NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PRECONDITIONING THE CONJUGATE GRADIENT METHOD FOR A HYBRIDIZED SUMMATION-BY-PARTS FINITE-DIFFERENCE DISCRETIZATION OF POISSON'S EQUATION WITH APPLICATION TO SEISMIC SIMULATIONS

by

Timothy P. James

June 2022

Thesis Advisor: Anthony Austin
Second Reader: Lucas C. Wilcox

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
PRECONDITIONING THE CONJUGATE GRADIENT METHOD FOR A HYBRIDIZED SUMMATION-BY-PARTS FINITE-DIFFERENCE DISCRETIZATION OF POISSON'S EQUATION WITH APPLICATION TO SEISMIC SIMULATIONS

5. FUNDING NUMBERS

6. AUTHOR(S) Timothy P. James

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

We consider the solution by the conjugate gradient (CG) method of the linear systems arising from a hybridized summation-by-parts finite-difference discretization of a Poisson problem that occurs in the simulation of seismic faults. We study the efficacy of three different preconditioning schemes—Jacobi, incomplete Cholesky, and algebraic multigrid—at reducing the number of CG iterations required for convergence, finding that while all three preconditioners are effective, the latter two are superior. Code for all of our studies, written in the Julia programming language, is provided.

14. SUBJECT TERMS
preconditioning, conjugate gradient, CG

15. NUMBER OF PAGES
83

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

PRECONDITIONING THE CONJUGATE GRADIENT METHOD FOR A HYBRIDIZED SUMMATION-BY-PARTS FINITE-DIFFERENCE DISCRETIZATION OF POISSON'S EQUATION WITH APPLICATION TO SEISMIC SIMULATIONS

Timothy P. James
Commander, United States Navy
BS, University of Alaska, Anchorage, 2004
MEng, Massachusetts Institute of Technology, 2012

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2022

Approved by: Anthony Austin
Advisor

Lucas C. Wilcox
Second Reader

Francis X. Giraldo
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

We consider the solution by the conjugate gradient (CG) method of the linear systems arising from a hybridized summation-by-parts finite-difference discretization of a Poisson problem that occurs in the simulation of seismic faults. We study the efficacy of three different preconditioning schemes—Jacobi, incomplete Cholesky, and algebraic multigrid—at reducing the number of CG iterations required for convergence, finding that while all three preconditioners are effective, the latter two are superior. Code for all of our studies, written in the Julia programming language, is provided.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Model Problem . . . 2
1.3 Discretization . . . 3
1.4 Area of Focus . . . 6

2 The Preconditioned Conjugate Gradient Method **7**
2.1 Conjugate Gradient Method . . . 7
2.2 Preconditioning . . . 9

3 Numerical Experiments and Results **15**
3.1 Experimental Setup . . . 15
3.2 Iterations . . . 16
3.3 Eigenvalues . . . 18
3.4 Runtime . . . 22

4 Conclusions / Future Research Areas **27**
4.1 Conclusions . . . 27
4.2 Future Research Areas . . . 27

Appendix A Modified Kozdon et al. Julia Code **29**

Appendix B Conjugate Gradient Implementation Julia Code **43**
B.1 Conjugate Gradient Julia Code . . . 43
B.2 Preconditioned Conjugate Gradient Julia Code . . . 46

Appendix C Julia Code Constructing Preconditioners **49**
C.1 Jacobi Julia Code . . . 49

C.2 Incomplete Cholesky Julia Code 50
C.3 Algebraic Multigrid Julia Code 51

Appendix D Tabular Results **53**

Appendix E Julia Plotting Code **59**

List of References **61**

Initial Distribution List **65**

List of Figures

Figure 1.1 Model domain 3

Figure 1.2 Sparsity pattern for the Equation (1.2) system matrix of dimension $n = 19,872$ 4

