Numerical Linear Algebra Problems in Structural Analysis

Kannan, Ramaseshan

2014

MIMS EPrint: **2014.57**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

NUMERICAL LINEAR ALGEBRA PROBLEMS IN STRUCTURAL ANALYSIS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2014

Ramaseshan Kannan
School of Mathematics

Contents

Word count 38952

List of Tables

List of Figures

List of Algorithms

The University of Manchester

Ramaseshan Kannan
Doctor of Philosophy
Numerical Linear Algebra problems in Structural Analysis
November 20, 2014

A range of numerical linear algebra problems that arise in finite element-based structural analysis are considered. These problems were encountered when implementing the finite element method in the software package Oasys GSA. We present novel solutions to these problems in the form of a new method for error detection, algorithms with superior numerical efficiency and algorithms with scalable performance on parallel computers. The solutions and their corresponding software implementations have been integrated into GSA's program code and we present results that demonstrate the use of these implementations by engineers to solve real-world structural analysis problems.

We start by introducing a new method for detecting the sources of ill conditioning in finite element stiffness matrices. The stiffness matrix can become ill-conditioned due to a variety of errors including user errors, i.e., errors in the definition of the finite element model when using a software package. We identify two classes of errors and develop a new method called model stability analysis for detecting where in the structural model these errors arise. Our method for debugging models uses the properties of the eigenvectors associated with the smallest and largest eigenvalues of the stiffness matrix to identify parts of the structural model that are badly defined. Through examples, we demonstrate the use of model stability analysis on real-life models and show that upon fixing the errors identified by the method, the condition number of the matrices typically drops from $O(10^{16})$ to $O(10^{8})$.

In the second part we introduce the symmetric definite generalized eigenproblem and the symmetric eigenproblem that are encountered in structural analysis. The symmetric definite generalized eigenvalue problem is to solve for the smallest eigenvalues and associated eigenvectors of $Kx = \lambda Mx$ for a symmetric positive-definite, sparse stiffness matrix K and a symmetric semidefinite mass matrix M. We analyse the existing solution algorithm, which uses the subspace iteration method. We improve the numerical efficiency of the algorithm by accelerating convergence via novel shift strategies and by reducing floating point operations using selective re-orthogonalization and locking of converged eigenvectors. The software implementation also benefits from improvements such as the use of faster and more robust libraries for linear algebra operations encountered during the iteration. We then develop an implementation of the subspace iteration method for solving the symmetric eigenvalue problem $Kx = \lambda x$ for the smallest and largest eigenvalues and their eigenvectors; this problem arises in model stability analysis.

In the next part we tackle the main bottleneck of sparse matrix computations in our eigensolvers: the sparse matrix-multiple vector multiplication kernel. We seek to obtain an algorithm that has high computational throughput for the operation $Y = AX$ for a square sparse A and a conforming skinny, dense X which, in turn, depends on the underlying storage format of A. Current sparse matrix formats and algorithms have high bandwidth requirements and poor reuse of cache and register loaded entries, which restricts their performance. We propose the mapped blocked row format: a bitmapped sparse matrix format that stores entries as blocks without a fill overhead, thereby offering blocking

without additional storage and bandwidth overheads. An efficient algorithm decodes bitmaps using de Bruijn sequences and minimizes the number of conditionals evaluated. Performance is compared with that of popular formats, including vendor implementations of sparse BLAS. Our sparse matrix multiple-vector multiplication algorithm achieves high throughput on all platforms and is implemented using platform-neutral optimizations.

The last part of the thesis deals with the B-orthogonalization problem, i.e., for a sparse symmetric positive definite $B \in \mathbb{R}^{n \times n}$ and a tall skinny matrix $X \in \mathbb{R}^{n \times \ell}$, we wish to rotate the columns of X such that $X^T B X = I$. This problem arises when the engineer wishes to orthonormalize the eigenvectors of the generalized eigenproblem such that they are orthogonal with respect to the stiffness matrix. Conventionally in the literature Gram-Schmidt like methods are employed for B-orthogonalization but they have poor cache efficiency. We recall a method that uses the Cholesky factorization of the inner product matrix $X^T B X$ and derive a stable algorithm that increases the parallel scalability through cache-reuse and is also numerically stable. Our experiments demonstrate orders of magnitude improvement in performance compared with Gram-Schmidt family of methods.

Declaration

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

Copyright Statement

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University's Policy on Presentation of Theses.

Acknowledgements

I would like to express my hearty gratitude to my supervisors Françoise Tisseur and Nicholas J. Higham for their expert guidance during the research and writing of this thesis and for their painstaking efforts in getting it to this form. They have also been very supportive of the setup of my PhD as I continued to work part-time at Arup alongside my research and have patiently worked around all constraints. What I have learnt from them goes beyond just numerical analysis and I could not have asked for finer people to work with.

At Arup, Chris Kaethner and Stephen Hendry have supported and backed my research endeavours from the time this project was a fledgling idea. Steve has shown me the ropes of FE solvers and our numerous discussions have contributed to the results in this thesis. Chris allowed me enormous flexibility and graciously tolerated my long absences, which made it feasible for me carry out this PhD.

Colleagues in the Manchester NA group and elsewhere have contributed ideas and helped with various opportunities all along; an incomplete list would include Nick Dingle, Iain Duff, Stefan Güttel, Amal Khabou, Lijing Lin, Yuji Nakatsukasa, David Silvester and Leo Taslaman. My office-mates Mary Aprahamian, Bahar Arslan, Sam Relton and Natasa Strabic gave me the pleasure of their company and kept me going when spirits were sagging. A special thanks goes to Jack Dongarra and his team at ICL, UTK for facilitating access to the Xeon Phi hardware and for valuable technical discussions.

My wife Miriam has ungrudgingly endured the unavailability of a full-time husband these last four years, something that is hard to express in a short span of words. She therefore deserves significant credit for this thesis and may also be held responsible for any copy-editing errors the reader may find, since she has proofread it.

To Miriam, who understood.

Publications

- The material covered in chapter 3 is based on the paper

  Ramaseshan Kannan, Stephen Hendry, Nicholas J. Higham, Françoise Tisseur, Detecting the causes of ill conditioning in structural finite element models, Computers & Structures, Volume 133, March 2014, Pages 79-89, ISSN 0045-7949, http://dx.doi.org/10.1016/j.compstruc.2013.11.014.

- Chapter 5 is based on the paper

  Kannan, R., Efficient sparse matrix multiple-vector multiplication using a bitmapped format, High Performance Computing (HiPC), 2013 20th International Conference on, pp.286,294, 18-21 Dec. 2013 http://dx.doi.org/10.1109/HiPC.2013.6799135

Chapter 1

Background

1.1 Introduction

This thesis deals with numerical linear algebra problems that arise in finite element based structural analysis applications.

The analysis and design of physical structures is one of the oldest, yet one of the most frequently encountered tasks in engineering. In its most basic form, structural analysis asks the question: 'How will a given physical structure and its components behave under the influence of loads and under the constraints of its environment?' These problems are routinely encountered in a range of engineering domains, for example, in aerospace, automotive, civil or mechanical engineering.

Within civil engineering we are interested in knowing how buildings or bridges will respond to loading acting upon them and whether they are strong and serviceable under the influence of these loads. The responses we are usually interested in are displacements, deflections, forces, bending moments and stresses while the loading can be from natural phenomena such as gravity or earthquakes, or it could be imposed by human-induced factors such as moving vehicles or vibrations from machinery. Once the response is predicted, the engineer turns to the capacity of the structure and designs it to withstand the forces, moments and stresses retrieved from the analysis. Designing involves computing optimal member sizes and structural layouts such that strength, stability and serviceability requirements are met to a set of prescribed standards. This modelling, analysis and design cycle is carried out iteratively for a variety of predicted simulations and the final

structural design arrived at is deemed to be safe and fit for the intended use.

Almost all nontrivial, modern-day structural analysis is carried out using the finite element (FE) method. In broad terms, as the name suggests, this method involves discretizing the mathematical model of the structure by dividing it into *finite elements*. A finite element [13, p. 4] "is a region in space in which a function ϕ is interpolated from nodal values of ϕ on the boundary of the region in such a way that interelement continuity of ϕ tends to be maintained in the assemblage". Within each element, the response is approximated using piecewise-smooth functions. Solving for these functions yields a set of simultaneous algebraic equations which can be linear or nonlinear depending on the behaviour modelled or the response being predicted.

More specifically, the FE-based analysis methodology is comprised of the following steps:

- Idealize the behaviour of the structure, loading and supports and identify initial conditions.
- Build a geometrical representation and create a mathematical model using differential equations.
- Discretize the model.
- Solve the discretization numerically for the desired response.

The solution must satisfy the conditions of equilibrium of forces, constitutive relations of materials and compatibility of strains. Where these criteria are not met, we must refine one or more steps in the above procedure and solve the problem again.

The equations that arise out of the discretization can be solved by recasting them as matrix equations and then employing matrix algorithms and floating point computations to find their solutions. Thus, numerical linear algebra (NLA) and numerical analysis play a central role in the entire structural analysis process.

1.2 Matrices in Structural Analysis

Historically, the appearance and use of matrices in structural analysis pre-dates the use of finite element method in structural mechanics. Engineers as far back as the 1930s [17]

used the Force-Displacement methods to formulate and solve stiffness matrices for models made up of beams. Felippa's paper "A historical outline of matrix structural analysis: a play in three acts" [20] presents an enlivening account of the history of matrices in structural analysis starting with the early, pre-FE approaches and following through to more modern formulations.

The most common implementation of the finite element method in structural engineering is the Direct Stiffness Method (DSM) [19, chap. 2]. DSM is a way of automating the FE procedure that allows setting up and handling the equations from the discretization using a library of element types. These element types are templates into which the practitioner can plug-in parameters like the geometry and material properties to obtain a matrix representing each part of the discretized model. The element type encapsulates the spatial dimensionality and the piecewise-smooth basis functions that approximate the differential equations. Spatial dimensionality means that elements can be defined in one-, two- or three-dimensional space. The basis functions, also known as shape functions in engineering literature, are usually polynomials of a low degree. The choice of the element type depends on the structural behaviour, the desired accuracy in the solution and the computational capability available at hand. Based on the element connectivity, the individual matrices are assembled into a global matrix called the Stiffness matrix (hence the name Direct Stiffness Method). The DSM therefore allows all parts of the discretization to be handled in a uniform manner regardless of their complexity. Since this property makes it amenable to computer programming, it is the method of choice for most FE structural analysis codes.

The stiffness matrix, often denoted by K, is a real, sparse matrix and is symmetric provided the behaviour being modelled is elastic. Once the global stiffness matrix is set up, we compute the structural response to a variety of loading. These loads can either be static or dynamic in nature and using physical laws of force equilibrium we obtain matrix equations involving the stiffness matrix and load and response vectors. NLA methods are then employed to solve these matrix equations. For example, for a vector $f \in \mathbb{R}^n$ representing static loads on the structure with stiffness $K \in \mathbb{R}^{n\times n}$, the displacement $u \in \mathbb{R}^{n\times n}$ is the solution of the linear system

$$Ku = f.$$

On the other hand, for a time varying load $f(t) \in \mathbb{R}^n$, the response $u(t) \in \mathbb{R}^n$, disregarding damping, is the solution to the ordinary differential equation

$$M\ddot{u} + Ku = f$$

where $M \in \mathbb{R}^{n \times n}$ represents the lumped mass of the structure at discretization points.

In this thesis, we consider NLA problems that have arisen in FE structural analysis and develop algorithms and software implementations to solve them. In particular, we are interested in techniques that help engineers to create structural models that return more accurate results or gain algorithmic efficiency or obtain results more quickly by making better use of parallel computer architectures.

1.3 Oasys GSA

The iterative modelling–analysis–design workflow described in Section 1.1 has to be realized in software. Such software should not only implement codes to build the mathematical model and solve it, but should also provide the user an interface, typically graphical, to model the structure geometrically, mesh it and allow 'post-processing' of results—visualization and reporting of forces, moments and stresses derived from the solution.

An example of such a program is Oasys GSA [49], which is a finite element-based structural analysis package for modelling, analysis and design of buildings, bridges, stadiums and other structures. GSA is a Windows, desktop-based program written in C and C++ and it supports the following analysis features:

- Materials: Isotropic, orthotropic and fabric.
- Analysis types:
  - Linear and dynamic analysis
  - Linear and nonlinear buckling analysis
  - Seismic response analysis
  - Static analysis of moving vehicle loads
  - Form finding analysis

Table 1.1: Element types in GSA

Element type	Number of nodes	Spatial dimensionality	Order of shape functions
SPRING	2	1-D	Linear
BEAM, BAR STRUT, TIE	2	1-D	Cubic
QUAD4	4	2-D	Linear
TRI3	3	2-D	Linear
QUAD8	8	2-D	Quadratic
TRI6	6	2-D	Quadratic

  - Dynamic relaxation analysis
  - Footfall and vibration analysis.

- Element types listed in Table 1.1 are supported.
- Constraints: Rigid constraints, rigid links, constraint equations.
- Loading: Point loads, line loads, area and surface loads, moving loads.

The problems we tackle in this thesis have all been encountered during the development of this software program. Therefore the algorithms and codes we present have also been implemented into this package and, with the exception of chapter 5, are currently being used by practitioners to analyse industrial structural models. The implementations are either new features in the package or improvements to existing features. In the cases where they are improvements to existing features they significantly reduce execution times needed for analysis and increase the numerical robustness of the solution methodology.

1.4 Definitions

1.4.1 Notation

Throughout, we make use of the Householder notation: capital letters such as A or K denote matrices, a subscripted lower case Roman letter such as a_{ij} indicates element (i, j) of A, lower case Roman letters such as x or u denote column vectors, Greek letters denote scalars, A^T denotes the transpose of the matrix A and $\|\cdot\|$ is any vector norm and its

corresponding subordinate matrix norm. Algorithms are presented in psuedocode and use syntax and keywords from the C and MATLAB programming languages. The expression $i = 1{:}n$ indicates that i can take the values from 1 to n and $A(p{:}q, r{:}s)$ represents a block of the matrix A from rows p to q and from columns r to s. $S^{(k)}$ represents the matrix S at the k-th step of an iterative algorithm and $\hat{S}$ is used to differentiate the computed value in, say, floating point arithmetic, from the exact arithmetic representation S.

1.4.2 Floating Point Arithmetic

A *floating point* number system F is a bounded subset of the real numbers whose elements have the representation

$$y = \pm m \times \beta^{e-t},$$

where m, β, e and t are integers known as the significand, base, exponent and precision, respectively. To ensure that a nonzero $y \in F$ is unique, m is selected such that $\beta^{t-1} \leq m \leq \beta^t - 1$. The quantity $u := \beta^{1-t}$ is called the *unit roundoff*. The boundedness of F implies the existence of the integers $e_{\min}$ and $e_{\max}$ such that $e_{\min} \leq e \leq e_{\max}$.

Theorem 1.4.1. *[33, Thm. 2.2] If $x \in \mathbb{R}$ lies in the range of F then*

$$fl(x) = x(1+\delta), \quad |\delta| < u. \qquad \square$$

The range of the nonzero floating point numbers in F is given by $\beta^{e_{\min}-1} \leq |y| \leq \beta^{e_{\max}}(1-\beta^{-t})$. For IEEE 754 single and double precision numbers [1], the values of β, t, e and u are given in Table 1.2.

Table 1.2: Floating point arithmetic parameters

Type	β	t	$e_{\min}$	$e_{\max}$	u
single	2	24	-125	128	6×10^{-8}
double	2	53	-1021	1024	1×10^{-16}

1.4.3 Norms

Norms are a means of obtaining a scalar measure of the size of a vector or a matrix. Vector norms are functions $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$ and satisfy the vector norm axioms:

- $\|x\| \geq 0$ for all $x \in \mathbb{C}^n$ and $\|x\| = 0$ if and only if $x = 0$.

- $\|\alpha x\| = |\alpha|\|x\|$ for all $\alpha \in \mathbb{C}$, $x \in \mathbb{C}^n$.
- $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in \mathbb{C}^n$.

The Hōlder p-norm is defined as

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

This definition gives the following vector norms:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|,$$
$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2} = (x^* x)^{1/2},$$
$$\|x\|_\infty = \max_{1 \leq i \leq n} |x_i|.$$

The 2-norm is invariant under orthogonal or unitary transformations, i.e.

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

for a unitary or orthogonal matrix Q.

1.4.4 Matrix Norms

Matrix norms are functions $\|\cdot\| : \mathbb{C}^{m \times n} \to \mathbb{R}$ and satisfy the matrix norm axioms:

- $\|A\| \geq 0$ for all $A \in \mathbb{C}^{m \times n}$, and $\|A\| = 0$ if and only if $A = 0$.
- $\|\alpha A\| = |\alpha|\|A\|$ where $\alpha \in \mathbb{C}$, $A \in \mathbb{C}^{m \times n}$.
- $\|A + B\| \leq \|A\| + \|B\|$ for all $A, B \in \mathbb{C}^{m \times n}$.

The simplest example is the Frobenius norm

$$\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}.$$

An important class of matrix norms are those subordinate to vector norms, defined as

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

for given vector norms on $\mathbb{C}^m$ in the numerator and $\mathbb{C}^n$ in the denominator. The following subordinate matrix norms are useful:

- 1-norm: $\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|$, the maximum column sum.
- ∞-norm: $\|A\|_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|$, the maximum row sum.
- 2-norm: $\|A\|_2 = (\rho(A^*A))^{1/2}$, the spectral norm, where the spectral radius

$$\rho(A) = \max\{|\lambda| : \det(A - \lambda I) = 0\}.$$

A matrix norm is consistent if

$$\|AB\| \le \|A\|\|B\|.$$

All the above matrix norms are consistent.

The Frobenius and 2-norm are invariant under orthogonal and unitary transformations, that is, for orthogonal or unitary matrices U and V,

$$\|UAV\|_2 = \|A\|_2 \text{ and } \|UAV\|_F = \|A\|_F.$$

1.4.5 Forward and Backward Errors

Error analysis in numerical linear algebra uses two types of errors known as *forward* and *backward errors*. Given a numerical algorithm that approximates the scalar function $y = f(x)$ by $\hat{y}$, the quantities $|f(x)-\hat{y}|$ and $|f(x)-\hat{y}|/|f(x)|$ are the absolute and relative forward errors.

However, we may wish to assess the quality of the computed solution by asking the question "For what set of data have we actually solved our problem?". If there exists a Δx such that $\hat{y} = f(x + \Delta x)$, then $|\Delta x|$ and $|\Delta x|/|x|$ are the absolute and relative backward errors, where if Δx is not unique we take the smallest such Δx.

If the forward or the backward errors are small, the algorithm is said to be forward or backward stable respectively.

1.4.6 Conditioning

The relationship between forward and backward error for a problem is governed by its conditioning, i.e., the sensitivity of the solution to perturbations in the data. If the function f in the previous section is twice continuously differentiable,

$$\hat{y} - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \frac{f''(x + \theta\Delta x)}{2!}(\Delta x^2), \;\; \theta \in (0, 1).$$

Hence for small a Δx,

$$\frac{y - y'}{y} \approx \frac{f'(x)\Delta x}{f(x)} = \left(\frac{f'(x)x}{f(x)}\right)\frac{\Delta x}{x},$$

and the magnification factor

$$c(x) = \left|\frac{xf'(x)}{f(x)}\right|$$

is called the relative *condition number* of f.

When backward error, forward error and the condition number are defined consistently, the following rule of thumb relates the three:

$$\text{forward error} \lesssim \text{condition number} \times \text{backward error}.$$

1.4.7 The Standard and Symmetric Eigenvalue Problems

The eigenvalue problem for $A \in \mathbb{C}^{n\times n}$, $x \neq 0 \in \mathbb{C}^n$ and $\lambda \in \mathbb{C}$ is defined as

$$Ax = \lambda x,$$

where λ is an eigenvalue and x the corresponding eigenvector. The equation $(A-\lambda I)x = 0$ is equivalent to $\det(A - \lambda I) = 0$, the solutions of which are the roots of the n degree polynomial $p(\lambda) = \det(A - \lambda I)$, called the characteristic polynomial of A.

When $A \in \mathbb{R}^{n\times n}$ is symmetric, all eigenvalues of A are real and there is an orthonormal basis of eigenvectors.

Theorem 1.4.2. *[25, Thm. 8.1.1] If $A \in \mathbb{R}^{n\times n}$ is symmetric, then there exists an orthogonal matrix $Q \in \mathbb{R}^{n\times n}$ such that*

$$Q^*AQ = \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n).$$

Moreover, for $k = 1 : n$, $AQ(:, k) = \lambda_k Q(:, k)$. □

1.4.8 Definiteness

A symmetric matrix A is said to be *positive definite* if $x^TAx > 0$ for any $x \neq 0$ and positive semidefinite if the inequality is not strict.

Theorem 1.4.3. *If $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite all its eigenvalues are real and positive.* □

The terms *negative definite* and negative semidefinite are used analogously whereas symmetric *indefinite* denotes a general symmetric matrix that can have both positive and negative eigenvalues.

1.4.9 Cholesky Factorization

For a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$, the Cholesky factorization is an important property that is useful as both a theoretical as well as an algorithmic tool.

Theorem 1.4.4. *[25, Thm. 4.2.5] If $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite, then there exists a unique upper triangular $R \in \mathbb{R}^{n\times n}$ with positive diagonal entries such that $A = R^T R$.* □

1.4.10 The QR Factorization

For $A \in \mathbb{R}^{m\times n}$ the QR factorization is given by

$$A = QR,$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{m\times n}$ is upper trapezoidal. The QR factorization always exists and an existence proof can be established as follows: if A is full rank and $A^T A = R^T R$ is a Cholesky factorization, then $A = AR^{-1} \cdot R$ is a QR factorization. The QR factorization is unique if A has full rank and we require R to have positive diagonal elements.

1.4.11 The Generalized and Symmetric Definite Generalized Eigenvalue Problems

The generalized eigenvalue problem (GEP) for $A, B \in \mathbb{C}^{n\times n}$ is of the form

$$Ax = \lambda Bx \tag{1.1}$$

where $\lambda \in \mathbb{C}$ is an eigenvalue and $x \neq 0 \in \mathbb{C}^n$ is an eigenvector. Analogously to the standard problem, λ is a root of the *pencil* $\det(A - \lambda B)$ and x is a corresponding null vector. This matrix pencil is *nonregular* if $\det(A - \lambda B)x = 0$ for all λ, else it is *regular*.

We can write (1.1) as

$$\alpha Ax = \beta Bx,$$

with $\lambda = \beta/\alpha$. If $\alpha = 0$, we call the eigenvalue infinite and this is a result of B being singular. For such an eigenvalue, a null vector of B is an eigenvector of (1.1). If $\alpha \neq 0$ then the eigenvalue is finite and $\beta = 0$ is a zero eigenvalue of the pencil. In this case, A is singular and a null vector of A is the eigenvector of (1.1) corresponding to the zero eigenvalue, provided the same vector does not belong to the null space of B. Where A and B do share a null vector, i.e., $Ax = Bx = 0$, $x \neq 0$, then $(\alpha A - \beta B)x = 0$ so α and β could take any value and the pencil is then nonregular.

When A and B are symmetric and B is positive definite, we call the problem the symmetric definite GEP. Using the Cholesky decomposition of $B := R^T R$, we can transform (1.1) into a standard eigenvalue problem $R^{-T} A R^{-1} y = \lambda y$, where $y = Rx$. Since $R^{-T} A R^{-1}$ is symmetric, the eigenvalues of the symmetric-definite GEP are real.

1.5 Problem Descriptions

We now list the problems discussed in the thesis. Although the issues tackled in each chapter have a common origin, we note that they are fairly self-contained investigations and therefore are presented as such. All chapters deal only with real matrices since that is the nature of our problems.

A significant component of this research is the programming of the software implementations of the algorithms we develop. Therefore we start the thesis with an introduction to software development techniques for numerical programming using the C++ programming language in **chapter 2**. We demonstrate that the use of these techniques results in high-level code that is readable, maintainable and yet has performance that is competitive with equivalent C and Fortran codes.

In **chapter 3**, we discuss the issue of ill-conditioning in stiffness matrices from an FE model. The presence of ill conditioning in these matrices was discovered when we implemented a condition number estimator in GSA, which reported the condition estimate of the stiffness matrix when engineers carried out linear/static analysis. We develop a new technique to detect the causes of ill-conditioning in these models which helps engineers

to rectify the errors that caused the ill-conditioning. We give the theoretical background for this method, which has also been implemented in the software package. We illustrate our discussion with real-life examples of structural models to which this tool has been applied.

In **chapter 4**, we investigate solution algorithms for the sparse, symmetric definite generalized eigenvalue problem that arises in structural dynamics and buckling analysis and the sparse, symmetric eigenvalue problem arising in model stability analysis. In the former, we seek to find the smallest eigenvalues and associated eigenvectors of the equation $Kx = \lambda Mx$, where K (the stiffness matrix) and M (the mass matrix) are symmetric and sparse and K is positive definite. For the latter, we wish to find the smallest and largest eigenvalues and corresponding eigenvectors for the problem $Kx = \lambda x$, where K is positive semidefinite. We use the subspace iteration method to compute these eigenpairs and investigate accelerating the algorithm through shifting and locking of converged vectors.

Chapter 5 follows on from chapter 4 by investigating the bottleneck of computations in our eigenvalue solvers: the sparse matrix multiple-vector product (SMMV) kernels. Our goal in this investigation is to obtain high computational throughput from the SMMV kernel, which in turn depends on the storage format of the sparse matrix. Current sparse matrix formats and algorithms have high bandwidth requirements and poor reuse of cache and register loaded entries, which restrict their performance. We propose the mapped blocked row format: a bitmapped sparse matrix format that stores entries as blocks without a fill overhead, thereby offering blocking without additional storage and bandwidth overheads. An efficient algorithm to decode bitmaps is developed and performance is compared with that of popular formats, including vendor implementations of sparse BLAS.

In **chapter 6**, we tackle the 'B-orthogonalization' problem that also arises in sparse eigenvalue solvers. For a tall, skinny $n \times \ell$ dense matrix X and a sparse symmetric positive-definite $n \times n$ matrix B, we wish to compute a dense matrix S of the same dimensions and column space as X such that $S^T BS = I$. The columns of the matrix X, in our case, may represent the eigenvectors of B although our solution algorithm works for any general matrix and assumes no specific properties. Current solvers in the literature

use the modified Gram-Schmidt method for B-orthogonalization but these have poor cache-efficiency. We derive a new algorithm based on the Cholesky decomposition of the matrix $X^T BX$, taking care to remove a stability limitation in the case where B and/or X are ill-conditioned. We show that our algorithm achieves better cache performance than existing algorithms by creating parallel multithreaded library implementations for manycore and multicore architectures.

We conclude the thesis with a summary of our findings in **chapter 7** and identify directions for future work.

Chapter 2

C++ for Numerical Libraries

A significant part of work for this thesis comprises of creating software implementations of the algorithms presented in standalone code. This software is designed for use in libraries in the lowest layer of GSA's software stack, a requirement that brings software engineering challenges. Our objective, therefore, is to create libraries that are comprehensible, maintainable and usable, i.e., they allow code to be composed in a way that maps closely to the mathematical operations they represent. In addition they must, of course, deliver high performance.

We use C++ for creating these implementations. C++ [65], an ISO standardized language, offers the dual advantages of performance benefits of C, of which it is a superset, and the benefits of a multi-paradigm language. Multi-paradigm (see [14, chap. 1] or [64]) simply means being able to use multiple programming styles like procedural, functional or object-oriented within the same programming environment. The programming paradigms most relevant to numerical software development are abstraction, encapsulation, compile-time genericity (templates) and task-based programming, which we combine to create code that satisfies the objectives identified above. We describe each of these paradigms briefly and demonstrate how they are implemented using C++ features. Our list is by no means exhaustive, but it illustrates a few key features of this versatile programming environment.

2.1 Abstraction Through Classes

In [64], Stroustrup defines abstraction as "the ability to represent concepts directly in a program and hide incidental details behind well-defined interfaces". We use C++ classes to provide abstraction and encapsulation, which allow high-level interfaces and syntaxes that naturally resemble mathematical operations. Consider, for example, the task of storing and manipulating dense matrices. Our data comprises the size of the matrix, the datatype of the scalar and the matrix entries held as an array of the scalar datatype. We need routines to carry out basic operations on this matrix, e.g. addition, multiplication or scaling. We can create a class that holds the data and bundle the operations with the class, making it easy to develop, maintain and use these routines. A minimal example of such a matrix class is listed below; it holds the entries as an array of `doubles`, has size `rows` × `cols`, and offers functions for multiplication, scaling and addition. Using operator overloading, we can define `operator*` for matrix multiplication and scaling and `operator+` for addition.

```
class Matrix
{
  int rows, cols;
  double *data;
  /* ..*/
public:
  Matrix();                     // constructor
  Matrix(int r,int c);          // constructor
  ~Matrix();                    // destructor
  double& Coeff(int r, int c) // element access
  {  return data[r*rows+c];  }

  const Matrix operator*(const Matrix& m, const double& d);
  const Matrix operator*(const Matrix& lhs, const Matrix& rhs);
  const Matrix operator+(const Matrix& lsh, const Matrix& rhs);
  /* .. */
};
```

The implementation of the `Matrix::operator*` function would contain the matrix multiplication code.

```
const Matrix operator*(const Matrix& lhs, const Matrix& rhs)
{
  Matrix temp(lhs.rows, rhs.cols);
  /* implementation of matrix multiplication
  ... */
  return temp;
}
```

A program that calls the `Matrix` class (the 'client code') to compute matrix multiplication would then have the following syntax.

```
Matrix A, B, C;
double alpha, beta;
// initialization, etc.
C = alpha*A*B + beta*C;
```

This contrasts in readability with, for example, using the BLAS interface in C code for achieving the same task:

```
double *A, *B, *C;
// initialization
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, A, k, B, n, beta, C, n);
```

Classes, return-by-reference and operator overloading are used extensively to allow chaining of function calls, which produces readable code. As an example, the following statement computes the sum of absolute values of the coefficients in a matrix `S`:

```
sum = S.cwiseAbs().colwise().sum().sum();
```

2.1.1 Generics and Template Meta-Programming

Generic programming can further extend abstraction by enabling the creation of class 'templates', i.e., code that generates other classes at compile time, based on template parameters supplied by the library user. We give three examples to illustrate this. In all cases, the goals are to obtain a single source base that reduces code replication, which leads to fewer bugs and reduced maintenance and also allows for uniformity in the client code.

Parametrizing the Datatype of Matrix

Our `Matrix` class can be extended to work with multiple precisions by making the scalar type a template parameter.

```
template <typename Scalar>
class Matrix
{
  int rows, cols;
  Scalar *data;
  /* ..*/
public:
  Matrix(int r, int c); // constructor
```

```
  ~Matrix();        // destructor

  const Matrix<Scalar> operator*(const Matrix<Scalar>& m,
                  const Scalar& d);
  const Matrix<Scalar> operator*(const Matrix<Scalar>& lhs,
                  const Matrix<Scalar>& rhs);
  const Matrix<Scalar> operator+(const Matrix<Scalar>& lhs,
                  const Matrix<Scalar>& rhs);
  /* .. */
};
```

Such an interface can now be used uniformly for all scalar types (real, complex, single precision or double precision), enabling the client code to stay the same in each case.

```
  Matrix<float> A, B, C;
  float alpha, beta;
  // initialization, etc.
  C = alpha*A*B + beta*C;
```

Fixed-sized Optimizations

Templates are also useful for optimizing performance. Since template parameters are available at compile-time, making this information available to the compiler can allow it to perform optimizations. For instance, the number of rows and columns of `Matrix` can be made optional template parameters, thereby enabling fixed sized matrices to be created at compile time.

```
template <typename Scalar, int RowsAtCompile=-1, int ColsAtCompile=-1>
class Matrix
{
  int rows, cols;
  Scalar *data;
  const static int nr_ = RowsAtCompile;
  const static int nc_ = ColsAtCompile;
  /* ..*/
public:
  /* construct and treat as fixed size if nr_ and nc_
    are greater than 0, else construct with runtime
    (r,c) and treat as dynamically sized. */
  Matrix();
  Matrix(int r, int c);
  /*.. */
};
```

The resultant class template can be used to create a 4×4 class type that can be instantiated as follows.

```
typedef Matrix<double,4,4> Matrixd4x4;
```

```
Matrixd4x4 A, B; // A and B are instances of a 4x4 Matrix
/* initialization, etc. */
Matrixd4x4 C = A*B;
```

This lets the compiler optimize the matrix multiplication code (in the previous section) by unrolling loops or by automatic vectorization, since it knows the loop bounds during compilation. In fact, since C++ templates are Turing complete, they can themselves be used to create a loop unroller, a technique we exploit in chapter 5.

Other examples of parameters that can be *templatized* are the storage order (row major vs. column major) or the structure (symmetric/banded/full).

