - \rho(x;A) = (A, P_y - P_x),$$

and we may observe that this is only a function of A and the difference of orthogonal projectors.

This representation allows us to provide an alternative proof to Theorem 3.5, which is given below:

Proof: Let $x, y \in \mathbf{R^n}$ with $x, y \neq 0$. Assume that x and y are linearly independent, otherwise $\rho(x;A) = \rho(y;A)$ and we are done. So assume that the span$\{x, y\}$ is 2–dimensional. Let $\mathcal{S} = \mathrm{span}\{x, y\}$ and let $P_{\mathcal{S}}$ be the orthogonal projection onto the subspace $\mathcal{S}$. Let $\tilde{A} = P_{\mathcal{S}}A|_{\mathcal{S}}$ and $\bar{A} = P_{\mathcal{S}}AP_{\mathcal{S}}$. Then

$\tilde{A}x = \bar{A}x$ and $\tilde{A}y = \bar{A}y$. We have

$$\begin{aligned} |\rho(y, A) - \rho(x, A)| &= |\rho(y, \tilde{A}) - \rho(x, \tilde{A})| && \text{(Lemma 3.4, part 1)} \\ &= |\rho(y, \bar{A}) - \rho(x, \bar{A})| \\ &= |(\bar{A}, P_y - P_x)| && \text{(Lemma 3.9)} \\ &= |\mathrm{trace}(\bar{A}(P_y - P_x))|. \end{aligned}$$

The operator $\bar{A}$ has at most two non-zero eigenvalues, which are the Ritz values of A w.r.t $\mathcal{S}$, namely $\widetilde{\lambda}_1$ and $\widetilde{\lambda}_2$ with $\widetilde{\lambda}_1 \leq \widetilde{\lambda}_2$. The two eigenvectors associate with these eigenvalues are in the the subspace $\mathcal{S}$, and the null space of the operator $\bar{A}$ includes the orthogonal complement of $\mathcal{S}$ which is an $n-2$ dimensional subspace. Therefore there exists a unitary matrix $U \in \mathbf{R}^{\mathbf{n}\times\mathbf{n}}$ such that

$$U^T \bar{A} U = \mathrm{diag}(\widetilde{\lambda}_1, \widetilde{\lambda}_2, 0, \ldots, 0).$$

Since the difference of the Rayleigh quotient is translation invariant, we have

$$\begin{aligned} |\rho(y, A) - \rho(x, A)| &= |\mathrm{trace}((\bar{A} - \widetilde{\lambda}_1 I)(P_y - P_x))| \\ &= |\mathrm{trace}(U^T(\bar{A} - \widetilde{\lambda}_1 I)UU^T(P_y - P_x)U)| \\ &= (\widetilde{\lambda}_2 - \widetilde{\lambda}_1)|(U^T(P_y - P_x)U)_{22}| \\ &\leq (\widetilde{\lambda}_2 - \widetilde{\lambda}_1)\|(U^T(P_y - P_x)U)\| \\ &= (\widetilde{\lambda}_2 - \widetilde{\lambda}_1)\|P_y - P_x\| \\ &= (\widetilde{\lambda}_2 - \widetilde{\lambda}_1)\sin(\angle\{x, y\}) \\ &\leq (\lambda_n - \lambda_1)\sin(\angle\{x, y\}). && \text{(Lemma 3.4, part 4)} \end{aligned}$$

In the third equation above we use the fact the absolute value of any diagonal element of a matrix is not greater than the 2–norm of the matrix. This follows since if $B \in \mathbf{R^{n\times n}}$, then $|b_{jj}| = |\rho(e_j;B)| \leq \|B\|$, where e_j is the jth standard basis vector. ■

Remark 3.1 *For a general matrix A, using the Cauchy–Schwarz inequality we have*

$$\begin{aligned} |\rho(y;A) - \rho(x;A)| &= |(A, P_y - P_x)| \\ &\leq \|A\|_F \|P_y - P_x\|_F \leq n\|A\|\|P_y - P_x\| \\ &= n\|A\|\sin(\angle\{x,y\}) \\ &= C\sin(\angle\{x,y\}). \end{aligned} \qquad (3.42)$$

Here $\|P_y - P_x\| = \sin(\angle\{x,y\})$ as in (2.5). This is the type of inequality that we are interested in, but the constant is large. If we put restrictions on the matrix A (.e.g., A is symmetric), we can obtain a much improved (smaller) constant as has been exhibited in Theorem 3.5.

Remark 3.2 *This concept allows for the generalization of the Rayleigh quotient from a function of a vector to a function of a subspace. Let $P_{\mathcal{X}}$ be the orthogonal projector onto the subspace $\mathcal{X}$, then let*

$$\rho(\mathcal{X}, A) = (A, P_{\mathcal{X}}).$$

This representation may have some useful properties. In particular we know that

$$|\rho(\mathcal{Y}, A) - \rho(\mathcal{X}, A)| \leq C\|P_{\mathcal{Y}} - P_{\mathcal{X}}\|$$

for some constant C, and when $\mathcal{X}$ and $\mathcal{Y}$ are both one dimensional, this definition reduces to the standard definition of Rayleigh quotient.

3.7 Conclusions Concerning Rayleigh Quotient Perturbations

Let us formulate here the main points of this section:

- There are two reasons for studying this problem: first, the results can be used in the design and analysis of algorithms, and second, a knowledge of the sensitivity of an eigenpair is of both theoretical and of practical interest.

- Considering Rayleigh quotient perturbations, initially restricting the domain to a 2–D subspace (i.e. performing "mini–dimensional" analysis) is a powerful technique that can yield interesting and significant results in the general case.

- This extension from 2–D to the general case requires a careful analysis of the properties of the Rayleigh quotient, the residual and the spectral condition number relative to the operators A and $P_{\mathcal{S}}A|_{\mathcal{S}}$. A rigorous proof is provided that provides a unique point of view.

- We address the case of a perturbation of general vector, as well as perturbations involving a general vector and an eigenvector, and perturbations involving subspaces. Several completely new results are presented. One of the interesting findings characterizes the perturbation of Ritz values for a symmetric positive definite matrix and two different subspaces, in terms of the spectral condition number and the largest principal angle between the subspaces.

- We also provide several alternative proofs, one of which uses a somewhat unique approach of expressing the Rayleigh quotient as a Frobenius inner product of matrices; $\rho(x;A) = (A, P_x) = \mathrm{trace}(AP_x)$, where P_x is the orthogonal projector onto x.

4. Implementation of a Preconditioned Eigensolver Using Hypre

4.1 Introduction to the Preconditioned Eigenvalue Solver

The goal of this project is to implement a scalable preconditioned eigenvalue solver for the solution of eigenvalue problems for large sparse symmetric matrices on massively parallel computers, using the Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG) [2, 36]. We take advantage of advances in the Scalable Linear Solvers project, in particular in multigrid technology and in incomplete factorizations (ILU) developed under the High Performance Preconditioners (Hypre) project [41], at the Lawrence Livermore National Laboratory, Center for Applied Scientific Computing (LLNL–CASC). The solver allows for the utilization of a set of Hypre preconditioners for solution of an eigenvalue problem.

In this chapter we discuss the implementation approach for a flexible "matrix-free" parallel algorithm, and the capabilities of the developed software. We also discuss testing and the performance of the software on a few simple test problems.

The LOBPCG Hypre software has been integrated into the Hypre software at LLNL and has been included in the recent Beta Release: Hypre–1.8.0b.

4.2 Background Concerning the LOBPCG Algorithm

The iterative method that has been implemented is LOBPCG. The LOBPCG solver finds one or more of the smallest eigenvalues and the corresponding eigenvectors of a symmetric positive definite matrix. The algorithm is “matrix-free” since the multiplication of a vector by the matrix A and an application of the preconditioner T to a vector are needed only as functions. For computing the smallest eigenpair, the algorithm implements a local optimization of a 3–term recurrence, where the Rayleigh–Ritz method is used during each iteration on a trial subspace with

$$x^{n+1} \in \operatorname{span}\{x^n,\ x^{n-1},\ Tr(x^n; A)\}.$$

Here T is the preconditioner, which approximates the inverse of A, and $r(x^n; A) = Ax^n - \rho(x^n; A)x^n$ is the residual as defined in (3.3). In the block version we are finding the m smallest eigenpairs of A. The Rayleigh-Ritz method is used during each iteration for a local optimization, which is initially a $3m$–dimensional trial subspace. By using a method that we will refer to as *soft locking*, which is described below, the size of this subspace, and hence the number of preconditioned solves and the number of matrix–vector multiplies are reduced as iterations progress.

4.3 The Block LOBPCG Algorithm

The LOBPCG algorithm described below, is a variant of Algorithm 4.1 in [36]. Here we solve a standard eigenvalue problem, where Algorithm 4.1 in [36] solves a generalized eigenvalue problem. LOBPCG works for the generalized eigenproblem $Ax = \lambda Bx$, but the present Hypre implementation is only available for $B = I$. The general case is a work in progress.

We present here a high level description of the LOBPCG algorithm:

```
ALGORITHM 4.1: Block LOBPCG Algorithm

Input: m starting vectors $x_1^{(0)}, \dots x_m^{(0)}$, devices to compute: $Ax$
    and $Tx$ for a given vector $x$, and the vector inner product $(x, y)$.

 1. Start: select $x_j^{(0)}$, and set $p_j^{(0)} = 0,\ j = 1, \dots, m$.
 2. Iterate: for $i = 0, \dots,$ Until Convergence Do:
 3.     $\lambda_j^{(i)} := (Ax_j^{(i)}, x_j^{(i)})/(x_j^{(i)}, x_j^{(i)}),\ j = 1, \dots, m$.
 4.     $r_j := A\,x_j^{(i)} - \lambda_j^{(i)} x_j^{(i)},\ j = 1, \dots, m$.
 5.     $w_j^{(i)} := Tr_j,\ j = 1, \dots, m$.
 6.     Use the Rayleigh–Ritz method on the trial subspace
        Span$\{w_1^{(i)}, \dots, w_m^{(i)}, x_1^{(i)}, \dots, x_m^{(i)}, p_1^{(i)}, \dots, p_m^{(i)}\}$.
 7.     $x_j^{(i+1)} := \sum_{k=1,\dots,m} \alpha_k^{(i)} w_k^{(i)} + \tau_k^{(i)} x_k^{(i)} + \gamma_k^{(i)} p_k^{(i)},\ j = 1, \dots, m$
        (the $j$-th Ritz vector corresponding to the $j$-th largest Ritz value).
 8.     $p_j^{(i+1)} := \sum_{k=1,\dots,m} \alpha_k^{(i)} w_k^{(i)} + \gamma_k^{(i)} p_k^{(i)}$.
 9. end for

Output: the approximations $\lambda_j^{(k)}$ and $x_j^{(k)}$ to the largest eigenvalues
    $\lambda_j$ and corresponding eigenvectors $x_j$, $j = 1, \dots, m$.
```

The column-vector

$$\left(\alpha_1^{(i)}, \dots, \alpha_m^{(i)},\ \tau_1^{(i)}, \dots, \tau_m^{(i)}\ \gamma_1^{(i)}, \dots, \gamma_m^{(i)}\right)^T$$

is the j-th eigenvector corresponding to the j-th largest eigenvalue of the $3m$-by-$3m$ eigenvalue problem of the Rayleigh–Ritz method. We observe that

$$p_j^{(i+1)} = x_j^{(i+1)} - \sum_{k=1,\dots,m} \tau_k^{(i)} x_k^{(i)},$$

and thus,

$$\begin{aligned} x_j^{(i+1)} &\in \operatorname{span}\{w_1^{(i)},\ldots,w_m^{(i)},x_1^{(i)},\ldots,x_m^{(i)},p_1^{(i)},\ldots,p_m^{(i)}\} \\ &= \operatorname{span}\{w_1^{(i)},\ldots,w_m^{(i)},x_1^{(i)},\ldots,x_m^{(i)},x_1^{(i-1)},\ldots,x_m^{(i-1)}\}. \end{aligned}$$

Considering the LOBPCG algorithm that is implemented in software (in MATLAB or in Hypre) for block size m, the main arrays are the eigenvectors X, the preconditioned residuals W and the conjugate directions P. Starting with the high level Algorithm 4.1, we have $X = [x_1, \ldots, x_m], W = [w_1, \ldots, w_m]$ and $P = [p_1, \ldots, p_m]$. Let $Z = [X \quad W \quad P]$ contain independent vectors (columns), then using the Rayleigh Ritz procedure we can solve the generalized $m \times m$ eigenvalue problem

$$Z^T AZy = \lambda Z^T Zy.$$

This generalized eigenvalue problems is small compared to the size of the matrix A, since the blocksize m is relatively small. We store the first m generalized eigenvectors in Y corresponding to the first m generalized eigenvalues in increasing order, where the columns of Y are $Z^T Z$–orthonormal. Then the first m orthonormal Ritz vectors are given by ZY, which are used to update X.

Using this approach, W and P may have vectors with small components, and this is especially true as LOBPCG converges. Consequently, the GRAM matrix may be ill–conditioned. This was a problem in MATLAB revision 3.2 (see [38]) of LOBPCG. A cure for this is to orthonormalize Z. However, this orthonormalization causes extra work and also will require more than one matrix–vector multiply per X vector, per iteration. To avoid extra matrix–vector multiplies, we can orthonormalize the columns of W and P only, during each iteration. The eigenvectors in X are kept orthonormalized because of the structure of the algorithm. With this approach we do not incur any extra matrix–vector multiplies and achieve improved stability.

Another improvement was implemented in MATLAB revision 3.3 (see [38]) of LOBPCG, which is also implemented in the Hypre code. This improvement excludes certain vectors from the Rayleigh Ritz procedure, so that the number of vectors to process is reduced as the iterations progress. In the LOBPCG algorithm this process will be referred to as *soft locking*, and is discussed below. There is a similar concept that is discussed in Section 4.3.4 of [2] called *locking* where vectors are frozen in the subspace iteration procedure. These frozen vectors are not changed and are "locked" as iterations progress, and the vectors that are retained in the iteration are kept orthogonal to those that are locked. In LOBPCG, the X vectors are all retained and are updated, but W and P are reduced in size. Hence the term "soft locking" is used, since the vectors in X are not actually frozen. However, the trial subspace is reduced in size.