Figure 1.3 Sparsity 6

Figure 3.1 Number of iterations required for convergence 17

Figure 3.2 Eigenvalues of trace Schur complement 19

Figure 3.3 Eigenvalues of displacements Schur complement 20

Figure 3.4 Eigenvalues of monolithic system 21

Figure 3.6 Seconds required for convergence 24

Figure 3.5 Comparison of runtime versus matrix dimension 25

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 2.1 Diagonality 12

Table 2.2 Non-zero entry comparison 13

Table 3.1 System size of each method (n) 15

Table 3.2 Iteration reduction 18

Table 3.3 Preconditioning effects on κ 22

Table 3.4 Trace Schur complement solve domination 22

Table D.1 Results for trace method 53

Table D.2 Results for displacements method 54

Table D.3 Results for monolithic method 55

Table D.4 Results for trace method using tolerance = 10^{-16} and infinite-norm 56

Table D.5 Results for displacements method using tolerance = 10^{-16} and infinite-norm 57

Table D.6 Results for monolithic method using tolerance = 10^{-16} and infinite-norm 58

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AMG algebraic multigrid

CG conjugate gradient

DG discontinuous Galerkin

IC incomplete Cholesky

NPS Naval Postgraduate School

PDE partial differential equation

SAT simultaneous approximation term

SBP summation-by-parts

USN U.S. Navy

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I'd like to express my appreciation to those who made it possible for me to earn a master's degree in Applied Mathematics, which has culminated in this thesis; the fulfillment of a life-long goal.

Without the support of the commanding officers of Naval Support Activity Monterey, **CAPT Rich Wiley** and **CAPT Paul Dale**, I would not have been allowed to attend the courses required for graduation. I am truly grateful for your support of my desire to further my education, even though doing so took me away from my duties as one of your department heads. Your confidence and trust in my abilities and tremendous leadership has been a great blessing to me.

Bard Mansager was instrumental in providing guidance on a degree plan that capitalized on my previous experience and education and that could fit into my busy schedule as the public works officer on the base. You not only provided sage advice, you were also a joy to talk with and always kept the mood light even when we were making decisions that would have significant impacts on my life. Thank you for your encouragement!

My deputy public works officer, **Matt Suess**, unfailingly covered my duties while I was in class. Your expertise, knowledge of the installation, and sound judgement are impeccable. I trust you completely and that allowed me to focus on learning while in class. I am tremendously grateful for your example, mentorship, and friendship. You have helped me become a better leader and naval officer.

Jeremy Kozdon suggested a topic that was interesting and provided advice and guidance throughout my research. I could not have completed my Julia code without your assistance. Thank you for your patience through numerous office calls and e-mails! Even after you left NPS you provided important feedback that improved my thesis. I learned a tremendous amount and wish you the best of luck to you adventures away from NPS.

I could not have completed this thesis without **Anthony Austin**'s instruction, guidance,

proofreading, and editing. Your critical eye and keen observations were essential in producing a final product I am proud of. Thank you for the hours and hours of discussion which helped (1) improve the clarity of the writing, and (2) deepen my understanding of key mathematical concepts and how to communicate a clear and compelling story. It has been a rewarding experience.

A big thanks to my second reader, **Lucas Wilcox**, for providing a balanced and essential different perspective. Your keen and insightful suggestions were very helpful in improving the clarity and accuracy of this thesis.

I thoroughly enjoyed learning from the NPS Applied Mathematics instructors. You are excellent educators and interesting people. Given the opportunity, I would enroll in more courses taught by you. Special thanks to **Frank Giraldo**, **Ralucca Gera**, **Jeremy Kozdon**, **David Canright**, and **Guillermo Owen**.

Most importantly, my wife, **Yuko**. You have always supported me in my efforts to better myself and pushed me to reach beyond my limits. You make me a better person. I love you and thank God we are on this journey together!