Explicit Vectorization

Templates are used not only for high level abstractions but also to simplify low level code like SIMD programming. Consider a snippet for computing the vectorized inner product of two `float` arrays `a` and `b`, of length `N` using SSE compiler intrinsics[1].

```
__m128 X, Y;
__m128 acc = _mm_setzero_ps();
float inner_prod(0.), temp[4];

for(int i(0); i<N; i+=4) {
  X = _mm_load_ps(&a[i]); // load chunk of 4 floats
  Y = _mm_load_ps(&b[i]);
  acc = _mm_add_ps(acc, _mm_mul_ps(X, Y)); // multiply X with Y
                       // and accumulate into acc
}
_mm_store_ps(&temp[0], acc); // store acc into an array of floats
inner_prod += temp[0] + temp[1] + temp[2] + temp[3]; // reduce
```

This code is specific to processors that support SSE instructions. The code for AVX or newer instructions would call different intrinsic functions and since the number of SIMD architectures is only increasing, this gives rise to repetition and extensive branching. The above can however be rewritten using Vc [41], a template-based C++ library for SIMD vectorization, as shown below. Vc provides the class `float_v` for handling SIMD 32-bit floating point vectors agnostic of the length of the vector or architecture.

```
float_v acc (Vc::Zero);

for(int i(0); i<N; i+=float_v::Size)
{
  float_v X(&a[i]);
```

[1] We assume `N` is a multiple of 4.

```
  float_v Y(&b[i]);
  acc += X * Y;
}
inner_prod = acc.sum();
```

The above snippet works for all SIMD instruction sets supported by Vc and has the same performance characteristics as the hand-crafted intrinsics code.

2.1.2 Reclaiming Performance using Expression Templates

Object orientation and generic programming are neither new nor unique to C++, but the reason object based-abstractions are not popular in linear algebra codes is that they can incur performance penalties. Consider the following expression in the light of the implementation of `operator*` in section 2.1.

```
C = alpha*A*B + beta*C;
```

This assignment of matrix-matrix multiplication incurs four temporary `Matrix` object creations: one each to store the results of `A*B` (`t1`), `alpha*t1` (`t2`), `beta*C` (`t3`) and `t2+t3` (`t4`). The temporary allocations and deallocations decrease performance because they destroy cache locality and incur data copying costs. To overcome this performance limitation, we use a technique called 'expression templates' [35].

The expression template approach involves creating a parse-tree of the whole expression, removing any temporary objects and deferring evaluating the expression until it is assigned to the target. An expression template-based multiplication routine would return a lightweight temporary object that acts as a placeholder for the operation:

```
class MatMul
{
  const Matrix& _lhs;
  const Matrix& _rhs;
public:
  MatMul(const& Matrix left, const Matrix& right)
    :_lhs(left), _rhs(right)
  {}

  int Rows() const
  { return _lhs.rows; }

  int Cols() const
  { return _rhs.cols; }

  double Coeff(int r, int c)
  {
```

```
    double val=0.;
    for(int i=0; i<_lhs.rows; ++i)
      val += _lhs.Coeff(r,i)*_rhs.Coeff(i,c);
    return val;
  }
 };
const MatMul operator*(const Matrix& lhs, const Matrix& rhs)
{
  return MatMul(lhs, rhs);
}
```

Instead of computing the matrix multiplication, `operator*` returns a type that has `const` references to the two operands. By overloading the assignment operator for `Matrix` to accept a `MatMul` argument, we can allow this type to be assigned to `C`.

```
Matrix C = A*B;
```

The assignment constructor is implemented such that the product is computed at the time the assignment is invoked.

```
class Matrix
{
  /* other code */
public:
  Matrix& operator=(const MatMul& prod)
  {
    _rows = prod.Rows();
    _cols = prod.Cols();
    for(int i=0; i<rows; ++i)
      for(int j=0; j<cols; ++j)
        this->Coeff(i, j) = prod.Coeff(i, j); // actual multiplication
                                              // happens here
    return *this;
  }
};
```

In [35], the authors present performance comparisons and demonstrate that expression template-based C++ codes are capable of achieving the same performance as hand-crafted C code.

Several C++ linear algebra libraries implement expression-templates, for example Boost uBlas[2], MTL[3] and Eigen [26]. We use Eigen for our work because of its extensive functionality and excellent performance [27]. Therefore, where we implement algorithms as library code, they are designed to use Eigen matrices as inputs and outputs and offer a similar level of genericity as Eigen does.

[2] http://www.boost.org/doc/libs/1_55_0b1/libs/numeric/ublas/doc/index.htm
[3] http://www.simunova.com/en/node/24

2.1.3 Implementation Example

As an example code that demonstrates the software benefits discussed above, we list an implementation developed using Eigen. In Chapter 3, we describe our use of Higham and Tisseur's block 1-norm estimation algorithm [34, Alg. 2.2] in GSA. This algorithm estimates the 1-norm of a matrix using only matrix-vector or matrix-matrix operations. It does not need access to the entries of the matrix as long as it can access a routine that computes the matrix vector products; therefore it can be used to estimate the norm of the inverse of a matrix.

This algorithm is implemented in the template class `OneNormEst` in Section 2.2. The template parameter is the data type of the scalar of the square matrix A whose norm we wish to estimate; therefore the class can work with matrices of both `float` and `double`. The `OneNormEst` class takes as constructor arguments the size of A and the number of vectors to be used for the estimation (t in [34, Alg. 2.2]). The equivalent FORTRAN implementation, `DLACN1`, for double precision only, is listed in [12]. On comparison, it is clear that `OneNormEst` is more concise, generic and comprehensible (assuming the programmer understands C++ syntax).

To invoke the norm estimation routine, the client calls `OneNormEst::ANorm`, which takes two anonymous functions as arguments. The anonymous functions, called *lambda* in C++, return the effect of applying A and A^T on a vector and emulate a functional-style of programming. We present a sample snippet that demonstrates their use. For a sparse symmetric matrix A held as an instance of `Eigen::SparseMatrix` and a matrix of column vectors X, also held as an instance of the dense matrix class `Eigen::Matrix`, the following lambda functor returns the product $Y = A \times X$.

```
// Create object A
Eigen::SparseMatrix<double> A(n, n);
/* initialize as a symmetric matrix object
...
*/

// declare lambda
auto applyA = [&] (Eigen::Matrix<double>& X, Eigen::Matrix<double>& Y)
{
  Y = A * X;
};
```

To invoke the norm estimator, we call the `ANorm` function and pass the lambda functor as

one of the arguments. These functors are then invoked within the `ANorm` function body at lines 38 and 99.

```
// norm estimator object
OneNormEst<double> norm_estimator(n);

// the output
double est(0.);

// call function ANorm
norm_estimator.ANorm(applyA, applyA, est);
```

Thus, the client code consists of a single call to the `ANorm` function with the appropriate lambda functions to retrieve the norm estimate. This contrasts with the reverse communication-style interface of `DLACN1`, which passes the control back to the calling code at the end of each iteration, resulting in code that looks more complicated than the lambda approach.

2.2 The `OneNormEst` class

```
1  template <typename Scalar>
2  class OneNormEst
3  {
4  public:
5    OneNormEst(const int& n, const int& t=4): m_n(n), m_t(t) {}
6    ~OneNormEst() {}
7
8    template<typename A_operator, typename A_transpose_operator>
9    bool ANorm(const A_operator& applyA, A_transpose_operator& ↩
       applyA_trans, Scalar& norm)
10   {
11     Eigen::Matrix<Scalar> X(m_n, m_t), Y(m_n, m_t), Z(m_n, m_t);
12
13     std::uniform_int_distribution<int> rand_gen(0, m_n-1);
14     auto random_plus_minus_1_func = [&](const Scalar& ) -> Scalar
15     {
16       if (rand_gen(engine)>m_n/2)
17         return 1;
18       else
19         return -1;
20     };
21
22     X = X.unaryExpr(random_plus_minus_1_func);
23     X.col(0).setOnes();
24     X /= m_n;
25
26     Eigen::Matrix<Scalar> S(m_n, m_t), S_old(m_n, m_t); // sign matrix
27     Eigen::MatrixXi prodS(m_t, m_t);
```

```
28      S.setZero();
29
30      Scalar est=0., est_old = 0.;
31
32      int ind_best(0);
33      std::vector<int> indices(m_n);
34      std::vector<int> ind_hist;
35      Eigen::VectorXd h(m_n);
36
37      for (int k(0); k<itmax; ++k) {
38        applyA(X, Y); // Y = A * X
39
40        int ind(-1);
41        for(int i(0); i<m_t; ++i) {
42          Scalar norm = Y.col(i).cwiseAbs().sum(); // norm = {||Y||}_1
43          if (norm > est) {
44            est = norm;
45            ind = indices[i];
46          }
47        }
48
49        if (ind != -1 && (est > est_old || k==1))
50          ind_best = ind;
51
52        if(ind_best > m_n) {
53          std::stringstream message;
54          message<<"OneNormEst: Error in accessing vector index";
55          throw std::exception(message.str().c_str());
56        }
57
58        if (est < est_old && k >= 1) {
59          norm = est_old;
60          return true;
61        }
62
63        est_old = est;
64        S_old = S;
65
66        // S = sign(Y)
67        S = Y.unaryExpr([&] (const Scalar& coeff) -> Scalar
68        {
69          if(coeff >= 0.) return 1.;
70          else       return -1.;
71        });
72
73        prodS = (S_old.transpose() * S).matrix().cast<int>();
74
75        // if all cols are parallel, prodS will have all entries = n
76        if (prodS.cwiseAbs().colwise().sum().sum() == m_n*m_n*m_t) {
77          norm = est;
78          return true;
79        }
80
81        // if S_old(i) is parallel to S(j), replace S(j) with random
```

```
82        for(int i(0); i<m_t; ++i) {
83          for(int j(0); j<m_t; ++j) {
84            if(prodS.coeff(i, j)==m_n)
85              S.col(j) = S.col(j).unaryExpr(random_plus_minus_1_func);
86          }
87        }
88
89        // if S(i) is parallel to S(j), replace S(j) with random
90        prodS = (S.transpose()*S).matrix().cast<int>();
91
92        for(int i(0); i<m_t; ++i) {
93          for(int j(i+1); j<m_t; ++j) {
94            if(prodS.coeff(i, j)==m_n)
95              S.col(j) = S.col(j).unaryExpr(random_plus_minus_1_func);
96          }
97        }
98
99        applyA_trans(S, Z);//Z = A_transpose * S
100
101       h.matrix() = Z.cwiseAbs().rowwise().maxCoeff();
102
103       if (k >= 1 && h.maxCoeff() == h.coeff(ind_best) ) {
104         norm = est;
105         return true;
106       }
107
108       for(int i(0); i<m_n; ++i)
109         indices[i] = i;
110
111       // sort indices by entries in h
112       std::sort(indices.begin(), indices.end(), [&](int& left, int& ↩
              right)
113       {
114         return h.coeff(left) > h.coeff(right);
115       });
116
117       int n_found(0);
118       for(auto i=indices.begin(); i!=indices.begin()+m_t; ++i) // is ↩
              indices contained in ind_hist?
119         if(std::find(ind_hist.begin(), ind_hist.end(), *i) != ind_hist↩
                .end())
120           ++n_found;
121
122       if(n_found == m_t) {
123         norm = est;
124         return true;
125       }
126
127       std::vector<int> new_indices;
128
129       if(k>0) {
130         new_indices.reserve(m_t);
131         int count(0);
```

```
132         for(auto it=indices.begin()+m_t; it!=indices.end() && count<m_t; ++it)
133         {
134           if(std::find(ind_hist.begin(), ind_hist.end(), *it) == ind_hist.end()) {
135             new_indices.push_back(*it); ++count;
136           }
137         }
138       }
139       assert(indices.size()>0);
140
141       X.setZero();
142       for(auto i=0; i<m_t && k>0; ++i) {
143         X.coeffRef(indices[i], i) = 1; //Set X(:, j) = e_ind-j
144       }
145       std::copy(new_indices.begin(), new_indices.end(), std::back_inserter(ind_hist));
146       new_indices.clear();
147     }
148     norm = est;
149     return true;
150   }
151 static int itmax=5;
152
153 private:
154   int m_n; // rows
155   int m_t; // cols
156
157 };
```

Chapter 3

Detecting the Causes of Ill Conditioning in Structural Finite Element Models

3.1 Introduction

Modelling structural response using the finite element (FE) method involves idealizing structural behaviour and dividing the structure into elements. The solution obtained from such a procedure is therefore inherently approximate. The errors are of a variety of types, such as error in the choice of mathematical model (when the physical system being modelled does not obey the assumptions of the mathematical model chosen), discretization errors (arising from representing infinite dimensional operators in finite spaces), and numerical errors (those caused by representing real numbers as finite precision numbers on a computer). Techniques for understanding, analysing and bounding these errors have developed in pace with the method itself and their study is part of any standard text on finite element analysis—see [7] or [13, chap. 18] for example. User errors (errors in the definition of the model within the software from say, erroneous input), on the other hand, have received significantly less attention in the literature. This is partly due to the practitioner using FE analyses being disconnected with the process of developing it or implementing it in software but mainly because such errors can arise arbitrarily, which poses a barrier to understanding and analysing them. Examples of such errors include

- Lack of connectivity: adjacent elements that are supposed to share a common node but are connected to different nodes that are coincident, resulting in one of the elements acquiring insufficient restraints.

- Failure to idealise member end connections correctly, which can occur if a beam is free to rotate about its axis, although in the physical model there is a nominal restraint against rotation.

- Modelling beam elements with large sections and/or very small lengths, often the result of importing FE assemblies from CAD models.

Irrespective of whether the error arose from approximation or from erroneous input data, it can lead to an ill-conditioned problem during its analysis, that is, one for which the stiffness matrix of the model has a large condition number with respect to inversion. In this work, we show how errors that lead to an ill-conditioned problem can be detected, i.e., we present a technique to identify the parts of the model that cause the ill conditioning. Our method has been implemented in Oasys GSA and we demonstrate its efficacy with examples of how it has been used to identify errors in user-generated ill-conditioned models.

3.2 Condition Number Estimation

Linear systems of equations arise in many problems in structural analysis. For a structural model with symmetric positive definite (or semidefinite) stiffness matrix $K \in \mathbb{R}^{n\times n}$, elastic static analysis for calculating the displacement $u \in \mathbb{R}^n$ under the action of loads $f \in \mathbb{R}^n$ yields the system $Ku = f$. Other types of analysis that involve solving linear equations with the stiffness matrix include dynamic, buckling, P–Delta, and nonlinear static analysis. Solving a linear system on a computer involves approximating the entries of K as floating point numbers, which introduces an error ΔK (we disregard, without loss of generality, the error introduced in f due to the same process). Denoting the corresponding change to u by Δu, the equation we solve is

$$(K + \Delta K)(u + \Delta u) = f.$$

Rearranging, taking norms and dropping the second order term $\Delta K \Delta u$ gives the inequality, correct to first order,

$$\frac{\|\Delta u\|}{\|u\|} \leq \kappa(K) \frac{\|\Delta K\|}{\|K\|}, \tag{3.1}$$

where $\kappa(K) = \|K\|\|K^{-1}\|$ is the condition number (with respect to inversion). The norm $\|\cdot\|$ can be any subordinate matrix norm, defined in terms of an underlying vector norm by $\|K\| = \max_{\|x\|=1} \|Kx\|$.

Condition numbers measure the maximum change of the solution to a problem with respect to small changes in the problem. Inequality (3.1) tells us that the relative error in u is bounded by the relative error in K times its condition number. This bound is attainable to first order for a given K and f [33, Thm. 7.2], so the change in the solution caused simply by storing K on the computer can be large if K is ill-conditioned. Rounding errors in the solution process can be shown to correspond to an increased $\|\Delta K\|$ in (3.1) and so are also magnified by as much as $\kappa(K)$.

In IEEE 754 double precision binary floating point arithmetic [1] we have a maximum of the equivalent of 16 significant decimal digits of precision available and we therefore have as little as $16 - \log_{10} \kappa(K)$ digits of accuracy in the computed solution. When a matrix has a condition number greater than 10^{16}, the solution algorithm can return results with no accuracy at all—such a matrix is numerically singular and linear systems with this matrix should not be solved. Therefore it is essential to compute or estimate the condition number of the stiffness matrix K to ensure it is well conditioned.

Since K is symmetric and positive definite (or positive semidefinite), its 2-norm condition number $\kappa_2(K)$ is the ratio of its extremal eigenvalues $\lambda_{\max} = \max_i \lambda_i$ and $\lambda_{\min} = \min_i \lambda_i$ [33, Chap. 6]:

$$\kappa_2(K) = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

Computing the maximum and minimum eigenvalues is an expensive operation, particularly since K is large. In practice we need just an estimate of the condition number that is of the correct order of magnitude, and we can choose any convenient norm to work with [33, chap. 15]. LAPACK [2] offers the `xLACON` routine that computes a lower bound for the 1-norm of a matrix, based on the algorithm of Higham [32]. Higham and Tisseur [34] develop a block generalization of this algorithm and demonstrate that it produces

estimates accurate to one or more significant digits at a nominal cost of a few matrix–vector multiplications. The algorithm does not need to access the elements of the matrix explicitly, as long as it can access a routine that returns the matrix–vector product. Thus it can be used to estimate the norm of the inverse of a matrix K as long as one can form the product $K^{-1}v =: g$, which is equivalent to solving the linear system $Kg = v$.

We incorporated a procedure in GSA that uses this algorithm to calculate an estimate of the 1-norm condition number of K. This method is invoked and the condition number is reported for all analysis types that involve assembling K. It first computes $\|K\|_1$ and then computes the estimate of $\|K^{-1}\|_1$, making use of the Cholesky factors of K. Since the analysis already requires the Cholesky factorization of K to be computed, the condition estimate comes at a nominal cost of a few triangular system solves.

Once the software had been released, users were informed when they had a model with an ill-conditioned stiffness matrix. As a result, the developers were subsequently faced with the question: "*how do we detect where in the FE model the ill conditioning lies?*". To answer this question we first need to look at the possible causes of ill conditioning.

3.3 Ill Conditioning in Structural Stiffness Matrices

The stiffness matrix is constructed by summing up stiffnesses at each degree of freedom (dof) in the model and each dof corresponds to a row (or a column) in the matrix. The dofs are based at nodes and each node has 6 dofs corresponding to 3 translations and 3 rotations. The stiffnesses at each dof come from elements that connect to the nodes at which the dofs are defined. The stiffness contributed by an element e is defined in its element stiffness matrix $K^{(e)}$ and depends on the element type and its physical and geometrical properties. Let the element stiffness matrix contributed by each element e in the domain Ω be $K^{(e)}$. The matrix $K^{(e)} \in \mathbb{R}^{n_e \times n_e}$ is symmetric and it is defined in a coordinate system (of displacements) and degrees of freedom (dofs) local to the element. To map the displacements to the global coordinate system we use the coordinate transformation matrix $T^{(e)}$; we also use a set m_e of n_e variables that map the locally numbered dofs to the global dof numbering. The stiffness matrix K is the sum of contributions from

all elements in the domain Ω transformed to global:

$$K = \sum_{e \in \Omega} G^{(e)}, \tag{3.2}$$

where

$$G^{(e)}(m_e, m_e) = {T^{(e)}}^T K^{(e)} T^{(e)}.$$

If an element connects dof i with dof j, then k_{ii}, k_{jj}, and $k_{ij} = k_{ji}$ are all nonzero, assuming an element with nonzero stiffness in directions of i and j connects the dofs. Because of its construction K is symmetric, with very few nonzero entries per row.

The matrix K becomes ill-conditioned when its columns are nearly linearly dependent. This can happen when

(a) the structure has one or more pairs or tuples of dofs that do not have sufficient connectivity with the rest of the model or

(b) certain dofs have stiffnesses disproportionate with the rest of the model.

We say a pair (i, j) of dofs connected to each other has insufficient connectivity when the stiffness contributions of terms k_{is} and k_{rj} for $r, s \in (1, n)$ with $r, s \notin (i, j)$ are either very small or at roundoff level compared with k_{ii}, k_{jj}, and k_{ij}. (The definition can be easily expanded for higher tuples of dofs.)

Possibility (a) occurs when elements do not have sufficient connectivity, for example a beam element that is connected to nodes at which there is no torsional restraint. Typically the resultant matrix would be singular since the structure is a mechanism, but it is possible that due to rounding during coordinate transformations, entries in columns k_i or k_j acquire small nonzero values. If i and j are the dofs corresponding to axial rotation at the two ends of the column member, such a model would result in a matrix with columns i and $j > i$ resembling (with k_i denoting the ith column of K)

$$k_i = [0, \ \ldots \ , 0, \ a, \ 0, \ \ldots, 0, \ -a, \ 0, \ldots]^T$$

and

$$k_j = [0, \ \ldots, \ \epsilon, \ -a, \ \epsilon, \ \ldots, \ \epsilon, \ a, \ 0, \ \ldots]^T,$$

where the entries $k_{ii} = k_{jj} = a > 0$ and $k_{ij} = k_{ji} = -a$ and all other entries ϵ in k_j arising from other dofs connected to dof j are such that $|\epsilon| \ll a$. More generally, a tuple $\mathcal{S} \subseteq \{1, \ldots, n\}$ of dofs can arise such that for $i \in \mathcal{S}$, k_{ij} is nonzero only for $j \in \mathcal{S}$.

The situation that would result in (b) is when certain elements that are disproportionately stiff in particular directions connect with more flexible elements in the neighbourhood. This results in a badly scaled matrix, and can be seen, for example, when beam or column members are split in numerous line elements—usually the result of importing a CAD drawing as an FE assembly.

It is impossible to obtain a full list of modelling scenarios that will result in an ill-conditioned stiffness matrix, but in section 3.5 we present a few examples of real life models we have encountered. For now, we focus on how the properties of the matrix can be exploited to identify the location of the anomalies that cause ill conditioning. By *location of anomalies*, we mean the identification of the errant dofs $\mathcal{S}$, and subsequently problematic elements, such that an examination of the model defined in the vicinity of these elements can help the user identify the issue.

Past work has focused on identifying mechanisms in finite element models. Mechanisms render the stiffness matrix singular, so the problem is the same as finding the null space of the matrix, though in the case of floating structures the matrix can also have a nontrivial null space corresponding to its rigid body modes. Farhat and Géradin [18] and Papadrakakis and Fragakis [50] deal with the computation of the null space of stiffness matrices using a combination of algebraic and geometric information specific to the discretization, whereas Shklarski and Toledo [58] use a graph theoretic approach for computing the null space.

Whilst the identification of mechanisms is useful, rectifying the error is a case of fixing the unconstrained dof. Moreover, since a mechanism results in a singular stiffness matrix, an attempt to find its Cholesky factors (during, say, linear static analysis) is likely to break down and hence signal the problem [33, Chap. 10]. An ill-conditioned matrix, however, poses more challenges. Ill conditioning can result from subtler errors that might be hard to detect, but their presence can lead to numerical inaccuracies in results.

Our technique for diagnosing user errors works for both ill conditioning errors as well as mechanisms. It provides the user with qualitative information about the location of these errors in the structure being modelled. The method uses eigenvectors of the stiffness matrix, so it does not need new algorithms to be deployed in software packages but rather

can make use of existing well-known techniques for solving eigenvalue problems.

3.4 Using Eigenvectors to Identify Cause of Ill Conditioning

We now describe our method to identify elements and dofs that cause ill conditioning in the stiffness matrix. The key insight is that the eigenvectors corresponding to extremal eigenvalues of K (which, with a slight abuse of notation, we will refer to as the *smallest/largest eigenvectors*) contain rich information about the dofs that cause ill conditioning. We assume the model does not have rigid body modes, i.e., K does not have a nontrivial null space corresponding to rigid body motion. If the ill conditioning of K is due to the reasons outlined in section 3.3, the smallest and/or the largest eigenvectors are *numerically sparse*. We call a normalized vector numerically sparse when it has only a small number of components significantly above the roundoff level. In the remainder of this section, we show that when ill conditioning is caused by insufficient connectivity the eigenvectors corresponding to the one or more smallest eigenvalues are numerically sparse, whereas when the ill conditioning is from the presence of elements with disproportionately large stiffnesses the largest eigenvectors exhibit numerical sparsity. If we define the inner product terms

$$v^{(e)} = \frac{1}{2} u_i(m_e)^T u_i(m_e), \quad e \in \Omega \tag{3.3}$$

and

$$s^{(e)} = \frac{1}{2} u_i(m_e)^T T^{(e)^T} K^{(e)} T^{(e)} u_i(m_e), \quad e \in \Omega \tag{3.4}$$

for a normalized eigenvector u_i and element e, then the elements that cause ill conditioning are those that have large relative values of either $v^{(e)}$ for the smallest eigenvectors or $s^{(e)}$ for the largest eigenvectors. The element-wise scalars $v^{(e)}$ and $s^{(e)}$, respectively, can be thought of as virtual kinetic and virtual strain energies associated with the modes of displacements defined by the eigenvectors; therefore we refer to them as virtual energies later on in the text.

Our method for identifying ill conditioning in a finite element assembly is outlined in Algorithm 3.1. We call it *model stability analysis*, since it analyses the numerical stability

of the underlying model.

Algorithm 3.1 Algorithm for model stability analysis.

This algorithm has the following user-defined parameters:

- $n_s \geq 0$: the number of smallest eigenpairs;
- $n_\ell \geq 0$: the number of largest eigenpairs;
- $\tau > 1$: the condition number threshold for triggering analysis;
- $\text{gf} \geq 1$: the order of the gap between a cluster of smallest eigenvalues and the next largest eigenvalue.

1. Compute a condition number estimate $\text{est} \approx \kappa_1(K)$.
2. If $\text{est} < \tau$, exit.
3. Issue ill conditioning warning.
4. Compute the n_s smallest eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_{n_s}$ and n_ℓ largest eigenvalues $\lambda_{n-n_\ell+1}, \ldots, \lambda_n$ of K and normalize the associated eigenvectors.
5. With the smallest eigenpairs: determine if a gap exists, i.e., if there is a $k < n_s$ such that
$$\frac{\lambda_{k-1}}{\lambda_k} > \text{gf} \times \frac{\lambda_k}{\lambda_{k+1}}$$
If no such k is found go to step 7.
6. For each eigenvector u_i, $i = 1$ to k, calculate $v^{(e)}$ for all elements.
7. With the largest eigenpairs: for each eigenvector u_i, $i = n - n_l$ to n, compute $s^{(e)}$.

Once the algorithm finishes executing, the user must find elements with large virtual energies for each eigenpair. Isolating elements with large relative values of $v^{(e)}$ or $s^{(e)}$ is based on visual inspection of the values. This is done by graphically colour-contouring the scalars on elements as discs whose radii and colour depend on the relative values of the scalars. Once the elements with large relative energies are identified the user must examine their definition (e.g., support conditions, nodal connectivity or section properties) for anomalies and fix discrepancies. This should result in $\kappa(K)$ decreasing, hence the proposed workflow is iterative: execute Algorithm 3.1, fix anomalies, check if $\text{est} < \tau$, and repeat the steps if necessary.

We now explain the reasoning behind our method by relating the ill conditioning to properties of the largest and smallest eigenvectors.

3.4.1 Smallest Eigenpairs

In the case of a few insufficiently connected dofs, the sparsity of eigenvectors stems from their continuity over perturbations. An ill conditioned matrix K is a small relative distance from a singular matrix $\widehat{K}$, as shown by the following result of Gastinel and Kahan [33, Thm. 6.5].

Theorem 3.4.1. *For any $n \times n$ matrix A the distance*

$$\operatorname{dist}(A) = \min \left\{ \frac{\|\Delta A\|}{\|A\|} : A + \Delta A \text{ singular} \right\}$$

is given by

$$\operatorname{dist}(A) = \kappa(A)^{-1},$$

where the norm is any subordinate matrix norm.

For an ill conditioned K, the eigenvectors of K corresponding to the smallest eigenvalues are the perturbed null vectors of a nearby singular matrix $\widehat{K}$, and since the null vectors of $\widehat{K}$ can reveal unconstrained dofs, as shown by Lemma 3.4.2 below, the smallest eigenvectors of K contain the same information as long as the perturbation is small.

We first show that the null vector of a singular matrix with a specific property has a predefined structure. We then demonstrate how a small perturbation to this singular matrix does not change the structure of the null vector. The proof is then generalized for an invariant subspace of a matrix and we show that under mild assumptions the eigenvectors of an ill conditioned matrix reveal dofs that are insufficiently restrained.

We start with the following lemma. All norms used throughout are 2-norms.

Lemma 3.4.2. *Let $A = [a_1, \ldots, a_n] \in \mathbb{R}^{n \times n}$ have columns $a_1, \ldots, a_{n-1}$ linearly independent and a_{n-1} and a_n dependent. Then*

$$\operatorname{Null}(A) = \{x \in \mathbb{R}^n \ : \ Ax = 0\} = \operatorname{span}\{u\},$$

where u has the form

$$u = [0, \ldots 0, \alpha, \beta]^T \tag{3.5}$$

for $\alpha, \beta \in \mathbb{R}$ with $\beta \neq 0$.

Proof. By assumption, $\operatorname{rank}(A) = n-1$ and hence $\dim \operatorname{null}(A) = 1$.

Write $a_n = \gamma a_{n-1}$ (this is always possible since $a_{n-1} \neq 0$). Then

$$Au = 0 \iff u_1 a_1 + u_2 a_2 + \cdots + u_{n-2} a_{n-2} + (u_{n-1} + \gamma u_n) a_{n-1} = 0,$$

and since $a_1, \ldots, a_{n-1}$ are linearly independent, $u_1 = \cdots = u_{n-2} = 0$ and $u_{n-1} = -\gamma u_n$. So $u \neq 0$ if $u_n \neq 0$ and in that case u has the required form. □

The matrix A in Lemma 3.4.2 has a simple eigenvalue 0 and the null vector u is a corresponding eigenvector. Now suppose A is, additionally, symmetric and consider a perturbed matrix $\tilde{A} = A + E$ for a symmetric perturbation E. The next result shows that if $\tilde{u}$ is the smallest eigenvector of $\tilde{A}$ then for small perturbations the structure of $\tilde{u}$ is similar to that of the null vector u.

Theorem 3.4.3. *Let A, as defined in Lemma 3.4.2, be symmetric positive semidefinite and perturbed with a symmetric $E \in \mathbb{R}^{n\times n}$ to $\tilde{A} = A + E$ such that $\tilde{A}$ is positive definite. Let $0, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of A in increasing order and u be a null vector of A with $\|u\| = 1$. Let t be the smallest eigenvalue of $\tilde{A}$ and $\tilde{u}$ be an eigenvector of unit 2-norm associated with t. Then, provided $t \ll \lambda_2$, $\tilde{u}$ has the structure $\psi\tilde{u}^T = [\delta_1, \ldots, \delta_{n-1}, \alpha + \delta_{n-1}, \beta + \delta_n]$ for some scalar $\psi \in \mathbb{R}$, with the δ_i of order $\|E\|/(\lambda_2 - t)$.*

Proof. Since the columns a_1 to a_{n-1} of A are linearly independent, the eigenvalue λ_2 is positive. Let $M := \operatorname{diag}(\lambda_2, \ldots, \lambda_n)$ and let an eigenvector matrix with orthonormal columns corresponding to eigenvalues in M be denoted by $U \in \mathbb{R}^{n\times(n-1)}$. We then have

$$\begin{bmatrix} u^T \\ U^T \end{bmatrix} A \begin{bmatrix} u & U \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & M \end{bmatrix}, \tag{3.6}$$

with $[u\ U]$ orthogonal and $M \in \mathbb{R}^{(n-1)\times(n-1)}$.

Let $(t, \tilde{u})$ be an eigenpair of $\tilde{A}$. Then $\tilde{A}\tilde{u} = t\tilde{u}$. We can write $\tilde{u} = \psi u + Up$, $\psi \in \mathbb{R}$ and $p \in \mathbb{R}^{n-1}$, with $\|[\psi\ p^T]\|_2 = 1$, to be determined. Noting that $Au = 0$ and $AU = UM$, we can rewrite $\tilde{A}\tilde{u} = t\tilde{u}$ as

$$UMp + \psi Eu + EUp = \psi tu + tUp. \tag{3.7}$$

Premultiplying (3.7) by U^T yields

$$Mp + \psi U^T Eu + U^T EUp = tp,$$

since $U^T u = 0$. Rearranging,

$$(tI - M)p = U^T E(\psi u + Up),$$

so

$$p = (tI - M)^{-1} U^T E(\psi u + Up),$$

as $tI - M$ is nonsingular because of the assumption $t < \lambda_2$. Taking norms on both sides gives

$$\|p\| \leq \|(tI - M)^{-1}\| \|E\| (|\psi| + \|p\|).$$

Since $|\lambda_2 - t| < |\lambda_i - t|$ for $i > 2$, $\|(tI - M)^{-1}\| = (\lambda_2 - t)^{-1}$. Therefore we can rewrite the inequality as:

$$\|p\| \leq \frac{\|E\|(\psi + \|p\|)}{\lambda_2 - t} \leq \frac{\sqrt{2}\|E\|}{\lambda_2 - t}, \tag{3.8}$$

since $|\psi| + \|p\| \leq \sqrt{2}(\psi^2 + \|p\|_2^2)^{1/2} = \sqrt{2}$ using Cauchy-Schwarz inequality. Since $t \ll \lambda_2$ from the assumptions of the theorem, $\|p\| = \|\tilde{u} - \psi u\|$ is small and hence $\tilde{u}$ has nearly the same structure as u.