Considering LOBPCG, assume we are at iteration k, then let $1 \leq l \leq m$ be the number of vectors in X that have a residual less than the tolerance. These are the vectors that are "locked". Let $I = \{i_1, \ldots i_{m-l}\}$ be the index set of vectors that are not locked. Let

$$X_I = [x_{i_1}, \ldots, x_{i_{m-l}}],$$

$$W_I = [w_{i_1}, \ldots, w_{i_{m-l}}],$$

and

$$P_I = [p_{i_1}, \ldots, p_{i_{m-l}}].$$

Then the columns of W and P, that are stored respectively in W_I and P_I, correspond (indexwise) to locked vectors in X. In soft locking all columns (vectors) in X are used in the Rayleigh Ritz procedure together with W_I and P_I. So the vectors that are used in the Rayleigh Ritz procedure are $Z = [X \;\; W_I \;\; P_I]$. The Rayleigh Ritz procedure then solves a $(m + 2(m - l)) \times (m + 2(m - l))$ problem, where l increases as the iterations progress, until $l = m$. In soft locking all of the eigenvectors in X may be updated during each iteration. There are $m - l$ matrix–vector multiplies and $m - l$ applications of the preconditioner per iteration.

There are cases below where we need to use this indexed notation to refer to the corresponding columns in AW, AP, AX respectively as AW_I, AP_I, AX_I, or the corresponding eigenvalues in the diagonal $(m - l) \times (m - l)$ matrix Λ_I.

Below is a detailed description of the LOBPCG algorithm that has been implemented in the Hypre code. This code implements both of the improvements that were discussed above: namely orthonormalization of W and P at each iteration to improve stability and soft locking to improve efficiency. The LOBPCG description in [36] and the description of Algorithm 4.1 are not nearly as complete as the description below.

Algorithm 4.2: **Block LOBPCG Algorithm in Hypre**

```
Input: m starting linearly independent vectors in $X \in \mathbf{R^{n\times m}}$, devices to
       compute $A*x$ and $T*x$.
 1. Allocate memory for the six matrices $X, AX, W, AW, P, AP \in \mathbf{R^{n\times m}}$.
 2. Initial settings for loop: $[Q,R] = qr(X); X = Q;\ AX = A*X;\ k = 0$.
 3. Compute initial Ritz vectors and initial residual:
    $[TMP, \Lambda] = eig(X^T * AX); X = X*TMP; AX = AX*TMP;$
    $W = AX - X*\Lambda$.
 4. Get norm of individual residuals and store in normR $\in \mathbf{R^m}$.
 5. do while (max(normR) > tol)
 6.     Find index set $I$ of normR for normR > tol. The index set $I$
        defines those residual vectors for which the norm is greater
        than the tolerance. Refer to the column vectors defined by $I$,
        that are a subset of the columns of $W$ and $P$, by $W_I$ and $P_I$.
 7.     Apply preconditioner: $W_I = T*W_I$.
 8.     Orthonormalize $W_I$: $[Q,R] = qr(W_I); W_I = Q$.
 9.     Update $AW_I$: $AW_I = A*W_I$.
10.     if $k > 0$
11.         Orthonormalize $P_I$: $[Q,R] = qr(P_I); P_I = Q;$
12.         Update $AP_I$: $AP_I = AP_I * R^{-1}$.
13.     end if
14.     Complete Rayleigh Ritz Procedure:
15.         if $k > 0$
16.             Compute $G = \begin{bmatrix} \Lambda & X^T*AW_I & X^T*AP_I \\ W_I^T*AX & W_I^T*AW_I & W_I^T*AP_I \\ P_I^T*AX & P_I^T*AW_I & P_I^T*AP_I \end{bmatrix}$.
```

Algorithm 4.2: **Block LOBPCG Algorithm in Hypre (continued)**

```
17.         Compute $M = \begin{bmatrix} I & X^T * W_I & X^T * P_I \\ W_I^T * X & I & W_I^T * P_I \\ P_I^T * X & P_I^T * W_I & I \end{bmatrix}$.
18.     else
19.         Compute $G = \begin{bmatrix} \Lambda & X^T * AW_I \\ W_I^T * AX & W_I^T * AW_I \end{bmatrix}$.
20.         Compute $M = \begin{bmatrix} I & X^T * W_I \\ W_I^T * X & I \end{bmatrix}$.
21.     end if
22.     Solve the generalized eigenvalue problem: $G * y = \lambda M * y$
        and store the first $m$ eigenvalues in increasing order in the
        diagonal matrix $\Lambda$ and store the corresponding $m$
        eigenvectors in $Y$.
23. if $k > 0$
24.     Partition $Y = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix}$ according to the number of columns in
        $X$, $W_I$, and $P_I$, respectively.
25.     Compute $X = X * Y_1 + W_I * Y_2 + P_I * Y_3$.
26.     Compute $AX = AX * Y_1 + AW_I * Y_2 + AP_I * Y_3$.
27.     Compute $P_I = W_I * Y_2 + P_I * Y_3$.
28.     Compute $AP_I = AW_I * Y_2 + AP_I * Y_3$.
29. else
30.     Partition $Y = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}$ according to the number of columns in
        $X$ and $W_I$ respectively.
```

ALGORITHM 4.2: **Block LOBPCG Algorithm in Hypre (continued)**

```
31.        Compute $X = X * Y_1 + W_I * Y_2$.
32.        Compute $AX = AX * Y_1 + AW_I * Y_2$.
33.        Compute $P_I = W_I$.
34.        Compute $AP_I = AW_I$.
35.    end if
36.    Compute new residuals: $W_I = AX_I - X_I * \Lambda_I$.
37.    Update residual normR for columns of $W$ in $W_I$.
38.    $k = k + 1$
39. end do
```

Output: *Eigenvectors X and eigenvalues Λ.*

4.4 LOBPCG Hypre Implementation Strategy

This project is based on some previously developed LOBPCG software including:

- MATLAB code,
- C–language code utilizing LAPACK/BLAS libraries,
- PETSc (Portable, Extensible Toolkit for Scientific Computing) code [3].

The experience gained from development of these initial codes was important and used extensively in developing the new parallel Hypre based code. The MATLAB lobpcg.m code is publicly available at the Web page: http://www-math. cudenver.edu/˜aknyazev/software. The author wrote and tested the initial C–language code. The C–language code was used in a Masters

Project by Dave Duran [17] involving normal coordinate analysis of a molecular structure. Both the earlier C-language implementation and the Hyper eigensolver implementation follow the general LOBPCG MATLAB functional organization quite closely. A portion of the initial C–language code was ported and used as a base for the Hypre code.

The Hypre eigensolver code is written in MPI [24], based C–language and uses Hypre and LAPACK libraries. It has been tested with Hypre version 1.6.0 and the LLNL Beta Release: Hypre–1.8.0b. The user interface or Applications Program Interface (API) to the solver is implemented using Hypre style object oriented function calls. The matrix–vector multiply and the preconditioned solve are done through user supplied functions. This approach provides significant flexibility, and the implementation illustrates that the LOBPCG algorithm can successfully and efficiently utilize parallel libraries.

Partition of vector data across the processors is determined by user input consisting of an initial array of parallel vectors. The same partitioning that is used for these vectors, is then used to setup and run the eigensolver. Partitioning of the matrix A and the preconditioner (also determined by the user) must be compatible with the partitioning of these initial vectors.

Preconditioning is implemented through calls to the Hypre preconditioned conjugate gradient method (PCG). LOBPCG has been tested with the following Hypre preconditioned linear solvers:

- AMG–PCG: algebraic multigrid
- DS–PCG: diagonal scaling

- ParaSails–PCG: approximate inverse of A is generated by attempting to minimize $\|I - AM\|_F$
- Schwarz–PCG: additive Schwarz
- Euclid–PCG: incomplete LU

In our tests, only a few (2-10) inner iterations typically provide the best performance and increasing the number of the inner iterations beyond this range, does not improve the final convergence. Figure 4.4 in Section 4.8 illustrates this observation.

4.5 Software Implementation Details

The implementation of this algorithm has been done using C–language using LAPACK/BLAS libraries and Hypre libraries (hypre–1.6.0). Hypre supports four conceptual interfaces: Struct, SStruct, FEM and IJ. This project uses the Linear–Algebraic Interface (IJ), since this interface supports applications with sparse linear systems. A block diagram of the high level software modules is given in Figure 4.1.

The test driver ij_es.c, is a modified version of the Hypre driver ij.c, which retains much of its functionality and capabilities. This driver is used to run the eigensolver software with control through command line parameters. Various test matrices can be internally generated that consist of different types of Laplacians, or a matrix market file may be read as input. The preconditioner can also be chosen by the user. An initial set of vectors can be generated randomly, as an identity matrix of vectors or a matrix market file may be input,

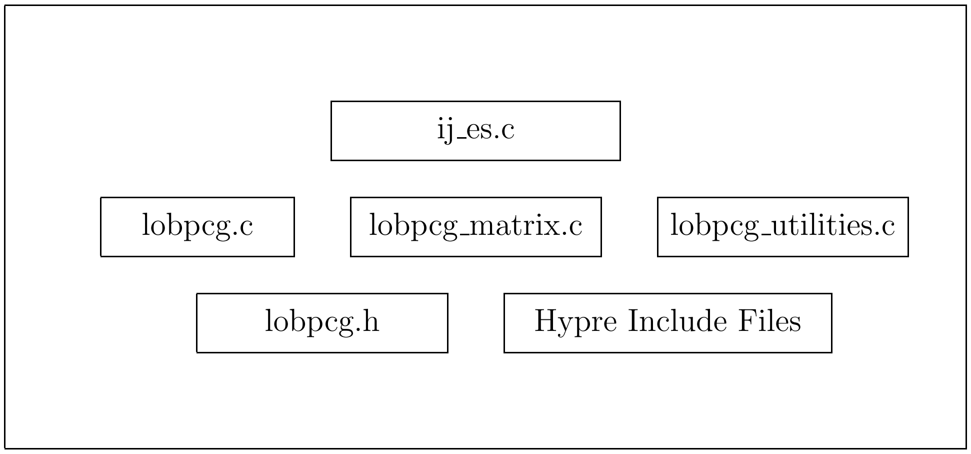


Figure 4.1: LOBPCG Hypre software modules

with it's columns used as initial vectors. Also various other parameters can be specified such as the convergence tolerance, the maximum number of iterations and the maximum number of inner iterations for the Hypre solver.

Of course a user would normally write code for a specific application using the LOBPCG API. In this case the test driver ij_es.c can serve as a template.

The routine lobpcg.c implements the main algorithm and the API routines. The routine lobpcg_matrix.c contains the functions that implement parallel Hypre vector operations. It also implements a parallel Modified Gram–Schmidt (MGS) orthonormalization algorithm. The routine lobpcg_utilities.c implements miscellaneous functions such as reading a matrix market file and broadcasting the data to all processors. We provide a listing of the primary functions for this Hypre implementation of LOBPCG in Appendix C.

This entire development amounts to approximately $4,000$ lines of non-commentary source code.

4.6 Applications Program Interface (API)

The user interface to the eigensolver has been implemented in a style similar to the Hypre standard API. This interface consists of object oriented functions that create an instance of the eigensolver, that run the eigensolver and finally that destroy the instance of the eigensolver. The functions that implement the eigensolver API are described in Appendix B.

4.7 Compiling and Testing

The LOBPCG Hypre code has been mainly developed and tested on a Beowulf cluster at CU Denver. This cluster computer includes 36 nodes, 2 processors per node, PIII 933MHz processors, 2GB memory per node running Linux Redhat and a 7.2SCI Dolpin interconnect. Make files for various hardware configurations and compilers are provided. LOBPCG code has been compiled and run on CU Denver cluster using scali and mpich libraries and using gcc, pgcc and mpicc compilers.

This code has also been compiled and run on various LLNL Open Computing Facility (OCF) Production Systems including ASCI blue, M&IC tera, gps and lx. LOBPCG has been tested on several LLNL clusters using Compaq and IBM hardware, running Unix and/or Linux.

A test script ij_es.sh has been developed, which is similar to the Hypre test script to ij_linear_solvers.sh in Hypre Release 1.6.0. This script can be run after compilation to determine basic sanity and correctness of the compile and to determine if there is compatibility for running this code in the computing environment of interest. The tests run in this script test combinations of user

```
ij_es.sh

mpirun -np 4 ij_es -9pt -solver 12 -pcgitr 2 -vrand 1
mpirun -np 4 ij_es -9pt -solver 12 -pcgitr 2 -vrand 2
mpirun -np 4 ij_es -9pt -solver 12 -pcgitr 2 -vrand 4
mpirun -np 4 ij_es -9pt -solver 12 -pcgitr 2 -vrand 8
mpirun -np 4 ij_es -27pt -solver 1 -vrand 5 -pcgitr 1
mpirun -np 4 ij_es -27pt -solver 1 -vrand 5 -pcgitr 2
mpirun -np 4 ij_es -27pt -solver 1 -vrand 5 -pcgitr 3
mpirun -np 4 ij_es -laplacian -n 10 10 10 -solver 43
-vin Xin.mtx -pcgitr 2
mpirun -np 4 ij_es -ain laplacian_10_10_10.mtx -solver 43
-vin Xin.mtx -pcgitr 2
mpirun -np 4 ij_es -ain laplacian_10_10_10.mtx
-tin laplacian_10_10_10.mtx -solver 43 -vin Xin.mtx -pcgitr 2
mpirun -np 4 ij_es -27pt -n 50 50 50 -solver 1
-vrand 40 -pcgitr 2
mpirun -np 4 ij_es -27pt -n 100 100 100 -solver 2 -vrand
4 -pcgitr 2
```

Table 4.1: LOBPCG test script

input parameters, and are executed using four processors and are summarized below in Table 4.1.

Overall testing has been done using a variety of internally generated Laplacians. Quality and accuracy of code, using multiple processors, seems to be quite good based in this limited initial testing.

4.8 Numerical Results

We first test scalability by varying the problem size so it is proportional to the number of processors. We use a 7–Point 3–D Laplacian where n is the number of grid points on each side of a cube. We vary n so that the number of non–zeroes nz, is proportional to the number of processors, varying n from $n = 100$ to $n = 252$. We set the block size to 1, set the maximum LOBPCG iterations to 10, set the maximum number of (inner) iterations of the preconditioned solver to 3, and use the Schwarz–PCG preconditioner. We measure the time for setup of the preconditioner, the time for applying the preconditioner and the time for remaining LOBPCG work. The preconditioner is setup before execution of LOBPCG. The time for application of the preconditioner is the sum of the time for each preconditioned/solve step during execution of the LOBPCG algorithm. This data is summarized in Table 4.2 and displayed as a bar chart in Figure 4.2. For this problem we observe good scalability.