CHAPTER 1:
Introduction

1.1 Problem Statement

In the last five years alone, U.S. military installations have incurred over $10 billion in damage from natural disasters [1]. For example, in July 2019, there were two earthquakes near Naval Air Weapons Station China Lake that caused extensive damage [2]. Because China Lake is home to one-of-a-kind capabilities for our military, with weapons testing and research that cannot be replicated elsewhere, in December 2019, Congress appropriated $2.9 billion in federal funding for rebuilding efforts [3]. If we are to minimize the impacts of devastating seismic events like the one at China Lake, we must improve our understanding of earthquake processes.

While most of the scientific community agree predictive earthquake models will forever remain out of reach, there are promising efforts underway aimed at better characterization of possible earthquakes and improving our understanding of seismic hazards resulting from long-term cycles of earthquakes [4], [5], [6]. This is accomplished by using computational models to simulate sequences of earthquakes. Through these simulations we look to better understand how faults interact; how the geometry of a fault (bends, branches, etc.) impacts the long-term behavior of a fault system; how non-uniform fault compositions affect development of fault stress; etc.

These simulation models must generate credible and reproducible results. In particular, the model must possess sufficient resolution for results based on it to be accurate. High-resolution models can be expensive to simulate. Accordingly, it is important not only to have high-fidelity models but also to ensure the numerical algorithms used for simulating them be efficient and scalable.

Our aim in this thesis is to explore a model problem that arises in forward simulations of seismic faults. Seismic simulations target one of two phases of the earthquake cycle: the interseismic phase (tens to hundreds of years between earthquakes) or the coseismic phase (seconds to minutes of active sliding of the fault—an earthquake). We will focus on the

interseismic phase. In this phase, acceleration in the medium is ignored. We will take an existing method for the interseismic problem from the literature and examine how it might be adapted to efficiently handle calculations at high resolutions by replacing one of its components that does not scale with one that has superior scalability.

1.2 Model Problem

If we start with the scalar wave equation in 2nd order form and set acceleration to zero, the resulting model problem, in which time is not present, is the following Poisson problem for a function u (the anti-plane fault displacement) in two spatial dimensions, taken from [7]:

$$\begin{aligned} -\nabla^2 u &= f && \text{on } \Omega, \\ u &= g_D && \text{on } \partial\Omega_D, \\ \mathbf{n}\cdot\nabla u &= g_N && \text{on } \partial\Omega_N, \\ \left.\begin{aligned} \{\{\mathbf{n}\cdot\nabla u\}\} &= 0 \\ [\![u]\!] &= \delta \end{aligned}\right\} & && \text{on } \Gamma_I. \end{aligned} \tag{1.1}$$

Here, $f(x, y)$ is a scalar source function. The domain boundary $\partial\Omega$ is partitioned into disjoint Dirichlet ($\partial\Omega_D$) and Neumann ($\partial\Omega_N$) segments. The functions g_D and g_N are the Dirichlet and Neumann boundary data, respectively. The vector $\mathbf{n}$ is the outward pointing unit normal, and the curve Γ_I is an internal interface introduced to model a fault, across which the normal component of ∇u is required to be continuous but there is a jump in the solution u by an amount δ. The symbols $\{\{w\}\}$ and $[\![w]\!]$ denote the sum and difference of the values of a scalar quantity w on either side of the interface, respectively.

We will solve Equation (1.1) on the square domain $\Omega = [-2, 2] \times [-2, 2]$ depicted in Figure 1.1. The thick red unit circle is the interface Γ_I, and the thin black lines depict the borders between blocks in the curvilinear quadrilateral mesh over which our discretization method will be applied; see the next section. The right and left sides of the square constitute the Dirichlet boundary and the top and bottom sides of the square, the Neumann boundary. To generate the source and boundary data, we use a manufactured solution taken from

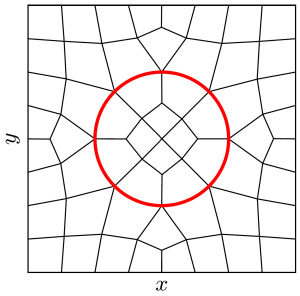


Figure 1.1. Domain. Source: [7]