□

We apply the result above to a symmetric positive semidefinite K with a simple eigenvalue 0 that is perturbed by a symmetric E such that $\tilde{K} = K + E$ is positive definite. If $(0, u)$ is the smallest eigenpair of K, with λ_2 being the next largest eigenvalue, then the perturbed eigenvector $\tilde{u}$ differs from the original null vector by a distance proportional to the product of the norm of the perturbation and the reciprocal of $d = \lambda_2$. In other words, if the null vector u has the structure defined in (3.5), its perturbation $\tilde{u}$ has the form

$$\tilde{u}^T = [\delta_1, \ldots, \delta_{n-1}, \alpha + \delta_{n-1}, \beta + \delta_n],$$

with δ_i of order $\|E\|_2/(\lambda_2 - t)$. Therefore, the smallest eigenvector of an ill-conditioned stiffness matrix will have large components corresponding to the nearly dependent columns. It is easy to show that elements that share these dofs have large values of the inner product/virtual energy $v^{(e)}$ as defined in (3.3). Let $\mathcal{D}$ be the set of dependent dofs and let element e have a mapping m_e that has a larger intersection with $\mathcal{D}$ than the mapping set of any other element in the assembly, i.e.,

$$|\mathcal{D} \cap m_e| > |\mathcal{D} \cap m_i| \quad \forall i \in \Omega, \quad i \neq e,$$

where $|\cdot|$ is the cardinality of the set. Then, $\tilde{u}(m_e)^T\tilde{u}(m_e) > \tilde{u}(m_i)^T\tilde{u}(m_i)$ for $i \in \Omega$, $i \neq e$.

We applied the bound in Theorem 3.4.3 to the special case of K with two columns dependent, resulting in a simple eigenvalue 0. More generally, there could be $p \geq 2$ columns that are dependent, corresponding to p dofs that are badly restrained. If K is scaled such that the largest eigenvalue is 1, then it has a cluster of eigenvalues close to 0 and these eigenvalues have eigenvectors with few large components. For eigenvalues that are clustered, the eigenvectors are sensitive to perturbations but the invariant subspace corresponding to the cluster is less sensitive. Our result is therefore easily generalized for a cluster of eigenvalues and the invariant subspace of the associated eigenvectors: the clustered eigenvalues of an ill conditioned symmetric matrix are a perturbation of repeated zero eigenvalues of a nearby singular matrix and the eigenvectors form a basis for the subspace that is in the neighbourhood of the null space. The following theorem, which is a special case of [25, Thm. 8.1.10], states this result. Here, we need the Frobenius norm, $\|A\|_F = \left(\sum_{i,j=1}^{n} |a_{ij}|^2\right)^{1/2}$.

Theorem 3.4.4. *Let a symmetric positive semidefinite matrix $A \in \mathbb{R}^{n\times n}$ have the eigendecomposition*

$$A = U^T\Lambda U,$$

where $\Lambda = \operatorname{diag}(\lambda_i)$ with $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ and U is a matrix of eigenvectors. Assume the spectrum of A is such that the first r eigenvalues in Λ are 0 and $\lambda_{r+1} > 0$, and partition $U = [U_1\ U_2]$, so that the columns of $U_1 \in \mathbb{R}^{n\times r}$ span the null space of A. Let $E \in \mathbb{R}^{n\times n}$ be a symmetric matrix and partition U^TEU conformably with U as

$$U^TEU = \begin{bmatrix} E_{11} & E_{12} \\ E_{21} & E_{22} \end{bmatrix}.$$

If

$$\|E\|_F \leq \frac{\lambda_{r+1}}{5},$$

then there exists a matrix $P \in \mathbb{R}^{(n-r)\times r}$ with

$$\|P\|_F \leq 4\frac{\|E_{21}\|_F}{\lambda_{r+1}}$$

such that the columns of $\tilde{U}_1 = (U_1 + U_2P)(I + P^TP)^{-\frac{1}{2}}$ form an orthonormal basis for a subspace invariant for $A + E$.

Theorem 3.4.4 suggests that where the ill conditioning is associated with a cluster of small eigenvalues we need to examine all the eigenvectors associated with the cluster to gain an understanding of the causes of ill conditioning. The identification of the gap is important and so is its size—a large gap ensures that the perturbed eigenvectors are "close" to the null vectors, and hence contain useful information about the errors.

3.4.2 Largest Eigenpairs

When K is ill-conditioned because a few elements possess large stiffnesses in comparison with other elements in the model, the largest eigenvectors are numerically sparse and we can use the element-wise virtual strain energy $s^{(e)}$ from (3.4) to identify such elements. The relationship between $\lambda_{\max}(K)$ and $\max_e\{\lambda_{\max}(K^{(e)})\}$ is already well known from the results of Wathen [68] and Fried [22]. Here, we show the connection between the largest eigenvector of K and the largest eigenvector of $K^{(e)}$ and use the relationship to demonstrate that the stiffest element has the largest virtual strain energy. We start with a localized sub-assembly of elements and then extend the argument for the entire model.

Consider an FE sub-assembly as in Figure 3.1, with a stiffness matrix S. We assume, without loss of generality, that each element in the sub-assembly has stiffness only in one direction at each end, so there is only one dof at nodes $d_1, \ldots, d_6$, and that no other coupling exists between these nodes and the rest of the model. Let the transformed stiffness matrix of element r in the direction of the dofs at the nodes be $k_r \left[\begin{smallmatrix} 1 & -1 \\ -1 & 1 \end{smallmatrix}\right]$. Since the ill conditioning scenario we are interested in is one where a few elements have much larger stiffnesses than most other neighbouring elements (that in turn have stiffnesses comparable with each other), let $k = k_1 = k_2 = k_4 = k_5$ and let element 3 be the stiffest, with $k_3 = \mu k$ for $\mu \gg 1$. Then the stiffness matrix S of the sub-assembly is

$$S = k \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & \mu+1 & -\mu & 0 \\ 0 & -\mu & \mu+1 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}. \tag{3.9}$$

We note here that the simplification of using elements with only one dof at each node is only to keep the dimension of S low and it does not affect the discussion: we can use any element as long as it has at least one dof about which it has a large relative stiffness.

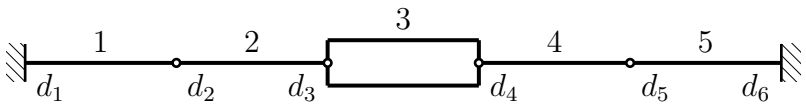


Figure 3.1: A sub-assembly. Element 3 is the stiffest, with a large difference in stiffness.

Table 3.1: $s^{(e)}$ for elements e in the sub-assembly.

Element e	1	2	3	4	5
$s^{(e)}$	1	$1+\gamma^2$	$4\gamma^2$	$1+\gamma^2$	1

We are interested in the largest eigenvector and the corresponding energies of elements when the assembly deforms in that mode.

Using the MATLAB Symbolic Math Toolbox the largest eigenvector of S is found to be (unnormalized)

$$u = [-1, \gamma, -\gamma, 1]^T, \text{ where } \gamma = \mu + (\sqrt{4\mu^2 - 4\mu + 5} - 1)/2.$$

Table 3.1 provides expressions for $s^{(e)}$ when the sub-assembly is in a displacement mode given by u. Clearly, $s^{(3)} > s^{(e)}$ for $e = 1, 2, 4, 5$ for all $\mu > 1$. Hence the stiffest element has the largest virtual energy. We also observe that the components of the eigenvector vary as polynomials of the ratio of stiffnesses, therefore the larger the variation in stiffness magnitudes, the larger the difference in the order of the components of the vector, making it more numerically sparse.

To generalize this observation, we show that when such an assembly is part of a larger structure, the largest eigenvector of the larger model has a form similar to that of the largest eigenvector of the sub-assembly.

Assume the structure in Figure 3.1 is embedded in an arbitrary FE model as in Figure

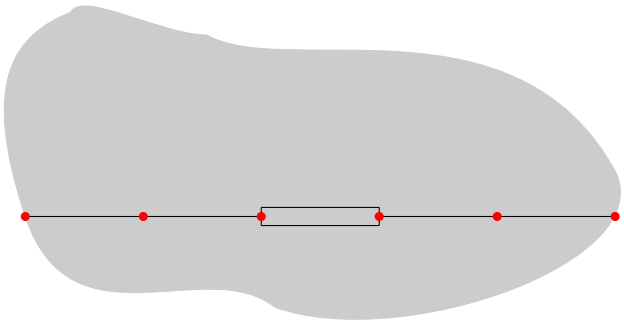


Figure 3.2: An arbitrary FE model.

3.2, with stiffnesses at nodes d_3 and d_4 larger than those at any other dof. The fixities at the ends of elements 1 and 5 in Figure 3.1 become shared nodes between the sub-assembly and the rest of the model. We represent the stiffness matrix of the entire structure as $K \in \mathbb{R}^{n\times n}$. Let b be the maximum number of nonzeros in any row of K. The quantity b represents the element connectivity in the structure.

We can order K as

$$K = \begin{bmatrix} M & F \\ F^T & S \end{bmatrix}, \tag{3.10}$$

where S is the same as in (3.9) and represents the sub-assembly and $M \in \mathbb{R}^{(n-4)\times(n-4)}$ contains dofs from the rest of the model. The stiffness terms for shared dofs are in $F \in \mathbb{R}^{(n-4)\times 4}$, which is of low rank and small norm compared with S (and can be written out explicitly). Then, assuming $b \ll \mu$, the largest eigenvector of the block-diagonal matrix

$$\widehat{K} = \begin{bmatrix} M & 0 \\ 0 & S \end{bmatrix} \tag{3.11}$$

is of the form $\hat{v}^T := [0, \ldots, 0, u] \in \mathbb{R}^n$, where u is the largest eigenvector of S. To show this we need to show that $\|M\|_2 < \|S\|_2$. If K is scaled using its construction in (3.2) such that

$$\max_{\substack{e\in\Omega \\ e\neq 2}} \|K^{(e)}\| = 1,$$

we have

$$\|K^{(3)}\| \gg 1,$$

since element 3 is the stiffest. Then informally, $\max_{i,j} |m_{ij}| = O(1)$ and $\max_{i,j} |s_{ij}| = O(\mu)$. Using [33, Prob. 6.14] and the definition of b and the symmetry of M, we have

$$\begin{aligned} \|M\|_2 &\leq b^{1/2} \max_j \|M(:,j)\|_2 \leq b^{1/2} \cdot b^{1/2} \max_{i,j} |m_{ij}| \\ &= b \max_{i,j} |m_{ij}| = O(b) < O(\mu) = \max_{i,j} |s_{ij}| \leq \|S\|_2, \end{aligned}$$

since $b \ll \mu$. This assumption is reasonable since the number of nonzeros is typically of a smaller order compared with the stiffness of the element causing the ill conditioning and conforms with our observations. We also note that b is akin to the factor $p_{\max}$ in [22,

Lemma 2] or $d_{\max}$ in [68, Equation 3.7]. The matrix

$$K = \hat{K} + \begin{bmatrix} 0 & F \\ F^T & 0 \end{bmatrix},$$

is a small perturbation of $\hat{K}$ (of norm ϵ, say). Therefore, using Theorem 3.4.3, the largest eigenvector v of K has a small distance (proportional to ϵ) from $\hat{v}$ and so retains the structure. This implies v is numerically sparse, leading to large relative values of virtual strain energies for elements that cause ill conditioning of the stiffness matrix.

3.5 Examples

The method described in section 3.4 has been implemented as an analysis option in GSA [49]. Eigenvalues and eigenvectors are computed using subspace iteration with deflation of converged vectors. The following examples illustrate how the feature has been used on structural models created by engineers on real-life projects. Since the illustrations make use of models created in GSA, we use software-specific definitions of common entities and concepts encountered in FE modelling and define new terms when we introduce them. In all but the last example, examining element virtual energies for the small eigenvalues revealed the causes of the ill conditioning.

3.5.1 Roof Truss for Performing Arts Arena

Figure 3.3 shows the roof truss model of a performing arts arena that was analysed in GSA. The roof is modelled as 'beam' and 'bar' elements, which are both one dimensional. Beam elements have 6 dofs at each end, 3 each for translation and rotation, whereas bars have two dofs corresponding to axial extensions only. The main trusses run along the shorter diameter of the oval, with longitudinal bracing connecting each truss at the top and bottom chord to provide lateral stability. The model is supported on pins and rollers at joints close to the circumference.

On an initial version of the model, the condition number was estimated to be $O(10^{17})$, meaning that the stiffness matrix was numerically singular. Model stability analysis was executed and the smallest eigenvalues of the matrix are $\lambda_1 = 0.0$, $\lambda_2 = 1.142 \times 10^{-13}$, and $\lambda_3 = 1.396 \times 10^5$.

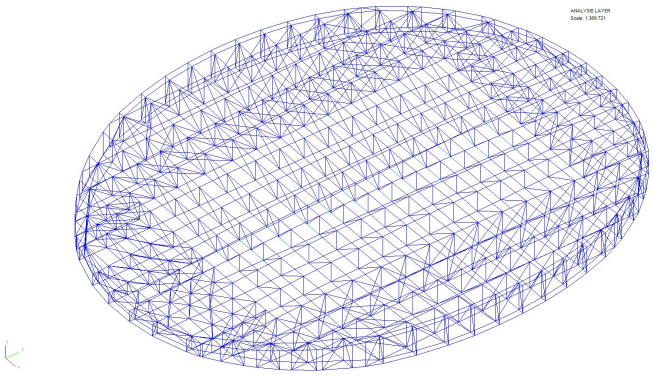
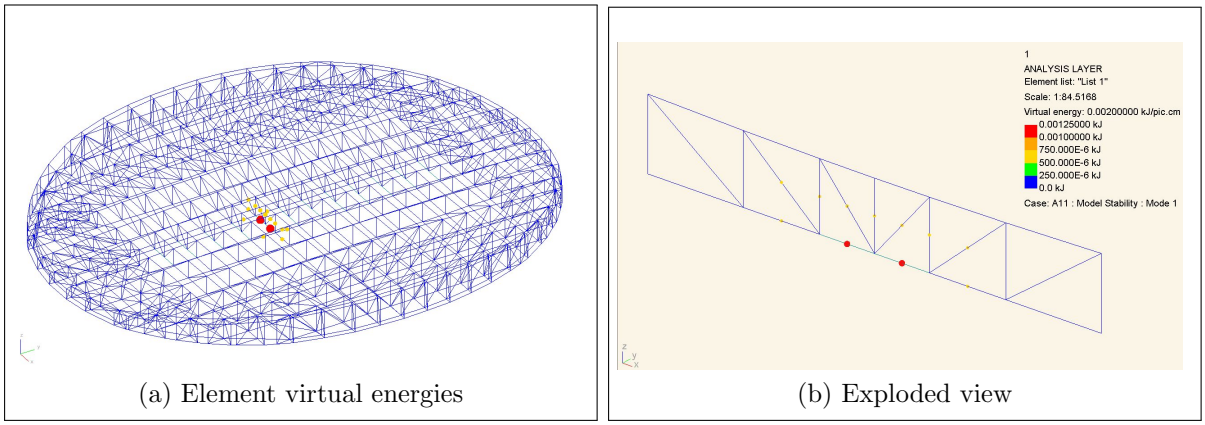


Figure 3.3: 3D view of the arena roof-truss model. The lines in blue represent truss elements.



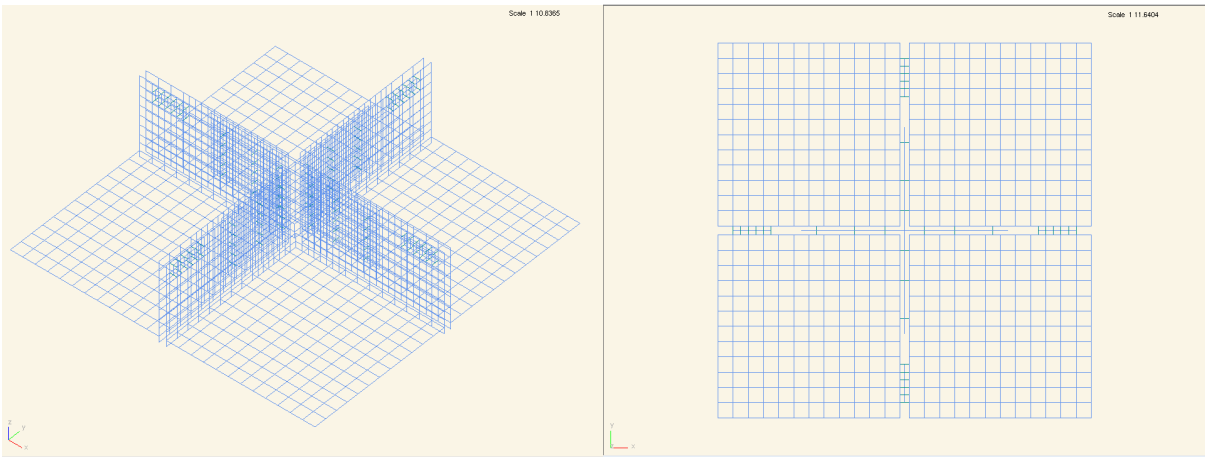
(a) Element virtual energies

(b) Exploded view

Figure 3.4: Roof truss model: virtual energies for first eigenvector.

Figure 3.4a shows the contour plot for element-wise virtual kinetic energies associated with the first eigenvector. The contour shows the largest values for exactly 6 amongst 850 elements in the model.

Figure 3.4b shows a closer view of three of those nodes along with the elements connected to them. Elements 589 and 590 are beams, whereas all other elements connecting at node 221 are bars, which is a modelling oversight. As a result, the dof corresponding to rotation about the X-axis at node 200 is unrestrained, which leads to a mechanism at the node. Restraining rotation about the X-axis at nodes 220 and 222 fixes the beams 589 and 590 from spinning about their axis. After the restraints are applied, the condition number estimate reported drops to $O(10^5)$.



(a) 3d view (b) Plan

Figure 3.5: Connection detail. Blue quadrilaterals are plate elements.

3.5.2 Steel Connection Detail

The model in Figure 3.5 is a steel connection detail. The flange and the web of the connection are modelled using parabolic (eight noded, quadrilateral) plate elements [59, sec. 12.11.7]. Adjacent webs are connected with bolts, modelled as beams.

Initially, the stiffness matrix for the static analysis had a condition number estimate of $O(10^{20})$. Model stability analysis returned a distribution of eigenvalues tabulated in Table 3.2. We notice a cluster of eigenvalues of order 10^{-11} and a large gap between the 40th and 41st eigenvalues. When element virtual kinetic energies ($v^{(e)}$ in (3.3)) associated with the first 40 eigenvectors are contoured on the model, a handful of elements are seen to have large relative values in each mode. Figures 3.6 and 3.7 illustrate four of these plots.

The matrix is numerically singular and it arises from the choice of the element type for modelling the web and their connection to the beams. Eight noded plate elements have only five dofs per node, three for displacements and two for bending: rotation about the normal axis (drilling dof) is not supported. When such an element is connected to a beam, which has six dofs, oriented perpendicular to it, the connecting node acquires an active dof in the rotational direction about its axis. Since the plate does not have stiffness in this direction, the dof is unconstrained and this renders the matrix singular. The 41st eigenvalue is not small compared with the first 40, so the element virtual energies for a corresponding eigenvector are more evenly distributed (mode 41, Figure 3.7).

Table 3.2: Eigenvalues of the stiffness matrix from the steel connection model.

λ_1	λ_2	...	λ_6	λ_7	λ_8	...
2.80E-12	2.30E-12		1.70E-11	5.40E-11	5.50E-11	
...	λ_{23}	...	λ_{40}	λ_{41}		
	7.28E-11		7.70E-11	1.99E+3		

Table 3.3: Eigenvalues of the stiffness matrix from the façade model.

λ_1	λ_2	λ_3	λ_4	λ_5	λ_6	λ_7	λ_8
1.001	1.001	1.001	1.001	1.001	1.001	171.788	271.649

Restraining the beams against rotation about their X-axis brings the condition number estimate of the stiffness matrix down to $O(10^8)$.

3.5.3 Façade Panels for a Building

Whereas the previous examples involved singular stiffness matrices, this example deals with an ill-conditioned matrix arising from erroneous nodal connectivity. Figure 3.8 shows a portion of a larger model of façade cladding of a structure that consists of about 32000 elements and 21000 nodes resting on pinned supports. Our screenshot reproduces only a small part. The glass façade panels are modelled using four noded plane-stress elements, supported on a grid of beam elements. Each panel rests on the grid of beams through springs at each of its four corners, as shown in the zoomed in view in Figure 3.9.

An early version of the model triggered a condition number warning upon linear static analysis. The estimated condition number was $O(10^{12})$. Model stability analysis reported eigenvalues tabulated in Table 3.3. We notice a cluster at 1.001 with six eigenvalues and a gap of $O(100)$ between the sixth and seventh eigenvalues. Correspondingly, the contours of element virtual energies show isolated large values at very few elements for the first six eigenpairs, whereas for the seventh eigenpair the energies are more evenly spread, as is evident from Figure 3.10.

On close examination of the elements, we discover an error in the nodal connectivity. Figure 3.11 shows the part of the model where nodes have large rotations. A zoomed-in view of nodal connectivity in Figure 3.11b reveals that the element is connected to a nearby node rather than to the node that connects the spring. The same error is repeated on the other node along the same edge of the element. As a result, the element is supported on only two corners on springs, which makes this part of the model highly

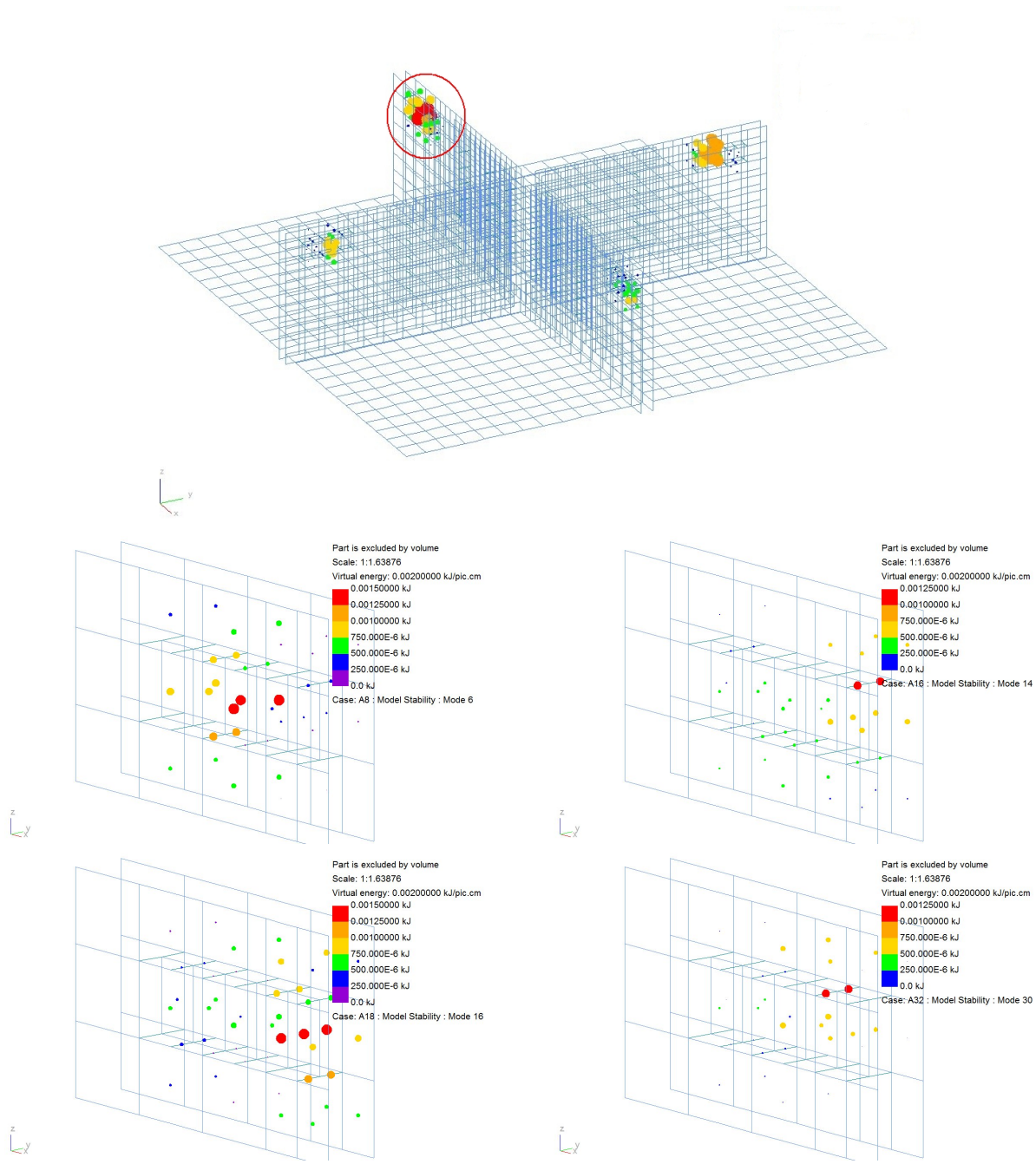


Figure 3.6: Contour plots for virtual energies for the connection model. Clockwise from top: energies associated with eigenvector 1, eigenvector 14, eigenvector 30, eigenvector 16 and eigenvector 6. Figures for eigenpairs 6, 14, 16 and 30 are magnified views of the encircled part in eigenvector 1.

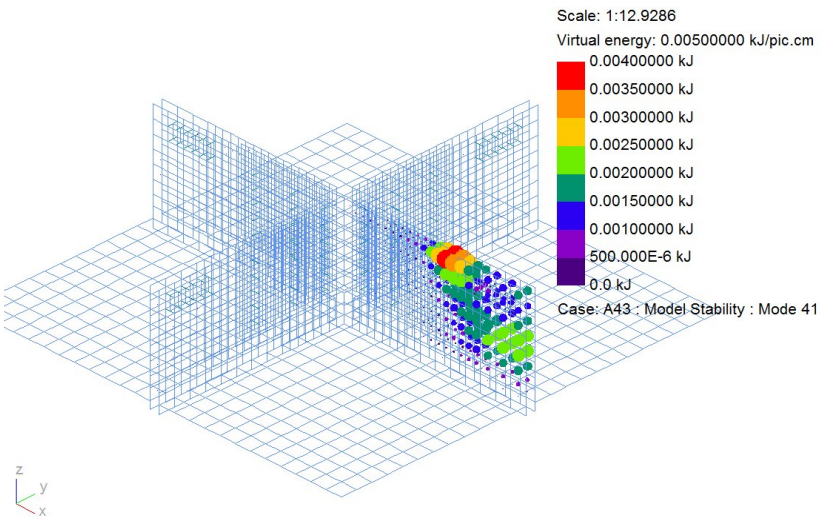


Figure 3.7: Contour plots for virtual energies for eigenvector corresponding to the 41st smallest eigenvalue for the connection model.

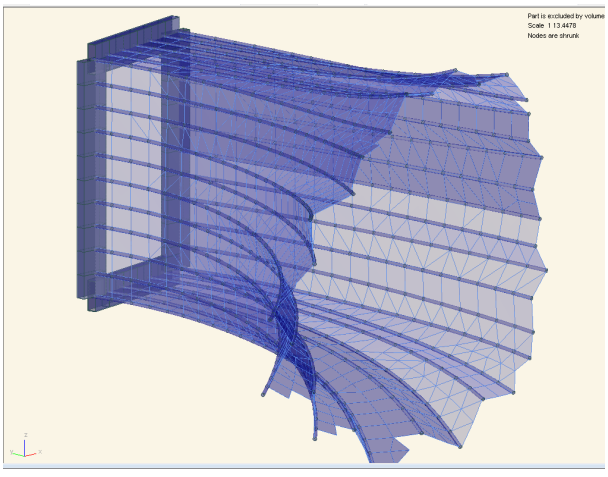


Figure 3.8: A small portion of the façade model.

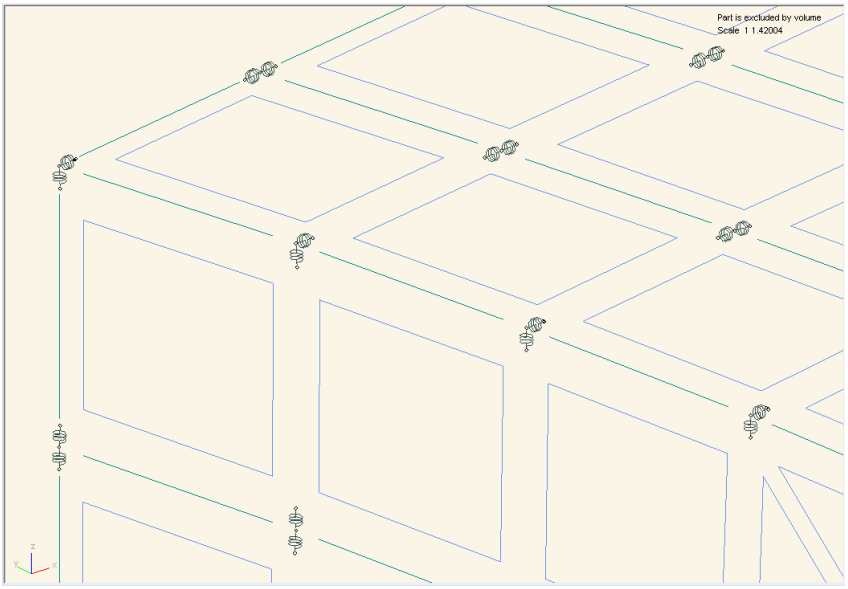


Figure 3.9: Close-up view of the element connectivity for façade model. Blue quadrilaterals are elements modelling the façade panels, green lines represent beams and springs are drawn as coils. Gaps between the elements are a graphic view setting and are only for visual clarity.

Table 3.4: Eigenvalues of stiffness matrix (of size n) from the Tall Building model.

λ_1	λ_2	...	λ_{25}	...	λ_{49}	λ_{50}	...	λ_{n-1}	λ_n
1.22E3	1.70E3	...	9.73E3	...	1.24E4	1.25E4	...	2.02E17	2.02E17

flexible compared to surrounding parts, but not a mechanism.

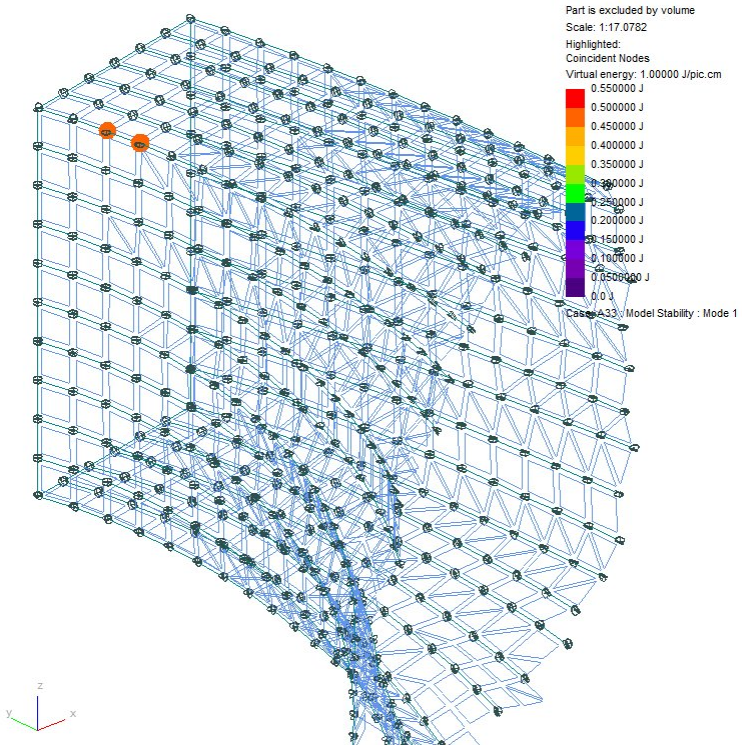
Connecting the plane element to the two springs at the corners reduces the estimated condition number of the model to $O(10^8)$.

3.5.4 Tall Building with Concrete Core

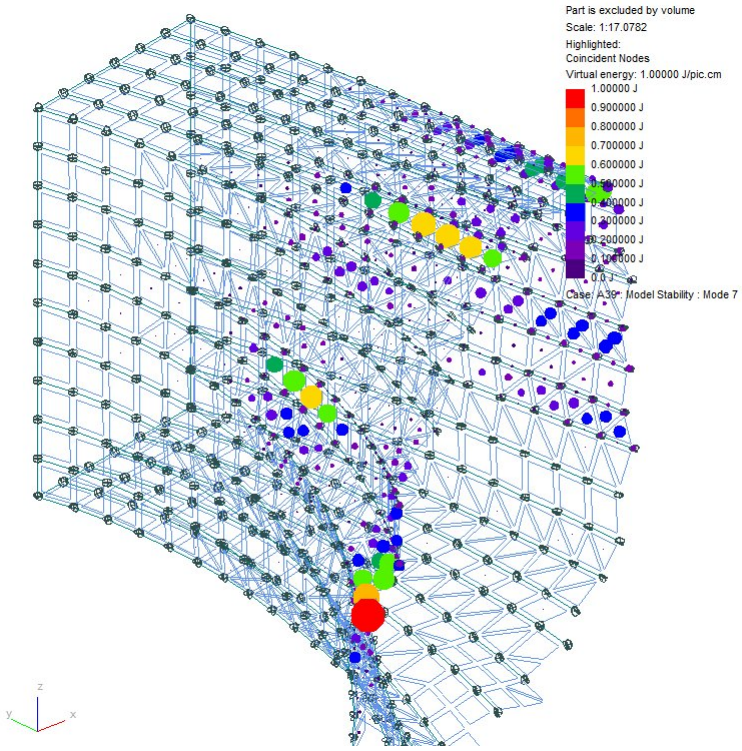
In this example, the ill conditioning was from the presence of elements that were very stiff in comparison with other elements. The GSA model in Figure 3.12 is of a tall building with a concrete core and has about 46000 elements using over a 1000 beam and slab sections with different properties. The model reported an initial condition number of order 10^{15}.