Nproc	n	nz	Precond. Setup	Apply Precond.	LOBPCG Linear Alg.	Total (sec)
2	100	6,940,000	918.5	155.7	17.7	1091.9
4	126	13,907,376	914.9	161.1	19.4	1095.4
8	159	27,986,067	937.9	163.9	20.3	1122.1
16	200	55,760,000	730.9	166.9	25.3	923.1
32	252	112,000,000	671.0	163.8	38.9	873.7

Table 4.2: Timing results – scalability

In the next numerical example we use a 7–Point 3–D Laplacian and vary the problem size so that $n = 79, 100, 126, 159, 200, 252$ to make the total number of non-zeros proportional to the number of processors. We set the block size to 1, the maximum number of Hyper Solver PCG iterations to 10, and we let the LOBPCG iterations run until convergence using the default tolerance of 10^{-6}. We use the Schwarz–PCG preconditioner. The detailed data for this experiment is summarized in Table 4.3. As we can see from this table, more iterations are required as the problem becomes larger.

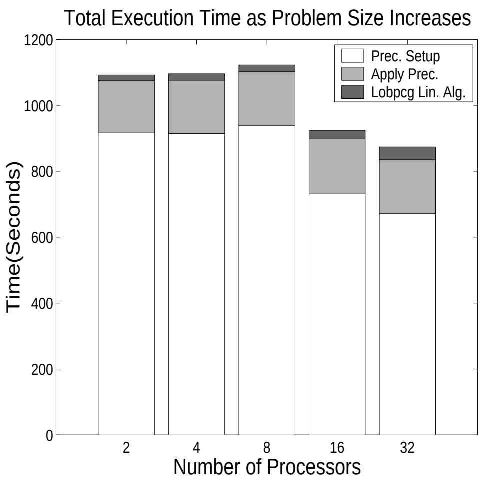


Figure 4.2: LOBPCG scalability as problem size increases

In Figure 4.3, on the left graph, we plot the total time to converge as the problem size and number of processors increase. This time includes 1) the time to setup the preconditioner, 2) the accumulated time to apply the preconditioner during LOBPCG execution and 3) the accumulated time for matrix vector multiplies and other linear algebra during LOBPCG execution. As we can see the time increases as the problem size increases, with the time being dominated by the application of the preconditioner. The accumulated time for matrix vector multiplies and other linear algebra is a relatively small proportion of the total time.

In Figure 4.3, on the right graph, we plot the total time/iteration as the problem size and number of processors increase for 1) the average time/iteration to apply the preconditioner during LOBPCG execution and 2) the average time/iteration for matrix vector multiplies and other linear algebra during LOBPCG execution. Good scalability is exhibited in this graph for the average time/iteration. Again the time/iteration for matrix vector multiplies and other linear algebra is a relatively small proportion of the total time/iteration.

We may observe that for the largest problem size, we are dealing with a sparse matrix of approximate size 16M $\times$ 16M and corresponding vectors of size 16M. This is a good test of the capability of LOBPCG to handle the linear algebra associated with large problems.

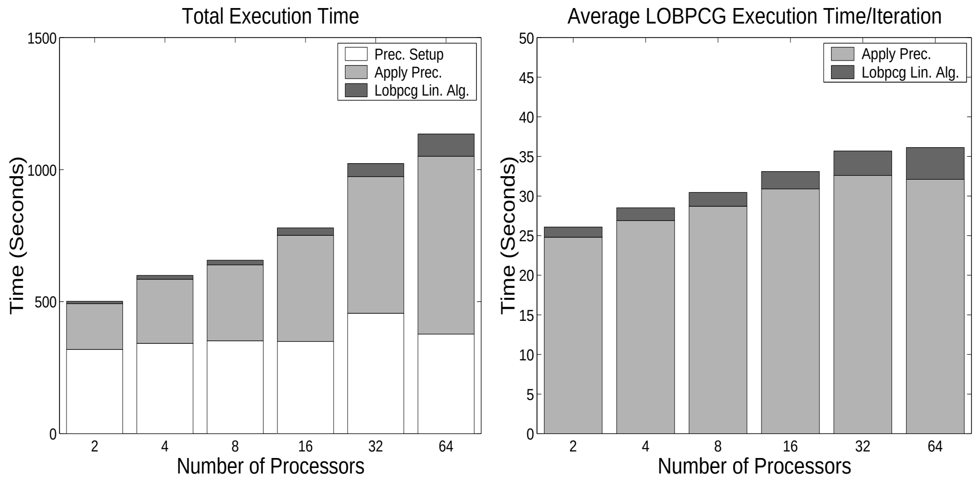


Figure 4.3: Execution time to converge as problem size increases

Nproc	Nz	Prec. setup (Seconds)	Apply Prec. (Seconds)	LOBPCG Lin. Alg. (Seconds)	Itr	Apply/Itr (Seconds)	Alg/Itr (Seconds)
2	3.4M	319	173.4	9.10	7	24.8	1.30
4	6.9M	342	242.8	14.5	9	26.9	1.61
8	13.9M	352	287.2	17.6	10	28.7	1.76
16	27.9M	349	401.9	28.6	13	30.9	2.20
32	55.8M	456	517.8	49.4	16	32.6	3.09
64	111.6M	377	673.4	84.5	21	32.1	4.03

Table 4.3: Scalability data for 3–D Laplacian

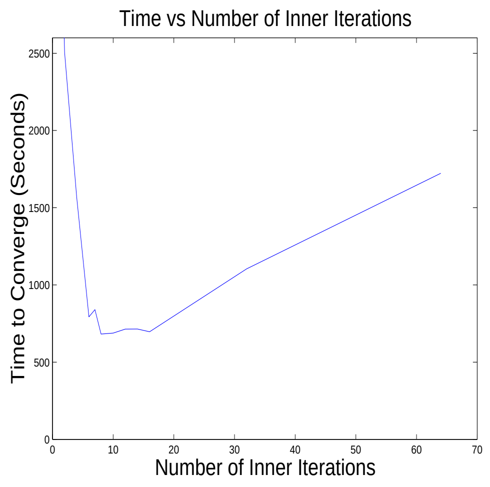


Figure 4.4: Performance versus quality of preconditioner

In the next numerical example we use a 7–Point 3–D Laplacian with $n = 200$, $nz = 55,760,000$ and with the Schwarz–PCG preconditioner. We run the problem on 10 processors. We run LOBPCG until a tolerance of 10^{-6} is achieved. We measure the execution time as we vary the quality of the preconditioner by changing the maximum number of iterations (inner iterations) that are used in the Hypre solver. The results of this experiment are displayed in Figure 4.4. We find that on this problem the optimal number of inner iterations is between 7 and 10.

In the next numerical example we use a 7–Point 3–D Laplacian with $n = 100$, $nz = 6,940,000$, the blocksize equal to 10 and we vary the type of preconditioner. We run the problem on 10 processors. We run LOBPCG until a tolerance of 10^{-10} is achieved. We illustrate convergence for six different runs as follows: no preconditioner, DS–PCG, ParaSails–PCG, Schwarz–PCG, Euclid–PCG and AMG–PCG. The results of this experiment are displayed in Figure 4.5. We plot the results in order of improving convergence. We also give the run time in seconds. On this problem the AMG–PCG preconditioner has the fastest run time and converges in the smallest number of iterations which is 30..

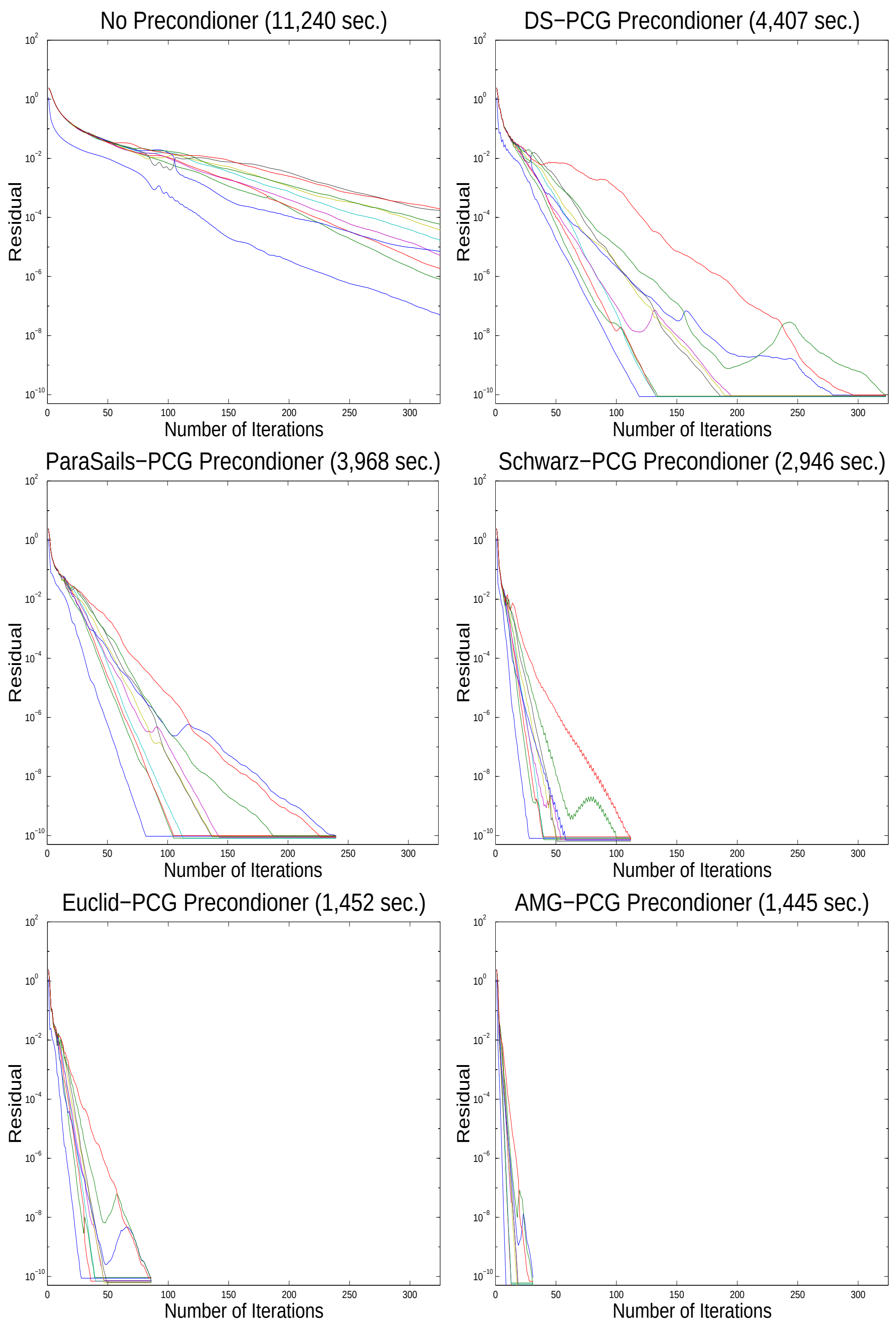


Figure 4.5: Convergence rates for different preconditioners

4.9 Conclusions Concerning the Hypre LOBPCG Implementation

Let us formulate here the main points of this section:

- This implementation illustrates that this "matrix-free" algorithm can be successfully implemented using parallel libraries that are designed to run on a great variety of multiprocessor platforms.
- We gain a significant leverage and flexibility in solving large sparse eigenvalue problems because the user can provide their own matrix–vector multiply and preconditioned solver functions and/or use the standard Hypre preconditioned solver functions.
- The user interface routines
  - Provide ease of use for a variety of users.
  - Have been developed with the goal of using the Hypre standard user interface.
- Initial scalability measurements look promising, but more testing is needed by other users. However, scalability is mainly dependent on the scalability within Hypre since most of the computations are involved with building and applying the preconditioner. In practical problems 90%–99% of the computational effort is required for building the preconditioner and in the applying the preconditioner during execution of the algorithm.
- Enhancements are possible to improve robustness, efficiency and readability/clarity of code.

- The LOBPCG Hypre software has been integrated into the Hypre software at LLNL and has been included in the recently released Hypre Beta Release – Hypre–1.8.0b, and is now available for users to test.

5. Overall Conclusions and Impact

5.1 Overall Conclusions

In this dissertation we focus on three related areas of research: 1) principal angles between subspaces including theory and numerical results, 2) Rayleigh quotient and Rayleigh Ritz perturbations and eigenvalue accuracy, and 3) parallel software implementation of the eigensolver LOBPCG using LLNL Hypre. Let us formulate here the main conclusions of this dissertation:

- Considering principal angles between subspaces the main points are:
  - A bug in the cosine–based algorithm for computing principal angles between subspaces, which prevents one from computing small angles accurately in computer arithmetic, is illustrated.
  - An algorithm is presented that computes all principal angles accurately in computer arithmetic and is proved to be equivalent to the standard algorithm in exact arithmetic.
  - A generalization of the algorithm to an arbitrary scalar product given by a symmetric positive definite matrix is suggested and justified theoretically.
  - Perturbation estimates for absolute errors in cosine and sine of principal angles, with improved constants and for an arbitrary scalar product, are derived.

  - A description of the code is given as well as results of numerical tests. The code is robust in practice and provides accurate angles for large-scale and ill-conditioned cases we tested numerically. It is also reasonably efficient for large-scale applications with $n \gg p$.
  - Our algorithms are "matrix-free"; i.e., they do not require storing in memory any matrices of the size n and are capable of dealing with A, which may be specified only as a function that multiplies A by a given vector.
  - MATLAB release 13 has implemented the SVD–based sine version of our algorithm.

- Considering Rayleigh quotient perturbations the main points are:
  - There are two reasons for studying this problem: first, the results can be used in the design and analysis of algorithms, and second, a knowledge of the sensitivity of an eigenpair is of both theoretical and of practical interest.
  - Considering Rayleigh quotient perturbations, initially restricting the domain to a 2–D subspace (i.e. performing "mini–dimensional" analysis) is a powerful technique that can yield interesting and significant results in the general case.
  - This extension from 2–D to the general case requires a careful analysis of the properties of the Rayleigh quotient, the residual and the spectral condition number relative to the operators A and $P_{\mathcal{S}}A|_{\mathcal{S}}$. A

rigorous proof is provided that provides a unique point of view.

  - We address the case of a perturbation of general vector, as well as perturbations involving a general vector and an eigenvector, and perturbations involving subspaces. Several completely new results are presented. One of the interesting findings characterizes the perturbation of Ritz values for a symmetric positive definite matrix and two different subspaces, in terms of the spectral condition number and the largest principal angle between the subspaces.
  - We also provide several alternative proofs, one of which uses a somewhat unique approach of expressing the Rayleigh quotient as a Frobenius inner product of matrices; $\rho(x; A) = (A, P_x) = \text{trace}(AP_x)$, where P_x is the orthogonal projector onto x.