Equation (22) of [7],

$$u(x,y) = \begin{cases} \frac{e}{1+e}(2 - e^{-r^2})r\sin\theta, & (x,y) \in \Omega_1, \\ (r-1)^2\cos\theta + (r-1)\sin\theta, & (x,y) \in \Omega_2, \end{cases}.$$

where $r = \sqrt{x^2+y^2}$ and $-\pi \leq \theta = \tan^{-1}(y/x) < \pi$, and Ω_1, Ω_2 denote the inside and the outside of the circle, respectively.

1.3 Discretization

We discretized Equation (1.1) using the combined summation-by-parts (SBP) finite difference method and simultaneous approximation term (SAT) method described in [7]. This method uses a (mapped) finite difference grid to discretize the Laplacian locally within each block of the mesh. Degrees of freedom belonging to grids of adjacent blocks are then coupled using SATs *a la* the numerical fluxes in a discontinuous Galerkin (DG) method [8]. This is a hybrid method in the sense that we build u from two unknown functions, one defined on the block volumes (interiors) and one defined on the block traces (boundaries). This leads to a linear system with the structure of Equation (1.2), which is amenable to Schur complement techniques (i.e., solving for the trace and volume functions independently of one another). As detailed in [9], [10], [11], [12], methods of this type have been particularly effective for simulating seismic faults, since interface conditions can be imposed weakly [7]. SBP-SAT methods have the same discrete structure as tensor product DG methods, except the derivative operators are banded finite difference matrices rather than spectral operators (i.e., not all the points are used to compute the derivatives at a given

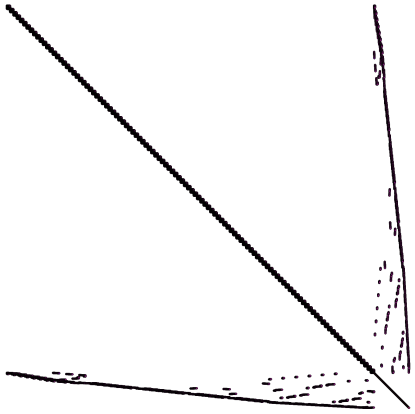


Figure 1.2. Sparsity pattern for the Equation (1.2) system matrix of dimension $n = 19,872$

point).

Applying this discretization to Equation (1.1) over the mesh depicted in Figure 1.1 results in a symmetric positive-definite linear system of the form

$$\begin{bmatrix} \bar{\mathbf{M}} & \bar{\mathbf{F}} \\ \bar{\mathbf{F}}^T & \bar{\mathbf{D}} \end{bmatrix} \begin{bmatrix} \bar{\mathbf{u}} \\ \bar{\boldsymbol{\lambda}} \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{g}} \\ \bar{\mathbf{g}}_\delta \end{bmatrix}, \tag{1.2}$$

where the vector $\bar{\mathbf{u}}$ contains the *volume variables* (i.e., the approximations to the values of the solution at the finite difference grid points within each mesh block), and the vector $\bar{\boldsymbol{\lambda}}$ contains the *trace variables* (i.e., the approximations to the solution at points along the interfaces between blocks). The vectors $\bar{\mathbf{g}}$ and $\bar{\mathbf{g}}_\delta$ that comprise the right-hand side incorporate the boundary data and source terms, and the interface data, respectively.

Figure 1.2 shows the non-zero structure of the matrix on the left-hand side of Equation (1.2) for a particular choice of the resolution of the finite difference grid. The block structure is readily apparent. Moreover, the matrix is extremely sparse. The $\bar{\mathbf{M}}$ block is block diagonal with banded blocks, and the $\bar{\mathbf{D}}$ block is diagonal.