Model stability analysis for up to the 50 smallest eigenvalues did not show a gap in the spectrum; a few of these are listed in Table 3.4. We therefore examine element virtual strain energies for the largest modes.

These highlight a handful of elements, as shown in Figure 3.13. An examination of the

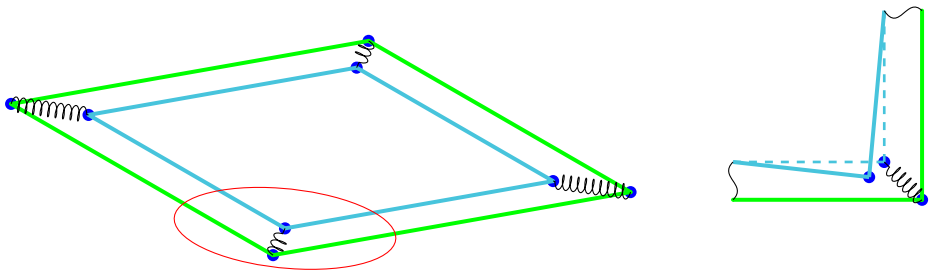


(a) Eigenpair 1



(b) Eigenpair 7

Figure 3.10: Element virtual energies for the first and seventh eigenpairs from the façade model.



(a) A closer view of element 13828

(b) Corner node connectivity

Figure 3.11: Close-up view of the element with large energy in Figure 3.10a.

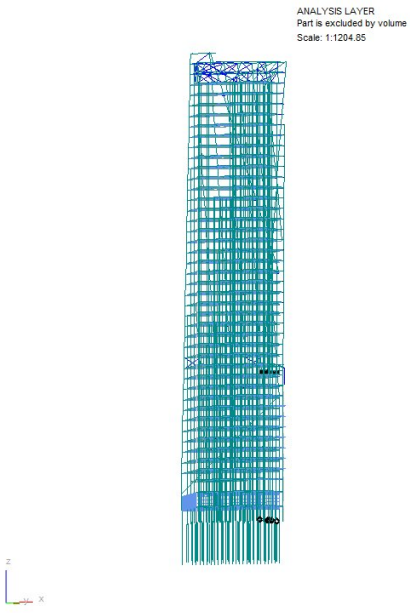


Figure 3.12: Tall building with stiff core.

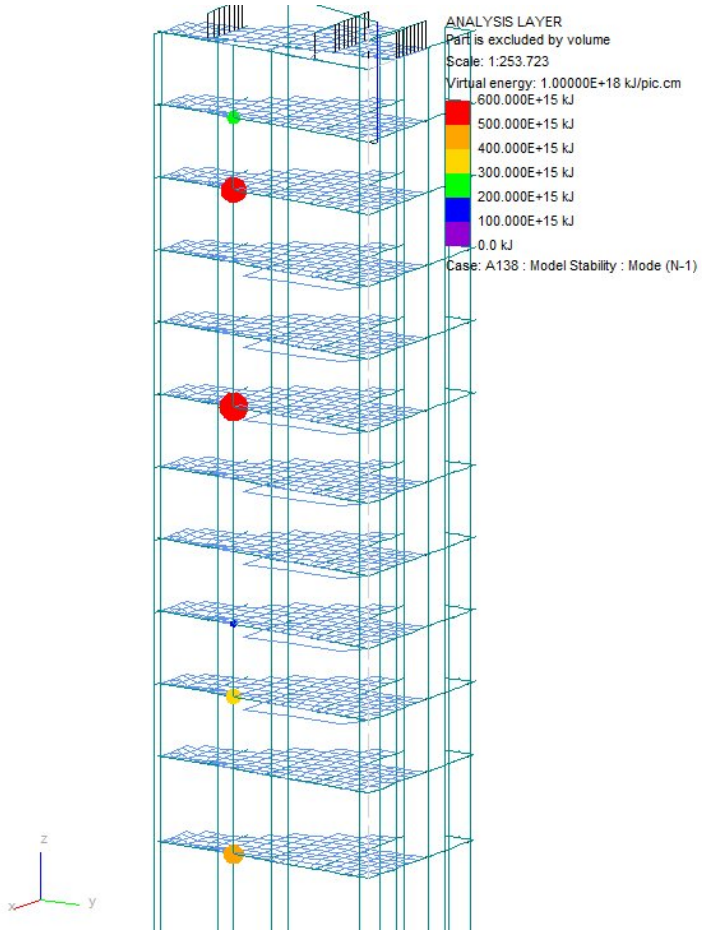


Figure 3.13: Contour of element virtual energies for the largest eigenvector of the stiffness matrix of tall building model.

section properties of the elements reveals that certain members are modelled as strings of short beam elements (Figure 3.14), resulting in these elements acquiring large stiffness values. Rectifying the model in this case involves replacing the string of elements by single beam elements wherever they occur, which results in the condition number decreasing by several orders of magnitude.

Unlike in the case of smallest eigenvectors, the number of largest eigenvectors to be examined is arbitrary. This is because identification of elements with "large" stiffness is relative to the stiffness of the other elements in the model. Developing a criterion for selecting the number of largest eigenpairs is left to future work. At present we recommend iterating by correcting anomalies, re-analysing the model to find the new condition number and repeating the process until the condition number falls below the threshold τ used in the method in section 3.4.

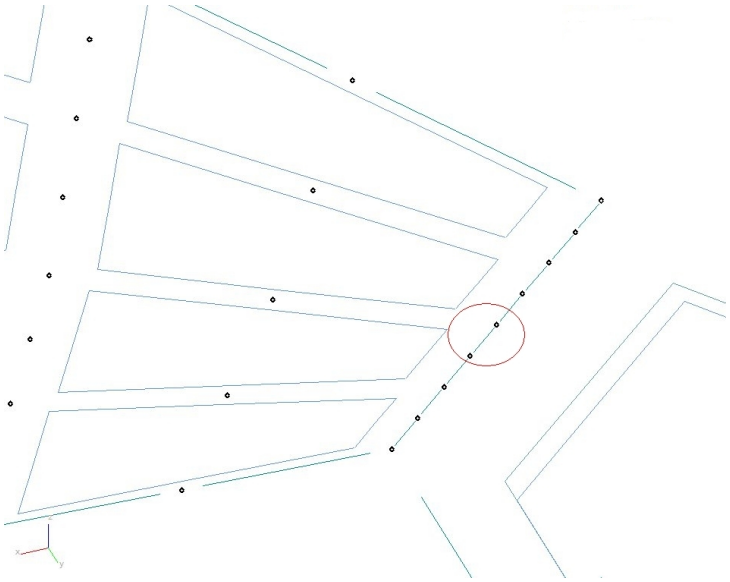


Figure 3.14: Magnified view of the element with high energy. The encircled beam has large stiffness.

3.6 Conclusion

We have demonstrated a method for detecting a class of ill-conditioned problems in structural analysis. These problems arise when there are dofs that contribute stiffnesses that are disproportional in magnitude and when there are tuples of dofs that have insufficient connectivity with the rest of the FE assembly, both of which are scenarios that commonly arise as user errors during model generation. We exploit the sparsity structure of the smallest and largest eigenpairs and use element virtual energies to highlight elements that cause the ill conditioning. This method has been implemented in a commercial FE analysis package and we have shown through examples of real-life models that it works in practice. Further work will focus on developing a criteria for determining the number of largest eigenpairs used for investigating ill conditioning from large element stiffnesses.

Chapter 4

The Subspace Iteration method for the Symmetric Definite GEP and Symmetric SEP

4.1 Introduction

This chapter presents the eigenvalue problems that GSA solves, and develops algorithms and the corresponding software implementations. These problems are as follows.

- **The Symmetric-Definite Generalized Eigenvalue Problem**

  The equilibrium equation governing the linear dynamic response of a system of finite elements can be written in the form $M\ddot{u} + Ku = 0$, [7, sec. 4.2], where

  - $M \in \mathbb{R}^{n\times n}$ represents the mass matrix of the structure
  - $K \in \mathbb{R}^{n\times n}$ represents the stiffness matrix of the structure; and
  - $u \in \mathbb{R}^n$ is the vector of displacements of degrees of freedom.

  Any solution of this equation is of the form $u = e^{iw}x$, $x \in \mathbb{C}^n$, which on substitution yields the symmetric-definite generalized eigenvalue problem (GEP)

  $$Kx = \lambda Mx, \qquad \lambda = \omega^2.$$

  The matrices K and M are both symmetric, sparse and typically large in size. Further, K is positive definite and M is semidefinite. The eigenvalues of interest

are the ones closest to 0 since the most critical modes of vibration are the ones with the lowest frequency. Since K is positive definite, all eigenvalues of our system are real and since M is semidefinite, they are all positive. Furthermore, we assume that eigenvalues of interest are finite.

- **The Symmetric Eigenvalue Problem**

  In chapter 5 we developed the Model Stability analysis method that uses the extremal eigenpairs of an ill conditioned stiffness matrix to detect the part of the structural model that causes ill conditioning. We compute a few smallest and largest eigenvalues and the corresponding eigenvectors of the positive definite (or semidefinite) stiffness matrix K, i.e., we solve the symmetric eigenvalue problem (SEP)

  $$Ku = \lambda u$$

  for a few eigenvalues with the smallest and largest absolute values.

4.2 Solution

For notational simplicity, we use A and B for the stiffness and mass matrices instead of K and M in the rest of this chapter. Owing to the potentially very large dimensions of A and B, it is not possible to find all the eigenvalues of the system since most *direct* [1] methods (i.e. methods based on similarity transformations) such as the QR algorithm and its variants, operate on the entire matrix and have $O(n^3)$ cost. Further, the eigenvectors of a sparse matrix are not necessarily sparse and hence it is expensive to hold all eigenvectors in memory. Therefore we resort to using methods that compute specific parts of the spectrum, for example, eigenpairs at the ends or in the interior. These *iterative* methods also take advantage of the sparse matrix storage.

A common technique is to transform the problems $Ax = \lambda Bx$ or $Ax = \lambda x$ into a reduced system of dimension m ($m < n$) using projections on invariant subspaces. The

[1] Direct methods must still iterate, since the problem of finding the eigenvalues is mathematically equivalent to finding the zeros of a polynomial for which no noniterative methods exist when the polynomial is of degree greater than 4. We can call a method *direct* if experience shows that it almost always converges in a fixed number of iterations.

eigenvectors of the smaller system can then be projected back to give an approximation to the eigenvectors of the original system.

The subspace iteration method, also known as the simultaneous iteration method in some literature, (see, for example, [61], [5]) is one such technique that is simple to implement and works robustly. By robust, we mean that the ability of the method to find the eigenvalues of interest does not depend on the conditioning of the problem.

4.3 Subspace Iteration

Subspace iteration (SI) is a block generalization of the power method applied simultaneously on many vectors, which converge to approximations to the eigenvectors of the matrix. For presentational simplicity, we consider the standard eigenvalue problem for $A \in \mathbb{R}^{n\times n}$. Beginning with an $n \times m$ matrix S_0, we compute the powers $A^k S_0$. It turns out that the powered subspace $\mathcal{R}(A^k S_0)$ contains an approximation to the dominant eigenvectors of A. However, the vectors of $A^k S_0$ will need to be orthogonalized otherwise they will tend towards linear dependence. Thus, in its simplest form the SI method to compute an invariant subspace of A is as follows:

1. Choose $S_0 = [s_1, \ldots, s_m]$ of full rank.

2. *Iteration:* for $k = 1, 2, \ldots$ till convergence,

   (a) compute $S_{k+1} = AS_k$,

   (b) form the QR factorization $S_{k+1} = QR$ and set $S_{k+1} = Q$.

Then, assuming the eigenvalues of A are ordered as

$$|\lambda_1| \geq \cdots \geq |\lambda_m| > |\lambda_{m+1}| \geq \cdots \geq |\lambda_n|,$$

with a gap $|\lambda_m| > |\lambda_{m+1}|$, the columns of S_k converge to a basis for the invariant subspace of A corresponding to the m dominant eigenvalues. Convergence is linear, with a convergence rate of $|\lambda_{m+1}/\lambda_m|$.

4.3.1 Proof of Convergence

We now establish the proof of convergence for SI. The following results will help us in constructing the proof. We first define the notion of *distance* between subspaces and state a result that provides a measure of this distance. Throughout the proof, we shall use the 2-norm.

Angle between subspaces Let $\mathcal{U}$ and $\mathcal{V}$ be two subspaces of $\mathbb{R}^n$ whose dimensions satisfy

$$p = \dim(\mathcal{U}) \geq \dim(\mathcal{V}) = q \geq 1.$$

The *principal angles* $\theta_1, \cdots, \theta_q \in [0, \pi/2]$ between $\mathcal{U}$ and $\mathcal{V}$ are defined recursively by

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

subject to $\|u\| = \|v\| = 1$ with $u^T u_i = 0,\ v^T v_i = 0$ for $i = 1 : k-1$.

Distance between Subspaces Let $\mathcal{U}$ and $\mathcal{V}$ be two m-dimensional subspaces of $\mathbb{R}^n$. We define the distance between $\mathcal{U}$ and $\mathcal{V}$ as

$$d(\mathcal{U}, \mathcal{V}) := \max_{\substack{u \in \mathcal{U} \\ \|u\|=1}} d(u, \mathcal{V}) = \max_{\substack{u \in \mathcal{U} \\ \|u\|=1}} \min_{v \in \mathcal{V}} \|u - v\|. \tag{4.1}$$

Theorem 4.3.1. *([69, Thm. 2.6.9]) Let $\mathcal{U}$ and $\mathcal{V}$ be m-dimensional subspaces and let U, $U^\perp$, V, $V^\perp$ be matrices with orthonormal columns such that $\mathcal{U} = \mathcal{R}(U)$, $\mathcal{U}^\perp = \mathcal{R}(U^\perp)$, $\mathcal{V} = \mathcal{R}(V)$, $\mathcal{V}^\perp = \mathcal{R}(V^\perp)$, where $\mathcal{R}(A)$ denotes the range of $A \in \mathbb{R}^{n \times m}$ given by $\mathcal{R}(A) = \{Ax \mid x \in \mathbb{R}^m\}$. Define θ_m as the largest principal angle between $\mathcal{U}$ and $\mathcal{V}$. Then*

$$d(\mathcal{U}, \mathcal{V}) = \|(U^\perp)^T V\| = \|U^T V^\perp\| = \|(V^\perp)^T U\| = \|V^T U^\perp\| = \sin\theta_m.$$

The following intermediate lemma provides a relationship for the distance between a subspace whose basis has a specific structure and the subspace spanned by unit vectors.

Lemma 4.3.2. *([69, Proposition 2.6.16]) Let $\hat{\mathcal{S}}$ be an m-dimensional subspace of $\mathbb{R}^n$, and suppose $\hat{\mathcal{S}} = \mathcal{R}(\hat{S})$, where $\hat{S} = \begin{bmatrix} I \\ X \end{bmatrix}$. Then*

$$\|X\| = \tan\theta_m,$$

where θ_m is the largest principal angle between $\mathcal{E}_m = \mathcal{R}(e_1, \cdots, e_m)$ (e_i being unit vectors of the appropriate dimension) and $\hat{\mathcal{S}}$.

Since it is often convenient to work with subspaces in terms of bases, the following results are useful when transforming a given subspace from one coordinate system to another.

Lemma 4.3.3. *Let $S \in \mathbb{R}^{n\times n}$ be nonsingular and let $\tilde{\mathcal{U}} = S^{-1}\mathcal{U}$ and $\tilde{\mathcal{V}} = S^{-1}\mathcal{V}$. Then $d(\mathcal{U},\mathcal{V}) \le \kappa(S)d(\tilde{\mathcal{U}},\tilde{\mathcal{V}})$.*

Proof. Pick $u \in \mathcal{U}$ such that $\|u\| = 1$ and the maximum is attained in (4.1), i.e., $d(\mathcal{U},\mathcal{V}) = d(u,\mathcal{V})$. If $d(u,\mathcal{V}) = \min_{v\in\mathcal{V}}\|u-v\|$ is attained at v', then $\|u-v'\| \le \|u-v\|$ for all $v \in \mathcal{V}$. So we can write

$$d(\mathcal{U},\mathcal{V}) \le \|u-v\|. \tag{4.2}$$

Then, let $\hat{u} = S^{-1}u \in \tilde{\mathcal{U}}$, $\alpha = \|\hat{u}\| > 0$, and $\tilde{u} = \alpha^{-1}\hat{u} \in \tilde{\mathcal{U}}$. Since $\alpha = \|S^{-1}u\| \le \|S^{-1}\|\|u\| = \|S^{-1}\|$, we have $\alpha \le \|S^{-1}\|$. Similarly, let $\hat{v} = S^{-1}v \in \mathcal{V}$ and $\hat{u} = \alpha^{-1}\hat{v}$. Substituting for u and v in $\|u-v\|$, we get

$$\begin{aligned}
\|u-v\| &= \|(\alpha S\tilde{u} - \alpha S\tilde{v})\| \\
&= \alpha\|S(\tilde{u}-\tilde{v})\| \\
&\le \alpha\|S\|\|\tilde{u}-\tilde{v}\| \\
&\le \|S^{-1}\|\|S\|\|\tilde{u}-\tilde{v}\| \\
&\le \kappa(S)\|\tilde{u}-\tilde{v}\|,
\end{aligned} \tag{4.3}$$

where $\kappa(S) = \|S^{-1}\|\|S\|$.

Now, pick $\tilde{v} \in \mathcal{V}$ such that $\|\tilde{u}-\tilde{v}\| = d(\tilde{u},\tilde{\mathcal{V}})$. Since $d(\tilde{\mathcal{U}},\tilde{\mathcal{V}}) = \max_{\tilde{w}\in\tilde{\mathcal{U}}} d(\tilde{w},\tilde{\mathcal{V}}) \ge \|\tilde{u}-\tilde{v}\|$, we have

$$\|\tilde{u}-\tilde{v}\| \le d(\tilde{\mathcal{U}},\tilde{\mathcal{V}}). \tag{4.4}$$

Combining (4.2), (4.3) and (4.4), we get $d(\mathcal{U},\mathcal{V}) \le \kappa(S)\|\tilde{u}-\tilde{v}\| \le \kappa(S)d(\tilde{\mathcal{U}},\tilde{\mathcal{V}})$. □

Lemma 4.3.4. *Let $\mathcal{U} = \mathcal{R}(U)$ and $\mathcal{V} = \mathcal{R}(V)$ be m-dimensional subspaces of $\mathbb{R}^n$ such that $\mathcal{U}\cap\mathcal{V} = \{0\}$. Let $A \in \mathbb{R}^{n\times n}$ be a nonsingular matrix and let $\tilde{\mathcal{U}} = A\mathcal{U}$, $\tilde{\mathcal{V}} = A\mathcal{V}$. Then,*

$$\tilde{\mathcal{U}}\cap\tilde{\mathcal{V}} = \{0\}. \tag{4.5}$$

Proof. The proof is by contradiction. Let there be an n-dimensional nonzero vector $x \in \tilde{\mathcal{U}}\cap\tilde{\mathcal{V}}$. Since $\tilde{\mathcal{U}} = \mathcal{R}(AU)$, there exists $u \in \mathbb{R}^m$ s.t. $x = AUu$ and similarly there

exists $v \in \mathbb{R}^m$ s.t. $x = AVv$. This implies $AUu = x = AVv$, which is impossible since $Uu \neq Vv$ for any $u, v \neq 0$. Hence $\tilde{\mathcal{U}} \cap \tilde{\mathcal{V}} = \{0\}$.

□

Theorem 4.3.5 (Convergence of subspace iteration)**.** *Let $A \in \mathbb{R}^{n\times n}$ be nonsingular and let p be a polynomial of degree $< n$. Let $\lambda_1, \ldots, \lambda_n$ denote the eigenvalues of A, ordered such that $|p(\lambda_1)| \geq |p(\lambda_2)| \geq \cdots \geq |p(\lambda_n)|$. Suppose m is an integer satisfying $1 \leq m < n$ for which $|p(\lambda_m)| > |p(\lambda_{m+1})|$ and let $\rho = |\frac{p(\lambda_{m+1})}{p(\lambda_m)}| < 1$. Let $\mathcal{U}$ and $\mathcal{V}$ be invariant subspaces of A associated with $\lambda_1, \ldots, \lambda_m$ and $\lambda_{m+1}, \ldots, \lambda_n$, respectively. Consider the subspace iteration*

$$\mathcal{S}_i = p(A)\mathcal{S}_{i-1}, \qquad i = 1, 2, 3, \ldots,$$

where $\mathcal{S}_0 = \mathcal{S}$ is an m-dimensional subspace satisfying $\mathcal{S} \cap \mathcal{V} = \{0\}$. Then there is a constant C such that

$$d(\mathcal{S}_i, \mathcal{U}) \leq C\rho^i, \qquad i = 1, 2, 3, \ldots.$$

Proof. Let J denote the Jordan form of A [69, Thm. 2.4.11]. Assume that the Jordan blocks are ordered in such a way that $J = \operatorname{diag}(J_1, J_2)$, where J_1 is $m \times m$ and contains the Jordan blocks corresponding to eigenvalues $\lambda_1, \ldots, \lambda_m$ and J_2 is $(n-m) \times (n-m)$ and contains blocks corresponding to $\lambda_{m+1}, \ldots, \lambda_n$.

We have $A = VJV^{-1}$ for some nonsingular matrix V. Partition $V = [V_1 \quad V_2]$ such that $V_1 \in \mathbb{R}^{n\times m}$. Then the equation $AV = VJ$ implies $AV_1 = V_1J_1$ and $AV_2 = V_2J_2$. $\mathcal{R}(V_1)$ and $\mathcal{R}(V_2)$ are therefore invariant subspaces of A associated with $\lambda_1, \ldots, \lambda_m$ and $\lambda_{m+1}, \ldots, \lambda_n$ respectively.

Now we make a change of coordinate system. For each i let $\tilde{\mathcal{S}}_i = V^{-1}\mathcal{S}_i$, $\tilde{\mathcal{U}} = V^{-1}\mathcal{U}$ and $\tilde{\mathcal{V}} = V^{-1}\mathcal{V}$. This yields $\tilde{\mathcal{U}} = V^{-1}\mathcal{R}(V_1) = \operatorname{span}\{e_1, \ldots, e_m\} = \mathcal{E}_m$ and $\tilde{\mathcal{V}} = \operatorname{span}\{e_{m+1}, \ldots, e_n\} = \mathcal{E}_m^{\perp}$. The subspace iteration

$$p(A)\mathcal{S}_{i-1} = \mathcal{S}_i$$

is equivalent to

$$p(J)\tilde{\mathcal{S}}_{i-1} = \tilde{\mathcal{S}}_i.$$

Let $\hat{p}_i(x) = p(x)^i$. Then,

$$\tilde{\mathcal{S}}_i = \hat{p}_i(J)\tilde{\mathcal{S}}_0. \tag{4.6}$$

Let $S = \left[\begin{smallmatrix} S_1 \\ S_2 \end{smallmatrix}\right] \in \mathbb{R}^{n\times m}$ be the full rank matrix representing the subspace vectors in $\tilde{\mathcal{S}}$, (i.e., $\mathcal{R}(S) = \tilde{\mathcal{S}} = \tilde{\mathcal{S}}_0$), partitioned such that S_1 is $m \times m$. Then S_1 is nonsingular. (To prove this, assume that S_1 is singular. Let $y \in \mathbb{R}^m \neq 0$ be such that $S_1 y = 0$. Then, $Sy = \left[\begin{smallmatrix} S_1 \\ S_2 \end{smallmatrix}\right] y = \left[\begin{smallmatrix} 0 \\ S_{2y} \end{smallmatrix}\right] \in \tilde{\mathcal{S}}$. But since $\left[\begin{smallmatrix} 0 \\ S_{2y} \end{smallmatrix}\right] \in \text{span}\{e_{m+1}, \ldots, e_n\}$, $Sy \in \tilde{\mathcal{V}}$. This implies $\tilde{\mathcal{S}} \cap \tilde{\mathcal{V}} \neq \{0\}$, which would mean $\mathcal{S} \cap \mathcal{V} \neq \{0\}$ by Lemma (4.3.4), which is a contradiction.)

Letting $X = S_2 S_1^{-1}$, we have

$$S = \begin{bmatrix} I \\ X \end{bmatrix} S_1,$$

so $\left[\begin{smallmatrix} I \\ X \end{smallmatrix}\right]$ represents $\tilde{\mathcal{S}}$ as well, in the sense that its columns span $\tilde{\mathcal{S}}$. Since we wish to analyse iteration (4.6), we note that

$$\hat{p}_i(J) \begin{bmatrix} I \\ X \end{bmatrix} = \begin{bmatrix} \hat{p}_i(J_1) \\ \hat{p}_i(J_2)X \end{bmatrix} = \begin{bmatrix} I \\ \hat{p}_i(J_2)X\hat{p}_i(J_1)^{-1} \end{bmatrix} \hat{p}_i(J_1), \tag{4.7}$$

assuming that $\hat{p_1}(J_1)$ is nonsingular. This assumption is valid as long as none of the eigenvalues of A are 0. Let

$$S_i = \begin{bmatrix} I \\ \hat{p}_i(J_2)X\hat{p}_i(J_1)^{-1} \end{bmatrix}.$$

Then, by (4.6) and (4.7), $\tilde{\mathcal{S}} = \mathcal{R}(S_i)$. Thus by Lemma (4.3.2),

$$\|\hat{p}_i(J_2)X\hat{p}_i(J_1)^{-1}\| = \tan\theta_m^{(i)},$$

where $\theta_m^{(i)}$ is the largest principal angle between $\tilde{\mathcal{S}}_i$ and $\tilde{\mathcal{U}}$. Consequently

$$d(\tilde{\mathcal{S}}, \tilde{\mathcal{U}}) = \sin\theta_m^{(i)} \leq \tan\theta_m^{(i)} \leq \|X\|\|\hat{p}_i(J_2)\|\|\hat{p}_i(J_1)^{-1}\|.$$

Using the result from Lemma (4.3.4), we have

$$d(\mathcal{S}_i, \mathcal{U}) \leq \kappa(V)\|X\|\|\hat{p}_i(J_2)\|\|\hat{p}_i(J_1)^{-1}\|. \tag{4.8}$$

The bound obtained above can be used to show convergence of the subspace iteration both when A is semisimple or non-semisimple. We show the convergence for the semisimple case. The convergence of non-semisimple A has been shown in [69, chap. 5].

For a semisimple matrix A, the Jordan blocks J_1 and J_2 are diagonal, i.e., $\hat{p}_i(J_2) = \text{diag}\{p(\lambda_{m+1})^i, \ldots, p(\lambda_n)^i\}$ and $\|\hat{p}_i(J_2)\| = |p(\lambda_{m+1})|^i$. Similarly, $\|\hat{p}_i(J_1)^{-1}\| = |p(\lambda_m)|^{-i}$.

Substituting the value of norms into (4.8), we get

$$d(\mathcal{S}_i, \mathcal{U}) \leq \kappa(V)\|X\| \frac{|p(\lambda_{m+1})|^i}{|p(\lambda_m)|^i} = C\rho^i.$$

□

Thus subspace iteration on A with m vectors, as shown above, converges with the rate $|p(\lambda_{m+1})/p(\lambda_m)|$, (which is proportional to $|\lambda_{m+1}/\lambda_m|^r$, r being the degree of p), to the dominant eigenvectors of A. In [62, chap. 6], Stewart shows informally that if the convergence of m eigenpairs is sought, iterating with $\ell\,(> m)$ subspace vectors results in a convergence rate

$$|\lambda_{\ell+1}/\lambda_m|. \tag{4.9}$$

This is potentially faster than iterating with m vectors and also overcomes the possibility of $|\lambda_m|$ being very close to $|\lambda_{m+1}|$, which can slow convergence.

4.4 Existing Software Implementations of SI

Much of the current state-of-the-art in subspace iteration methods is aimed at the standard eigenvalue problem and has been comprehensively summarized by Stewart in [62, Chap. 6] (we note that the reference is not recent but the material covered in it stays current). The use of subspace iteration originated in Bauer's *Treppperniteration* [10] in 1955. Early implementations of the algorithm for the standard eigenproblem were published by Rutishauser [53] and by Stewart [60]. More recently, Duff and Scott [16] published the algorithm `EB12` (part of the Harwell Subroutine Library [63]) to compute selected eigenvalues and eigenvectors of a standard eigenproblem $Ax = \lambda x$ for sparse unsymmetric matrices. The program uses novel iteration controls and stopping criteria and allows the user to provide a function to form the sparse matrix-vector product $A \times x$ instead of supplying an explicitly stored A. Chebyshev polynomials are employed for accelerating convergence. Bai and Stewart published `SRRIT` [3], [4], initially in 1978 and then a revised version in 1997. The algorithm is similar to those of `EB12` but `SRRIT` focusses on being modifiable by allowing the user access to control parameters and by being more modular. Even though the algorithms were developed for standard eigenproblems, the authors state they can be used for the generalized case $Ax = \lambda Bx$ by solving $Ax = By$

for subspace vectors x, in effect forming $A^{-1}B$. However such transformations do not preserve properties like symmetry.

Within the engineering literature, Bathe pioneered a variant of subspace iteration for the generalized eigenvalue problem, published as a series of algorithms in the 70's and early 80's [9], [7], [8], and subsequently revisited and improved upon by other engineers (for example, [74], [39]). A key deficiency common to most of these works is the lack of orthogonalization of subspace vectors, without which they tend towards linear dependence after a few iterations. They also rely on heuristics for details such as shifting strategies, which raises questions about the robustness of these implementations to a wide range of problems. Despite these misgivings, SI remains a very popular algorithm within the engineering FE community and this is the primary reason for its existing implementation in GSA.

4.5 Implementation for Symmetric Definite GEP

In this section we describe our implementation of SI for solving the symmetric definite GEP in GSA.

4.5.1 Existing GSA Implementation

Prior to the work in this chapter, GSA implemented the algorithm `SSPACE`, as described in [7, chap. 9]. This is listed in Algorithm 4.2.

The algorithm projects the matrices A and B onto the subspace $\mathcal{S}$ using S_i to obtain matrices M and N respectively. It then solves for the eigenpairs (Q, Λ) of the projected system. The call to the Jacobi solver invokes the generalized Jacobi method as implemented in 'Subroutine Jacobi' from [7, p. 924]. The generalized Jacobi method solves for the eigenvalues Λ and eigenvectors Q of the projected problem $MQ = NQ\Lambda$. The eigenvectors in Q are multiplied by U to obtain a better approximation of the next iteration of subspace vectors.

The algorithm works with $\ell\ (> m)$ subspace vectors to obtain a faster convergence. The value of ℓ is chosen to be $\min(m + 8, 2m)$ [7, p. 960]. The starting subspace vectors are determined as follows:

Algorithm 4.2 Subspace Iteration Method (`SSPACE`)

Given $A \in \mathbb{R}^{n\times n}$ symmetric and positive definite, $B \in \mathbb{R}^{n\times n}$ symmetric, the number of eigenvalues desired m, convergence tolerance tol, the starting subspace vectors $S_1 \in \mathbb{R}^{n\times \ell}$, the number of Jacobi sweeps nJacobi, the tolerance for the Jacobi iterations tolJacobi and maximum number of iterations iter, this algorithm solves the symmetric generalized eigenvalue problem $AS = BS\Lambda$ for m smallest eigenvalues Λ and corresponding eigenvectors S.

```
1  Set p = 0
2  Factorize A = LDL^T
3  for k = 1:iter
4       T = A^{-1} S_k. (triangular system solves)
5       M = T^T A T.
6       N = T^T B T.
7       Call Jacobi(M, N, Λ, Q, nJacobi, tolJacobi) to compute the
8       eigendecomposition MQ = NQΛ for eigenvectors Q and eigenvalues Λ.
9       S = BTQ
10      for i = 1:m (convergence test)
11           if |(λ_i^(k) − λ_i^(k−1))/λ_i^(k)| ≤ tol
12                p = p + 1
13           end.
14      end.
15      if p = m
16           quit
17      end
18 end
```

- The first vector comprises of all diagonal entries from B, i.e.,

$$s_1^T = [b_{11}, b_{22}, \ldots, b_{nn}].$$

- The vectors $s_2, \ldots, s_\ell$ are unit vectors e_i of appropriate dimensions, where $i = r_1 : r_p$ and $r_j,\ j = 1 : p$ correspond to the p smallest values of b_{gg}/a_{gg} for $g \in (1, n)$.

The main issues with the algorithm are:

- There is no orthogonalization of the subspace vectors, hence the final eigenvector approximations are not guaranteed to be orthonormal.

- The convergence criterion checks eigenvalue estimates against the values obtained in the previous iteration and does not check the residuals $r_i = \|As_i - \lambda_i Bs_i\|_2$. Though the program does report the unscaled residuals after the iteration, it does not ensure the resulting vectors approximate the eigenvectors of the pair (A, B).