- Considering the parallel software implementation of the eigensolver LOBPCG using LLNL Hypre the main points are:
  - This implementation illustrates that this “matrix-free” algorithm can be successfully implemented using parallel libraries that are designed to run on a great variety of multiprocessor platforms.
  - We gain a significant leverage and flexibility in solving large sparse eigenvalue problems because the user can provide their own matrix–vector multiply and preconditioned solver functions and/or use the standard Hypre preconditioned solver functions.
  - The user interface routines

    * Provide ease of use for a variety of users.
    * Have been developed with the goal of using the Hypre standard user interface.

  - Initial scalability measurements look promising, but more testing is needed by other users. However, scalability is mainly dependent on the scalability within Hypre since most of the computations are involved with building and applying the preconditioner. In practical problems 95%–99% of the computational effort is required for building the preconditioner and in the applying the preconditioner during execution of the algorithm.
  - Enhancements are possible to improve robustness, efficiency and readability/clarity of code.
  - The LOBPCG Hypre software has been integrated into the Hypre software at LLNL and has been included in the recently released Hypre Beta Release – Hypre–1.8.0b, and so is now available for users to test.

5.2 Impact

Concerning computation of principal angles between subspaces, we propose several MATLAB based algorithms that compute both small and large angles accurately, and illustrate their practical robustness with numerical examples. We prove basic perturbation theorems for absolute errors for sine and cosine of principal angles with improved constants. And MATLAB release 13 has implemented our SVD–based algorithm for the sine. The impact of this work is to provide a practical algorithm and code for computation that is generally available. This work also provides a rigorous background and theory, and a framework for understanding concepts involving angles between subspaces..

Concerning Rayleigh quotient and Rayleigh Ritz perturbations and eigenvalue accuracy, some relevant and interesting perturbations are discussed and proofs are included. The main impact of this work is that several completely new results are obtained, and this work underscores the value of doing a 2–D "mini–dimensional" analysis. These results have definite practical application, since the Rayleigh quotient is unarguably the most important function used in the analysis and computation of eigenvalues of symmetric matrices.

Concerning the parallel software implementation of the LOBPCG algorithm using Hypre, the major impact is that the user gains significant power, leverage and flexibility in solving large sparse symmetric eigenvalue problems because this LOBPCG implementation provides a flexible "matrix-free" parallel algorithm, which allows for the utilization of Hypre parallel libraries and high performance preconditioners that Hypre provides. This parallel framework also allows the user to provide their own preoconditioned solvers in a flexible manner.

6. Future Work Directions

Concerning principal angles between subspaces, one area of future work is a sparse implementation of Algorithm 2.2. This sparse implementation has possible application to information retrieval and/or web search engines. Also, there are some interesting and useful extensions to these ideas to angles between closed subspaces of a Hilbert space [15].

Concerning Rayleigh quotient perturbations and eigenvalue accuracy, there are several areas of future work. First the concept of "mini–dimensional" 2–D analysis proved to be an effective approach in this work, and there are other areas where this concept can be beneficially applied. Also can

$$|\rho(x) - \rho(y)| \leq \left[\|(I - P_x)AP_x\| + \|(I - P_y)AP_y\|\right] \tan(\angle\{x, y\})$$

be used to prove

$$|\alpha_j - \beta_j| \leq \left[\|(I - P_{\mathcal{X}})AP_{\mathcal{X}}\| + \|(I - P_{\mathcal{Y}})AP_{\mathcal{Y}}\|\right] \tan(\angle\{\mathcal{X}, \mathcal{Y}\}) \qquad j = 1, \ldots, m$$

where $\alpha_j, \beta_j \quad j = 1, \ldots, m$ are the Ritz values for A w.r.t $\mathcal{X}$ and $\mathcal{Y}$? Currently this is a conjecture.

Concerning the implementation of a preconditioned eigensolver using Hypre, there is an opportunity to make enhancements based on feedback from users. Also, one area of possible new work is the implement of the generalized eigenvalue problem $Ax = \lambda Bx$.

A MATLAB Code for SUBSPACEA.M and ORTHA.M

```
function [theta,varargout] = subspacea(F,G,A)
%SUBSPACEA angles between subspaces
%  subspacea(F,G,A)
%  Finds all min(size(orth(F),2),size(orth(G),2)) principal angles
%  between two subspaces spanned by the columns of matrices F and G
%  in the A-based scalar product x'*A*y, where A
%  is Hermitian and positive definite.
%  COS of principal angles is called canonical correlations in statistics.
%  [theta,U,V] = subspacea(F,G,A) also computes left and right
%  principal (canonical) vectors - columns of U and V, respectively.
%
%  If F and G are vectors of unit length and A=I,
%  the angle is ACOS(F'*G) in exact arithmetic.
%  If A is not provided as a third argument, than A=I and
%  the function gives the same largest angle as SUBSPACE.m by Andrew Knyazev,
%  see http://www.mathworks.com/matlabcentral/fileexchange
%  /Files.jsp?type=category&id=&fileId=54
%  or ftp://ftp.mathworks.com/pub/contrib/v5/linalg/subspace.m.
%  MATLAB's SUBSPACE.m function, Rev. 5.5-5.8 has a bug and fails to compute
%  small angles accurately.
%
%  The optional parameter A is a Hermitian and positive definite matrix,
%  or a corresponding function. When A is a function, it must accept a
%  matrix as an argument.
%  This code requires ORTHA.m, Revision 1.5.5 or above,
%  which is included. The standard MATLAB version of ORTH.m
%  is used for orthonormalization, but could be replaced by QR.m.
%
%  The algorithm is described in A. V. Knyazev, M. E. Argentati,
%  Principal Angles between Subspaces in an $A$-Based Scalar Product:
%  Algorithms and Perturbation Estimates, Submitted to SIAM SISC.
%  See http://www-math.cudenver.edu/~aknyazev/research/papers/
%
%  Tested under MATLAB R10-12.1
%  Copyright (c) 2001 Freeware for noncommercial use.
%  Andrew Knyazev, Rico Argentati
%  Contact email: knyazev@na-net.ornl.gov
%  $Revision: 4.4 $  $Date: 2001/10/4


threshold=sqrt(2)/2; % Define threshold for determining when an angle is small

if size(F,1) ~= size(G,1)
   error(['The row dimension ' int2str(size(F,1)) ...
         ' of the matrix F is not the same as ' int2str(size(G,1)) ...
         ' the row dimension of G'])
end

if nargin<3  % Compute angles using standard inner product

   % Trivial column scaling first, if ORTH.m is used later
   for i=1:size(F,2),
     normi=norm(F(:,i),inf);
     %Adjustment makes tol consistent with experimental results
     if normi > eps^.981
       F(:,i)=F(:,i)/normi;
       % Else orth will take care of this
     end
   end
   for i=1:size(G,2),
     normi=norm(G(:,i),inf);
     %Adjustment makes tol consistent with experimental results
     if normi > eps^.981
       G(:,i)=G(:,i)/normi;
       % Else orth will take care of this
     end
   end

  % Compute angle using standard inner product

  QF = orth(F);      %This can also be done using QR.m, in which case
```

```
QG = orth(G);      %the column scaling above is not needed

q = min(size(QF,2),size(QG,2));
[Ys,s,Zs] = svd(QF'*QG,0);
if size(s,1)==1
  % make sure s is column for output
  s=s(1);
end
s = min(diag(s),1);
theta = max(acos(s),0);
U = QF*Ys;
V = QG*Zs;
indexsmall = s > threshold;
if max(indexsmall) % Check for small angles and recompute only small
  RF = U(:,indexsmall);
  RG = V(:,indexsmall);
  %[Yx,x,Zx] = svd(RG-RF*(RF'*RG),0);
  [Yx,x,Zx] = svd(RG-QF*(QF'*RG),0); % Provides more accurate results
  if size(x,1)==1
    % make sure x is column for output
    x=x(1);
  end
  Tmp = fliplr(RG*Zx);
  V(:,indexsmall) = Tmp(:,indexsmall);
  U(:,indexsmall) = RF*(RF'*V(:,indexsmall))*...
  diag(1./s(indexsmall));
  x = diag(x);
  thetasmall=flipud(max(asin(min(x,1)),0));
  theta(indexsmall) = thetasmall(indexsmall);
end

% Compute angle using inner product relative to A
else
  [m,n] = size(F);
  if ~isstr(A)
    [mA,mA] = size(A);
    if any(size(A) ~= mA)
      error('Matrix A must be a square matrix or a string.')
    end
  if size(A) ~= m
     error(['The size ' int2str(size(A)) ...
           ' of the matrix A is not the same as ' int2str(m) ...
           ' - the number of rows of F'])
  end
end

[QF,AQF]=ortha(A,F);
[QG,AQG]=ortha(A,G);
q = min(size(QF,2),size(QG,2));
[Ys,s,Zs] = svd(QF'*AQG,0);
if size(s,1)==1
  % make sure s is column for output
  s=s(1);
end
s=min(diag(s),1);
theta = max(acos(s),0);
U = QF*Ys;
V = QG*Zs;
indexsmall = s > threshold;
if max(indexsmall) % Check for small angles and recompute only small
  RG = V(:,indexsmall);
  AV = AQG*Zs;
  ARG = AV(:,indexsmall);
  RF = U(:,indexsmall);
  %S=RG-RF*(RF'*(ARG));
  S=RG-QF*(QF'*(ARG));% A bit more cost, but seems more accurate

  % Normalize, so ortha would not delete wanted vectors
  for i=1:size(S,2),
    normSi=norm(S(:,i),inf);
    %Adjustment makes tol consistent with experimental results
    if normSi > eps^1.981
      QS(:,i)=S(:,i)/normSi;
```

```
        % Else ortha will take care of this
      end
    end

    [QS,AQS]=ortha(A,QS);
    [Yx,x,Zx] = svd(AQS'*S);
    if size(x,1)==1
      % make sure x is column for output
      x=x(1);
    end
    x = max(diag(x),0);

    Tmp  = fliplr(RG*Zx);
    ATmp = fliplr(ARG*Zx);
    V(:,indexsmall) = Tmp(:,indexsmall);
    AVindexsmall = ATmp(:,indexsmall);
    U(:,indexsmall) = RF*(RF'*AVindexsmall)*...
                       diag(1./s(indexsmall));
    thetasmall=flipud(max(asin(min(x,1)),0));

    %Add zeros if necessary
    if sum(indexsmall)-size(thetasmall,1)>0
      thetasmall=[zeros(sum(indexsmall)-size(thetasmall,1),1)',...
           thetasmall']';
    end

    theta(indexsmall) = thetasmall(indexsmall);
  end
end
varargout(1)={U(:,1:q)};
varargout(2)={V(:,1:q)};

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Q,varargout]=ortha(A,X)
%ORTHA Orthonormalization Relative to matrix A
%  Q=ortha(A,X)
%  Q=ortha('Afunc',X)
%  Computes an orthonormal basis Q for the range of X, relative to the
%  scalar product using a positive definite and selfadjoint matrix A.
%  That is, Q'*A*Q = I, the columns of Q span the same space as
%  columns of X, and rank(Q)=rank(X).
%
%  [Q,AQ]=ortha(A,X) also gives AQ = A*Q.
%
%  Required input arguments:
%  A : either an m x m positive definite and selfadjoint matrix A
%  or a linear operator A=A(v) that is positive definite selfadjoint;
%  X : m x n matrix containing vectors to be orthonormalized relative
%  to A.
%
%  ortha(eye(m),X) spans the same space as orth(X)
%
%  Examples:
%  [q,Aq]=ortha(hilb(20),eye(20,5))
%  computes 5 column-vectors q spanned by the first 5 coordinate vectors,
%  and orthonormal with respect to the scalar product given by the
%  20x20 Hilbert matrix,
%  while an attempt to orthogonalize (in the same scalar product)
%  all 20 coordinate vectors using
%  [q,Aq]=ortha(hilb(20),eye(20))
%  gives 14 column-vectors out of 20.
%  Note that rank(hilb(20)) = 13 in double precision.
%
%  Algorithm:
%  X=orth(X), [U,S,V]=SVD(X'*A*X), then Q=X*U*S^(-1/2)
%  If A is ill conditioned an extra step is performed to
%  improve the result. This extra step is performed only
%  if a test indicates that the program is running on a
%  machine that supports higher precison arithmetic
%  (greater than 64 bit precision).
%
%  See also ORTH, SVD
%
```

```
%  Copyright (c) 2001 Freeware for noncommercial use.
%  Andrew Knyazev, Rico Argentati
%  Contact email: knyazev@na-net.ornl.gov
%  $Revision: 1.5.8 $  $Date: 2001/8/28
%  Tested under MATLAB R10-12.1

% Check input parameter A
[m,n] = size(X);
if ~isstr(A)
  [mA,mA] = size(A);
  if any(size(A) ~= mA)
    error('Matrix A must be a square matrix or a string.')
  end
  if size(A) ~= m
    error(['The size ' int2str(size(A)) ...
           ' of the matrix A does not match with ' int2str(m) ...
           ' - the number of rows of X'])
  end
end

% Normalize, so ORTH below would not delete wanted vectors
for i=1:size(X,2),
  normXi=norm(X(:,i),inf);
  %Adjustment makes tol consistent with experimental results
  if normXi > eps^.981
    X(:,i)=X(:,i)/normXi;
    % Else orth will take care of this
  end
end

% Make sure X is full rank and orthonormalize
X=orth(X); %This can also be done using QR.m, in which case
           %the column scaling above is not needed

%Set tolerance
[m,n]=size(X);
tol=max(m,n)*eps;

% Compute an A-orthonormal basis
if ~isstr(A)
  AX = A*X;
else
  AX = feval(A,X);
end
XAX = X'*AX;

XAX = 0.5.*(XAX' + XAX);
[U,S,V]=svd(XAX);

if n>1 s=diag(S);
  elseif n==1, s=S(1);
  else s=0;
end

%Adjustment makes tol consistent with experimental results
threshold1=max(m,n)*max(s)*eps^1.1;

r=sum(s>threshold1);
s(r+1:size(s,1))=1;
S=diag(1./sqrt(s),0);
X=X*U*S;
AX=AX*U*S;
XAX = X'*AX;

% Check error against tolerance
error=normest(XAX(1:r,1:r)-eye(r));
% Check internal precision, e.g., 80bit FPU registers of P3/P4
precision_test=[1 eps/1024 -1]*[1 1 1]';
if error<tol | precision_test==0;
  Q=X(:,1:r);
  varargout(1)={AX(:,1:r)};
  return
end
```

```

% Complete another iteration to improve accuracy
% if this machine supports higher internal precision
if ~isstr(A)
  AX = A*X;
else
  AX = feval(A,X);
end
XAX = X'*AX;

XAX = 0.5.*(XAX' + XAX);
[U,S,V]=svd(XAX);

if n>1 s=diag(S);
  elseif n==1, s=S(1);
  else s=0;
end

threshold2=max(m,n)*max(s)*eps;
r=sum(s>threshold2);
S=diag(1./sqrt(s(1:r)),0);
Q=X*U(:,1:r)*S(1:r,1:r);
varargout(1)={AX*U(:,1:r)*S(1:r,1:r)};
```