At the high grid resolutions demanded by applications, the system Equation (1.2) can be large and thus expensive to solve. To reduce the size of these systems, the authors of [7] introduced a Schur complement technique. The idea is to perform block Gaussian elimination on Equation (1.2) to eliminate either the volume variables or the trace variables,

leaving a smaller linear system for the remaining variables that were not eliminated. Since either set of variables can be eliminated, we have three approaches to solving Equation (1.2):

- **Monolithic** – We solve Equation (1.2) as-is without eliminating variables.
- **Trace** – We solve the first block equation in Equation (1.2) for the volume variables, obtaining

$$\bar{\mathbf{u}} = \bar{\mathbf{M}}^{-1}\bar{\mathbf{g}} - \bar{\mathbf{M}}^{-1}\bar{\mathbf{F}}\bar{\boldsymbol{\lambda}}. \tag{1.3}$$

  Substituting this into the second block equation then yields the Schur complement system for the trace variables,

$$(\bar{\mathbf{D}} - \bar{\mathbf{F}}^T\bar{\mathbf{M}}^{-1}\bar{\mathbf{F}})\bar{\boldsymbol{\lambda}} = \bar{\mathbf{g}}_\delta - \bar{\mathbf{F}}^T\bar{\mathbf{M}}^{-1}\bar{\mathbf{g}}. \tag{1.4}$$

  We first solve Equation (1.4) for $\bar{\boldsymbol{\lambda}}$ and then compute $\bar{\mathbf{u}}$ from Equation (1.3). As $\bar{\mathbf{M}}$ is block diagonal, the latter can be done extremely cheaply. The bulk of the work goes into solving the Schur complement system, which nevertheless is considerably smaller than the original system, as it only involves unknowns along the interfaces between blocks.
- **Displacements** – We solve the second block equation in Equation (1.2) for the trace variables, giving

$$\bar{\boldsymbol{\lambda}} = \bar{\mathbf{D}}^{-1}\bar{\mathbf{g}}_\delta - \bar{\mathbf{D}}^{-1}\bar{\mathbf{F}}^T\bar{\mathbf{u}}. \tag{1.5}$$

  Substituting this into the first block equation then yields the Schur complement system for the volume variables,

$$(\bar{\mathbf{M}} - \bar{\mathbf{F}}\bar{\mathbf{D}}^{-1}\bar{\mathbf{F}}^T)\bar{\mathbf{u}} = \bar{\mathbf{g}} - \bar{\mathbf{F}}\bar{\mathbf{D}}^{-1}\bar{\mathbf{g}}_\delta. \tag{1.6}$$

  We first solve Equation (1.6) for $\bar{\mathbf{u}}$ and then compute $\bar{\boldsymbol{\lambda}}$ from Equation (1.5). As with the trace method, the bulk of the work for this method goes into solving the Schur complement system. This method also benefits from a reduction in system size, though not to the same extent as the trace method.

Even though they are formed from products of the blocks of Equation (1.2), the Schur complements $\bar{\mathbf{D}} - \bar{\mathbf{F}}^T\bar{\mathbf{M}}^{-1}\bar{\mathbf{F}}$ and $\bar{\mathbf{M}} - \bar{\mathbf{F}}\bar{\mathbf{D}}^{-1}\bar{\mathbf{F}}^T$ are still quite sparse. Figure 1.3 depicts their sparsity patterns.

(a) (b)

Figure 1.3. Sparsity patterns for the Schur complements of the trace (a) and displacements (b) methods. Note that while the plots are the same size, the Schur complement for the trace method is significantly smaller than that for the displacements method; they have dimensions $n = 1,728$ and $n = 18,144$, respectively.

1.4 Area of Focus

In [7], the authors used a direct method to solve the linear systems that arise in each of these three approaches. This worked well enough for their purposes; however, at sufficiently large scales, direct solvers are not a viable option, especially for problems in three spatial dimensions. Our focus in this thesis is to explore developing iterative approaches to solving these systems that will allow scaling up to larger problem sizes (i.e., finer discretizations and higher accuracy).