- Since the algorithm uses a fixed value of ℓ to compute m eigenpairs, convergence can be arbitrarily slow and the user has no opportunity to use any *a priori* knowledge of the spectrum to accelerate the convergence.

4.5.2 Increasing the Efficiency

The main focus of the new algorithm is to improve on the following aspects:

- Increased efficiency of the iteration by shifting the spectrum, delaying orthogonalization and locking converged vectors,

- Increased reliability of the computed solution by using a better convergence test and orthogonalization,

- Use of modern industry-standard software libraries for better efficiency and

- Greater user control of iteration parameters and better feedback on the progress of the iteration and on the quality of the computed results.

The following subsections describe these improvements. We start with a simple implementation for the GEP and make a series of changes to arrive at a final, efficient

algorithm. For all algorithms, the inputs are the matrices $A \in \mathbb{R}^{n\times n}$ (symmetric, positive definite), $B \in \mathbb{R}^{n\times n}$ (symmetric, semidefinite), m (the number of eigenvalues desired), iter (the maximum number of iterations), tol (the convergence tolerance) and $S \in \mathbb{R}^{n\times \ell}$(the starting vectors).

We first describe the convergence test used, since it is common throughout our experiments.

The Convergence Criterion

The convergence test for a given Ritz pair (λ_i, s_i) for our SI algorithms is

$$\frac{\|As_i - \lambda_i Bs_i\|}{\|s_i\|(\|A\| + |\lambda_i|\|B\|)} \leq \text{tol}, \tag{4.10}$$

where tol is a user defined tolerance and the norm is the 1-norm. This expression is the standard normwise backward error for an approximate eigenpair (λ_i, s_i) for the generalized eigenproblem [31] and ensures that we solve the nearby system

$$(A + \Delta A)s_i = \lambda_i(B + \Delta B)s_i,$$

where $\|\Delta A\| \leq \text{tol}\|A\|$ and $\|\Delta B\| \leq \text{tol}\|B\|$.

This is a more reliable test of convergence of the iterative solution than the convergence test of Algorithm 4.2. It gives an upper bound for the backward error in the system we have actually solved for, i.e., if we set *tol* to u, the machine precision, we have solved a problem within the rounding distance of the original problem. The tolerance is supplied as user input with a default value of 10^{-10}.

The Basic Algorithm

We start with a naïve implementation of the SI method in Alg. 4.3.

The powering of subspace vectors by A and B is achieved by multiplying them by B and then solving for $AS = T$. The multiplication with A^{-1} is realized using LDL^T factorization and back substitution. We use the parallel sparse direct solver 'Pardiso' [57], [56] from the Intel Math Kernel Library (MKL) package [37] for this purpose. Pardiso uses 1×1 and 2×2 Bunch and Kaufman [57] pivoting for symmetric indefinite matrices.

As we keep multiplying the vectors in S by $A^{-1}B$, they start aligning themselves along the dominant eigenvector of the system. This can potentially make the matrix S

Algorithm 4.3 Basic subspace iteration with orthonormalization and Schur–Rayleigh–Ritz refinement.

Given symmetric, pos-def $A \in \mathbb{R}^{n\times n}$, symmetric, semidefinite $B \in \mathbb{R}^{n\times n}$, starting vectors $S \in \mathbb{R}^{n\times \ell}$, this algorithm calculates m smallest eigenvalues and associated approximate eigenvectors of the GEP $AS = BS\Lambda$.

1 Determine $\|A\|_1$ and $\|B\|_1$.
2 Compute the factorization $A = LDL^T$.
3 Set $p = 0$.
4 **for** k = 1:iter
5 $T = BS$.
6 $S = A\backslash T$.
7 $S = QR; S = Q$ (orthonormalization using QR factorization).
8 $M = S^T AS; N = S^T BS$.
9 Calculate the eigendecomposition $MV = NV\Lambda$.
10 Reorder Λ and V in ascending order according to absolute values in diag(Λ).
11 $S = SV$.
12 Test convergence for (s_i, λ_{ii}), $i = 1{:}\, m$ using relation (4.10).
13 Set $p =$ number of pairs converged.
14 **if** $p = m$, exit.
15 **end**

ill-conditioned and lead to loss of information in the projection step. To avoid such a situation, we must orthogonalize the vectors to ensure the projection uses an orthogonal basis. This is done by factorizing S into orthonormal Q and upper triangular R using the QR algorithm. S is then set to Q.

The projection steps, which Stewart [62] calls Schur-Rayleigh-Ritz (SRR) refinement, are equivalent to calculating the Rayleigh quotient matrices for the Ritz vectors in S. Once we have a projected pair (M, N), we solve the generalized eigenproblem $MV = NV\Lambda$ for eigenvectors q_i and eigenvalues λ_{ii}. We then reorder the eigenvalues and eigenvectors based on $|\lambda_{ii}|$. We reorder because the dominant eigenpairs converge first and hence we can bring them to the left. The Ritz vectors in S are then refined by multiplying back by the ordered vectors in V.

The QR factorization and generalized eigenproblem are solved using the LAPACK routines `dgeqrf` and `dsygvd` [2] respectively. `dgeqrf` uses a blocked Householder QR algorithm similar to the one described in [24, sec. 5.2.2] and `dsygvd` uses a divide and conquer algorithm [24, sec. 8.5.4].

Locking

Locking refers to isolating parts of the solution that have converged and not modifying them any further. Once the first p eigenpairs have converged, we can lock these vectors in the powering step to save computational cost. If $S = [S_1 \quad S_2]$ and if the vectors in S_1 have converged, the powering step 5 in Alg. 4.3 can be modified as

$$[T_1 \quad T_2] = [S_1 \quad A^{-1}BS_2].$$

We must, however, use the entire subspace for the SRR projection/refinement steps. Therefore we arrive at Algorithm 4.4 (the missing parts are same as algorithm 4.3).

Algorithm 4.4 Code fragment for locking.

```
1  ···
2  p = 0
3  for k = 1:iter
4       S_2 = S(:, p+1:ℓ)
5       T_2 = B S_2
6       S_2 = A\T_2
7       S(:, p+1:ℓ) = S_2
8       S = QR; S = Q (orthonormalization using QR factorization)
9       M = S^T A S; N = S^T B S
10      ···
11      Test convergence.
12      Set p = number of pairs converged.
13      if p == m, exit
14 end
```

Shifting

The idea with shifting is to power the vectors in S with a shifted system $(A - \mu B)^{-1}$ instead of A^{-1} so as to accelerate convergence. When we shift A close to an eigenvalue and repeatedly multiply S by its inverse, the Ritz vectors rapidly grow richer in the direction of the eigenvector corresponding to the eigenvalue closest to the shift. More specifically, if each vector s_i converges with the ratio $c|\lambda_i/\lambda_\ell|$ for some constant c in unshifted SI, then by shifting to a $\mu > 0 \in \mathbb{R}$, we can decrease the convergence ratio to $c|(\lambda_i - \mu)/(\lambda_\ell - \mu)|$. For a detailed analysis, see, for example, [24, sec. 7.6.1].

The shift must be chosen carefully taking into account the following:

- Shifting would necessitate factorizing the shifted matrix $A-\mu B$, which is an $O(n^3)$ operation. Hence the shift must cause *enough*[2] acceleration to justify the cost.
- Shifting to an exact eigenvalue λ_i results in A becoming singular.
- Since in SI eigenvalues converge from left to right, shifting can result in missing a few eigenvalues.
- When shifting to accelerate convergence for λ_p, choosing $\mu \gg \lambda_p$ can have the detrimental effect of increasing the convergence ratio instead of decreasing it.

A natural choice for the shift is the location $(\lambda_{p+1} + \lambda_p)/2$, where p is the number of eigenpairs that have converged at any stage. But we would like for the shift to accelerate convergence not just for the pth eigenpair but also for the succeeding ones.

We therefore derive a novel shifting strategy to ensure rapid convergence of eigenpairs as follows. The upper bound for the convergence ratio (for a shifted pair (λ_p, s_p)) for any shift to be beneficial is λ_p/λ_ℓ, the natural rate without any shifting. Hence we have the requirement

$$\left|\frac{\lambda_p - \mu}{\lambda_\ell - \mu}\right| \leq \frac{\lambda_p}{\lambda_\ell}.$$

Since the eigenvalues we are interested in are positive and finite, we have

$$\mu \leq \mu_{limiting} = \frac{2\lambda_p\lambda_\ell}{\lambda_p + \lambda_\ell}.$$

The shift μ can then be chosen as

$$\mu = \lambda_r := \max\{\lambda_i \mid \lambda_i < \mu_{limiting}\ , i \in (p, \ell]\}, \tag{4.11}$$

using the estimates of λ_i's we have from previous iterations.

In practice though, we find that this value of μ could sometimes be aggressive, i.e., it may not cause enough decrease in the convergence rate. Since the convergence of the iteration is sensitive to the location of the shift, it is possible for an aggressive shift to stagnate the progress of the iteration. In such an event, the shift is brought back to its safe, conservative value of

$$(\lambda_{p+1} + \lambda_p)/2. \tag{4.12}$$

[2]We cannot measure the exact acceleration in convergence.

The variable *count* (algorithm 4.5) keeps track of the number of iterations the algorithm spends without any change in the number of converged pairs.

It must also be ensured that μ is not too close to any of the existing eigenvalue estimates to avoid forming a singular matrix $A - \mu B$. If μ calculated from (4.11) returns a value too close to any λ_i for $i \in (p, r)$, then r in (4.11) is decremented continuously till a safe value of μ is obtained (Algorithm 4.5 step 9).

Finally, since the convergence rates depends on the distribution of the eigenvalues, the choice of shifting is made a user-controllable parameter as follows.

- **Aggressive:** using (4.11) for μ and moving to conservative (using relation (4.12) for μ) shifting if the iteration stagnates. This is the default strategy.
- **Conservative:** relation (4.12) shifts only.
- **None**.

In the event of non-convergence, this gives the user an opportunity to restart the iteration with a different shifting strategy.

Delayed Orthonogonalization

In SI, we multiply S by $A^{-1}B$ and then orthonormalize it to prevent it from becoming ill-conditioned. However, the orthogonalization need not be carried out at every iteration and can be put off until there is an imminent possibility of ill conditioning resulting from the columns pointing in the same direction. Therefore we can orthogonalize the columns of the matrix $(A^{-1}B)^p S$ instead of $(A^{-1}B)S$, which amounts to a significant saving of flops. The power p has to be chosen optimally since a value too small would result in unnecessary work whereas if the orthogonalization is put off for too long we might lose information about the subspace.

The loss of orthogonality is measured by observing the inner product matrix $X := S^T S$, with the columns of S set to unit 1-norm. For an S with perfectly orthonormal columns,

$$\|X - I\| = O(u).$$

Algorithm 4.5 Code fragment for shifting.

```
1  ...
2  Set p = p_prev = 0; count = 0; μ = 0.
3  for k = 1:iter
4      if count > 1
5          μ = (λ_p + λ_{p+1})/2
6      elseif p > 0
7          μ_limiting = 2λ_p λ_ℓ/(λ_p + λ_ℓ).
8          Pick λ_r using (4.11). Set μ = λ_r.
9          if μ ≈ λ_i for any i ∈ (p + 1, r)
10             Update r = r − 1 and repeat from 9.
11         end
12     end
13     A = A − μB
14     T = BS
15     S = A\T
16     ...
17     for i = 1: n, λ_i = λ_i + μ, end
18     ...
19     p_prev = p
20     Test convergence.
21     Set p = number of pairs converged.
22     if p = m
23         exit
24     else
25         if p_prev = p, count = count + 1
26         else count = 0
27     end.
28 end
```

Therefore we orthogonalize S when $\|X - I\|_1$ becomes large enough. We use a threshold of $\sqrt{u}$ for this check, i.e., we orthogonalize S when

$$\|S^T S - I\|_1 > \sqrt{u}.$$

4.6 Implementation for Symmetric SEP

In the case of the standard eigenvalue problem, we are required to compute both the smallest and the largest eigenvalues and eigenvectors of the semidefinite matrix A. Therefore we use SI in two passes. In the first pass, we compute the smallest eigenvalues of A using inverse subspace iteration, i.e. we power the subspace with A^{-1}. In the next pass, we compute the largest eigenvalues of A using powers of A. Since A can sometimes be singular we use a user-defined shift to avoid solving singular systems when powering by A^{-1}.

Algorithm 4.6 lists the details of our implementation. The algorithm locks converged eigenvectors in S_s and S_ℓ during the iteration to save the redundant work of powering them. The algorithm uses the driver routine `dsyevr` from LAPACK to compute the eigenvalues Λ_s and Λ_ℓ and eigenvectors Q of the projected symmetric matrix M.

To test convergence for an eigenpair (λ, s), we calculate the scaled residual

$$\rho = \frac{\|As - \lambda s\|_1}{|\lambda| \|A\|_1}, \qquad \text{where } \|s\|_1 = 1, \tag{4.13}$$

and compare it with the user-supplied tolerance.

4.7 `esol`: The Eigenvalue Solver Library

We combine the implementations in sections 4.5 and 4.6 in a C++ library called `esol`. Our library is written using `Eigen` and makes use of LAPACK and Intel MKL [37]. It accepts the stiffness and mass matrices as sparse symmetric `Eigen::Sparse` matrix objects stored in the Compressed Sparse Row format and returns eigenvalues as an array of doubles and eigenvectors as an `Eigen::Dense` matrix objects. In addition, we also compute the quantity

$$\text{sr} = \frac{\|As_i - \lambda_i B s_i\|_2}{\|As_i\|_2} \tag{4.14}$$

Algorithm 4.6 Subspace iteration for computing the smallest and largest eigenpairs of A

Given a sparse, symmetric matrix $A \in \mathbb{R}^{n \times n}$, a tolerance tol and shift α, this algorithm computes the n_s smallest eigenvalues $\text{diag}(\Lambda_s)$ and n_l largest eigenvalues $\text{diag}(\Lambda_\ell)$ of A and their corresponding eigenvectors S_s and S_ℓ, respectively.

1 Initialize $S_s \in \mathbb{R}^{n \times 2n_s}$ and $S_\ell \in \mathbb{R}^{n \times 2n_\ell}$ with random vectors.
2 % Smallest eigenvalues
3 **Set** $t = 0$ % Number of converged eigenpairs
4 Compute the factorization $A - \alpha I = LDL^T$.
5 **Do**
6 $S_1 = S_s[:, 1{:}t]$. $S_2 = S_s[:, t+1{:}n_s]$.
7 $S_s = [S_1 \quad (A - \alpha I)^{-1} S_2]$ % triangular system solve using factorization in 4.
8 Orthonormalize S_s
9 $M = S_s^T A S_s$
10 Compute all eigenvalues Λ_s and eigenvectors Q of M
11 $S_s[:, t+1{:}n_s] = S_2 Q$
12 Test convergence and **Set** t = number of converged eigenpairs.
13 **Repeat Until** $t = n_s$
14 % Largest eigenvalues
15 **Set** $t = 0$
16 **Do**
17 $S_1 = S_\ell[:, 1{:}t]$. $S_2 = S_\ell[:, t+1{:}n_\ell]$.
18 $S_\ell = [S_1 \quad A S_2]$
19 Orthonormalize S_ℓ
20 $M = S_\ell^T A S_\ell$.
21 Compute all eigenvalues Λ_ℓ and eigenvectors Q of M
22 $S_\ell[:, t+1{:}n_s] = S_2 Q$
23 Test convergence and **Set** t = number of converged eigenpairs.
24 **Repeat Until** $t = n_\ell$

for each computed eigenpair (λ_1, s_i). The scaled residual sr for an eigenpair is used by Bathe [7, p. 884] as a measure of error in the computed solution, since it signifies the ratio of the out of balance forces to the nodal forces. Additionally, it also returns the backward errors for each computed eigenpair calculated from equations (4.10) and (4.13) and streams the progress of the iteration to the console.

The eigensolver `esol` offers the user control over the following aspects of the algorithms:

- ℓ, the number of subspace vectors for GEP,
- the choice of shifting strategy – aggressive/conservative/none – to be used,
- convergence tolerance tol and
- the shift α used in SEP.

4.8 Numerical Experiments and Results

The performance of `esol` (GEP) was measured for a variety of real-world structural models analysed in GSA and compared with `SSPACE`, the implementation of Algorithm 4.2. Here we present the results of this comparison for a few problems of different sizes that are representative of the range of models that are analysed in GSA. The models vary not only in size (of the stiffness and mass matrices) but also differ in the number of eigenpairs required for satisfying design criteria. The tests were conducted on a workstation with Intel Sandy Bridge-based Core i7 processor with 4 cores and 2 hyperthreads per core and 24 GB of RAM. We used vendor-supplied multithreaded sparse BLAS by linking to Intel MKL version 10.3.

Table 4.1 lists the results of this comparison. We note that since the convergence test used in `esol` (4.10) is different from the convergence test used in Algorithm 4.2, the comparison is not exactly 'like-for-like'. Therefore we also compute the largest angle between eigenvectors computed by each algorithm, i.e., for each eigenvector $s_{i,\texttt{esol}}$ and $s_{i,\texttt{SSPACE}}$, we compute

$$\text{maxangle} = \max_i \left| \arccos \left(\frac{s_{i,\texttt{esol}}^T s_{i,\texttt{SSPACE}}}{\|s_{i,\texttt{esol}}\| \|s_{i,\texttt{SSPACE}}\|} \right) \right|$$

and report it in Table 4.1. We compare the performance of both algorithms for computing 15 and 100 eigenvalues and eigenvectors. Where the algorithm ran for longer than 4 hours, we stop the iteration and report the time taken as being greater than 14400 seconds.

Table 4.1: Runtime comparison between `esol` and `SSPACE` (Alg. 4.2)

Model	Degrees of freedom (problem size)	Time (sec)				Angle (in degrees)
		15 eigenpairs		100 eigenpairs		
		`SSPACE`	`esol`	`SSPACE`	`esol`	
bairport	67537	82.1	19.9	1334.1	132.12	0.03427
tallbuilding	267176	2798.6	95.5	13279	669.8	0
limest	298831	139.5	67	> 14400	396.8	0.0213
RG	1,446,230	> 14400	102	> 14400	1818.3	0.01417

As it can be seen, `esol` is about 2x faster when only 15 eigenpairs are requested but for larger problems, the difference between the runtime is more than 10x. This can be attributed to efficiency gains from shifting, delayed orthogonalization and locking. We note that despite the relatively expensive operation of factorizing $A - \alpha I$ into LDL^T at every shift the benefits outweigh the costs. Because the convergence ratio at any point in the iteration depends on the ratio λ_p/λ_ℓ, decreasing the ratio results in a reduction in the number of iterations.

Also of interest is the effect of shifting strategy on the speed of convergence. In Table 4.2, we list the performance for computing 150 eigenpairs for each problem with the 'conservative' and 'aggressive' shifting strategy described in section 4.5.2. In all cases but 'tallbuilding' using the aggressive shifting strategy accelerates convergence. Since the actual effect of the shifts depends on the distribution of the eigenvalues, we therefore give the control of this parameter to the user and use a heuristic as a default.

Table 4.2: Runtimes for different shifting strategies for computing 150 eigenpairs

Model	Time (sec)	
	Conservative	Aggressive
bairport	172	160
tallbuilding	1045	1070
limest	550	498
RG	17206	16340

4.9 Conclusions

We have created an efficient implementation of the subspace iteration method for solving the sparse symmetric-definite GEP and symmetric SEP that arise in structural analysis applications. Our implementation supersedes the previous implementation of the same method in Oasys GSA and is between 2 and 10 times faster in our tests.

The Subspace Iteration method is simple to implement yet is numerically robust and our convergence test ensures the computed eigenpairs have $O(u)$ backward error. Since it is a block method, it benefits from level 3 BLAS during the powering and projection steps. Our implementation also uses optimizations brought about by a novel shifting strategy, locking and delaying orthogonalization of subspace vectors.

Further work will involve investigating the use of an iterative solution scheme to solve linear systems during inverse subspace iteration as proposed in [72], [73]. These methods link the accuracy of the linear system solves to the convergence of the main iteration. An important advantage of this approach is that it eliminates the dependence on a sparse direct solver and only uses sparse matrix vector multiplication as the main computational kernel. This allows us to work with different sparse matrix formats for storing the sparse matrix and these can bring greater computational throughput to the execution. The main challenge with these methods, however, is their dependence on a suitable preconditioner for the type of problems encountered in GSA. The practical applicability for an industrial-strength black-box solver will therefore need to be carefully evaluated.

Chapter 5

Efficient Sparse Matrix Multiple-Vector Multiplication using a Bitmapped Format

5.1 Introduction

The sparse matrix × vector (SpMV) and sparse matrix × multiple-vector (SMMV) multiplication routines are key kernels in many sparse matrix computations used in numerical linear algebra, including iterative linear solvers and sparse eigenvalue solvers. For example, in the subspace iteration method used for solving for a few eigenvalues of a large sparse matrix A, one forms the Rayleigh quotient (projection) matrix $M = S^T AS$, where $A \in \mathbb{R}^{n \times n}$ and $S \in \mathbb{R}^{n \times p}$ is a dense matrix with $p \ll n$. The computational bottleneck in such algorithms is the formation of the SMMV products. SpMV/SMMV routines typically utilize only a fraction of the processor's peak performance. The reasons for the low utilisation are a) indexing overheads associated with storing and accessing elements of sparse matrices and b) irregular memory accesses leading to low reuse of entries loaded in caches and registers.

Obtaining higher performance from these kernels is an area of active research owing to challenges posed by hardware trends over the last two decades and significant attention has been paid to techniques that address the challenges. This hardware trend is outlined by McKee in the famous note 'The Memory Wall' [71], [47] and can be summarized as

follows: the amount of computational power available (both the CPU cycle time and the total number of available cores) is increasing with a rate that is much higher than the rate of increase of memory bandwidth. It will therefore lead to a scenario where performance bottlenecks arise not because of processors' speeds but from the rate of transfer of data to them. The implication of this trend is that there is an increasing need for devising algorithms, methods and storage formats that obtain higher processor utilization by reducing communication. Such techniques and approaches will hold the key for achieving good scalability in serial and parallel execution, both on existing and emerging architectures.

In this chapter we introduce a blocked sparse format with an accompanying SMMV algorithm that is motivated by the above discussion of reducing communication cost. The format improves on an existing blocked sparse format by retaining its advantages whilst avoiding the drawbacks. An algorithm that computes SMMV products efficiently for a matrix in this format is developed and its performance is compared with the existing blocked and unblocked formats. The algorithm achieves superior performance over these formats on both Intel and AMD based x86-platforms and holds promise for use in a variety of sparse matrix applications. The current discussion is in the context of Oasys GSA.

5.2 Overview and Comparison of Compressed Sparse Row and Block Compressed Sparse Row Formats

The Compressed Sparse Row (CSR) format [6] (or its variant, the Compressed Sparse Column format) can be regarded as the *de-facto* standard format for storing and manipulating sparse matrices. The CSR format stores the nonzero entries in an array `val` of the relevant datatype (single precision, double precision or integers). The column indices of the entries are stored in `col_idx` and the row indices are inferred from the markers

into `col_idx`, stored as `row_start`. For example, with array indices starting with 0:

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & a_{32} & 0 \end{pmatrix}$$

$$\texttt{val} = (a_{00}, a_{01}, a_{02}, a_{03}, a_{10}, a_{11}, a_{22}, a_{23}, a_{32})$$

$$\texttt{col\_idx} = (0, 1, 2, 3, 0, 1, 2, 3, 2)$$

$$\texttt{row\_start} = (0, 4, 6, 8, 9)$$

If the number of nonzeros in A is z and if A is stored in double precision, the storage cost for A in the CSR format is $3z + n + 1$ words. (We assume a word size of 32 bits for the entire discussion.) An unoptimized SpMV algorithm is shown in snippet 5.7.

Algorithm 5.7 Compute $y = y + Ax$ for a matrix A stored in CSR format and conforming vectors x and y stored as arrays.

```
1  for i = 0 to n − 1
2        yi = y[i]
3        for j = row_start[i] to row_start[i + 1]
4              yi += val[j] * x[col_idx[j]]
5        y[i] = yi
```

The SpMV implementation in Algorithm 5.7 suffers from the problem of irregular memory use, which results in reduced data locality and poor reuse of entries loaded in registers. It performs a single floating point addition and multiplication for every entry `val` and `x` loaded, thus has a low 'computational intensity', i.e., it performs too few flops for every word of data loaded. This aspect of CSR SpMV has been well studied in the past, see [66] or [51] for example.

The Block Compressed Sparse Row (BCSR) format [51] is intended to improve the register reuse of the CSR. The BCSR format stores nonzero entries as dense blocks in a contiguous array `val`. These blocks are of size $r \times c$ where r and c are respectively the number of rows and columns in the dense blocks. For indexing it stores the column position of the blocks in an array (`col_idx`) and row-start positions in `col_idx` in `row_start`.

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & a_{32} & 0 \end{pmatrix}$$

$$\begin{aligned} \texttt{r} = 2, \texttt{c} = 2 \\ \texttt{val} = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, 0, 0, \\ a_{22}, a_{23}, a_{32}, 0) \\ \texttt{col\_idx} = (0, 1, 1) \\ \texttt{row\_start} = (0, 2, 3) \end{aligned}$$

A sparse matrix stored in the BCSR format takes up $\approxeq 2brc + b + \frac{n}{r}$ words to store, where b is the number of nonzero blocks (for a given r and c) stored when the matrix is held in BCSR.

Algorithm 5.8 Compute $y = y + Ax$ for a BCSR matrix A with $r, c = 2$ and bm block-rows and vectors x, y.

```
1  for i = 1 to bm
2      ir = i × r
3      y0 = y[ir]
4      y1 = y[ir + 1]
5      for j = row_start[i] to row_start[i + 1]
6          jc = col_idx[j] × c
7          y0 += val[0] * x[jc]
8          y1 += val[2] * x[jc]
9          y0 += val[1] * x[jc + 1]
10         y1 += val[3] * x[jc + 1]
11         increment pointer val by 4
12     y[ir] = y0
13     y[ir + 1] = y1
```

The SpMV routine for BCSR with $r, c = 2$ is presented in Algorithm 5.8. Since the `val` array is stored as a sequence of blocks, the algorithm loads all entries in a block into registers and multiplies them with corresponding entries in `x`. The increase in spatial locality results in the reuse of register–loaded entries of `x`, reducing the total number of cache accesses. The inner loop that forms the product of the block with the corresponding

part of the vector is fully unrolled, reducing branch penalties and allowing the processor to prefetch data. There is, however, a tradeoff involved in selecting the right block size for BCSR. The reuse of loaded registers increases in number with an increase in r and c but having larger block sizes may increase the fill, leading to higher bandwidth costs and extra computations involving zeros, which decrease performance. The amount of fill for a given block size depends on the distribution of the nonzeros in the matrix. The efficiency gains from increasing the block size will depend on the size of the registers and the cache hierarchy of the machine the code is running on. There is, therefore, an optimal block size for a given matrix (or set of matrices with the same nonzero structure) and a given architecture. This suggests using a tuning based approach to picking the optimum block size and such approaches have been studied extensively in [36], [67] and [42]. The dependence of the performance of the SpMV routine on the structure of the matrix make it a complex and tedious process to extract enough performance gains to offset the overheads of maintaining and manipulating the matrix in a different format, such as implementing kernels for other common matrix operations for a given block size. Additionally, the prospect of storing zeros increases the storage costs, making it dependent on the matrix structure, which is information that is available only at runtime. The arguments above motivate a blocked format that offers the benefits of BCSR's performance without the associated storage and bandwidth penalties.

5.3 The Mapped Blocked Row Format

We now introduce the mapped blocked row sparse (MBR) format for storing sparse matrices. For a matrix

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & a_{32} & 0 \end{pmatrix},$$

we represent the 2×2 block on the top right as a combination of the nonzero elements and a boolean matrix representing the nonzero structure:

$$\begin{matrix} a_{02} & a_{03} \\ 0 & 0 \end{matrix} \quad \Rightarrow \quad \begin{matrix} a_{02} & a_{03} \end{matrix} + \begin{matrix} 1 & 1 \\ 0 & 0 \end{matrix}.$$

The bit sequence `0011` can be represented in decimal as 3 and this representation is stored in a separate array. Thus, for our example, the MBR representation can be written as follows:

$$\begin{aligned} \texttt{r} = 2, \texttt{c} &= 2 \\ \texttt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \\ \texttt{col\_idx} &= (0, 1, 1) \\ \texttt{b\_map} &= (15, 3, 7) \\ \texttt{row\_start} &= (0, 2, 3) \end{aligned}$$

The bit structures of the blocks are stored in `b_map`, the array of their corresponding decimal representations. The bitmaps are encoded in left-to-right and top-to-bottom order, with first position in the block (i.e. the bit on top left) being stored as the lowest bit and the bottom right position being the highest.

The datatype `maptype` of the `b_map` array can be chosen to fit the size of the blocks; hence if the block size is 8×5, 40 bits are required and a `__int64` [48] will be used but if the block size is 4×4, a `maptype` of `short` will suffice for the 16 bits needed. For block sizes where the number of bits are less than the corresponding variable that stores the bitmap, the excess bits are left unused. With built-in C++ datatypes, up to 8×8 blocks can be supported. Larger blocks can be constructed by combining two or more adjacent instances in an array of built-in types or by using a dynamic bitset class like `boost::dynamic_bitset` [1].

The storage cost of an $n \times n$ matrix in the MBR format is bounded by

$$\underbrace{2z}_{\texttt{val}} + \underbrace{b}_{\texttt{col\_idx}} + \overbrace{\frac{b}{\delta}}^{\texttt{b\_map}} + \underbrace{\frac{n}{r}}_{\texttt{row\_start}} \quad \text{words}$$

where z is the number of nonzeros, b the number of blocks and r the size of the blocks. δ is the ratio $\frac{\texttt{sizeof(int)}}{\texttt{sizeof(maptype)}}$ to convert the size of `maptype` into words. b lies in a range that is given by the following lemma.

Lemma 5.3.1. *For an $n \times n$ sparse matrix with z nonzeros and at least one nonzero per row and per column, the minimum number $b_{\min}$ of $r \times r$ blocks required to pack the entries is* $\text{ceil}(z/r^2)$ *and the maximum $b_{\max}$ is* $\min(z, \text{ceil}(n/r)^2)$.

[1] http://www.boost.org/doc/libs/1_36_0/libs/dynamic_bitset/dynamic_bitset.html

Proof. Since $z > n$, z entries can be arranged such that there is at least one nonzero per row and column. This can be done in z/r^2 blocks but no less, hence $b_{\min} = \text{ceil}(z/r^2)$ is the minimum number of blocks. The $n \times n$ matrix contains $\text{ceil}(\frac{n}{r})^2$ blocks. If $z > \frac{n^2}{r^2}$, then $b_{\max} = \text{ceil}(\frac{n}{r})^2$, otherwise each nonzero can occupy a block of its own, so we have $b_{\max} = z$ blocks. □

Although these bounds would be seldom attained in practice, they can provide an intuitive feel for when a particular format can become advantageous or disadvantageous. The storage costs of CSR, BCSR and MBR are compared in Table 5.1. For the lower

Table 5.1: Comparison of storage bounds for CSR, BCSR and MBR.

	CSR	BCSR	MBR
	$3z + n$	$2br^2 + b + \frac{n}{r}$	$2z + b\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$
Lower bound	$3z + n$	$2z + \frac{z}{r^2} + \frac{n}{r}$	$2z + \frac{z}{r^2}\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$
Upper bound	$3z + n$	$2n^2 + \frac{n^2}{r^2} + \frac{n}{r}$	$2z + \frac{n^2}{r^2}\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$

bound of b, the MBR format takes up storage comparable with the CSR format but more than BCSR. For b close to the upper bound, MBR requires significantly less storage than BCSR term but more than CSR. However, for all our test matrices, the number of blocks arising from the conversion to blocked formats resulted in MBR requiring less storage than both BCSR and CSR. Table 5.2 lists the storage costs (in words) and their ratios arising for 8×8 blocking of the test matrices (the matrices are introduced in Table 6.1).

Table 5.2: Ratio of storage for MBR to BCSR and MBR to CSR formats for 8×8 blocks.

Matrix	n	b	$\frac{\text{MBR}}{\text{BCSR}}$	$\frac{\text{MBR}}{\text{CSR}}$
sp_hub	143,460	249,267	0.171	0.759
rajat29	643,994	991,244	0.1	0.839
nlpkkt80	1,062,400	2,451,872	0.205	0.744
hamrle3	1,447,360	906,839	0.119	0.774
ecology1	1,000,000	622,750	0.149	0.75
dielFilterV3	1,102,824	11,352,283	0.145	0.791
dielFilterV2	1,157,456	8,106,718	0.116	0.828
asic_680k	682,862	728,334	0.106	0.814

5.3.1 Similarity with Other Formats

Buluç et al. independently propose a format called the 'bitmasked CSB' [11], based around the idea of storing blocks that are compressed using a bit structure representation. The format partitions the matrix into "compressed sparse" blocks of bit-mapped register blocks, resulting in two levels of blocking. The nonzero entries in each register block are stored contiguously and their positions within the block are marked using a bitwise representation. There is no storage of zeros (i.e. the fill), which improves on BCSR in the same way that MBR does, but the CSB SpMV algorithm does perform multiply-add operations on zeros so as to avoid conditionals. The storage cost for MBR is slightly less than that of bitmasked CSB because of higher bookkeeping arising from two levels of blocking and there are subtle differences in the encoding of bit structure of the blocks. In order to perform the multiplication of a block with the relevant chunk of a vector, bitmasked CSB uses SIMD instructions to load matrix entries, which we avoid. Instead, in MBR SpMV, we minimize the conditionals evaluated using de Bruijn sequences. Overall, whilst bitmasked CSB is geared towards optimizing parallel execution, the aim of this work is to obtain a high throughput for multiple vectors in the sequential case.