B LOBPCG Hypre API Functions

```
I. Create/Destroy  and Setup Functions
int
HYPRE_LobpcgCreate(HYPRE_LobpcgData *lobpcg);
Create Lopbcg solver.

int
HYPRE_LobpcgDestroy(HYPRE_LobpcgData lobpcg);
Destroy Lobpcg solver.

int
HYPRE_LobpcgSetup(HYPRE_LobpcgData lobpcg);
Setup Lobpcg solver.

int
HYPRE_LobpcgSetVerbose(HYPRE_LobpcgData lobpcg);
(Optional) Set verbose mode. Displays different levels of detail.
=0 (no output), =1 (standard output) default, =2 (detailed output).

int
HYPRE_LobpcgSetRandom(HYPRE_LobpcgData lobpcg);
(Optional) Set flag to randomized array of input vectors.

int
HYPRE_LobpcgSetEye(HYPRE_LobpcgData lobpcg);
(Optional) Set flag to initialized array of input vectors to m x bsize
identity matrix, still stored as array of vectors.

int
int HYPRE_LobpcgSetOrthCheck(HYPRE_LobpcgData lobpcg);
(Optional) Compute Frobenius norm of V'V-I to check
final orthogonality of eigenvectors.

int
HYPRE_LobpcgSetMaxIterations(HYPRE_LobpcgData lobpcg,int max_iter);
(Optional) Set maximum number of iterations. Default is 500.

int
HYPRE_LobpcgSetTolerance(HYPRE_LobpcgData lobpcg,double tol);
(Optional) Set tolerance for maximum residual to stop. Default is 1E-6.

int
HYPRE_LobpcgSetBlocksize(HYPRE_LobpcgData lobpcg,int bsize);
(Optional) Set block size for number of initial eigenvectors.
Default is 1.

int
HYPRE_LobpcgSetSolverFunction(HYPRE_LobpcgData lobpcg,
    int (*FunctSolver)(HYPRE_ParVector x,HYPRE_ParVector y));
(Optional) Set name of solver function. Solves y=inv(T)*x with user defined
preconditioner. If this function is not called, then no
preconditioner is used (y=x).


II. Lobpcg Solver
int
HYPRE_LobpcgSolve(HYPRE_LobpcgData lobpcgdata,
    int (*FunctA)(HYPRE_ParVector x,HYPRE_ParVector y),
    HYPRE_ParVector *v,double **eigval);
Solve the system. The parameters are defined as:
lobpcgdata - lobpcg solve context,
FunctA - function that computes matrix-vector multiply y=A*x (A is n x n),
x and y are parallel vectors that must be compatible with A.
v - pointer to array of parallel initial eigenvectors (n x bsize),
eigval - pointer to array to store eigenvalues (bsize x 1).


III. Output Functions
int
HYPRE_LobpcgGetIterations(HYPRE_LobpcgData lobpcg,int *iterations);
(Optional) Return the number of iterations taken.
```

```
int
int HYPRE_LobpcgGetOrthCheckNorm(HYPRE_LobpcgData lobpcg,double *orth_frob_norm);
(Optional) Get value of Frobenius norm of V'V-I to check
final orthogonality of eigenvectors. This must be setup using
HYPRE_LobpcgSetOrthCheck.

int
HYPRE_LobpcgGetEigval(HYPRE_LobpcgData lobpcg,double **eigval);
(Optional) Return pointer to array of eigenvalues (bsize x 1).

int
HYPRE_LobpcgGetResvec(HYPRE_LobpcgData lobpcg,double ***resvec);
(Optional) Return pointer to array of residual vector history (bsize x iterations).

int
HYPRE_LobpcgGetEigvalHistory(HYPRE_LobpcgData lobpcg,double ***eigvalhistory);
(Optional) Return pointer to array of eigenvalue history (bsize x iterations).

int
HYPRE_LobpcgGetBlocksize(HYPRE_LobpcgData lobpcg,int *bsize);
(Optional) Return blocksize.
```