As the matrix in Equation (1.2) is symmetric positive-definite, the natural iterative method to consider is the conjugate gradient (CG) method. In Chapter 2, we review the basic properties of this method, including the need for preconditioning and the basic properties of three common preconditioners: Jacobi, incomplete Cholesky (IC), and algebraic multigrid (AMG). In Chapter 3, we apply the CG method with these three preconditioners to the solution of our model problem and compare the efficacy of each. Finally, in Chapter 4, we discuss conclusions and suggested future research areas.

CHAPTER 2:
The Preconditioned Conjugate Gradient Method

2.1 Conjugate Gradient Method

The CG method is perhaps the most widely used method for solving $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is large, sparse, real, symmetric, and positive-definite. The basic form of this method, taken from [13], is presented in Algorithm 2.1. Since the linear systems in Equations (1.1), (1.4), and (1.6) are symmetric positive-definite, CG is a natural choice for solving them. Our Julia implementation of this algorithm can be found in Appendix B.

CG can be viewed as an optimization method for minimizing the quadratic form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c \tag{2.1}$$

over $\mathbf{x} \in \mathbb{R}^n$, where n is the dimension of $\mathbf{A}$. Since

$$\nabla f = \mathbf{Ax} - \mathbf{b},$$

Algorithm 2.1: The Conjugate Gradient Method

Input: $\mathbf{A}$, $\mathbf{b}$, initial guess $\mathbf{x}^{(0)}$
Output: Approximate solution to $\mathbf{Ax} = \mathbf{b}$

$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{Ax}^{(0)}$, $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$
for $k = 0, 1, 2, \ldots$ *until convergence* **do**
  $\alpha_k = \dfrac{\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)} \rangle}{\langle \mathbf{Ap}^{(k)}, \mathbf{p}^{(k)} \rangle}$
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$
  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{Ap}^{(k)}$
  if *converged* **then**
    return $\mathbf{x}^{(k+1)}$
  $\beta_k = \dfrac{\langle \mathbf{r}^{(k+1)}, \mathbf{r}^{(k+1)} \rangle}{\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)} \rangle}$
  $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta_k \mathbf{p}^{(k)}$

at the minimum, we have $\mathbf{Ax} - \mathbf{b} = 0$ (i.e., $\mathbf{Ax} = \mathbf{b}$).

The most basic method for minimizing 2.1 is the *steepest descent method*. Steepest descent seeks a solution by selecting a starting point and iteratively moving to a local minimum by following the direction of greatest slope. Unfortunately, this method converges too slowly for serious use. The following theorem from [14] provides an explicit bound on the number of iterations required to achieve a given reduction in the magnitude of the error using this method. The error is measured using the so-called $\mathbf{A}$-*norm*, defined as $\|\mathbf{x}\|_A = \mathbf{x}^T\mathbf{Ax} \in \mathbb{R}^n$.

Theorem 1 *Let $\mathbf{x}_i$ be the i^{th} steepest descent approximation to the exact solution $\mathbf{x}$, starting from an initial guess x_0, and define the errors as $\mathbf{e}_i := \mathbf{x}_i - \mathbf{x}$. Let $\mathbf{A}$ be positive-definite with $\lambda_n := \lambda_{\min}(\mathbf{A})$, $\lambda_1 := \lambda_{\max}(\mathbf{A})$ and 2-norm condition number $\kappa = \kappa(\mathbf{A}) = \lambda_1/\lambda_n$. Then,*

$$\|e_i\|_{\mathbf{A}} \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^i \|e_0\|_{\mathbf{A}}.$$

For example, if we wanted to reduce our error by just one order of magnitude (i.e., $\|e_i\|_{\mathbf{A}}/\|e_0\|_{\mathbf{A}} = 1/10$), for a matrix with $\kappa = 1000$ (a modest number) it would take approximately $\log(1/10)/\log(999/1001) \approx 1151$ iterations.