5.4 SpMV and SMMV with MBR

As noted in the first section, the SMMV kernels are employed in sparse eigensolvers, and our interest is in their eventual use in GSA. This software is required to run on a variety of x86 architectures both old and new. The aim with implementing SpMV and SMMV for the MBR format, therefore, was to obtain a clean but efficient routine subject to the following considerations.

- Not employing optimizations that are specific to a platform or hardware, for example, prefetching.

- Obtaining kernels with parameters such as the type of scalar (single, double, higher precision), block sizes, number of dense vectors and `maptype` datatypes bound at compile time. C++ templates offer metaprogramming techniques that allow such kernels to be generated at the time of compilation from a single source base. This

approach is advantageous compared with the use of external code generators for generating kernels in a parameter space since the programmer can write code for generating kernels in the same environment as generated code, thus making it easier to maintain and update.

- Focussing on sequential execution; this will inform the approach for a parallel version, which will be tackled as future work.

The programming language used was C++, using templates for kernel generation and to provide abstractions for optimizations like loop unrolling. The compilers used were Microsoft Visual C++ and Intel C++.

A naïve algorithm for SpMV for MBR is shown in Algorithm 5.9. The notation `*val` indicates dereferencing the C pointer `val` to obtain the value of the nonzero entry it currently points to. Let z_b be the number of nonzeros in a given block, represented as

Algorithm 5.9 SpMV for a matrix stored in MBR with block dimensions (r, c) and vectors x, y.

```
1  for each block row bi
2      for each block column bj in block row bi
3          map = b_map_bj
4          for each bit map_p in map
5              if map_p = 1
6                  i := p%r + bi × r, j := p%c + bj × c
7                  y(i) += *val × x(j)
8                  increment val
9              end
10         end
11     end
12 end
```

set bits in `map`. It is easy to show that the minimum number of register loads required to multiply a block by a corresponding chunk of vector x and add the result to y will be $\mathcal{O}(3z_b)$ in the worst case. Algorithm 5.9 attains this minimum and also minimizes flops since it enters the block–vector product loop (steps 6–8) exactly z_b times.

However, the algorithm is inefficient when implemented, because steps 4–11 contain conditionals that are evaluated r^2 times, which the compiler cannot optimize. The presence of conditionals also implies a high number of branch mis-predictions at runtime since

the blocks can have varying sparsity patterns. Mispredicted branches necessitate removal of partially-completed instructions from the CPU's pipeline, resulting in wasted cycles. Furthermore, step 6 is an expensive operation because of modulo (remainder) calculation (denoted using the binary operator `%`) and the loop does not do enough work to amortize it.

We introduce a set of optimizations to overcome these inefficiencies.

5.4.1 Optimal Iteration over Blocks

The `for` loop in step 4 and the `if` statement in line 5 contain conditionals (in case of the `for` loop, the conditional is the end-of-loop check) that present challenges to branch predictors. The true/false pattern of the second conditional is the same as bit-pattern of the block, the repeatability of which decreases with increasing size of the block.

One workaround is to use code replication for each bit pattern for a given block size, such that all possible bit structures are covered exhaustively. It would then be possible to write unrolled code for each configuration, thus completely avoiding conditionals. However this quickly becomes impractical since there are 2^r arrangements for a block of size $r \times r$ and generating explicit code can become unmanageable. Additionally, since it is desirable to not restrict the kernel to square blocks, the number of configurations to be covered increases even further. The solution therefore is to minimize the number of conditionals evaluated and fuse the end-of-loop check with the check for the set bit. In other words, instead of looping r^2 times over each block (and evaluating r^2 conditionals), we loop over them z_b times, thus evaluating only z_b conditionals. Algorithm 5.10 shows the modification to the relevant section from Algorithm 5.9.

Algorithm 5.10 A modification to steps 4–8 Algorithm 5.9.

```
1      ⋮
2  for each set bit p in map
3      i := p%r + bi × r, j := p%c + bj × c
4      y(i) += *val × x(j)
5      increment val
```

The key detail is iterating over *set* bits in 'for each'. This is achieved by determining the positions of trailing set bits using constant time operations. Once the position is

determined, the bit is unset and the process is repeated till all bits are zero. To determine the positions of trailing bits, we first isolate the trailing bit and then use de Bruijn sequences to find its position in the word, based on a technique proposed by Leiserson et al in [43].

A de Bruijn sequence of n bits, where n is a power of 2, is a constant where all contiguous substrings of length $\log_2 n$ are unique. For $n = 8$, such a constant could be $C :=$ `00011101`, which has all substrings of length 3 (`000`, `001`, `011`,..., `101`, `010`, `100`) unique. A lone bit in an 8-bit word, say x, can occupy any position from 0 to 7, which can be expressed in 3 bits. Therefore, by operating x on C, a 3-bit distinct word can be generated. This word is hashed to the corresponding position of `1` in x and such a hash table is stored for all positions, at compile time. At run time, the procedure is repeated for an x with a bit at an unknown position to yield a 3-bit word, which can then be looked up in the hash table.

Algorithm 5.11 Looping over set bits for a bitmap x of length 8 bits.

```
1  Pick an 8 bit de Bruijn sequence C and generate its hashtable h
2  while map ≠ 0
3      y = map & (−map)
4      z = C × y
5      z = z >> (8 − log2 8)
6      p = h(z)
7      compute i and j from p and multiply (Alg. 5.9 steps 6–8)
8      ⋮
9      map = map & (∼y)
10 end
```

Algorithm 5.11 lists the realization of 'for each' and the decoding of *map*. Step 3 isolates the trailing bit into y using the two's complement of *map*, steps $4 - 6$ calculate the index of the bit and step 9 clears the trailing bit. The operators used in the algorithm are standard C/C++ bitwise operation symbols: `&` for bitwise `AND`, `<<` and `>>` for left shift and right shift by the number denoted by the second operand, $\sim$ for bit complement. Steps 3–6 can be carried out in constant time and bitwise operations execute in a constant number of clock cycles [21]. The constant time decoding, combined with a reduction in the number of conditionals evaluated, gives huge performance gains.

5.4.2 Unrolled Loops for Multiple Vectors

At every iteration of the inner loop, Algorithm 5.10 calculates the position of the nonzero entry in the block and multiply-adds with the source and destination vector. The cost of index-calculation and looping can be amortized by increasing the amount of work done per nonzero decoded. This can be achieved by multiplying multiple vectors per iteration of the loop.

Algorithm 5.12 Multiplying multiple vectors in inner loops.

```
1  Given ... and x_1 ··· x_ℓ , y_1 ··· y_ℓ ∈ R^n
2  for each block row bi
3      for each block column bj in row bi
4          map = b_map_bj
5          for each set bit p in map
6              i := p%r + bi × r, j := p%c + bj × c
7              y_1(i) += *val × x_1(j)
8                  ⋮
9              y_ℓ(i) += *val × x_ℓ(j)
10             increment val
```

For the kernel to work with any value of ℓ, either the mutiply-add operations would need to be in a loop, which would be inefficient at runtime or we would need to write ℓ versions of the kernel, which can be tedious and expensive to maintain. This is overcome by using templates that generate code for many kernels at compile time, each varying in the number of vectors and each unrolling the innermost loop ℓ times. There is a close relationship between the performance of the kernel, the size of blocks (r, c) and the number of vectors multiplied ℓ. For a given block size, performance increases with increase in the number of vectors in the inner loop, since it increases the reuse of the nonzero values loaded in the registers and on lower levels of cache, till such a point where loading more vector entries displaces the vectors previously loaded, thereby destroying the benefit blocking brings in the first place. This suggests a need for tuning based either on comparative performance of the kernels or on heuristics gathered from the architecture (or indeed, on both). We use the former approach, leaving the investigation of the latter to future work.

5.5 Numerical Experiments

The experimental setup consists of x86-based machines based on Intel and AMD platforms intended to be representative of the target architectures the kernel will eventually run on. The code is compiled using the Intel C++ compiler v12.1 Update 1 on Windows with full optimization turned on. We note however, that this does not apply to the pre-compiled Intel MKL code (used for benchmarking) that takes advantage of vectorization using SIMD instructions available on all our test platforms. The test platforms consist of machines based on AMD Opteron, Intel Xeon and Intel Sandy Bridge processors. The AMD Opteron 6220, belonging to the Bulldozer architecture, is an 8-core processor with a clock speed of 3.0 GHz and 16 MB of shared L3. Each core also has access to 48 KB of L1 cache and 1000 KB of L2 cache. The Intel Harpertown-based Xeon E5450 on the other hand has access to 12 MB of L2 cache, shared between 4 cores on a single processor, each operating at 3 GHz. Each core has 256 KB of L1 cache. Both the Opteron and Xeon support the 128-bit SSE4 instruction set that allows operating on 2 double-precision floating point numbers in a single instruction. The third test platform is the Intel Core i7 2600 processor, based on the recent Sandy Bridge architecture, which is the second generation in the Intel Core line of CPUs. This processor has 4 cores sharing 8 MB of shared L3 cache with two levels private caches of 32 KB and 256 KB for each core. The cores operate at a peak clock speed of 3.8 GHz, with Turbo Boost turned off. The Core i7 processor uses the AVX instruction set, supporting 256-bit wide registers that enable operating on 4 double precision variables in a single instruction.

The test matrices consist of a set of matrices from the University of Florida Sparse Matrix collection [15] as well as from problems solved in Oasys GSA. These are listed in Table 6.1.

The final algorithm for MBR SMMV was a combination of all optimizations described in the previous section. The C++ implementation of this was run on the matrices via a test harness for different values of block sizes upto a maximum of 8×8 and multiple vectors. Where matrix sizes are not multiples of the block size, the matrix is padded with zeros on the right and on the bottom, such that the increased dimensions are a multiple. This merely involves modifying the arrays storing the indices and does not affect the storage or the performance, since the blocks are sparse. The performance of

Table 5.3: Test matrices used for benchmarking SMMV algorithms.

	Matrix	Source	Dimension	Nonzeros	Application
1	ASIC_680k	U.Florida	682,862	3,871,773	Circuit simulation
2	atmosmodm	U.Florida	1,489,752	10,319,760	Atmospheric modeling
3	circuit5M	U.Florida	5,558,326	59,524,291	Circuit simulation
4	dielfilterV2real	U.Florida	1,157,456	48,538,952	Electromagnetics
5	dielfilterV3real	U.Florida	1,102,824	89,306,020	Electromagnetics
6	ecology1	U.Florida	1,000,000	4,996,000	Landscape ecology
7	G3_circuit	U.Florida	1,585,478	7,660,826	Circuit simulation
8	hamrle3	U.Florida	1,447,360	5,514,242	Circuit simulation
9	nlpkkt80	U.Florida	1,062,400	28,704,672	Optimization
10	rajat29	U.Florida	643,994	4,866,270	Circuit simulation
11	sp_hub	GSA	143,460	2,365,036	Structural engineering
12	watercube	GSA	68,598	1,439,940	Structural engineering

the implementation was compared with that of CSR SMMV and BCSR SMMV. For all our tests, a near-exclusive access is simulated by ensuring that the test harness is the only data-intensive, user-driven program running on the system during the course of benchmarking.

We do not study the effect of reordering on the performance of the kernels, since in applications, the specific choice of the reordering algorithm may not always be governed by SMMV, instead it could be governed by other operations that the application performs. We do however note that any reordering approach that decreases the bandwidth of a matrix[2] will, in general, increase performance of a blocked format. Furthermore, we do not consider the costs of conversion between various formats since applications can generate a sparse matrix in the MBR format and use the format for an entire application, thereby negating expensive data transformations. This does, however, necessitate the availability of software for matrix manipulations and factorizations that the application uses.

In the case of CSR, a standard SpMV implementation based on Algorithm 5.7 and the functions available from the Intel MKL library [37] are used for comparison. The sparse BLAS Level 2 and Level 3 routines available within the MKL library are regarded as highly optimized implementations and achieve performance higher than corresponding reference implementations, especially on Intel platforms. The library offers the functions `mkl_cspblas_dcsrgemv` and `mkl_dcsrmm` that perform SpMV and SMMV operations.

[2]In this context, the term bandwidth refers to the maximum distance of a matrix nonzero element from the diagonal.

Since our objective is to obtain benchmarks for SMMV, `mkl_dcsrmm` would appear to be the right candidate. However, in almost all our experiments, `mkl_cspblas_dcsrgemv` outperformed `mkl_dcsrmm`, hence `mkl_cspblas_dcsrgemv` was used as the benchmark. Since the MKL library is closed-source software, it is not possible to determine why `mkl_dcsrmm` is not optimized to take advantage of multiple vectors.

For BCSR SMMV, an implementation of Algorithm 5.8 that is optimized for multiple vectors is used. This uses unrolled loops and multiplies each block with multiple vectors. The right number of vectors to be multiplied within each loop depends on the architecture and has been studied in [42]. Similar to the MBR algorithm, the optimal number of vectors depends on the architecture and needs to be selected via tuning. For the purpose of this comparison, we run BCSR SMMV kernels with fixed blocks of sizes 4×4 and 8×8, with increasing number of multiple vectors, going from 1 to 20, and select the best performance as being the representative performance of the format.

5.5.1 Performance against Number of Vectors

The performance of the MBR format depends on amortizing the cost from decoding the blocks, and this is achieved by multiplying multiple vectors. Therefore it is important to know the behaviour of the algorithm with respect to varying ℓ, the number of vectors. Figures 5.1 and 5.2 present how the performance varies on the Core i7 and Opteron respectively. In both cases, there is a sharp increase in performance initially, followed by a plateau and then a drop as the implementations go from single-vector kernels to ones handling 20 vectors. The reason for this behaviour is that when the number of vectors is increased, more vector entries stay loaded in the cache, reducing misses when the algorithm tries to load them again. The performance peaks and starts decreasing when the number of vectors reaches a point where loading more entries into the cache displaces previously loaded entries, leading to increased misses. The performance peaks for a different number of vectors, depending on the size of the cache hierarchy and to a lesser extent, matrix sizes and sparsity patterns.

On the Opteron, most matrices exhibit peak performance around the range of 12 to 16 vectors whilst on i7, the range is around 5 to 6. Both processors have a comparable L1 cache size but the Opteron has almost four times L2 cache as Core i7. This allows for

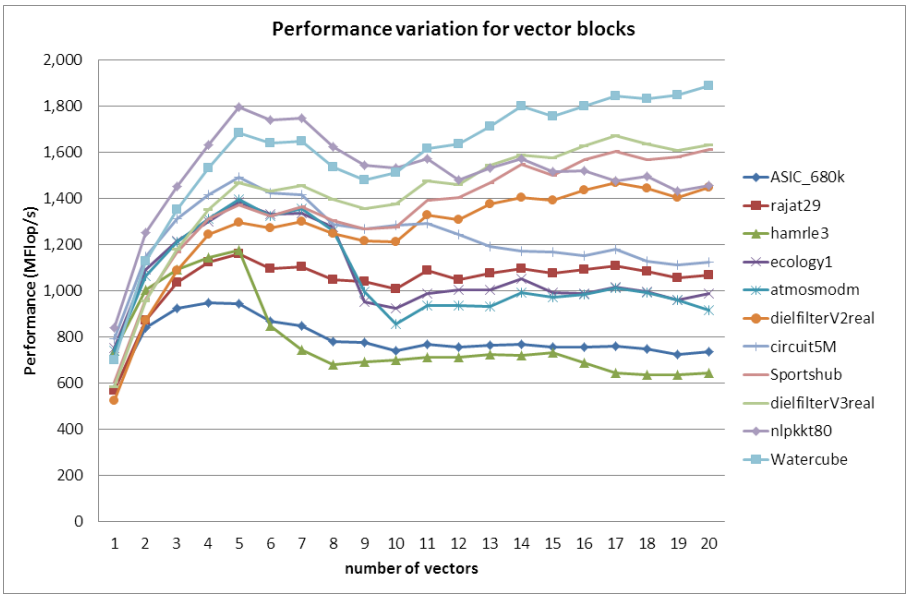


Figure 5.1: Performance variation of MBR SMMV across multiple vectors for test matrices on Intel Core i7 2600

more reuse of loaded entries and hence the performance tops at a higher value of ℓ.

A small number of matrices in graph 5.1 show an increase in performance after hitting a trough in the post-peak part of their curves. These matrices are watercube, sp_hub, dielfilterV2real and dielfilterV3real, from structural engineering and circuit simulation applications. They have the highest nonzero density amongst all matrices but are within the lower half when arranged by increasing order of matrix sizes. This combination of smaller sizes and low sparsity could result in higher performance—the size ensures that a larger number of vector chunks or entire vectors are resident in L3 caches, whereas the higher density results in higher flops per loaded vector entry. Indeed, the best performance is attained for watercube on both processors, which is the smallest matrix but has the highest nonzero density.

5.5.2 Performance Comparison with Other Kernels

The performance of the kernels is compared with that of other SMMV routines and the results for different platforms are presented in Figures 5.3, 5.4 and 5.5 for Xeon, Core i7 and Opteron respectively. In the case of MBR and BSR, the kernels used are the ones that yield the maximum performance for the given matrix and for the given block size, i.e., the peaks of graphs for each matrix in figures 5.1 or 5.2.

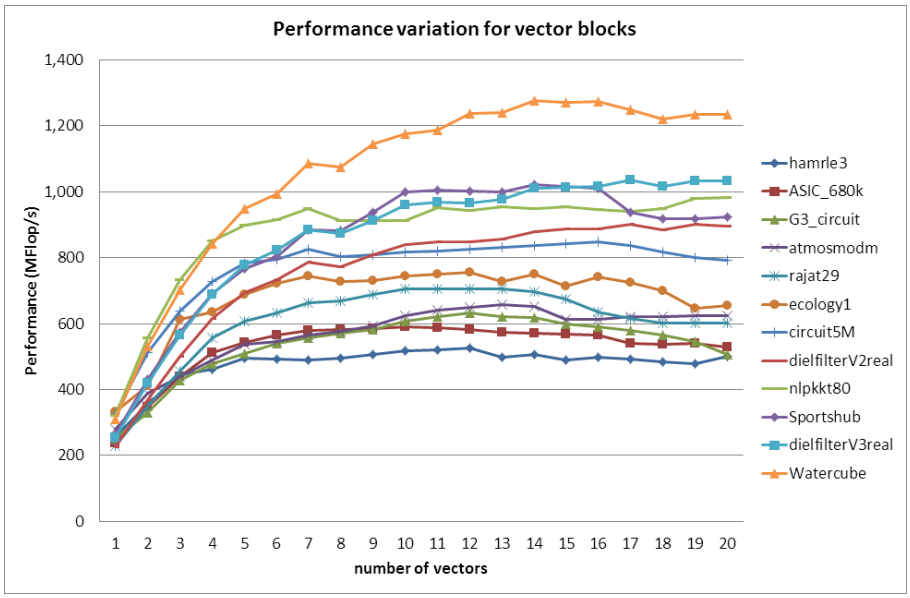


Figure 5.2: Performance variation of MBR SMMV across multiple vectors for test matrices on AMD Opteron 6220

On the Xeon, MBR is faster than MKL by factors of 1.3 to 1.9. It is also more efficient than BCSR for all matrices except watercube. The Xeon has the largest L2 cache of all test platforms. The large L2 cache and the small size of the matrix ensures that BCSR is faster, since it has fully unrolled loops with no conditionals, thus ensuring very regular data access patterns that aid prefetching. Furthermore, watercube also has the highest nonzero density and highest number of entries-per-block (z/b from Table 5.2) so the ratio of redundant flops (i.e. operations involving 0 entries) to useful flops is low, which helps the BCSR routine.

The performance trends for Core i7 are somewhat similar to those of Xeon, but a key difference is that it is the only architecture where MKL outperforms MBR for some matrices. The MKL to MBR performance ratios vary from 0.72 to 1.71. MKL is faster than MBR on four matrices: dielfilterV2real, dielfilterV3real, ASIC_680k and nlpkkt80, which come from different applications. There are no specific properties of these matrices or their sparsity patterns that gives us a suitable explanation for why MBR is slower. For two of the four matrices–dielfilterV2real and dielfilterV3real–the MBR SMMV performance vs. number of vectors graph (Figure 5.1) indicates that higher performance could be gained by using more than 20 vectors, although such a kernel may not always be relevant, especially in applications with small number of right hand sides. Evidently,

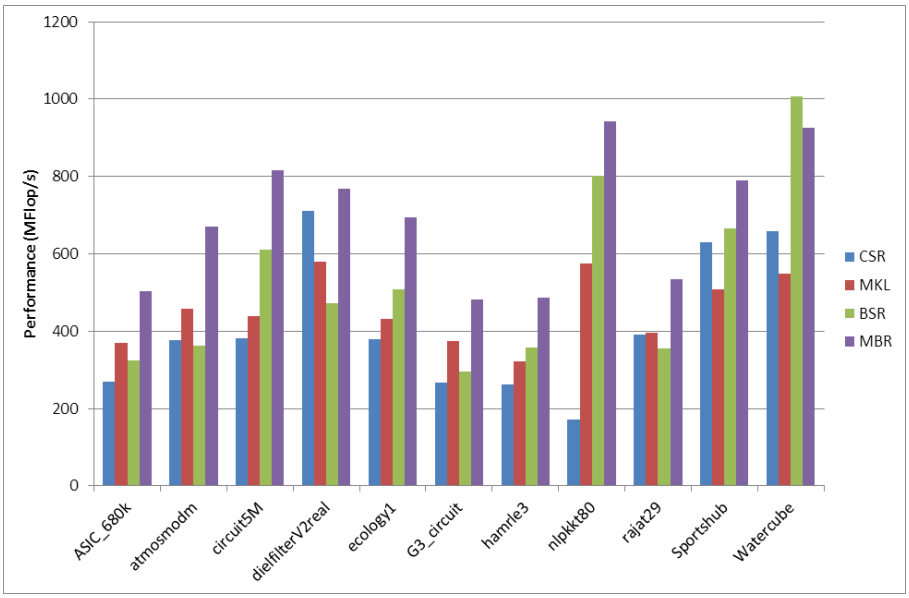


Figure 5.3: Performance comparison of MBR SMMV on Intel Xeon E5450

MKL's use of AVX instructions on Sandy Bridge allows for good efficiency gains that lead to a higher throughput. It will need a closer evaluation using experimental data from hardware counters combined with performance modelling to explain the reasons for this discrepancy, which will be looked at in future work.

Finally, on the AMD processor, MBR outperforms MKL by factors of 1.5 to 3.2 and BCSR by factors of 1.3 to 3.9. This demonstrates that while the MKL routines use architecture–specific and platform–specific optimization to gain efficiency, MBR SMMV is capable of attaining high efficiency through platform–neutral optimizations that deliver a good performance on all platforms.

5.6 Conclusions and Future Work

This work introduces mapped blocked row as a practical blocked sparse format that can be used with sparse matrix programs. The storage requirements of the format have been studied and they are significantly less than the two popular formats we have compared with. The MBR format offers the distinct advantage of being a blocked format that does not incur the computational and storage overheads of other formats. This holds promise for applications that involve very large problem sizes where holding the matrix in memory is an issue, for example iterative solvers for linear systems.

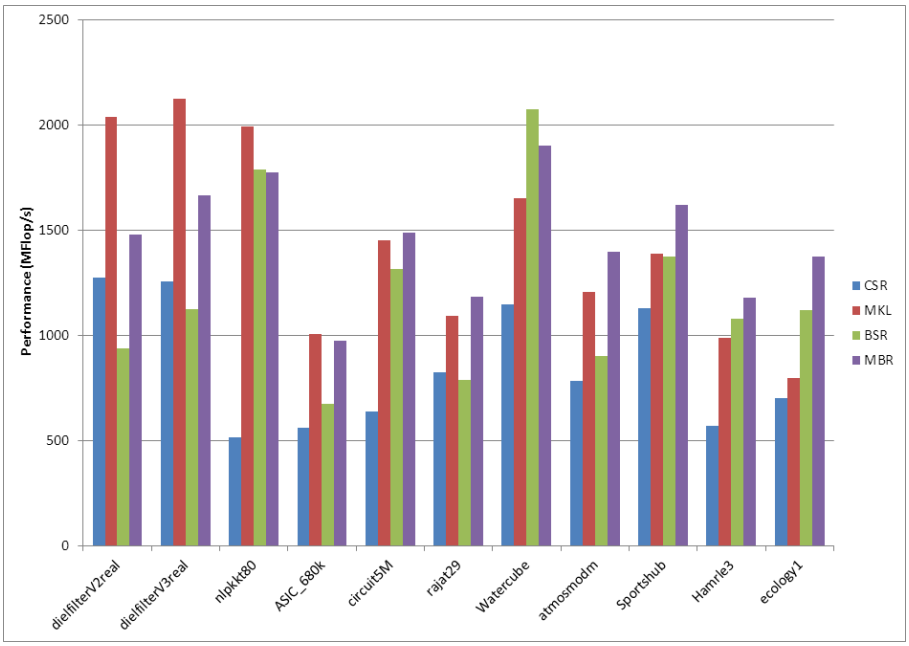


Figure 5.4: Performance comparison of MBR SMMV on Intel Core i7 2600

A fast algorithm has been developed for multiplying sparse matrices in the MBR format, with several optimizations for minimizing loop traversals and evaluations of conditionals, for increasing cache reuse and to amortize the decoding costs. By virtue of operating on a blocked format, the algorithm obtains high computational intensity. A C++ implementation of the algorithm offers compile–time parameters like the block size, number of vectors and the datatypes of the scalars and of the bitmap, making it generic in scope for a wide range of applications. The templates also makes it possible to produce code that has fully unrolled loops and kernels that bind to parameters at compile-time, unifying the code generator with the generated code for greater transparency and maintainability.

The performance results presented in the previous section prove that these performance optimizations can achieve good efficiency gains on all platforms by increasing register and cache reuse. The reference implementation attains performance over 3× that of the Intel MKL libraries and better performance on most test platforms over existing optimized BCSR and CSR implementations. There is ample scope to tune performance by modifying parameters such as the block size and effects of such tuning will be the topic of future work. A key motivation for communication reducing algorithms is the desire for improved parallel scalability. This article has focussed on establishing the performance

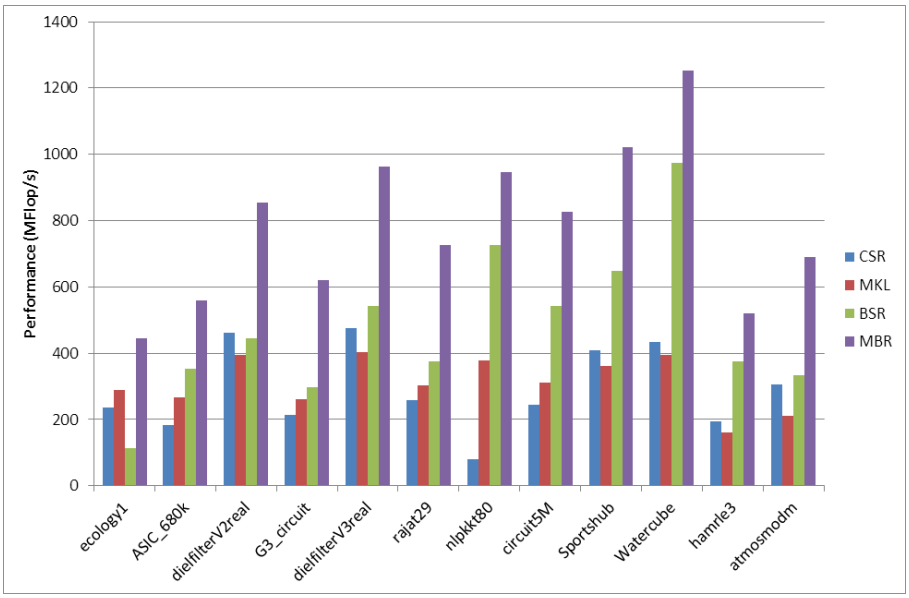


Figure 5.5: Performance comparison of MBR SMMV on AMD Opteron 6220

of the MBR format and the algorithm for sequential execution, paving the way for its parallelization, which will be explored in future work. Also of interest is the question "to what extent is a blocked sparse format of relevance for sparse direct solutions?" and whether it can offer advantages for storing and manipulating the factors from a Cholesky or symmetric indefinite factorization. These will be examined in due course.

Chapter 6

Cache-efficient B-orthogonalization for Multicore and Manycore Architectures

6.1 Introduction

Let B be a symmetric positive definite $n \times n$ matrix and X be a tall skinny matrix $n \times \ell$ ($n \gg \ell$) matrix of full column rank. We wish to compute a matrix S, with the same dimensions and column space as X, such that S is orthogonal with respect to the inner product defined by B, i.e. $S^T BS = I$. We call this problem the 'B-orthogonalization of a matrix' in this discussion. There are several applications for the B-orthogonalization problem. The one most relevant to us is the Lanczos method for the sparse GEP [5], [30], where it is used to transform the generalized eigenproblem into a standard eigenproblem. A related use occurs in structural engineering when we compute the eigenvectors of the equation $Kx = \lambda Mx$. If the eigenvalues of interest are the ones closest to the origin, an algorithm like the subspace iteration method (chapter 4) might compute the eigenvectors to be M-orthogonal. However, if the engineer requires the eigenvectors to be K-orthogonal, the K-orthogonalization can be performed as a post-processing step after solving the eigenvalue problem. Yet another use is in the weighted GMRES method [28]. A more extensive list of applications can be found in [52] and the references therein.

For the case of $B = I$, the problem reduces to the well known QR factorization:

computing a set of orthonormal columns Q spanning the same space as X such that $X = QR$, where R is an upper triangular matrix. Several methods exist for the QR factorization and their efficiency and stability is well studied in the literature, including classical and modified Gram-Schmidt and the Householder QR factorization [24, chap. 5].

To solve the B-orthogonalization problem, however, we do not have an analogue of the Householder QR method to use. This is because the product of two B-orthogonal matrices is not in general B-orthogonal. Since B is positive definite, an intuitive approach is to compute the Cholesky factor of $B = U^T U$. We can then find an orthonormal basis Y of the matrix XU using QR factorization $XU = YR$ and obtain the B-orthogonal matrix $S = U^{-1}Y$. This can however become expensive for large scale problems, since the $O(n^3)$ cost of the Cholesky factorization governs the total cost. In many applications, $\ell \ll n$ and we would have done nothing to take advantage of this structure. Therefore a frequently used method is the Gram-Schmidt method that uses B-inner products to orthogonalize the columns of X; this has been studied in [52]. Gram-Schmidt B-orthogonalization works by computing projections using matrix-vector and vector-vector products and consecutively B-orthogonalizes columns of X against the previously computed columns.

A drawback of the Gram-Schmidt scheme is its use of vector-vector and matrix-vector operations, which lead to cache-inefficient implementations and offer limited parallelization opportunities. These reasons motivate the need for more efficient strategies and we look at a class of methods that use either a Cholesky-like factorization or the eigenvalue decomposition (EVD) of the positive definite matrix $X^T BX$ for the orthgonalization. By Cholesky-like factorizations, we mean either the Cholesky or pivoted Cholesky factorization or their square root-free counterpart, the LDL^T factorization. In contrast to Gram-Schmidt, these methods use matrix-matrix or matrix-block-vector operations and subsequently have better parallel scalability. In our experiments, we observe that this results in orders-of-magnitude speedup over Gram-Schmidt methods both in the sequential as well as in the parallel case and that the method is stable under reasonable assumptions. Lowery and Langou [45] present the stability analysis for the Cholesky-based B-orthogonalization. Here we present algorithms that have a smaller loss of orthogonality than current Cholesky B-orthogonalization methods in the presence of ill-conditioning

in X and/or B and we develop a software library implementation for multicore and manycore architectures for a sparse B.

6.2 EVD and Cholesky B-orthogonalization

We now introduce our method for B-orthogonalization using the EVD and Cholesky-like factorization. We start by computing the symmetric positive definite $\ell \times \ell$ matrix

$$M := X^T BX. \tag{6.1}$$

If $M = Q\Lambda Q^T$ is the EVD of M, then the main idea is to form the matrix

$$S := XQ\Lambda^{-1/2}. \tag{6.2}$$

Mathematically, S is then B-orthogonal since

$$S^T BS = \Lambda^{-1/2} Q^T X^T BXQ\Lambda^{-1/2} = \Lambda^{-1/2} Q^T MQ\Lambda^{-1/2} = I.$$

Analogously, the Cholesky based variant uses the Cholesky factor of M to form a B-orthogonal matrix S as follows:

$$S = XR^{-1}, \qquad R = \texttt{chol}(M). \tag{6.3}$$

For both methods to work numerically, we need M to be numerically nonsingular, i.e. $\kappa(X^T BX) < u^{-1}$, where u is the machine precision. Forming the product $X^T BX$ may lead to loss of information about the column space of X but we recover this information by multiplying with X when we form S in the final step. In finite precision, the columns of S may not be perfectly B-orthogonal because of rounding errors in the process, but it is easy to show that the computed S is more B-orthogonal than X. Indeed if S is close to being B-orthogonal, the inner product matrix $S^T BS$ will be nearly diagonal with the diagonal entries close to 1, i.e., $\kappa(S^T BS)$ is close to 1. Therefore, to show S is more B-orthogonal than X, it suffices to show that $S^T BS$ is better conditioned than $X^T BX$. Assume the eigendecomposition of $X^T BX$ yields

$$X^T BX = \tilde{Q}\widehat{\Lambda}\tilde{Q}^T + E \tag{6.4}$$

where $\widehat{\Lambda}$ is the matrix of computed eigenvalues for some exactly orthogonal $\tilde{Q}$ and E is the error matrix of appropriate dimensions. We also assume, without loss of generality, that X is scaled such that $\|\widehat{\Lambda}\|_2 = 1$. If X^TBX is full rank,

$$\kappa_2(X^TBX) = O\left(\frac{1}{d}\right),$$

where d is the smallest diagonal entry in $\widehat{\Lambda}$.