C Primary LOBPCG Functions – lobpcg.c

```
/*BHEADER**********************************************************************
 * lobpcg.c
 *
 * $Revision: 1.8 $
 * Date: 10/7/2002
 * Authors: M. Argentati and A. Knyazev
 *********************************************************************EHEADER*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <assert.h>
#include <stdarg.h>
#include <float.h>
#include "lobpcg.h"

/*--------------------------------------------------------------------------*/
/* HYPRE includes                                                           */
/*--------------------------------------------------------------------------*/
#include <utilities/fortran.h>

/* function prototypes for functions that are passed to lobpcg */
/*void dsygv_(int *itype, char *jobz, char *uplo, int *n,
        double *a, int *lda, double *b, int *ldb, double *w,
        double *work, int *lwork, int *info);*/
void hypre_F90_NAME_BLAS(dsygv, DSYGV)(int *itype, char *jobz, char *uplo, int *n,
        double *a, int *lda, double *b, int *ldb, double *w,
        double *work, int *lwork, /*@out@*/ int *info);

int Func_AMult(Matx *B,Matx *C,int *idx); /* added by MEA 9/23/02 */
int Func_TPrec(Matx *R,int *idx);

int (*FunctA_ptr)(HYPRE_ParVector x,HYPRE_ParVector y);
int (*FunctPrec_ptr)(HYPRE_ParVector x,HYPRE_ParVector y);

/* this needs to be available to other program modules */
HYPRE_ParVector temp_global_vector;
Matx *temp_global_data; /* this needs to be available to other program modules */
static Matx *TMP_Global1;
static Matx *TMP_Global2;
static int rowcount_global;

int HYPRE_LobpcgSolve(HYPRE_LobpcgData lobpcgdata,
    int (*FunctA)(HYPRE_ParVector x,HYPRE_ParVector y),
    HYPRE_ParVector *v,double **eigval1)
{

  Matx *X,*eigval,*resvec,*eigvalhistory; /* Matrix pointers */
  Matx *TMP,*TMP1,*TMP2; /* Matrices for misc calculations */
  extern HYPRE_ParVector temp_global_vector;
  int i,j,max_iter,bsize,verbose,rand_vec,eye_vec;
  double tol;
  double minus_one=-1;
  int (*FuncT)(Matx *B,int *idx)=NULL;
  int *partitioning,*part2;
  int mypid,nprocs;

  MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  /* store function pointers */
  FunctA_ptr=FunctA;
  HYPRE_LobpcgGetSolverFunction(lobpcgdata,&FunctPrec_ptr);

  /* get lobpcg parameters */
  HYPRE_LobpcgGetMaxIterations(lobpcgdata,&max_iter);
  HYPRE_LobpcgGetTolerance(lobpcgdata,&tol);
  HYPRE_LobpcgGetVerbose(lobpcgdata,&verbose);
  HYPRE_LobpcgGetRandom(lobpcgdata,&rand_vec);
  HYPRE_LobpcgGetEye(lobpcgdata,&eye_vec);
```

```
    HYPRE_LobpcgGetBlocksize(lobpcgdata,&bsize);

    /* initialize detailed verbose status for data collection */
    if (verbose==2)
    {
      verbose2(0);    /* turn off */
      collect_data(3,0,0);
    }
    else verbose2(2); /* turn off */

    /* derive partitioning from v */
    partitioning=hypre_ParVectorPartitioning((hypre_ParVector *)  v[0]);
    hypre_LobpcgSetGetPartition(0,&partitioning);
    rowcount_global=partitioning[nprocs];

    /* check time test */
    time_functions(1,0,rowcount_global,0,FunctA);

    /* create global temp vector to use in solve function */
    part2=CopyPartition(partitioning);
    if (verbose2(1)==TRUE) collect_data(0,HYPRE_ParVectorCreate_Data,0);
    HYPRE_ParVectorCreate(MPI_COMM_WORLD,rowcount_global,
      part2,&temp_global_vector);
    HYPRE_ParVectorInitialize(temp_global_vector);

    /* create global temporary data containing bsize parallel hypre vectors */
    /* these may be used in other programs for lobpcg */
    temp_global_data=Mat_Alloc1();
    Mat_Init(temp_global_data,rowcount_global,bsize,rowcount_global*bsize,
      HYPRE_VECTORS,GENERAL);

    /* check to see if we need to randomize v or set to identity */
    if (rand_vec==TRUE)
    {
      Init_Rand_Vectors(v,partitioning,rowcount_global,bsize);
    }
    else if (eye_vec==TRUE)
    {
      Init_Eye_Vectors(v,partitioning,rowcount_global,bsize);

    }

    /* Setup X array for initial eigenvectors */
    X=Mat_Alloc1();
    Mat_Init(X,rowcount_global,bsize,rowcount_global*bsize,HYPRE_VECTORS,GENERAL);
    for (i=0; i<bsize; i++){
       if (verbose2(1)==TRUE) collect_data(0,HYPRE_ParVectorCopy_Data,0);
       HYPRE_ParVectorCopy(v[i],X->vsPar[i]);

    }

    TMP_Global1=Mat_Alloc1();
    TMP_Global2=Mat_Alloc1();
    resvec=Mat_Alloc1();
    eigval=Mat_Alloc1();
    eigvalhistory=Mat_Alloc1();
    TMP=Mat_Alloc1();
    TMP1=Mat_Alloc1();
    TMP2=Mat_Alloc1();

    if (FunctPrec_ptr!=NULL) FuncT=Func_TPrec;

    /* call main lobpcg solver */
    lobpcg(X,Func_AMult,FuncT,tol,&max_iter,verbose,eigval,eigvalhistory,resvec);

    /* check orthogonality of eigenvectors */
    if (hypre_LobpcgOrthCheck((hypre_LobpcgData *) lobpcgdata)==TRUE)
    {
      Mat_Trans_Mult(X,X,TMP);
      Mat_Eye(bsize,TMP1);
      Mat_Add(TMP,TMP1,minus_one,TMP2);
      hypre_LobpcgOrthFrobNorm((hypre_LobpcgData *) lobpcgdata)=Mat_Norm_Frob(TMP2);
    }
```

```

  /* get v vectors back */
  for (i=0; i<bsize; i++){
     if (verbose2(1)==TRUE) collect_data(0,HYPRE_ParVectorCopy_Data,0);
     HYPRE_ParVectorCopy(X->vsPar[i],v[i]);

  }

  /* store output */
  /* max_iter is returned from lobpcg as the actual number of iterations */
  hypre_LobpcgIterations((hypre_LobpcgData *) lobpcgdata)=max_iter;
  hypRe_LobpcgEigval((hypre_LobpcgData *) lobpcgdata)=
    (Double *)calloc(bsize,sizeof(double));
  hypre_LobpcgResvec((hypre_LobpcgData *) lobpcgdata)=Mymalloc(bsize,max_iter);
  hypre_LobpcgEigvalHistory((hypre_LobpcgData *) lobpcgdata)=Mymalloc(bsize,max_iter);

  for (i=0; i<bsize; i++){
    ((hypre_LobpcgData *) lobpcgdata)->eigval[i]=eigval->val[i][0];
    for (j=0; j<max_iter; j++){
      ((hypre_LobpcgData *) lobpcgdata)->resvec[i][j]=resvec->val[i][j];
      ((hypre_LobpcgData *) lobpcgdata)->eigvalhistory[i][j]=eigvalhistory->val[i][j];
    }
  }

  /* save eigenvalues to output */
  HYPRE_LobpcgGetEigval(lobpcgdata,eigval1);

  /* free all memory associated with these pointers */
  if (verbose2(1)==TRUE) collect_data(0,HYPRE_ParVectorDestroy_Data,0);
  HYPRE_ParVectorDestroy(temp_global_vector);

  Mat_Free(temp_global_data);free(temp_global_data);
  Mat_Free(X);free(X);
  Mat_Free(resvec);free(resvec);
  Mat_Free(eigvalhistory);free(eigvalhistory);
  Mat_Free(eigval);free(eigval);
  Mat_Free(TMP);free(TMP);
  Mat_Free(TMP1);free(TMP1);
  Mat_Free(TMP2);free(TMP2);
  Mat_Free(TMP_Global1);free(TMP_Global1);
  Mat_Free(TMP_Global2);free(TMP_Global2);

  /* print out execution statistics */
  if (verbose==2 && Get_Rank()==0)
    Display_Execution_Statistics();

  return 0;
}

/*****************************************************************************/
int Func_AMult(Matx *B,Matx *C,int *idx)
{
  /* return C=A*B according to index */
  int i;
  for (i=0; i<B->n; i++) {
    if (idx[i]>0){
      FunctA_ptr(B->vsPar[i],C->vsPar[i]);
      if (verbose2(1)==TRUE) collect_data(0,NUMBER_A_MULTIPLIES,0);
    }
  }
  return 0;
}

/*****************************************************************************/
int Func_TPrec(Matx *R,int *idx)
{
  /* Solve A*X=R, return R=X using  */
  int i;
  extern HYPRE_ParVector temp_global_vector;

  for (i=0; i<R->n; i++) {
    if (idx[i]>0)
    {
      /* this next copy is not required, but improves performance substantially */
      if (verbose2(1)==TRUE) collect_data(0,HYPRE_ParVectorCopy_Data,0);
      HYPRE_ParVectorCopy(R->vsPar[i],temp_global_vector);
      FunctPrec_ptr(R->vsPar[i],temp_global_vector);
      if (verbose2(1)==TRUE) collect_data(0,HYPRE_ParVectorCopy_Data,0);
      HYPRE_ParVectorCopy(temp_global_vector,R->vsPar[i]);
      if (verbose2(1)==TRUE) collect_data(0,NUMBER_SOLVES,0);
    }
  }

  return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgCreate
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgCreate(HYPRE_LobpcgData *lobpcg)
{
   HYPRE_LobpcgData lobpcg2;

   /* allocate memory */
   if (!(*lobpcg=(HYPRE_LobpcgData) malloc(sizeof(hypre_LobpcgData)))) {
     fprintf(stderr, "Out of memory\n");
     abort();
   }

   lobpcg2=*lobpcg;

   /* set defaults */
   HYPRE_LobpcgSetMaxIterations(lobpcg2,LOBPCG_DEFAULT_MAXITR);
   HYPRE_LobpcgSetTolerance(lobpcg2,LOBPCG_DEFAULT_TOL);
   HYPRE_LobpcgSetBlocksize(lobpcg2,LOBPCG_DEFAULT_BSIZE);
   HYPRE_LobpcgSetSolverFunction(lobpcg2,NULL);
   hypre_LobpcgVerbose((hypre_LobpcgData *) lobpcg2)=LOBPCG_DEFAULT_VERBOSE;
   hypre_LobpcgRandom((hypre_LobpcgData *) lobpcg2)=LOBPCG_DEFAULT_RANDOM;
   hypre_LobpcgEye((hypre_LobpcgData *) lobpcg2)=LOBPCG_DEFAULT_EYE;
   hypre_LobpcgOrthCheck((hypre_LobpcgData *) lobpcg2)=LOBPCG_DEFAULT_ORTH_CHECK;

   /* initialize pointers */
   hypre_LobpcgEigval((hypre_LobpcgData *) lobpcg2)=NULL;
   hypre_LobpcgResvec((hypre_LobpcgData *) lobpcg2)=NULL;
   hypre_LobpcgEigvalHistory((hypre_LobpcgData *) lobpcg2)=NULL;
   hypre_LobpcgPartition((hypre_LobpcgData *) lobpcg2)=NULL;

   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetup
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetup(HYPRE_LobpcgData lobpcg)
{
  if(!lobpcg) abort();
  /* future use */
  return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgDestroy
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgDestroy(HYPRE_LobpcgData lobpcg)
{

   /* free memory */
   if (((hypre_LobpcgData *) lobpcg)->eigval!=NULL)
      free(((hypre_LobpcgData *) lobpcg)->eigval);
   if (((hypre_LobpcgData *) lobpcg)->resvec != NULL){
      if (((hypre_LobpcgData *) lobpcg)->resvec[0] != NULL)
         free(((hypre_LobpcgData *) lobpcg)->resvec[0]);
      free(((hypre_LobpcgData *) lobpcg)->resvec);
   }
   if (((hypre_LobpcgData *) lobpcg)->eigvalhistory != NULL){
      if (((hypre_LobpcgData *) lobpcg)->eigvalhistory[0] != NULL)
         free(((hypre_LobpcgData *) lobpcg)->eigvalhistory[0]);
      free(((hypre_LobpcgData *) lobpcg)->eigvalhistory);
   }
   if (((hypre_LobpcgData *) lobpcg)->partition!=NULL)
      free(((hypre_LobpcgData *) lobpcg)->partition);
   free((hypre_LobpcgData *) lobpcg);

   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetVerbose
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetVerbose(HYPRE_LobpcgData lobpcg,int verbose)
{
   hypre_LobpcgVerbose((hypre_LobpcgData *) lobpcg)=verbose;
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetRandom
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetRandom(HYPRE_LobpcgData lobpcg)
{
   hypre_LobpcgRandom((hypre_LobpcgData *) lobpcg)=TRUE;
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetEye
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetEye(HYPRE_LobpcgData lobpcg)
{
   hypre_LobpcgEye((hypre_LobpcgData *) lobpcg)=TRUE;
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetOrthCheck
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetOrthCheck(HYPRE_LobpcgData lobpcg)
{
   hypre_LobpcgOrthCheck((hypre_LobpcgData *) lobpcg)=TRUE;
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetMaxIterations
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetMaxIterations(HYPRE_LobpcgData lobpcg,int max_iter)
{
   hypre_LobpcgMaxIterations((hypre_LobpcgData *) lobpcg)=max_iter;
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetTolerance
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetTolerance(HYPRE_LobpcgData lobpcg,double tol)
{
   hypre_LobpcgTol((hypre_LobpcgData *) lobpcg)=tol;
   return 0;
}

/*--------------------------------------------------------------------------
```

```
 * HYPRE_LobpcgSetBlocksize
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetBlocksize(HYPRE_LobpcgData lobpcg,int bsize)
{
   hypre_LobpcgBlocksize((hypre_LobpcgData *) lobpcg)=bsize;
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgSetSolverFunction
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgSetSolverFunction(HYPRE_LobpcgData lobpcg,
   int (*FunctSolver)(HYPRE_ParVector x,HYPRE_ParVector y))
{
   hypre_LobpcgFunctSolver((hypre_LobpcgData *) lobpcg)=FunctSolver;
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetSolverFunction
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetSolverFunction(HYPRE_LobpcgData lobpcg,
   int (**FunctSolver)(HYPRE_ParVector x,HYPRE_ParVector y))
{
   *FunctSolver=hypre_LobpcgFunctSolver((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetMaxIterations
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetMaxIterations(HYPRE_LobpcgData lobpcg,int *max_iter)
{
   *max_iter=hypre_LobpcgMaxIterations((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetOrthCheckNorm
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetOrthCheckNorm(HYPRE_LobpcgData lobpcg,double *orth_frob_norm)
{
   *orth_frob_norm=hypre_LobpcgOrthFrobNorm((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetIterations
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetIterations(HYPRE_LobpcgData lobpcg,int *iterations)
{
   *iterations=hypre_LobpcgIterations((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetTolerance
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetTolerance(HYPRE_LobpcgData lobpcg,double *tol)
{
   *tol=hypre_LobpcgTol((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetVerbose
 *--------------------------------------------------------------------------*/
```

```
int
HYPRE_LobpcgGetVerbose(HYPRE_LobpcgData lobpcg,int *verbose)
{
   *verbose=hypre_LobpcgVerbose((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetRandom
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetRandom(HYPRE_LobpcgData lobpcg,int *rand_vec)
{
   *rand_vec=hypre_LobpcgRandom((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetEye
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetEye(HYPRE_LobpcgData lobpcg,int *eye_vec)
{
   *eye_vec=hypre_LobpcgEye((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetEigval
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetEigval(HYPRE_LobpcgData lobpcg,double **eigval)
{
   *eigval=hypre_LobpcgEigval((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetResvec
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetResvec(HYPRE_LobpcgData lobpcg,double ***resvec)
{
   *resvec=hypre_LobpcgResvec((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetEigvalHistory
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetEigvalHistory(HYPRE_LobpcgData lobpcg,double ***eigvalhistory)
{
   *eigvalhistory=hypre_LobpcgEigvalHistory((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * HYPRE_LobpcgGetBlocksize
 *--------------------------------------------------------------------------*/
int
HYPRE_LobpcgGetBlocksize(HYPRE_LobpcgData lobpcg,int *bsize)
{
   *bsize=hypre_LobpcgBlocksize((hypre_LobpcgData *) lobpcg);
   return 0;
}

/*--------------------------------------------------------------------------
 * hypre_LobpcgSetGetPartitioning
 * Store or retrieve partitioning to use in lobpcg for parallel vectors
 *--------------------------------------------------------------------------*/
int
hypre_LobpcgSetGetPartition(int action,int **part)
{
```

```
static int *partitioning;

   /* store partitioning */
   if (action==0)
   {
      partitioning=*part;
      return 0;
   }
   /* get partitioning */
   else if (action==1)
   {
      *part=partitioning;
      if (part==NULL) {
        fprintf(stderr, "hypre_LobpcgSetGetPartition function failed");
        abort();
      }
      return 0;
   }
   else
   {
     fprintf(stderr, "hypre_LobpcgSetGetPartition function failed");
     abort();
     return(1);
   }
}


/*****************************************************************************
* LOBPCG.C - eigenvalue solver implemented using C-language
* and LAPACK Fotran calls. By M.E. Argentati and Andrew Knyazev
*
* Revision: 1.00-C03 Date: 2002/7/8
* Modified by AK, 2002/7/14
*
* This implementation is a simplified (for B=I) version of the
* preconditioned conjugate gradient method (Algorithm 5.1) described in
* A. V. Knyazev, Toward the Optimal Preconditioned Eigensolver:
* Locally Optimal Block Preconditioned Conjugate Gradient Method,
* SIAM Journal on Scientific Computing 23 (2001), no. 2, pp. 517-541.
* See http://epubs.siam.org/sam-bin/getfile/SISC/articles/36612.pdf.
*
* It is also logically similar to a MATLAB inplementation - LOBPCG.m
* Revision: 4.0 Beta 4 $  $Date: 2001/8/25, by A. V. Knyazev, which
* can be found at http://www-math.cudenver.edu/~aknyazev/software/CG/.
*
* This program also uses some available software including several
* Fortran routines from LAPACK (http://www.netlib.org/lapack/),
* The author would like to acknowledge use of this software and
* thank the authors.
*
*Required input/output:
*   X - initial approximation to eigenvectors, colunm-matrix n-by-bsize
        Eigenvectors are returned in X.
*   FuncA - name of function that multiplies a matrix A times a vector
*           FuncA(Matx *B, Matx *C, int *idx) for C=A*B using index idx.
*   nargs - the number of optional parameters (0-7)
*   mytol - tolerance, by default, mytol=n*sqrt(eps)
*   maxit - max number of iterations, by default, maxit = min(n,20)
*   verbose - =0 (no output), =1 (standard output) default, =2 (detailed output)
*   lambda - Matrix (vector) of eigenvalues (bsize)
*   lambdahistory - history of eigenvalues iterates (bsize x maxit)
*   resvec - history of residuals (bsize x maxit)
*
*Optional function input:
*   FuncT - preconditioner, is the name of a function, default T=I
*           Funct(Matx *R,int *idx), usually solve T*X=R, return R=X
*           using index idx.
*
* Examples:
* All parameters:
* lobpcg(X,FuncA,7,FuncT,mytol,maxit,verbose,lambda,lambdahistory,resvec);
*
* All parameters, but no preconditioner:
* lobpcg(X,FuncA,NULL,mytol,maxit,verbose,lambda,lambdahistory,resvec);
```

```
******************************************************************************/

int
lobpcg(X, FuncA, FuncT, mytol, maxit_ptr, verbose, lambda_out, lambdahistory, resvec)
     Matx *X, *lambda_out, *lambdahistory, *resvec;
     int (*FuncA)(Matx *B, Matx *c, int *idx), (*FuncT)(Matx *B,int *idx),
         *maxit_ptr, verbose;
     double mytol;
{
  /* initialize variables */
  double minus_one=-1;
  int maxit=0;
  /* storage for the following matrices must be allocated
     before the call to lobpcg */
  Matx    *AX,*R,*AR,*P,*AP;                   /* Matrix pointers */
  Matx    *RR1,*D,*TMP,*TMP1,*TMP2;
  Matx    *Temp_Dense;
  extern Matx *temp_global_data;

  int    i,j,k,n;
  int    bsize=0,bsizeiaf=0;
  int    Amult_count=0; /* count matrix A multiplies */
  int    Tmult_solve_count=0; /* count T matrix multiplies of prec. solves */
  int    *idx;       /* index to keep track of kept vectors */

  double *lambda, *y,*normR;

  /* allocate storage */
  AX=   Mat_Alloc1();
  R=    Mat_Alloc1();