The reason for this appallingly slow rate of convergence is because as we get closer to the solution, the steepest descent directions all tend to point in the same general direction (i.e., the method does not explore enough of the space to be efficient). The CG method improves on steepest descent by demanding the directions in which we step be $\mathbf{A}$-*orthogonal*: orthogonal with respect to the inner product $\langle x, y\rangle_A = y^T\mathbf{A}x$. The idea is to prevent the search directions from overlapping too much with one other. The result is an improved convergence rate [14].

Theorem 2 *Let $\mathbf{x}_i$ be the i^{th} conjugate gradient approximation to the exact solution $\mathbf{x}$, starting from an initial guess x_0, and define the errors as $\mathbf{e}_i := \mathbf{x}_i - \mathbf{x}$. Let $\mathbf{A}$ be positive-definite with $\lambda_n := \lambda_{\min}(\mathbf{A})$, $\lambda_1 := \lambda_{\max}(\mathbf{A})$ and the 2-norm condition number $\kappa = \kappa(\mathbf{A}) = \lambda_1/\lambda_n$. Then,*

$$\|e_i\|_{\mathbf{A}} \leq 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^i \|e_0\|_{\mathbf{A}}.$$

Using our same example, if we wanted to reduce our error by one order of magnitude

(i.e., $\|e_i\|_{\mathbf{A}}/\|e_0\|_{\mathbf{A}} = 1/10$), for a matrix with $\kappa = 1000$ this method would take approximately $\log(1/20)/\log\left((\sqrt{1000}-1)/(\sqrt{1000}+1)\right) \approx 47$ iterations. That's a 96% reduction in iterations compared to steepest descent!

It turns out that CG actually converges to the exact solution if run to n iterations [14]. In practice, however, this is never done due to the excessive time required to do this for real-world problems (very large n).

2.2 Preconditioning

Even with the improvement in convergence rate realized with CG (over steepest descent), it is still too slow for many problems of practical interest. For instance, per a back-of-the-envelope calculation of the same type as those performed in the previous section, if κ is on the order of 10^6 (far from unheard of in real-world problems), CG will take over 1000 steps to reduce the $\mathbf{A}$-norm of the error by a factor of 10 and many more to reach convergence starting from a general initial guess.

The remedy for this is *preconditioning*. The idea is to replace the system of equations we are given with another one that has the same solution but for which CG converges more rapidly. Specifically, we will employ *left-preconditioning*, which replaces $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b} \tag{2.2}$$

for some matrix $\mathbf{M}$.

How should we pick $\mathbf{M}$? Naturally, we want to reduce the number of CG steps needed for convergence, and to accomplish this, Theorem 2 suggests that we should take $\mathbf{M}$ to minimize $\kappa(\mathbf{M}^{-1}\mathbf{A})$. If we could, we would choose $\mathbf{M} = \mathbf{A}$, which would make $\kappa(\mathbf{M}^{-1}\mathbf{A}) = 1$, and CG would then converge in one step, but building this $\mathbf{M}$ would be just as expensive as solving our original system. More generally, it can be shown that CG converges to the exact solution in no more than m iterations, where m is the number of distinct eigenvalues of $\mathbf{A}$ [14]. Thus, all we actually need to do is construct $\mathbf{M}$ so that the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$ are clustered; we do not actually need to reduce κ to achieve fast convergence [13].

Additionally, we need to choose $\mathbf{M}$ so that $\mathbf{M}^{-1}$ can be computed efficiently (i.e., so that

we can easily solve systems of the form $\mathbf{Mz} = \mathbf{r}$). This is necessary to prevent each preconditioned CG iteration from becoming too expensive in terms of both time and computer memory. It does no good to reduce the number of iterations needed to reach convergence by a factor of 10 only for each iteration to become 10 times as expensive.