If $\widehat{S} := X\tilde{Q}\widehat{\Lambda}^{-1/2}$, we have

$$\begin{aligned}\widehat{S}^TB\widehat{S} &= \widehat{\Lambda}^{-\frac{1}{2}}\tilde{Q}X^TBX\tilde{Q}^T\widehat{\Lambda}^{-\frac{1}{2}} \\ &= \widehat{\Lambda}^{-\frac{1}{2}}\tilde{Q}^T(\tilde{Q}\widehat{\Lambda}\tilde{Q}^T + E)\tilde{Q}^T\widehat{\Lambda}^{-\frac{1}{2}} \\ &= I + \widehat{\Lambda}^{-\frac{1}{2}}\tilde{Q}E\tilde{Q}^T\widehat{\Lambda}^{-\frac{1}{2}} =: N.\end{aligned}$$

To bound $\kappa_2(N)$, we need bounds on $\|N\|_2$ and $\|N^{-1}\|_2$. The former is straightforward: if $\|E\|_2 = \epsilon$, then

$$\|N\|_2 \leq 1 + \frac{\epsilon}{d}. \tag{6.5}$$

To bound $\|N^{-1}\|_2$, we use [24, Lemma 2.3.3] to observe that

$$\|N^{-1}\|_2 \leq \frac{1}{1 - \|E\|\|\widehat{\Lambda}^{-1}\|_2} = \frac{1}{1 - \epsilon/d} \approx 1 + \frac{\epsilon}{d}.$$

Therefore

$$\kappa_2(\widehat{S}^TB\widehat{S}) \approx \left(1 + \frac{\epsilon}{d}\right)^2 = 1 + \frac{2\epsilon}{d} + O(\epsilon^2) \tag{6.6}$$

and assuming $\epsilon \ll d$, $\kappa_2(\widehat{S}^TB\widehat{S}) < \kappa_2(X^TBX)$ as required.

The analysis also allows us to bound the loss of orthogonality $\|\widehat{S}^TB\widehat{S} - I\|_2$ informally. If we assume B is normalized such that $\|B\|_2 = 1$, from (6.5) we have

$$\|\widehat{S}^TB\widehat{S} - I\|_2 \leq \frac{\epsilon}{d} \leq \frac{\epsilon}{\lambda_{\min}(B)} = \epsilon\kappa_2(B).$$

The bound is sharp as it is attained if $\widehat{S}$ contains the eigenvectors associated with the smallest and the largest eigenvalues of B. We present a more formal bound in [40]. The stability of the Cholesky-based process has been analysed in [45] and the following result provides an upper bound for the loss of orthogonality. The bound is shown to be tight in the experiments of [45].

Theorem 6.2.1. *[45, Thm. 2.1] The computed factors $\widehat{S}$ and $\widehat{R}$ in the sequence (6.3) satisfy*

$$\|\widehat{S}^T B\widehat{S} - I\|_2 \leq cn\ell u\|\widehat{R}^{-1}\|_2\|X\|_2(\|\widehat{R}^{-1}\|_2\|BX\|_2 + \|\widehat{S}\|_2\|B\|_2) + O(u^2).$$

The EVD and Cholesky B-orthogonalization process has an obvious limitation. In the case where B and/or X have large enough condition numbers such that $\kappa_2(X^TBX) > O(u^{-1})$ the algorithm is unstable, since Λ could have negative or $O(u)$ entries and therefore $\Lambda^{-1/2}$ could be ill-conditioned or may not be real. This affects the practical applicability of the algorithm, since in real world applications we often encounter ill conditioned B or X. To stabilize the algorithm, we make the following modifications.

- We run the process more than once.
- We perturb Λ such that entries smaller than a certain threshold are replaced by a small positive multiple of u.

As a result, Λ, and subsequently, S, stays numerically nonsingular and the algorithm does not break down. The updated procedure is listed in Algorithm 6.13.

Algorithm 6.13 EVD: Stable EVD B-orthogonalization

Given a symmetric positive definite $B \in \mathbb{R}^{n\times n}$ and an $n\times\ell$ matrix X of full rank, compute $S \in \mathbb{R}^{n\times\ell}$ s.t. $\|S^TBS - I\| \leq$ tol for a tolerance tol and stability threshold h.

```
1   k = 1.
2   S^(k) = X.
3   M = S^(k)^T B S^(k)
4   while (‖M − I‖_1 ≤ tol and k < maxiter)
5       [Q, Λ] = eig(M).
6       for i = 1:ℓ
7           if |λ_i| < h, λ_i = λ_i + h.
8       S^(k+1) = S^(k) Q Λ^(−1/2)
9       M = S^(k+1)^T B S^(k+1).
10      k = k + 1.
11  S = S^(k)
```

On line 4, a tolerance of tol $= u\kappa(B)$ for the loss of orthogonality is a reasonable target. In our experiments, a value of 2 or 3 for maxiter was sufficient to achieve this and further iterations did not make the residual smaller. The stability threshold h is a

modest multiple of u; we use $100u$. To see why the perturbation technique works, we can recast the analysis that led to (6.6) such that E is the perturbation we induce, i.e. E is a diagonal matrix with $e_{ii} = h$ where $\widehat{\lambda}_{ii} = O(u)$ and 0 everywhere else. Then, from (6.6), the condition number of the resultant product $\widehat{S}^T B \widehat{S}$ is

$$\kappa(\widehat{S}^T B \widehat{S}) \approx 1 + \frac{2\epsilon}{d} = 1 + \frac{2h}{O(u)},$$

which, qualitatively, is significantly better than $\kappa(X^T BX) = O(1/d) = O(1/u)$ that we had started with. We present a more detailed analysis of algorithm 6.13 in [40]; for the rest of this article we focus on efficiency gains that our algorithm offers over Gram-Schmidt. The efficiency is a result of using level 3 BLAS operations to compute the inner product matrix, subsequent to which we can work with an $\ell \times \ell$ matrix. The EVD in Algorithm 6.13 can be computed using either the divide and conquer algorithm or the QR algorithm [24, chap. 8], both of which are available through LAPACK [2] and require reducing the matrix M to tridiagonal form, which costs $4\ell^3/3$ flops. We can avoid this cost by replacing the EVD by a Cholesky factorization, which can be computed in as few as $\ell^3/3$ flops and has excellent numerical stability [33, chap. 10]. Since the matrix M can be semidefinite, we use a pivoted Cholesky factorization that computes the factorization of a permuted matrix $P^T MP = R^T R$, as given by the following theorem.

Theorem 6.2.2. *[33, Thm. 10.9] Let $A \in \mathbb{R}^{n\times n}$ be a positive semidefinite matrix of rank r. (a) There exists at least one upper triangular R with nonnegative diagonal elements such that $A = R^T R$. (b) There is a permutation Π such that $\Pi^T A \Pi$ has a unique Cholesky factorization, which takes the form*

$$\Pi^T A \Pi = R^T R, \quad R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}, \tag{6.7}$$

where R_{11} is $r \times r$ upper triangular with positive diagonal elements.

In [46], Lucas presented a pivoted form of Cholesky factorization for semidefinite matrices that was implemented [29] and incorporated into LAPACK version 3.2 as the routine `xPSTRF`. We make use of this routine to derive Algorithm 6.14.

As with algorithm 6.13, we use a value of $h = 100u$. In our numerical experiments, Algorithm 6.14 was stable for a range of problems. As with Algorithm 6.13, it requires at most 2 or 3 iterations.

Algorithm 6.14 CHOLP: Stable pivoted-Cholesky B-orthogonalization

Given a symmetric positive definite $B \in \mathbb{R}^{n\times n}$ and an $n\times\ell$ matrix X of full rank, compute $S \in \mathbb{R}^{n\times\ell}$ s.t. $\|S^TBS - I\| \leq$ tol for a tolerance tol and stability threshold h.

$k = 1$.
$S^{(k)} = X$.
$M = {S^{(k)}}^T BS^{(k)}$.
while ($\|M - I\|_1 \leq$ tol **and** $k <$ maxiter)
    $[R, P] =$ `dpstrf(`M`)`.
    for $i = 1{:}\,\ell$
        if $|r_{ii}| < h$, $r_{ii} = r_{ii} + h$.
    $S^{(k+1)} = S^{(k)}PR^{-1}$.
    $M = {S^{(k+1)}}^T BS^{(k+1)}$
    $k = k + 1$.
$S = S^{(k)}$.

6.3 Gram-Schmidt B-orthogonalization

We briefly summarize the Gram-Schmidt family of B-orthogonalization methods here, which is a generalization of the Gram-Schmidt family for the standard inner product. Whilst the latter have been studied in great detail (see [23] and references therein), the B-inner product case has been analysed in Rozložník et al. in [52].

To form a B-orthogonal S from X using classical Gram-Schmidt (CGS), we compute

$$s_i = s'_i / r_{ii}, \tag{6.8}$$

where

$$s'_i = x_i - \sum_{j=1}^{i-1} r_{ij}x_j, \qquad r_{ij} = s_j^T Bx_i, \qquad r_{ii} = \|s'_i\|_B.$$

The CGS process is known to suffer from loss of orthogonality, which is usually remedied in one of the following two ways. The first is to change the order of the computations such that when s_i is computed at the ith step all remaining vectors in X are orthogonalized against it; this alteration is known as the modified Gram-Schmidt method. The other remedy is to reorthogonalize the vectors computed from CGS, i.e. run CGS twice, which is sufficient to ensure a small loss of orthogonality, a principle informally referred to as "twice-is-enough" [23]. The twice-CGS algorithm (CGS2) has a smaller loss of orthogonality compared with MGS B-orthogonalzation [52]; we therefore use this method for our comparisons. The CGS2 algorithm is listed in Alg 6.15.

Algorithm 6.15 CGS2: Gram-Schmidt B-orthogonalization with reorthogonalization

Given a symmetric positive definite $B \in \mathbb{R}^{n\times n}$ and an $n\times\ell$ matrix X of full rank, compute $S \in \mathbb{R}^{n\times\ell}$ with the same column space as X s.t. $\|S^TBS - I\| < \|X^TBX - I\|$.

$k = 0$.
$S^{(k)} = X$.
while $(k < 2)$
  for $i = 1{:}\,\ell$
  $s_i^{(k+1)} = s_i^{(k)}$.
  for $j = 1{:}\,i-1$
    $r = {s_j}^{(k+1)^T} B s_i^{(k)}$.
    $s_i^{(k+1)} = s_i^{(k+1)} - r s_j^{(k+1)}$.
  $s_i^{(k+1)} = s_i^{(k+1)} / {s_i^{(k+1)}}^T B s_i^{(k+1)}$.
  $k = k + 1$.
$S = S^{(k)}$.

The next theorem bounds the loss of orthogonality for the CGS2 algorithm.

Theorem 6.3.1. *[52, Thm. 5.2] Assuming $cn^{3/2}\ell u\kappa^{1/2}(B)\kappa(BX) < 1$, the computed $\widehat{S}$ from Alg. 6.15 satisfies*

$$\|\widehat{S}^TB\widehat{S} - I\|_2 \le cn^{3/2}\ell u\kappa(B). \tag{6.9}$$

We note that (6.9) does not depend on $\kappa(X)$, which allows the CGS2 method to handle problems where X has a large condition number. This contrasts with the Cholesky-based bound of Theorem 6.2.1. However, we observe in our experiments that for both CGS2 and the pivoted Cholesky version, the loss of orthogonality varies proportionally with $u\kappa(B)$.

6.4 Parallel Implementation

Since the main motivation for using Cholesky based B-orthogonalization was its superior cache-efficiency, we are interested in knowing how its performance compares with Gram-Schmidt and how well it scales with increasing problem size. In particular, we are interested in accelerator-based heterogeneous architectures and SMP-based shared memory models. The use of accelerators/coprocessors has gained significant popularity

in recent years and as many as 53 computers in the November 2013 Top500[1] list of supercomputers use them, including the supercomputers ranked 1 and 2. This is because they not only achieve good performance but also reduce energy usage per computation. Indeed all 10 of the top supercomputers in the Green500 list[2] are based on heterogeneous memory. The other factor that motivates using accelerators is the use of thread-based programming models, which is intuitively more natural than the distributed memory programming.

Amongst accelerators, Intel's Xeon Phi coprocessor, also known as Knight's Corner, is a recent development that is becoming popular in HPC applications. Based on Intel's Many Integrated Core architecture, Xeon Phi is a coprocessor with more than 50 cores that support multiple in-order hardware threads with 512-bit wide vector units and with local caches attached to them. A block diagram is shown in Figure 6.1. Being an x86 SMP-on-a-board architecture, Xeon Phi offers the possibility to use the same tools, programming languages and programming models as on x86 CPUs, like OpenMP or Intel Cilk. This is a crucial difference between Xeon Phi and GPU accelerators, which use programming extensions and frameworks like CUDA or OpenCL that are significantly different.

The MIC architecture supports a new SIMD instruction set that is different from the SSE and AVX instruction sets found on x86 CPU chips. Many of these instructions are syntactically similar to their CPU counterparts although a key difference is the addition of a masking flag parameter to include/exclude operands from the vectors. Of the new instructions, the ones of interest are scatter and gather of SIMD vectors from unaligned noncontiguous memory locations, prefetch hints and fused multiply-add instructions.

Our library implementation of Algorithm 6.14 is called `chol_borth`. It is written in C and C++, using Eigen [26] for providing a C++ library interface and C for low-level SIMD intrinsics. `chol_borth` takes as inputs a sparse positive definite matrix B, a dense matrix X and a stability threshold for perturbation. We also created an implementation of Algorithm 6.15 for comparison, using multithreaded sparse BLAS kernels for parallelization.

The main bottleneck in the execution of Algorithm 6.14 is the formation of the inner

[1] http://www.top500.org/lists/2013/11/highlights/
[2] http://www.green500.org/lists/green201311

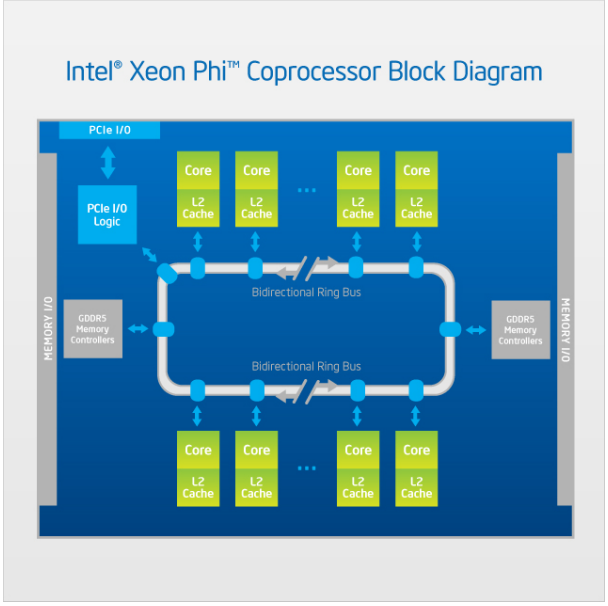


Figure 6.1: High level view of Xeon Phi

product matrix S^TBS on line 9. This involves formation of a Sparse Matrix-Vector product, which has been studied extensively in literature, see, for example, the references in chapter 5. More recently its performance has also been studied for Xeon Phi in [44] and [55]. In this work we aim to create a library implementation. Hence, unlike the cited articles, we avoid changing the storage format of the sparse matrix and only use the Compressed Sparse Row (CSR) storage. This is because the B-orthogonalization algorithm is usually part of a "larger" program, for example, a routine for computing the eigenpairs of a sparse GEP, and hence the storage format needs may be governed by other operations in the parent algorithm, for example, a sparse direct solution, and such software may not exist for the new format. Changing the sparse matrix format to accelerate the B-orthogonalization may also slow down other parts of the calling program that are not optimized to work with a different format.

The inner product $M = X^TBX$ can be computed in three ways depending on how loops are ordered. In the first (Algorithm 6.16), we simply evaluate a temporary dense matrix $Y = BX$ and then form $M = X^TY$. Both the matrices are computed using multithreaded BLAS kernels.

For larger problems or smaller cache hierarchies, it may not be efficient to form Y if it causes S to be evicted. Therefore, alternatively, we loop over the columns of X, evaluate a Sparse Matrix-Vector product (SpMV) and use the resultant vector to compute an

Algorithm 6.16 FULL_SpMM

Given Sparse $B \in \mathbb{R}^{n\times n}$ and Dense $S \in \mathbb{R}^{n\times \ell}$, compute $M = S^T BS$.

```
1    Y = BS /*dcsrmm */
2    M = Y^T S /* dgemm */
```

inner product with X, thereby reusing the computed vector that is loaded in the cache. Algorithm 6.17 presents this approach.

Algorithm 6.17 SpMV

Given Sparse $B \in \mathbb{R}^{n\times n}$ and Dense $S \in \mathbb{R}^{n\times \ell}$, compute $M = S^T BS$.

```
1  for i = 1:ℓ
2      y = Bs_i /* dcsrmv */
3      for j = 1:i
4          M(j,i) = s_j^T y /*ddot */
```

We can also design a method to reuse the loaded entries of B as well as the computed entries of BS. This can be done by evaluating M in blocks, which divides the columns of X into ℓ/b blocks made up of b contiguous columns. For each block $X(:, k\colon k+b)$, we compute $Y_k = BX(:, k\colon k+b)$ first and then form $X(:, k\colon \ell)^T Y_k$. Each block becomes a work chunk for a thread in the OpenMP thread pool. Since the number of blocks is dependent on the problem size, it is possible for number of threads to exceed the number of blocks. Therefore, we create a nested threaded region where each inner product between rows of B and a block is processed on a different thread.

Algorithm 6.18 BLOCK_SpMV

Given Sparse $B \in \mathbb{R}^{n\times n}$, Dense $X \in \mathbb{R}^{n\times \ell}$ and block size b, compute $M = X^T BX$.

```
1  for block index jj = 1:ℓ/b in Parallel
2      X_jj = X(:, jj:jj+1)
3      for i = 1:n in Parallel
4          Y_jj = b_i^T X_jj /* sprow Alg. 6.19 */
5      M(jj:ℓ/b, jj:jj+1) = X(:, jj:ℓ/b)^T Y_jj /*dgemm*/
```

The inner product `sprow` between the sparse rows of B and block column X_{jj} is evaluated using a SIMD kernel shown in Algorithm 6.19. This algorithm makes of use of the new `gather` instruction that is supported on MIC, which allows loading words from

unaligned memory locations given by a vector of indices *simd_ind*. Firstly, the inner product loop is peeled to reach an aligned location in `val` and `col`. Then we load aligned data from `val` and `col` into MIC's 512-bit SIMD vectors *simd_val* and *simd_ind*, which we use for a `gather`. Subsequently, we multiply the loaded data using `mul` and finally sum the vector entries with a `reduce`. The inner loops that multiply individual vectors in a block are unrolled using compiler pragmas.

Algorithm 6.19 `sprow`

Given row i of sparse B held as CSR (`row_ptr, col, val`) and X_{jj} held as array `x` as in Alg. 6.18, compute $Y_{jj} = b_i^T X_{jj}$, held as array `y`.

```
1  // peel loop to reach aligned position
2  aligned_start = row_ptr[i] + row_ptr[i]%8
3  for j = row_ptr[i]: aligned_start
4      for k = 1: b Unroll
5          y[k × n + i] = y[k × n + i] + val[j] × x[k × n + col[j]]
6  aligned_end = row_ptr[i + 1] + 8 − row_ptr[i + 1]%8
7  for j = aligned_start: aligned_end
8      simd_ind = load(&col[j])
9      simd_vals = load(&vals[j])
10     for k = 1: b Unroll
11         simd_x = gather(simd_ind, &x[k × n])
12         simd_y = mul(simd_vals, simd_x)
13         y[k × n + i] = y[k × n + i] + reduce(simd_y)
14     j = j + 16
15 for j = aligned_end: row_ptr[i + 1]
16     for k = 1: b Unroll
17         y[k × n + i] = y[k × n + i] + val[j] × x[k × n + col[j]]
```

The other operations in Algorithm 6.14 are parallelized either using OpenMP loops or using threaded BLAS from MKL.

6.5 Experiments

The library `chol_borth` was compiled using Intel C++ Compiler version 14.0 and we use CMAKE as the build system for cross compilation for MIC. The code is compiled with `-O3` optimization level, with autovectorization turned on for both CPU and MIC. We used the following hardware for the experiments.

Table 6.1: Test matrices used for benchmarking `chol_borth`.

Name	Application	Size	Nonzeros	Nonzeros/row
apache2	finite difference	715,176	2,766,523	3.86
bairport	finite element	67,537	774,378	11.46
bone010	model reduction	986,703	47,851,783	48.50
G3_circuit	circuit simulation	1,585,478	7,660,826	4.83
Geo_1438	finite element	1,437,960	60,236,322	41.89
parabolic_fem	CFD	525,825	2,100,225	3.99
serena	finite element	1,391,349	64,131,971	46.09
shipsec8	finite element	114,919	3,303,553	28.74
watercube	finite element	68,598	1,439,940	20.99

Xeon Phi 7120P The instance of MIC we use is the 7120P. This coprocessor is connected through the system PCI bus and has 61 cores connected to each other and to the memory controllers via a bidirectional ring. The card we use has 16 GB of memory. Each core has a 64-bit Pentium based architecture that has been upgraded to support 4-way in-order multithreading with full coherent 32 KB L1 cache and 512 KB L2 cache. Much of the processing power of the core comes from the 512-bit SIMD Vector Processing Unit that can operate on 8 doubles in a single cycle. The measured peak `dgemm` performance of the coprocessor is 950 GFlops/sec.

Xeon E5-2670 Our shared memory system has Intel Sandy Bridge-based symmetric multiprocessors. There are 2 processors with 8 cores per processor, each with 2 hyperthreads that share a large 20 MB L3 cache. Each core also has access to 32 KB of L1 cache and 256 KB of L2 cache and runs at 2.6 GHz. They support the AVX instruction set with 256-bit wide SIMD registers, resulting in 10.4 GFlop/s of double precision performance per core or 83.2 GFlop/s per processor and a `dgemm` performance of 59 GFlop/sec.

We run experiments to gauge both numerical stability and computational performance. The former is tested using MATLAB implementations of Algorithms 6.14 and 6.15, with specially-constructed B and X matrices. For measuring computational performance, we use `chol_borth` where the test problems are positive definite sparse matrices from structural FE models and from the University of Florida [15] collection, as listed in Table 6.1.

6.5.1 Loss of Orthogonality

We are interested in the loss of orthogonality in using Algorithm 6.14 and how it compares with Algorithm 6.15 and, for reference, Algorithm 6.13. We study this by increasing the conditioning of the problem, i.e. $\kappa_2(X^TBX)$ and plot the loss of orthogonality $\|S^TBS - I\|_2$ against $\kappa_2(B)$. We construct B of size 5000×5000 using the MATLAB command `gallery('randsvd')`. We take 500 vectors in X and consider three cases:

1. X has random vectors and has a condition number that increases with $\kappa_2(B)$. It is generated using randsvd.

2. X has linear combinations of the eigenvectors of unit 2-norms associated with the $\ell/2$ smallest and $\ell/2$ largest eigenvalues of B, with $\kappa_2(X) = \kappa_2(B)$. It is generated in the following way:

```
% k_B is the condition number of B
X = gallery('randsvd',[n l],k_B);
[~,R] = qr(X,0);
[V,D] = eig(B);
% Fill X with extremal evectors
X = [V(:, 1:l/2) V(:, (n-l/2+1):n)];
% Multiply by R to obtain desired condition number
X = X*R;
```

3. Same as 2 but with $\kappa_2(X)$ varying as $\sqrt{\kappa_2(B)}$.

Cases 2 and 3 represent the worst case bound for the loss of orthogonality. The variation of $\kappa_2(X)$ with $\kappa_2(B)$ in case 2 will indicate the behaviour in the extreme scenario, whereas case 3 is for comparison with experiments in [45].

Figure 6.2 shows the loss of orthogonality for the random X case and the loss of orthogonality is very similar for pivoted Cholesky and CGS2. The EVD based version has a larger error when $\kappa_2(B)$ grows towards $O(u^{-1})$, a phenomenon we are currently unable to explain. In Figure 6.3, we plot the worst case, i.e., when X contains extremal eigenvectors of B. As expected, all three algorithms behave similarly and the error is independent of the conditioning of X. They lose almost all orthogonality as $\kappa(B)$ reaches

$O(u)^{-1}$ but we observe that when $\kappa(X)$ varies as $\kappa(B)$, the algorithms are still usable up to $\kappa(B) = 10^{15}$. `chol_borth` always takes 2 or 3 iterations to reach the desired tolerance for orthogonality.

6.5.2 Performance Comparison with CGS2

The motivation for investigating Cholesky-based B-orthogonalization was the efficiency of the latter, hence it is important to know what performance gains can be achieved against CGS2. Therefore we measure the performance of `chol_borth` on CPU and MIC by varying the size of X and compare the results with those from CGS2. On the CPU, the tests are run by scheduling all threads on a single processor by using the OpenMP 'compact' thread affinity, which is set using `KMP_SET_AFFINITY`. On MIC, we use OpenMP's 'balanced' thread affinity and place 2 threads each on 60 cores using `KMP_PLACE_THREADS`. Since we are interested in the speedup that is brought about by `chol_borth`, we use X with 16 and 256 vectors in order to estimate the lower and upper bounds of the efficiency gains.

Table 6.2 shows the results for our performance comparisons. On the CPU, `chol_borth` was faster than CGS2 by a minimum of 3.7 times for 16 vectors and a maximum of 40 times for 256 vectors. This contrasts with the MIC where we gain over 8x performance for the smallest problem and over 200x for X with 256 vectors. The large performance gains on MIC show the importance of minimizing irregular accesses during manycore/heterogeneous execution and demonstrate the importance of targeting cache-efficiency in algorithm development.

On the MIC, with apache, `chol_borth` gained the highest speedup over CGS2 when the problem size was increased. The reason for this is the low number of nonzeros-per-row for the apache2 matrix. On the other hand, bone010 gained the least since it has the highest ratio of nonzeros per row. On CPUs, CGS2 benefits from the larger cache hierarchy which leads to greater reuse, hence the improvements performance of `chol_borth` over CGS2 are not as drastic.

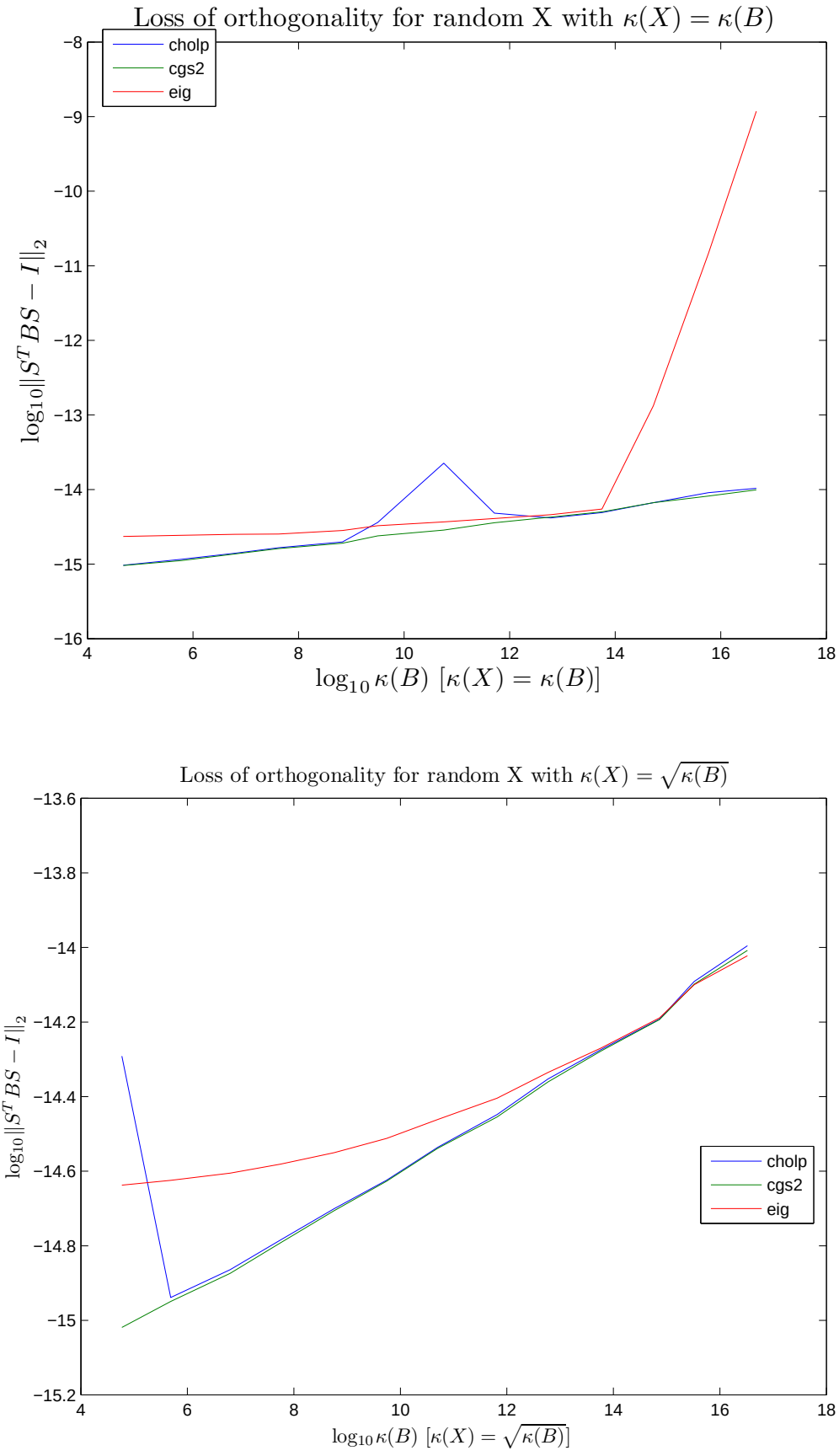
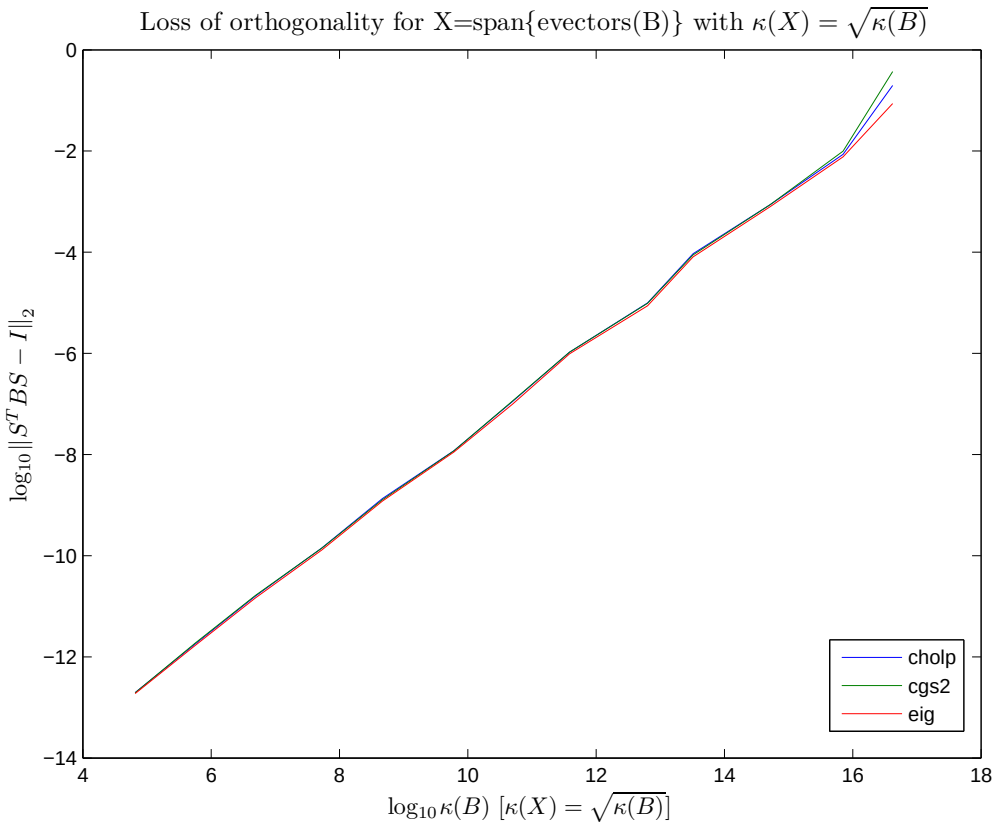


Figure 6.2: Loss of orthogonality for random X.