  AR=   Mat_Alloc1();
  P=    Mat_Alloc1();
  AP=   Mat_Alloc1();
  TMP1= Mat_Alloc1();
  TMP2= Mat_Alloc1();
  TMP=  Mat_Alloc1();
  RR1=  Mat_Alloc1();
  D=    Mat_Alloc1();
  Temp_Dense=Mat_Alloc1();

  /* get size of A from X */
  n=Mat_Size(X,1);
  bsize=Mat_Size(X,2);

  /* check bsize */
  if (n<5*bsize && Get_Rank()==0){
    fprintf(stderr,
     "The problem size is too small compared to the block size for LOBPCG.\n");
    exit(EXIT_FAILURE);
  }

  /* set defaults */
  if (mytol<DBL_EPSILON) mytol=sqrt(DBL_EPSILON)*n;
  if (maxit_ptr != NULL) maxit=*maxit_ptr;
  if (maxit==0) maxit=n<20 ? n:20;

  Mat_Init_Dense(lambda_out,bsize,1,GENERAL);
  Mat_Init_Dense(lambdahistory,bsize,maxit+1,GENERAL);
  Mat_Init_Dense(resvec,bsize,maxit+1,GENERAL);

  if (bsize > 0) {
    /* allocate lambda */
    lambda=(double *)calloc((long)3*bsize,sizeof(double));
    y=(double *)calloc((long)3*bsize,sizeof(double));
    /* allocate norm of R vector */
    normR=(double *)calloc((long)bsize,sizeof(double));
    /* allocate index */
    idx=(int *)calloc((long)bsize,sizeof(int));
    for (i=0; i<bsize; i++) idx[i]=1; /* initialize index */
  }
  else {
    fprintf(stderr, "The block size is wrong.\n");
    exit(EXIT_FAILURE);
  }

  /* perform orthonormalization of X */
  Qr2(X,RR1,idx);

  /* generate AX */
  Mat_Copy(X,AX);     /* initialize AX */
  if (misc_flags(1,0)!=TRUE)
  {
    FuncA(X,AX,idx);    /* AX=A*X */
    Amult_count=Amult_count+bsize;
  }

  /* initialize global data */
  Mat_Copy(X,TMP_Global1);
  Mat_Copy(X,TMP_Global2);

  /* compute initial eigenvalues */
  Mat_Trans_Mult(X,AX,TMP1);     /* X'*AX */
  Mat_Sym(TMP1);                 /* (TMP1+TMP1')/2 */
  Mat_Eye(bsize,TMP2);           /* identity */
  myeig1(TMP1,TMP2,TMP,lambda);  /* TMP has eignevectors */
  assert(lambda!=NULL);

  Mat_Mult2(X,TMP,idx);        /* X=X(idx)*TMP */
  Mat_Mult2(AX,TMP,idx);       /* AX=AX(idx)*TMP */

  /* compute initial residuals */
  Mat_Diag(lambda,bsize,TMP1);   /* diagonal matrix of eigenvalues */
  Mat_Mult(X,TMP1,TMP_Global1);
  Mat_Add(AX,TMP_Global1,minus_one,R);  /* R=AX-A*eigs */
  Mat_Norm2_Col(R,normR);
  assert(normR!=NULL);

  /* initialize AR  and P and AP using the same size and format as X */
  Mat_Init(AR,X->m,X->n,X->nz,X->mat_storage_type,X->mat_type);
  Mat_Init(P,X->m,X->n,X->nz,X->mat_storage_type,X->mat_type);
  Mat_Init(AP,X->m,X->n,X->nz,X->mat_storage_type,X->mat_type);

  k=1;         /* iteration count */

  /* store auxillary information */
  if (resvec != NULL){
    for (i=0; i<bsize; i++) resvec->val[i][k-1]=normR[i];
  }
  if (lambdahistory != NULL){
    for (i=0; i<bsize; i++) lambdahistory->val[i][k-1]=lambda[i];
  }

  printf("\n");
  if (verbose==2 && Get_Rank()==0){
    for (i=0; i<bsize; i++) printf("Initial eigenvalues lambda %22.16e\n",lambda[i]);
    for (i=0; i<bsize; i++) printf("Initial residuals %12.6e\n",normR[i]);
  }
  else if (verbose==1 && Get_Rank()==0)
    printf("Initial Max. Residual %12.6e\n",Max_Vec(normR,bsize));

  /* increment data collection mode */
  if (verbose2(1)==TRUE) collect_data(1,0,0);

  /* main loop of CG method */
  while (Max_Vec(normR,bsize) - mytol >  DBL_EPSILON && k<maxit+1)
  {
    /* increment data collection mode */
    if ((verbose2(1)==TRUE) && (k==2)) collect_data(1,0,0);

    /* generate index of vectors to keep */
    bsizeiaf=0;
    for (i=0; i<bsize; i++){
      if (normR[i] - mytol > DBL_EPSILON){
        idx[i]=1;
        ++bsizeiaf;
      }
      else idx[i]=0;
    }
    assert(idx!=NULL);

    /* check for precondioner */
    if ((FuncT != NULL) && (misc_flags(1,1)!=TRUE))
    {
      FuncT(R,idx);
      Tmult_solve_count=Tmult_solve_count+bsizeiaf;
    }

    /* orthonormalize R to increase stability  */
    Qr2(R,RR1,idx);

    /* compute AR */
    if (misc_flags(1,0)!=TRUE)
    {
      FuncA(R,AR,idx);
      Amult_count=Amult_count+bsizeiaf;
    }
    else Mat_Copy(R,AR);

    /* compute AP */
    if (k>1){
      Qr2(P,RR1,idx);
      Mat_Inv_Triu(RR1,Temp_Dense); /* TMP=inv(RR1) */
      Mat_Mult2(AP,Temp_Dense,idx);   /* AP(idx)=AP(idx)*TMP */
    }

    /* Raleigh-Ritz proceedure */
    if (bsize != bsizeiaf)
    {
      Mat_Get_Col(R,TMP_Global2,idx);
      Mat_Get_Col(AR,temp_global_data,idx);
      Mat_Get_Col(P,TMP_Global1,idx);
      Mat_Copy(TMP_Global1,P);
      Mat_Get_Col(AP,TMP_Global1,idx);
      Mat_Copy(TMP_Global1,AP);
      rr(X,AX,TMP_Global2,temp_global_data,P,AP,lambda,idx,bsize,k,0);
    }
    else rr(X,AX,R,AR,P,AP,lambda,idx,bsize,k,0);

    /* get eigenvalues corresponding to index */
    j=0;
    for (i=0; i<bsize; i++){
      if (idx[i]>0){
        y[j]=lambda[i];
        ++j;
      }
    }
    assert(j==bsizeiaf);
    assert(y!=NULL);

    /* compute residuals */
    Mat_Diag(y,bsizeiaf,Temp_Dense);
    Mat_Get_Col(X,TMP_Global1,idx);
    Mat_Mult(TMP_Global1,Temp_Dense,TMP_Global2);
    Mat_Get_Col(AX,TMP_Global1,idx);
    Mat_Add(TMP_Global1,TMP_Global2,minus_one,temp_global_data);
    Mat_Put_Col(temp_global_data,R,idx);
    Mat_Norm2_Col(R,normR);

    /* store auxillary information */
    if (resvec != NULL){
      for (i=0; i<bsize; i++) resvec->val[i][k]=normR[i];
    }
    if (lambdahistory != NULL){
      for (i=0; i<bsize; i++) lambdahistory->val[i][k]=lambda[i];
    }

    if (verbose==2 && Get_Rank()==0){
      printf("Iteration %d \tbsize %d\n",k,bsizeiaf);
      for (i=0; i<bsize; i++) printf("Eigenvalue lambda %22.16e\n",lambda[i]);
      for (i=0; i<bsize; i++) printf("Residual %12.6e\n",normR[i]);
    }
    else if (verbose==1 && Get_Rank()==0)
    printf("Iteration %d \tbsize %d \tmaxres %12.6e\n",
        k,bsizeiaf,Max_Vec(normR,bsize));
    ++k;
  }

  /* increment data collection mode if only one iteration */
  if ((verbose2(1)==TRUE) && (k==2)) collect_data(1,0,0);

  /* increment data collection mode */
  if (verbose2(1)==TRUE) collect_data(1,0,0);

  /* call rr once more to release memory */
  rr(X,AX,R,AR,P,AP,lambda,idx,bsize,k,1);

  /* return actual number of iterations */
  if (maxit_ptr != NULL) *maxit_ptr=k;
  else {
    fprintf(stderr, "The number of iterations is empty.\n");
    exit(EXIT_FAILURE);
  }

  if (verbose==1 && Get_Rank()==0){
    printf("\n");
    for (i=0; i<bsize; i++) printf("Eigenvalue lambda %22.16e\n",lambda[i]);
    for (i=0; i<bsize; i++) printf("Residual %12.6e\n",normR[i]);
  }

  /* free all memory associated with these pointers */
  free(lambda);
  free(y);
  free(normR);
  free(idx);
  Mat_Free(AX);free(AX);
  Mat_Free(R);free(R);
  Mat_Free(AR);free(AR);
  Mat_Free(P);free(P);
  Mat_Free(AP);free(AP);
  Mat_Free(TMP1);free(TMP1);
  Mat_Free(TMP2);free(TMP2);
  Mat_Free(TMP);free(TMP);
  Mat_Free(RR1);free(RR1);
  Mat_Free(D);free(D);
  Mat_Free(Temp_Dense);free(Temp_Dense);

  return 0;
}

/*************************************************************************/
int rr(Matx *U,Matx *LU,Matx *R,Matx *LR,Matx *P,Matx *LP,
       double *lambda,int *idx,int bsize,int k,int last_flag)
{
  /* The Rayleigh-Ritz method for triplets U, R, P */

  static Matx *GL;
  static Matx *GM;
  static Matx *GL12;
  static Matx *GL13;
  static Matx *GL22;
  static Matx *GL23;
  static Matx *GL33;
  static Matx *GM12;
  static Matx *GM13;
  static Matx *GM23;
  static Matx *GU;
  static Matx *D;
  static Matx *Temp_Dense;

  int i,n;
  int bsizeU,bsizeR,bsizeP;
  int restart;

  static int exec_first=0;

  if (exec_first==0){
    GL=   Mat_Alloc1();
    GM=   Mat_Alloc1();
    GL12= Mat_Alloc1();
    GL13= Mat_Alloc1();
    GL22= Mat_Alloc1();
    GL23= Mat_Alloc1();
    GL33= Mat_Alloc1();
    GM12= Mat_Alloc1();
    GM13= Mat_Alloc1();
    GM23= Mat_Alloc1();
    GU=   Mat_Alloc1();
    D=    Mat_Alloc1();
    Temp_Dense=Mat_Alloc1();
    exec_first=1;
  }

  /* cleanup */
  if ((last_flag !=0) && (exec_first==1))
  {
    Mat_Free(GL);free(GL);
    Mat_Free(GM);free(GM);
    Mat_Free(GL12);free(GL12);
    Mat_Free(GL13);free(GL13);
    Mat_Free(GL23);free(GL23);
    Mat_Free(GL33);free(GL33);
    Mat_Free(GM12);free(GM12);
    Mat_Free(GM13);free(GM13);
    Mat_Free(GM23);free(GM23);
    Mat_Free(GU);free(GU);
    Mat_Free(D);free(D);
    Mat_Free(Temp_Dense);free(Temp_Dense);
    return 0;
  }

  /* setup and get sizes */
  bsizeU=Mat_Size(U,2);
  assert(bsize==bsizeU);

  bsizeR=0;
  for (i=0; i<bsize; i++){
    if (idx[i]>0) ++bsizeR;
  }

  if (k==1){
    bsizeP=0;
    restart=1;
  }
  else {
    bsizeP=bsizeR; /* these must be the same size */
    restart=0;
  }

  Mat_Trans_Mult(LU,R,GL12);
  Mat_Trans_Mult(LR,R,GL22);
  Mat_Trans_Mult(U,R,GM12);
  Mat_Sym(GL22);

  if (restart==0){
    /* form GL */
    Mat_Trans_Mult(LU,P,GL13);
    Mat_Trans_Mult(LR,P,GL23);
    Mat_Trans_Mult(LP,P,GL33);
    Mat_Sym(GL33);
    Mat_Diag(lambda,bsizeU,D);

    n=bsize+bsizeR+bsizeP;
    Mat_Init_Dense(GL,n,n,SYMMETRIC);

    Mat_Copy_MN(D,GL,0,0);
    Mat_Copy_MN(GL12,GL,0,bsizeU);
    Mat_Copy_MN(GL13,GL,0,bsizeU+bsizeR);
    Mat_Copy_MN(GL22,GL,bsizeU,bsizeU);
    Mat_Copy_MN(GL23,GL,bsizeU,bsizeU+bsizeR);
    Mat_Copy_MN(GL33,GL,bsizeU+bsizeR,bsizeU+bsizeR);
    Mat_Trans(GL12,D);
    Mat_Copy_MN(D,GL,bsizeU,0);
    Mat_Trans(GL13,D);
    Mat_Copy_MN(D,GL,bsizeU+bsizeR,0);
    Mat_Trans(GL23,D);
    Mat_Copy_MN(D,GL,bsizeU+bsizeR,bsizeU);

    /* form GM */
    Mat_Trans_Mult(U,P,GM13);
    Mat_Trans_Mult(R,P,GM23);
    Mat_Init_Dense(GM,n,n,SYMMETRIC);

    Mat_Eye(bsizeU,D);
    Mat_Copy_MN(D,GM,0,0);
    Mat_Eye(bsizeR,D);
    Mat_Copy_MN(D,GM,bsizeU,bsizeU);
    Mat_Eye(bsizeP,D);
    Mat_Copy_MN(D,GM,bsizeU+bsizeR,bsizeU+bsizeR);
    Mat_Copy_MN(GM12,GM,0,bsizeU);
    Mat_Copy_MN(GM13,GM,0,bsizeU+bsizeR);
    Mat_Copy_MN(GM23,GM,bsizeU,bsizeU+bsizeR);
    Mat_Trans(GM12,D);
    Mat_Copy_MN(D,GM,bsizeU,0);
    Mat_Trans(GM13,D);
    Mat_Copy_MN(D,GM,bsizeU+bsizeR,0);
    Mat_Trans(GM23,D);
    Mat_Copy_MN(D,GM,bsizeU+bsizeR,bsizeU);
  }
  else
  {
    /* form GL */
    n=bsizeU+bsizeR;
    Mat_Init_Dense(GL,n,n,SYMMETRIC);
    Mat_Diag(lambda,bsizeU,D);
    Mat_Copy_MN(D,GL,0,0);
    Mat_Copy_MN(GL12,GL,0,bsizeU);
    Mat_Copy_MN(GL22,GL,bsizeU,bsizeU);
    Mat_Trans(GL12,D);
    Mat_Copy_MN(D,GL,bsizeU,0);

    /* form GM */
    Mat_Init_Dense(GM,n,n,SYMMETRIC);
    Mat_Eye(bsizeU,D);
    Mat_Copy_MN(D,GM,0,0);
    Mat_Eye(bsizeR,D);
    Mat_Copy_MN(D,GM,bsizeU,bsizeU);
    Mat_Copy_MN(GM12,GM,0,bsizeU);
    Mat_Trans(GM12,D);
    Mat_Copy_MN(D,GM,bsizeU,0);

  }

  /* solve generalized eigenvalue problem */
  myeig1(GL,GM,GU,lambda);
  Mat_Copy_Cols(GU,Temp_Dense,0,bsizeU-1);
  Mat_Copy(Temp_Dense,GU);

  Mat_Copy_Rows(GU,Temp_Dense,bsizeU,bsizeU+bsizeR-1);
  Mat_Mult(R,Temp_Dense,TMP_Global1);
  Mat_Copy(TMP_Global1,R);
  Mat_Mult(LR,Temp_Dense,TMP_Global1);
  Mat_Copy(TMP_Global1,LR);

  if (restart==0){
    Mat_Copy_Rows(GU,Temp_Dense,bsizeU+bsizeR,bsizeU+bsizeR+bsizeP-1);
    Mat_Mult(P,Temp_Dense,TMP_Global1);
    Mat_Add(R,TMP_Global1,1,P);

    Mat_Mult(LP,Temp_Dense,TMP_Global1);
    Mat_Add(LR,TMP_Global1,1,LP);

    Mat_Copy_Rows(GU,Temp_Dense,0,bsizeU-1);
    Mat_Mult(U,Temp_Dense,TMP_Global1);
    Mat_Add(P,TMP_Global1,1,U);

    Mat_Mult(LU,Temp_Dense,TMP_Global1);
    Mat_Add(LP,TMP_Global1,1,LU);
  }
  else
  {
    Mat_Copy(R,P);
    Mat_Copy(LR,LP);

    Mat_Copy_Rows(GU,Temp_Dense,0,bsizeU-1);
    Mat_Mult(U,Temp_Dense,TMP_Global1);
    Mat_Add(R,TMP_Global1,1,U);

    Mat_Mult(LU,Temp_Dense,TMP_Global1);
    Mat_Add(LR,TMP_Global1,1,LU);
  }

  return 0;
}

/*------------------------------------------------------------------------*/
/* myeig1                                                                 */
/*------------------------------------------------------------------------*/
int myeig1(Matx *A, Matx *B, Matx *X,double *lambda)
{
/*------------------------------------------------------------------------
 * We deal with a blas portability issue (it's actually a
 * C-calling-fortran issue) by using the following macro to call the blas:
 * hypre_F90_NAME_BLAS(name, NAME)();
 * So, for dsygv, we use:
 *
 *  hypre_F90_NAME_BLAS(dsygv, DSYGV)();
 *
 * This helps to get portability on some other platforms that
 * such as on Cray computers.
 * The include file fortran.h is needed.
 *------------------------------------------------------------------------*/

  int i,j,n,lda,lwork,info,itype,ldb;
  char jobz,uplo;
  double *a,*b,*work,temp;



  /* do some checks */
  assert(A->m==A->n);
  assert(B->m==B->n);
  assert(A->m==B->m);
  assert(A->mat_storage_type==DENSE);
  assert(B->mat_storage_type==DENSE);

  n=A->n;
  lda=n;
  ldb=n;
  jobz='V';
  uplo='U';
  lwork=10*n;
  itype=1;

  Mat_Init_Dense(X,n,n,GENERAL);

  /* allocate memory */
    a=(double *)calloc((long)n*n,sizeof(double));
    b=(double *)calloc((long)n*n,sizeof(double));
    work=(double *)calloc((long)lwork,sizeof(double));
  if (a==NULL || b==NULL || work==NULL){
    fprintf(stderr, "Out of memory.\n");
    abort();
  }

  /* convert C-style to Fortran-style storage */
  /* column major order */
  for(i=0;i<n;++i){
    for(j=0;j<n;++j){
      a[j+n*i]=A->val[j][i];
      b[j+n*i]=B->val[j][i];
    }
  }
  assert(a!=NULL);assert(b!=NULL);

  /* compute generalized eigenvalues and eigenvectors of A*x=lambda*B*x */
  /*dsygv_(&itype, &jobz, &uplo, &n, a, &lda, b, &ldb,
     lambda, &work[0], &lwork, &info);a */
  hypre_F90_NAME_BLAS(dsygv, DSYGV)(&itype, &jobz, &uplo, &n, a, &lda, b, &ldb,
     lambda, &work[0], &lwork, &info);

  /* compute transpose of A */
  for (i=0;i<n;i++){
    for (j=0;j<i;j++){
    temp=A->val[i][j];
    A->val[i][j]=A->val[j][i];
    A->val[j][i]=temp;
    }
  }

  /* load X */
  /* convert Fortran-style to C-style storage */
  /* row  major order */
  for(j=0;j<n;++j){
    for(i=0;i<n;++i){
      X->val[i][j]=a[i+n*j];
    }
  }

  /* check error condition */
  if (info!=0) fprintf(stderr, "problem in dsygv eigensolver, info=%d\n",info);
  Trouble_Check(0,info);

  free(a);
  free(b);
  free(work);
  return info;
}

/******************************************************************************
*     SUBROUTINE DSYGV( ITYPE, JOBZ, UPLO, N, A, LDA, B, LDB, W, WORK,
     $                  LWORK, INFO )
*
*  -- LAPACK driver routine (version 3.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     June 30, 1999
*
*     .. Scalar Arguments ..
      CHARACTER          JOBZ, UPLO
      INTEGER            INFO, ITYPE, LDA, LDB, LWORK, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), B( LDB, * ), W( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  DSYGV computes all the eigenvalues, and optionally, the eigenvectors
*  of a real generalized symmetric-definite eigenproblem, of the form
*  A*x=(lambda)*B*x,  A*Bx=(lambda)*x,  or B*A*x=(lambda)*x.
*  Here A and B are assumed to be symmetric and B is also
*  positive definite.
*
*  Arguments
*  =========
*
*  ITYPE   (input) INTEGER
*          Specifies the problem type to be solved:
*          = 1:  A*x = (lambda)*B*x
*          = 2:  A*B*x = (lambda)*x
*          = 3:  B*A*x = (lambda)*x
*
*  JOBZ    (input) CHARACTER*1
*          = 'N':  Compute eigenvalues only;
*          = 'V':  Compute eigenvalues and eigenvectors.
*
*  UPLO    (input) CHARACTER*1
*          = 'U':  Upper triangles of A and B are stored;
*          = 'L':  Lower triangles of A and B are stored.
*
*  N       (input) INTEGER
*          The order of the matrices A and B.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA, N)
*          On entry, the symmetric matrix A.  If UPLO = 'U', the
*          leading N-by-N upper triangular part of A contains the
*          upper triangular part of the matrix A.  If UPLO = 'L',
*          the leading N-by-N lower triangular part of A contains
*          the lower triangular part of the matrix A.
*
*          On exit, if JOBZ = 'V', then if INFO = 0, A contains the
*          matrix Z of eigenvectors.  The eigenvectors are normalized
*          as follows:
*          if ITYPE = 1 or 2, Z**T*B*Z = I;
*          if ITYPE = 3, Z**T*inv(B)*Z = I.
*          If JOBZ = 'N', then on exit the upper triangle (if UPLO='U')
*          or the lower triangle (if UPLO='L') of A, including the
*          diagonal, is destroyed.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  B       (input/output) DOUBLE PRECISION array, dimension (LDB, N)
*          On entry, the symmetric positive definite matrix B.
*          If UPLO = 'U', the leading N-by-N upper triangular part of B
*          contains the upper triangular part of the matrix B.
*          If UPLO = 'L', the leading N-by-N lower triangular part of B
*          contains the lower triangular part of the matrix B.
*
*          On exit, if INFO <= N, the part of B containing the matrix is
*          overwritten by the triangular factor U or L from the Cholesky
*          factorization B = U**T*U or B = L*L**T.
*
*  LDB     (input) INTEGER
*          The leading dimension of the array B.  LDB >= max(1,N).
*
*  W       (output) DOUBLE PRECISION array, dimension (N)
*          If INFO = 0, the eigenvalues in ascending order.
*
*  WORK    (workspace/output) DOUBLE PRECISION array, dimension (LWORK)
*          On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
*
*  LWORK   (input) INTEGER
*          The length of the array WORK.  LWORK >= max(1,3*N-1).
*          For optimal efficiency, LWORK >= (NB+2)*N,
*          where NB is the blocksize for DSYTRD returned by ILAENV.
*
*          If LWORK = -1, then a workspace query is assumed; the routine
*          only calculates the optimal size of the WORK array, returns
*          this value as the first entry of the WORK array, and no error
*          message related to LWORK is issued by XERBLA.
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value
*          > 0:  DPOTRF or DSYEV returned an error code:
*             <= N:  if INFO = i, DSYEV failed to converge;
*                    i off-diagonal elements of an intermediate
*                    tridiagonal form did not converge to zero;
*             > N:   if INFO = N + i, for 1 <= i <= N, then the leading
*                    minor of order i of B is not positive definite.
*                    The factorization of B could not be completed and
*                    no eigenvalues or eigenvectors were computed.
*
*  ===================================================================
*****************************************************************************/

/****************************************************************************/
int Trouble_Check(int mode,int test)
{
  static int trouble=0;
  if (mode==0)
  {
     if (test!=0) trouble=1;
     else trouble=0;
  }
  return trouble;
}
```