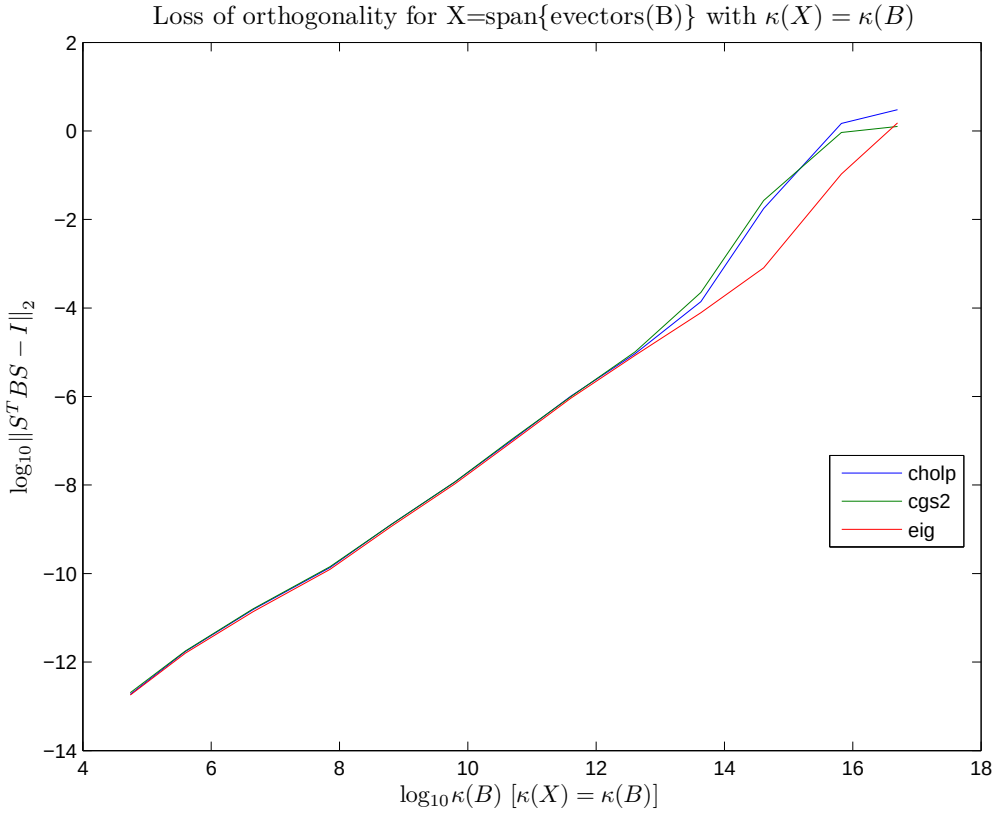


Figure 6.3: Loss of orthogonality for X with eigenvectors of B.

Table 6.2: Performance ratios of `chol_borth` vs CGS2 for small and large problems on CPU and MIC (ratios of runtime of each algorithm).

Name	CPU		MIC	
	$\ell = 16$	$\ell = 256$	$\ell = 16$	$\ell = 256$
apache2	5.7	40.9	16.6	219.4
bairport	3.7	28.8	9.9	24.9
bone010	3.7	28.8	6.9	9.4
G3_circuit	6.6	15.6	14.6	31.8
Geo_1438	4.1	23.9	15.5	66.5
parabolic_fem	17.7	39.7	15.7	65.5
serena	4.0	39.8	8.1	87.2
shipsec8	3.8	21.4	14.1	99.7
watercube	3.7	25.1	14.1	99.7

6.5.3 Scalablity

We are also interested to know the execution rates that `chol_borth` is capable of on MIC and CPUs and how these scale with increase in the number of vectors ℓ to be orthognalized. In section 6.4, we discussed three strategies for parallelizing the formation of the inner product matrix, which is the main bottleneck in `chol_borth`. These are FULL_SpMM (Algorithm 6.16), SINGLE_SpMV (Algorithm 6.17) and BLOCK_SpMV (Algorithm 6.18). We profile `chol_borth` while using each of these strategies for increasing sizes of X. All tests are run with the same thread placement and affinity settings that were used in the previous section.

In Figure 6.4, we show the effect of increasing ℓ on the CPU and Figure 6.5 contains the comparisons for MIC.

On the CPU, FULL_SpMM is the most efficient strategy on most problems. BLOCK_SpMV comes close for larger problems, i.e., when $\ell = 256$. We attribute the efficiency of FULL_SpMM to the large L3 cache on CPUs that ensures that more entries from $Y = BX$ stay loaded before they are multiplied with X^T. The inner product operation has a flop count of $2z\ell + 2\ell^2 n$, z being the number of nonzeros in B. Therefore for large ℓ, the cost of $X^T Y$ dominates and since FULL_SpMM uses `dgemm` for this operation, it is the most efficient. The matrix bone010 has the lowest flop rate because it has the highest nonzeros per row, which displace the computed values of Y from the cache. As the problem size decreases, SINGLE_SpMV starts becoming as competitive as FULL_SpMM.

On MIC, SINGLE_SpMV is more efficient for some problems than FULL_SpMM

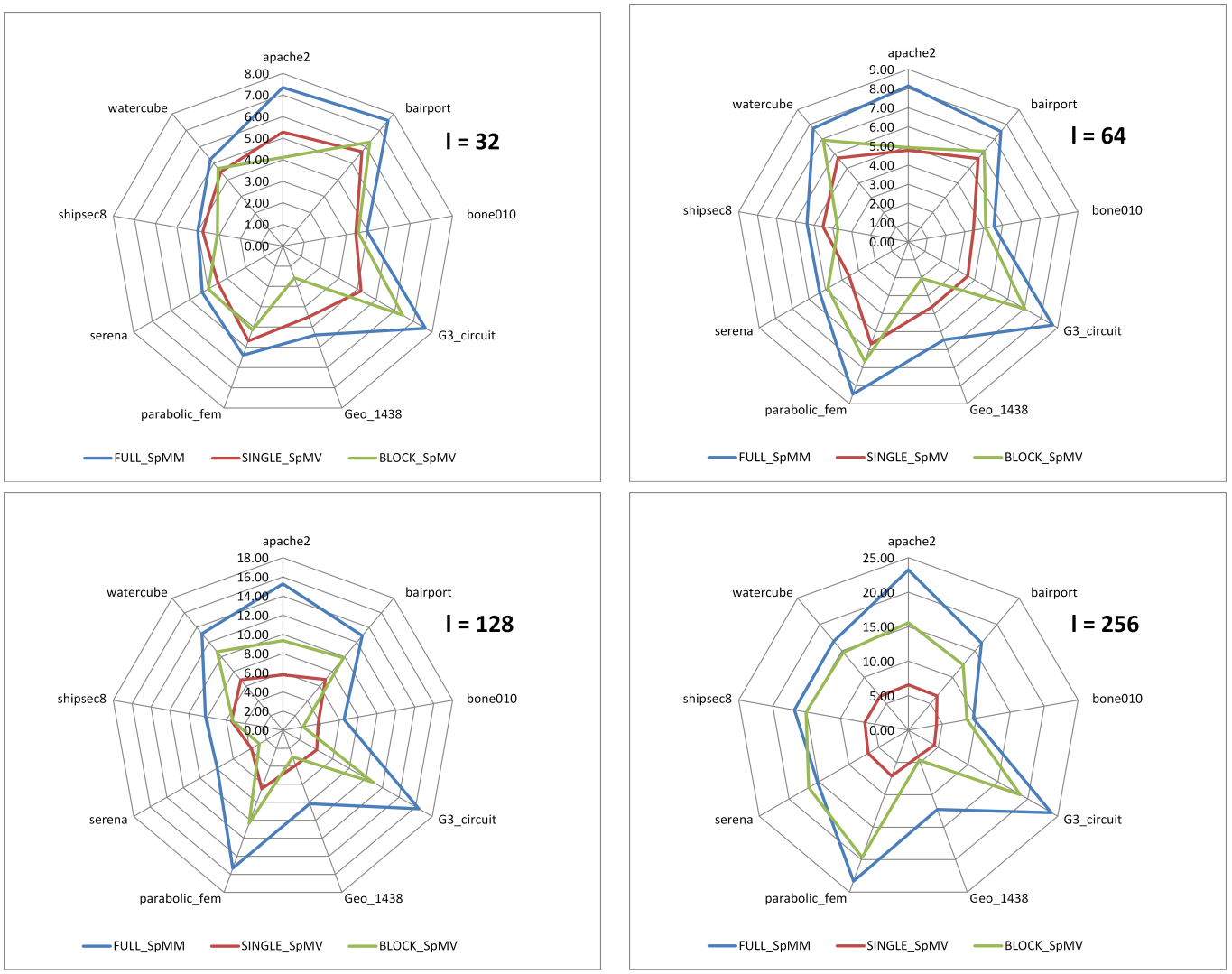


Figure 6.4: Execution rates for various ℓ on CPU. Performance in GFlops/sec.

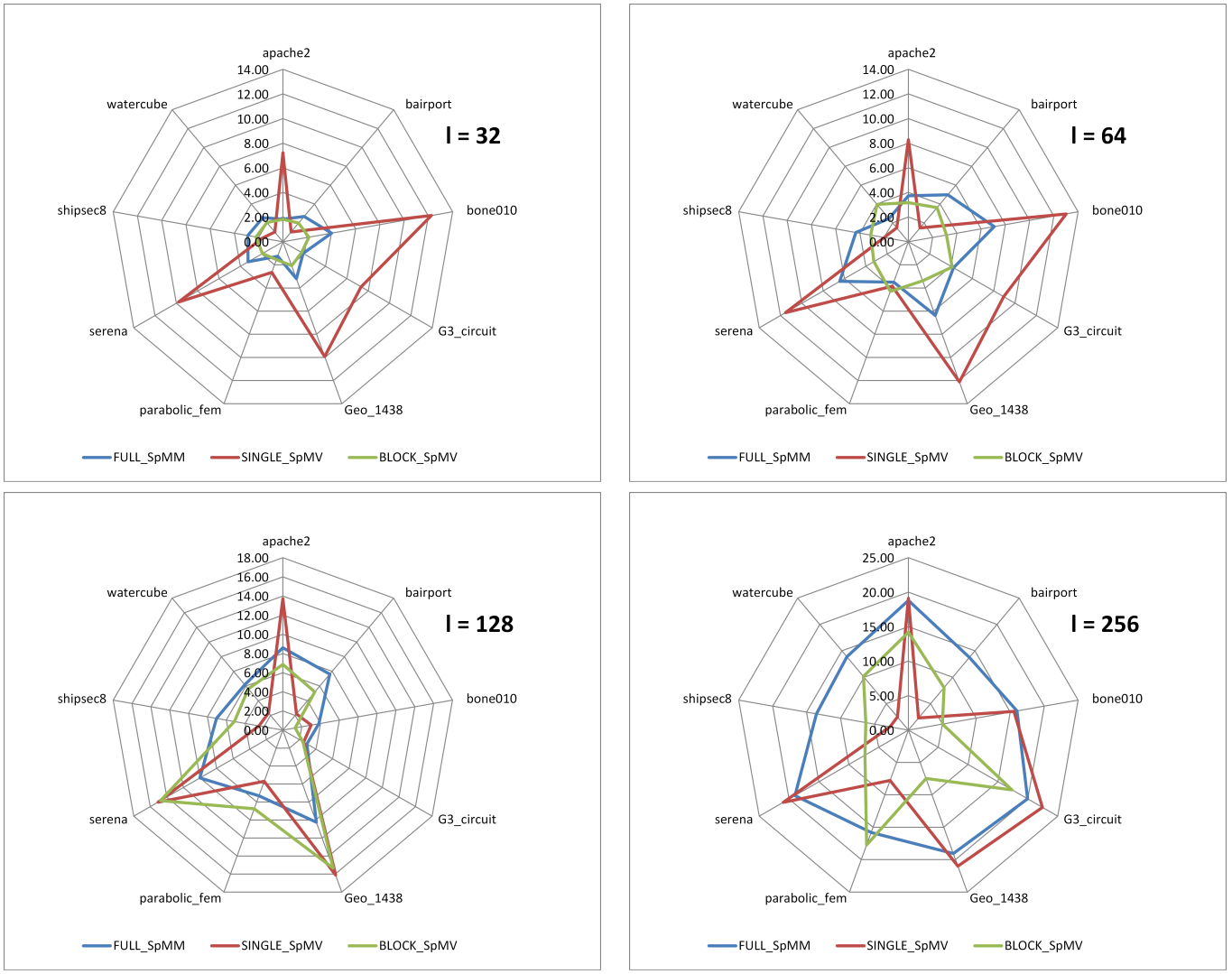


Figure 6.5: Execution rates for various ℓ on MIC. Performance in GFlops/sec.

because of the smaller cache sizes. For large ($n > 10^9$) matrices like G3_circuit and Geo_1438, SINGLE_SpMV is always more efficient, which is a result of the 512 KB L2 cache being too small to keep elements of more than 1 vector loaded, therefore multiplying more than 1 vector destroys any locality. For other matrices, SINGLE_SpMV is more efficient as ℓ gets smaller.

Our BLOCK_SpMV implementation is competitive on MIC for large ℓ but for smaller sizes of X, it is far slower than the fastest algorithm of the three. We use thread local storage for the column blocks Y_{jj} in Algorithm 6.18, with the expectation that memory is allocated on the closest controller, thereby reducing the latency; but we cannot be certain if this happens in practice. To better understand the reason for its slowness, we must analyse the cache misses for BLOCK_SpMV execution, an exercise we leave for [40]. The performance of any blocked algorithm is sensitive to the size of the block; in our experiments we used a fixed size of 8 but varying the blocksize to suit the sparsity pattern can improve performance.

Another factor that has a significant impact on the performance on MIC is the thread placement. The Xeon Phi has 4 in-order threads per core but using all 4 with a memory latency-bound computation increases contention. We therefore run 2 threads per core with a 'balanced' distribution, which placed threads 0 and 1 on core 1, threads 2 and 3 on core 2, and so on. Other thread placement strategies yield different results. The thread placement options together with variable block sizes offer a large search space for choosing the optimal runtime parameters to achieve the best performance with BLOCK_SpMV– tuning of these parameters will be looked at in our future work.

6.6 Conclusion

We have developed a new, stable Cholesky-based B-orthogonalization algorithm that overcomes the stability limitation using perturbations and iteration. For a random matrix X the algorithm is experimentally shown to have a loss of precision that is independent of $\kappa(B)$ and is no larger than a modest multiple of $O(u)$. In the worst case of $\kappa(B) = \kappa(X)$, our experiments show that the loss of orthogonality is proportional to $\kappa(B)$ but independent of $\kappa(X)$, similar to Gram-Schmidt with reorthogonalization. In all cases the

algorithm needs only 2 or 3 iterations.

The pivoted Cholesky-based algorithm has superior efficiency compared with Gram-Schmidt. In testing our multithreaded implementations for CPUs and Intel's MIC-based accelerator, we find the Cholesky implementations outperform Gram-Schmidt by a factor of up to 40 and 200 respectively.

Even though the Xeon Phi card has a peak `dgemm` performance of 1.1 TFlop/s, our kernels have been able to extract only up to 21 GFlops/s with manual vectorization. This contrasts with the 24 GFlop/s we achieve on an SSE-enabled CPU using just compiler auto-vectorization, which suggests that computations on Xeon Phi are memory-latency bound. Better performance is possible by using a different sparse matrix format [55],[44] to reduce bottlenecks arising from memory-latency overheads.

The incentive for choosing Intel's MIC accelerator over GPGPU-based accelerators is that we can use the same code base as the CPU to exploit the coprocessor. We find that while this is sufficient for getting an unoptimized version running on the MIC, such code may not necessarily return the best performance. Since tuning the program to extract good performance is a non-trivial task, the final code may diverge significantly from the CPU code, thereby negating the advantages of the single source base. Architectural choices that have been made in the design of Xeon Phi imply that several runtime parameters must have optimal values to get good performance, making the performance sensitive to the execution settings, especially when dealing with sparse matrix computations. Future iterations of the Intel MIC architecture [38] will feature low-powered cores sharing the same main memory as the CPU cores which will reduce the latency problems that restrict the performance of our algorithm.

Chapter 7

Summary

The conditioning of a problem that is solved numerically using floating point computing is critical for the accuracy and the reliability of the results. We developed a novel method called Model Stability Analysis (Alg. 3.1) to debug the causes of ill conditioning in stiffness matrices arising from finite element applications. This method has been implemented in the FE software package Oasys GSA and has been successfully used by practitioners to detect where in their models the cause of the ill conditioning lies. The existence of ill conditioning came to light because we introduced the estimation of the condition number of the stiffness matrix, a practise that we have not seen in other finite element codes both commercial and academic. A major source of ill conditioning is errors made by practitioners while creating FE models. The properties of the eigenvectors of ill-conditioned stiffness matrices led us back to the parts of the model that are badly defined. In our examples the condition numbers of stiffness matrices reduced to $O(10^8)$ from $O(10^{16})$ when the identified errors were fixed.

Although our method identifies elements that cause the ill conditioning, rectifying the errors is a manual task. The engineer must sift through all plausible causes to identify the remedy. An automatic, algorithmic fix for user errors is highly desirable and this is an open question that needs to be addressed in future. The applicability of our method to other finite element applications, for example in convection-diffusion problems or in nonlinear finite element analysis, is also of interest since user errors are a common occurrence wherever practitioners use software packages as black-boxes. In problems where the discretizations yield non-symmetric matrices, the singular vectors of

the smallest or largest singular values might be of more interest than the eigenvectors.

The subspace iteration method for sparse eigenvalue problems is popular within the engineering community because of its ease of implementation and robustness in finding eigenvalues in the desired range. The previous implementation of subspace iteration in GSA, Algorithm 4.2, was unoptimized and dated. Algorithms 4.3, 4.4 and 4.5, which have been implemented in the package, improved the numerical efficiency of the solution by an order of magnitude. They also increased the users' confidence in the results by using a robust shift strategy and by using a better convergence test (4.10) that related the convergence tolerance to the backward error in the solution. The speed improvements over the previous implementation were especially significant: about 10x for large problems with more than a million degrees of freedom and where a large number of eigenpairs were sought. By making these improvements to an existing implementation of subspace iteration we could take advantage of existing program codes wherever possible, thereby reducing the development and maintenance time.

It is possible to accelerate convergence further through other shift strategies like Chebyshev acceleration [54] and by using the inexact subspace iteration family of algorithms. The latter has the advantage of depending only on sparse matrix vector multiplication kernels for the entire iteration and significant speedups are possible, especially on emerging parallel architectures using different storage formats. However inexact methods need good preconditioners and real world problems are often ill-conditioned, as our experience demonstrates. Hence the challenge is to use these methods to develop algorithms that can be used without significant intervention from the user.

An important consideration while designing numerical algorithms for recent and emerging computer architectures is that the cost of moving data far exceeds the cost of performing flops on it. Therefore algorithms and their software implementations need to be economical with data movement and should seek to increase the computational intensity, i.e., the ratio of flops to bytes moved. Our new sparse matrix format, Mapped Blocked Row, combines the advantages of being a blocked format without any of the disadvantages that affect these formats, namely fill-in. In addition to a memory efficient storage format, a sparse matrix multiple-vector multiplication kernel for computing $Y = AX$ for a sparse A and a dense X also needs a fast algorithm to load the matrix and

vector entries. Our kernel is an implementation of Algorithms 5.9, 5.10 and 5.11 and it outperforms the Intel MKL sparse kernel by a factor of up to 2x on Intel processors and up to 3x on AMD processors. To minimize the conditionals evaluated, we iterate over only set bits in a bitmapped block using de Bruijn sequences. The kernel library uses C++ templates to parametrize scalar types, number of right hand vectors and the size of the blocks.

The natural extension of our sparse matrix kernel is an algorithm that takes advantage of multithreading. Since our blocked format has sparse blocks it lends itself well to variable blocking, i.e., using blocks of variable dimensions for different parts of a sparse matrix and this area is worthy of attention in future work. Yet another strand of investigation would be the applicability of the format to other sparse matrix operations such as sparse direct solvers.

Our investigation of B-orthogonalization algorithms is also motivated by the theme of developing cache-efficient algorithms that reduce data movement. The orthogonalization of basis vectors with respect to a nonstandard inner product crops up in structural analysis and current approaches use the Gram-Schmidt family of algorithms which rely on matrix-vector operations (level 2 BLAS). Our Cholesky-based B-orthogonalization approach involves both numerical and computational contributions. On the numerical front, we developed Algorithm 6.14 that is stable in practice and converges to a tolerable loss of orthogonality within two or three iterations, even though the Cholesky-based orthogonalization method is inherently unstable. We do so by using a combination of perturbing $O(u)$ diagonal entries and by iterating to reduce the residual of orthogonality $\|S^T BS - I\|_2$. On the computational front, our algorithm gains efficiency because it is rich in level 3 BLAS operations. Our multithreaded implementations for the CPU and Intel Xeon Phi coprocesor are fully vectorized using compiler intrinsics in the inner product loop of the kernel (Algorithm 6.19). As a result the stabilized Cholesky B-orthogonal implementation is 40 and 200 times faster than the Gram-Schmidt based implementations on the CPU and Xeon Phi respectively.

Future work lies in the area of improving the performance of the inner product kernel $X^T BX$ by using sparse matrix formats that are designed for the Xeon Phi architecture. However since Intel's next iteration of this architecture will be significantly different,

obtaining optimal performance on this platform might be a moving target.

Bibliography

[1] *IEEE Std 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 2008. 1-58 pp.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Third edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999. ISBN 0-89871-447-8 (paperback).

[3] Z. Bai and G.W. Stewart. SRRIT – a Fortran subroutine to calculate the dominant invariant subspace of a nonsymmetric matrix. Technical report, Department of Computer Science, University of Maryland, College Park, Maryland, USA., 1992.

[4] Z. Bai and G.W. Stewart. Algorithm 776: SRRIT: A Fortran subroutine to calculate the dominant invariant subspace of a nonsymmetric matrix. *ACM Trans. Math. Softw.*, 23:494–513, 1997.

[5] 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, Philadelphia, PA, USA, 2000.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994.

[7] Klaus-Jürgen Bathe. *Finite Element Procedures*. Prentice-Hall International, Englewood Cliffs, NJ, USA, 1996.

[8] Klaus-Jürgen Bathe and Seshadri Ramaswamy. An accelerated subspace method. *Comput. Methods in Appl. Mech. Engrg.*, 23:313–331, 1980.

[9] Klaus-Jürgen Bathe and Edward L. Wilson. Solution methods for eigenvalue problems in structural mechanics. *International Journal for Numerical Methods in Engineering*, 6(2):213–226, 1973.

[10] Friedrich L. Bauer. Das Verfahren der Treppeniteration und verwandte Verfahren zur Lösung algebraischer Eigenwertprobleme. *Zeitschrift für angewandte Mathematik und Physik ZAMP*, 8(3):214–235, 1957.

[11] Aydin Buluc, Samuel Williams, Lenny Oliker, and James Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *International Parallel & Distributed Processing Symposium (IPDPS)*, IEEE International, May 2011, pages 721–723.

[12] Sheung Hun Cheng and Nicholas J. Higham. Implementation for LAPACK of a block algorithm for matrix 1-norm estimation. Numerical Analysis Report No. 393, Manchester Centre for Computational Mathematics, Manchester, England, August 2001. 19 pp. LAPACK Working Note 152.

[13] Robert D. Cook, David S. Malkus, and Michael E. Plesha. *Concepts and Applications of Finite Element Analysis*. Third edition, Wiley, New York, 1989. ISBN 0471847887.

[14] James O. Coplien. *Multi-Paradigm Design*. PhD thesis, Faculteit Wetenschappen, Vrije Universiteit Brussel, July 2000. Available from `https://sites.google.com/a/gertrudandcope.com/info/Publications`.

[15] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[16] I. S. Duff and J. A. Scott. Computing selected eigenvalues of sparse unsymmetric matrices using subspace iteration. *ACM Transactions on Mathematical Software*, 19:137–159, 1993.

[17] W.J. Duncan and A.R. Collar. LXXIV. A method for the solution of oscillation problems by matrices. *Philosophical Magazine Series 7*, 17(115):865–909, 1934.

[18] Charbel Farhat and Michel Géradin. On the general solution by a direct method of a large-scale singular system of linear equations: application to the analysis of floating structures. *International Journal for Numerical Methods in Engineering*, 41 (4):675–696, 1998.

[19] C.A. Felippa. Introduction to Finite Element Methods. Available from http://www.colorado.edu/engineering/cas/courses.d/IFEM.d/. Course notes for the graduate course ASEN 5007: Finite Element Methods.

[20] C.A. Felippa. A historical outline of matrix structural analysis: a play in three acts. *Computers and Structures*, 79(14):1313–1324, 2001.

[21] Agner Fog. Optimizing software in C++. http://www.agner.org/optimize/optimizing_cpp.pdf, Feb 2012. Retrieved on August 07, 2012.

[22] I. Fried. Bounds on the extremal eigenvalues of the finite element stiffness and mass matrices and their spectral condition number. *Journal of Sound and Vibration*, 22 (4):407–418, 1972.

[23] Luc Giraud, Julien Langou, and Miroslav Rozložník. On the round-off error analysis of the Gram-Schmidt algorithm with reorthogonalization. Technical Report TR/PA/02/33, CERFACS, Toulouse, France, 2002.

[24] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Third edition, Johns Hopkins University Press, Baltimore, MD, USA, 1996. xxvii+694 pp. ISBN 0-8018-5413-X (hardback), 0-8018-5414-8 (paperback).

[25] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Fourth edition, Johns Hopkins University Press, Baltimore, MD, USA, 2013. xxi+756 pp. ISBN 978-1-4214-0794-4.

[26] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[27] Gaël Guennebaud, Benoît Jacob, et al. Benchmarking results for Eigen vs. other libraries. http://eigen.tuxfamily.org/index.php?title=Benchmark, July 2011.

[28] Stefan Güttel and Jennifer Pestana. Some observations on weighted GMRES. *Numerical Algorithms*, pages 1–20, 2014.

[29] Sven Hammarling, Nicholas J. Higham, and Craig Lucas. LAPACK-style codes for pivoted Cholesky and QR updating. In *Applied Parallel Computing. State of the Art in Scientific Computing. 8th International Workshop, PARA 2006*, Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, number 4699 in *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 2007, pages 137–146.

[30] V. Hernández, J.E. Román, A. Tomás, and V. Vidal. Lanczos Methods in SLEPc. Technical Report SLEPc Technical Report STR-5, Universidad Politecnica De Valencia, October 2006. Available at `http://www.grycap.upv.es/slepc/documentation/reports/str5.pdf`.

[31] Desmond J. Higham and Nicholas J. Higham. Structured backward error and condition of generalized eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 20(2):493–512, 1998.

[32] Nicholas J. Higham. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation (Algorithm 674). *ACM Trans. Math. Software*, 14(4):381–396, 1988.

[33] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. xxx+680 pp. ISBN 0-89871-521-0.

[34] Nicholas J. Higham and Françoise Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM J. Matrix Anal. Appl.*, 21(4):1185–1201, 2000.

[35] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. Expression Templates Revisited: A Performance Analysis of Current Methodologies. *SIAM J. Sci. Comput.*, 34(2):42–69, 2012.

[36] Eun-Jin Im and Katherine A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on*

Computational Science, volume 2073 of *LNCS*, San Francisco, CA, May 2001, pages 127–136. Springer.

[37] Intel Corporation. Intel Math Kernel Library (Intel MKL). http://software.intel.com/en-us/articles/intel-mkl/, 2012. Version 10.3 Update 7.

[38] Intel Corporation. Intel re-architects the fundamental building block for high-performance computing. http://bit.ly/intel-knightslanding-announce, June 2014.

[39] Hyung-Jo Jung, Man-Cheol Kim, and In-Won Lee. An improved subspace iteration method with shifting. *Computers & Structures*, 70(6):625–633, 1999.

[40] Ramaseshan Kannan and Yuji Nakatsukasa. Cache-efficient B-orthogonalization using Cholesky factorization and its parallel implementation for manycore architectures. *In preparation*, Aug. 2014.

[41] Matthias Kretz and Volker Lindenstruth. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, 2012.

[42] Benjamin C. Lee, Richard W. Vuduc, James W. Demmel, Katherine A. Yelick, Michael deLorimier, and Lijue Zhong. Performance optimizations and bounds for sparse symmetric matrix-multiple vector multiply. Technical Report UCB/CSD-03-1297, University of California, Berkeley, CA, USA, November 2003.

[43] Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de Bruijn Sequences to Index a 1 in a Computer World. Technical report, MIT, July 1998. Unpublished manuscript, available from http://supertech.csail.mit.edu/papers/debruijn.pdf.

[44] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, New York, NY, USA, 2013, pages 273–282. ACM.

[45] Bradley R. Lowery and Julien Langou. Stability analysis of QR factorization in an Oblique Inner Product. *arXiv preprint arXiv:1401.5171*, 2014. Available from http://arxiv.org/abs/1401.5171.

[46] Craig Lucas. *Algorithms for Cholesky and QR factorizations, and the semidefinite generalized eigenvalue problem*. PhD thesis, School of Mathematics, University of Manchester, Manchester, England, 2004.

[47] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, New York, NY, USA, 2004, pages 162–167. ACM.

[48] Microsoft Corporation. *Visual C++ Language reference, available from http://msdn.microsoft.com/en-us/library/vstudio/3bstk3k5(v=vs.100).aspx*, 2011. Retrieved on January 26, 2012.

[49] Oasys Limited. Oasys GSA, available from http://www.oasys-software.com/gsa. Retrieved on January 20, 2012.

[50] M. Papadrakakis and Y. Fragakis. An integrated geometric–algebraic method for solving semi-definite problems in structural mechanics. *Comput. Methods in Appl. Mech. Engrg.*, 190(49-50):6513 – 6532, 2001.

[51] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, New York, NY, USA, 1999. ACM.

[52] Miroslav Rozložník, Miroslav Tůma, Alicja Smoktunowicz, and Jiří Kopal. Numerical stability of orthogonalization methods with a non-standard inner product. *BIT Numerical Mathematics*, 52(4):1035–1058, 2012.

[53] H. Rutishauser. Simultaneous iteration method for symmetric matrices. *Numerische Mathematik*, 16:205–223, 1970.

[54] Youcef Saad. Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems. *Mathematics of Computation*, 42(166):pp. 567–588, 1984.

[55] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. In *Proc of the 10th Int'l Conf. on Parallel Processing and Applied Mathematics (PPAM)*, September 2013, page 10. (to appear).

[56] O. Schenk and K. Gartner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electron. Trans. Numer. Anal.*, 23:158–179, 2006.

[57] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.*, 20:475–487, 2004.

[58] Gil Shklarski and Sivan Toledo. Computing the null space of finite element problems. *Comput. Methods in Appl. Mech. Engrg.*, 198(37-40):3084–3095, 2009.

[59] Oasys Software. *GSA Version 8.6 reference manual.* Arup, 13 Fitzroy Street London W1T 4BQ, 2012. Available from `http://www.oasys-software.com/media/Manuals/Latest_Manuals/gsa8.6_manual.pdf`.

[60] G. W. Stewart. Simultaneous iteration for computing invariant subspaces of non-Hermitian matrices. *Numer. Math.*, 25:123–136, 1976.

[61] G. W. Stewart. *Matrix Algorithms. Volume I: Basic Decompositions.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[62] G. W. Stewart. *Matrix Algorithms. Volume II: Eigensystems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. xix+469 pp. ISBN 0-89871-503-2.

[63] Numerical Analysis Group STFC Rutherford Appleton Laboratory. Hsl, a collection of fortran codes for large-scale scientific computation. `http://www.hsl.rl.ac.uk/index.html`.

[64] Bjarne Stroustrup. Why C++ is Not Just an Object-oriented Programming Language. In *Addendum to the Proceedings of the 10th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '95, New York, NY, USA, 1995, pages 1–13. ACM.

[65] Bjarne Stroustrup. *The C++ Programming Language*. Fourth edition, Addison-Wesley, 2013. xiv+1346 pp. ISBN 978-0-321-56384-2.

[66] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–726, 1997.

[67] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[68] A.J. Wathen. An analysis of some element-by-element techniques. *Comput. Methods in Appl. Mech. Engrg.*, 74(3):271–287, 1989.

[69] David S. Watkins. *The Matrix Eigenvalue Problem*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.

[70] David S. Watkins. *Fundamentals of Matrix Computations*. Third edition, Wiley, New York, 2010.

[71] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[72] Fei Xue and Howard C. Elman. Fast inexact subspace iteration for generalized eigenvalue problems with spectral transformation. *Linear Algebra and its Applications*, 435(3):601–622, 2011.

[73] Qiang Ye and Ping Zhang. Inexact inverse subspace iteration for generalized eigenvalue problems. *Linear Algebra and its Applications*, 434(7):1697–1715, 2011.

[74] Qian-Cheng Zhao, Pu Chen, Wen-Bo Peng, Yu-Cai Gong, and Ming-Wu Yuan. Accelerated subspace iteration with aggressive shift. *Computers & Structures*, 85 (19-20):1562–1578, 2007.