REFERENCES

[1] N. I. Akhiezer and I. M. Glazman. *Theory of Linear Operators in Hilbert Space*. Dover Publications Inc., New York, 1993. Translated from the Russian and with a preface by Merlynd Nestell, Reprint of the 1961 and 1963 translations, Two volumes bound as one.

[2] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.

[3] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. *PETSc Users Manual*. Argonne National Laboratory, anl-95/11 - revision 2.1.5 edition, 2002.

[4] Jesse Barlow and James Demmel. Computing accurate eigensystems of scaled diagonally dominant matrices. *SIAM J. Numer. Anal.*, 27(3):762–791, 1990.

[5] Åke Björck and Gene H. Golub. Numerical methods for computing angles between linear subspaces. *Math. Comp.*, 27:579–594, 1973.

[6] James H. Bramble, Andrew Knyazev, and Joseph E. Pasciak. A subspace preconditioning algorithm for eigenvector/eigenvalue computation. *Advances in Computational Mathematics*, 6(2):159–189, 1996.

[7] Françoise Chatelin. *Eigenvalues of Matrices*. John Wiley & Sons Ltd., Chichester, 1993. With exercises by Mario Ahués and the author, Translated from the French and with additional material by Walter Ledermann.

[8] J. Dauxois and G. M. Nkiet. Canonical analysis of two Euclidean subspaces and its applications. *Linear Algebra Appl.*, 264:355–388, 1997.

[9] C. Davis and W. M. Kahan. The rotation of eigenvectors by a perturbation. III. *SIAM J. Numer. Anal.*, 7(1):1–46, 1970.

[10] Lokenath Debnath and Piotr Mikusinski. *Introduction to Hilbert Spaces with Applications*. Academic Press, San Diego, CA, 1999.

[11] James Demmel, Ming Gu, Stanley Eisenstat, Ivan Slapničar, Krešimir Veselić, and Zlatko Drmač. Computing the singular value decomposition with high relative accuracy. *Linear Algebra Appl.*, 299(1-3):21–80, 1999.

[12] James Demmel and Krešimir Veselić. Jacobi's method is more accurate than *QR*. *SIAM J. Matrix Anal. Appl.*, 13(4):1204–1245, 1992.

[13] James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, PA, 1997.

[14] Frank Deutsch. The angle between subspaces of a Hilbert space. In *Approximation Theory, Wavelets and Applications (Maratea, 1994)*, pages 107–130. Kluwer Acad. Publ., Dordrecht, 1995.

[15] Frank Deutsch. The angle between subspaces of a Hilbert space. In *Approximation Theory, Wavelets and Applications (Maratea, 1994)*, pages 107–130. Kluwer Academic Publishing, Dordrecht, 1995.

[16] Zlatko Drmač. On principal angles between subspaces of Euclidean space. *SIAM J. Matrix Anal. Appl.*, 22(1):173–194 (electronic), 2000.

[17] David A. Duran. Use of lobpcg in normal coordinate analysis for molecular structures. Masters Project – University of Colorado, Denver, February, 2002.

[18] Donald A. Flanders. Angles between flat subspaces of a real n-dimensional Euclidean space. In *Studies and Essays Presented to R. Courant on His 60th Birthday, January 8, 1948*, pages 129–138. Interscience Publishers, Inc., New York, 1948.

[19] I.M. Glazman and Ju. I. Ljubic. *Finite–Dimensional Linear Analysis: A Systematic Presentation in Problem Form.* The MIT Press, Cambridge, MA, 1974.

[20] I. C. Gohberg and M. G. Kreĭn. *Introduction to the Theory of Linear Nonselfadjoint Operators.* American Mathematical Society, Providence, R.I., 1969. Translated from the Russian by A. Feinstein. Translations of Mathematical Monographs, Vol. 18.

[21] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[22] Gene H. Golub and Hong Yuan Zha. Perturbation analysis of the canonical correlations of matrix pairs. *Linear Algebra Appl.*, 210:3–28, 1994.

[23] Gene H. Golub and Hong Yuan Zha. The canonical correlations of matrix pairs and their numerical computation. In *Linear Algebra for Signal Processing (Minneapolis, MN, 1992)*, pages 27–49. Springer, New York, 1995.

[24] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface.* The MIT Press, Cambridge, MA, second edition, 1999.

[25] P. R. Halmos. Two subspaces. *Trans. Amer. Math. Soc.*, 144:381–389, 1969.

[26] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1996.

[27] Roger A. Horn and Charles R. Johnson. *Matrix Analysis.* Cambridge University Press, New York, NY, 1990.

[28] Roger A. Horn and Charles R. Johnson. *Topics in Matrix Analysis.* Cambridge University Press, New York, NY, 1999.

[29] H. Hotelling. Relation between two sets of variables. *Biometrica*, 28:322–377, 1936.

[30] T. Kato. *Perturbation Theory for Linear Operators.* Springer–Verlag, New–York, 1976.

[31] A. V. Knyazev. *Computation of Eigenvalues and Eigenvectors for Mesh Problems: Algorithms and Error Estimates.* Dept. Numerical Math. USSR Academy of Sciences, Moscow, 1986. (In Russian).

[32] A. V. Knyazev. Convergence rate estimates for iterative methods for mesh symmetric eigenvalue problem. *Soviet J. Numerical Analysis and Math. Modelling*, 2(5):371–396, 1987.

[33] A. V. Knyazev. New estimates for Ritz vectors. *Math. Comp.*, 66(219):985–995, 1997.

[34] A. V. Knyazev. Preconditioned eigensolvers—an oxymoron? *Electron. Trans. Numer. Anal.*, 7:104–123 (electronic), 1998. Large scale eigenvalue problems (Argonne, IL, 1997).

[35] A. V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.

[36] A. V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing*, 23(2):517–541, 2001.

[37] A. V. Knyazev and M. E. Argentati. Rayleigh quotient and ritz value perturbations with respect to a trial subspace. *Preprint submitted to Linear Algebra and its Applications.*

[38] A. V. Knyazev and K. Neymeyr. A geometric theory for preconditioned inverse iteration, III: A short and sharp convergence estimate for generalized eigenvalue problems. *Linear Algebra Appl.*, 358:95–114, 2003.

[39] A. V. Knyazev and A. L. Skorokhodov. On exact estimates of the convergence rate of the steepest ascent method in the symmetric eigenvalue problem. *Linear Algebra and Applications*, 154–156:245–257, 1991.

[40] Andrew V. Knyazev and Merico E. Argentati. Principal angles between subspaces in an a-based scalar product: Algorithms and perturbation estimates. *SIAM Journal on Scientific Computing*, 23(6):2009–2041, 2002.

[41] Lawrence Livermore National Laboratory, Center for Applied Scientific Computing (CASC), University of California. *HYPRE User's Manual - Software Version 1.6.0*, 1998.

[42] Ren-Cang Li. Relative perturbation theory. II. Eigenspace and singular subspace variations. *SIAM J. Matrix Anal. Appl.*, 20(2):471–492 (electronic), 1999.

[43] Elizabeth R. Jessup Michael W. Berry, Zlatko Drmac. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[44] Klaus Neymeyr. A geometric theory for preconditioned inverse iteration. I: Extrema of the Rayleigh quotient. *Linear Algebra Appl.*, 322:61–85, 2001.

[45] Klaus Neymeyr. A geometric theory for preconditioned inverse iteration, II: Sharp convergence estimates. *Linear Algebra Appl.*, 322:87–104, 2001.

[46] Klaus Neymeyr. A geometric theory for preconditioned inverse iteration applied to a subspace. *Mathematics of Computation*, 71:197–216, 2002. Technical report SBF, July 1999.

[47] C. C. Paige and M. Wei. History and generality of the CS decomposition. *Linear Algebra Appl.*, 208/209:303–326, 1994.

[48] Beresford N. Parlett. *The Symmetric Eigenvalue Problem.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998. Corrected reprint of the 1980 original.

[49] Beresford N. Parlett. *The Symmetric Eigenvalue Problem.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[50] G. W. Stewart. Computing the CS decomposition of a partitioned orthonormal matrix. *Numer. Math.*, 40(3):297–306, 1982.

[51] G. W. Stewart and Ji Guang Sun. *Matrix Perturbation Theory.* Academic Press Inc., Boston, MA, 1990.

[52] G.W. Stewart. *Matrix Algorithms Volume II: Eigensystems.* SIAM, Philadelphia, PA, 2001.

[53] Ji Guang Sun. Perturbation of angles between linear subspaces. *J. Comput. Math.*, 5(1):58–61, 1987.

[54] Charles Van Loan. Computing the CS and the generalized singular value decompositions. *Numer. Math.*, 46(4):479–491, 1985.

[55] P. A. Wedin. On angles between subspaces of a finite-dimensional inner product space. In Bo Kågström and Axel Ruhe, editors, *Matrix Pencils. Proceedings of the Conference Held at Pite Havsbad, March 22–24, 1982*, pages 263–285. Springer-Verlag, Berlin, 1983.

[56] Harald K. Wimmer. Canonical angles of unitary spaces and perturbations of direct complements. *Linear Algebra Appl.*, 287(1-3):373–379, 1999. Special issue celebrating the 60th birthday of Ludwig Elsner.