There is just one remaining issue: The CG method only works for symmetric positive-definite matrices, but $\mathbf{M}^{-1}\mathbf{A}$ need be neither symmetric nor positive-definite even if $\mathbf{M}$ and $\mathbf{A}$ both are. One way to fix this is to perform *two-sided preconditioning* in which $\mathbf{M}$ is taken to be symmetric positive-definite and we factor it in a Cholesky decomposition as $\mathbf{M} = \mathbf{H}\mathbf{H}^T$. We then consider the system

$$\mathbf{H}^{-1}\mathbf{A}\mathbf{H}^{-T}\mathbf{v} = \mathbf{H}^{-1}\mathbf{b}, \text{ where } \mathbf{v} = \mathbf{H}^T\mathbf{x}. \tag{2.3}$$

Because the matrix $\mathbf{H}^{-1}\mathbf{A}\mathbf{H}^{-T}$ is symmetric positive-definite, CG can be applied. Unfortunately, it is not always convenient to have to factor $\mathbf{M}$ in this way. We can dispense with this by a clever rewriting of the algorithm, detailed in [13]. The outcome is the algorithm given in Algorithm 2.2, and our implementation of this algorithm is in Appendix B.

To summarize, we need $\mathbf{M}$ to be

1. symmetric positive-definite,
2. easy to invert, and
3. a good approximation to $\mathbf{A}$.

By the third statement, we mean $\kappa(\mathbf{M}^{-1}\mathbf{A})$ is small, or at least the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$ are clustered.

Algorithm 2.2: The Preconditioned Conjugate Gradient Method

Input: $\mathbf{A}$, $\mathbf{b}$, preconditioner $\mathbf{M}$, initial guess $\mathbf{x}^{(0)}$

Output: Approximate solution to $\mathbf{Ax} = \mathbf{b}$

$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{Ax}^{(0)}$; solve $M\mathbf{z}^{(0)} = \mathbf{r}^{(0)}$; $\mathbf{p}^{(0)} = \mathbf{z}^{(0)}$

for $k = 0, 1, 2, \ldots$ *until convergence* **do**

  $\alpha_k = \dfrac{\langle \mathbf{z}^{(k)}, \mathbf{r}^{(k)} \rangle}{\langle \mathbf{Ap}^{(k)}, \mathbf{p}^{(k)} \rangle}$

  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$

  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{Ap}^{(k)}$

  if *converged* **then**

    return $\mathbf{x}^{(k+1)}$

  Solve $M\mathbf{z}^{(k+1)} = \mathbf{r}^{(k+1)}$.

  $\beta_k = \dfrac{\langle \mathbf{z}^{(k+1)}, \mathbf{r}^{(k+1)} \rangle}{\langle \mathbf{z}^{(k)}, \mathbf{r}^{(k)} \rangle}$

  $\mathbf{p}^{(k+1)} = \mathbf{z}^{(k+1)} + \beta_k \mathbf{p}^{(k)}$

We now examine three choices for $\mathbf{M}$ that are common in practice.

2.2.1 Jacobi Preconditioning

In *Jacobi* preconditioning we take $\mathbf{M} = \mathbf{D}$, where $\mathbf{D}$ is a diagonal matrix whose main diagonal entries are equal to the main diagonal entries of $\mathbf{A}$. It is clearly symmetric positive-definite, satisfying condition 1, and trivial to invert, satisfying condition 2. Jacobi preconditioning is derived from the classical stationary iteration of the same name [15].

The closer $\mathbf{A}$ is to being diagonally dominant (meaning the sum of the absolute values of the diagonal entries of $\mathbf{A}$ is much greater than the sum of the absolute values of the off-diagonal entries of $\mathbf{A}$; ratio is $\gg 1$) the closer the Jacobi preconditioner will be to approximating $\mathbf{A}$ and satisfying condition 3. Though $\mathbf{A}$ is not strictly diagonally dominant in our case (Table 2.1 shows the relative sizes of the sums of the diagonal and off-diagonal entries for a particular instantiation of these matrices), Jacobi preconditioning is so inexpensive it is worth trying. The code we wrote to build the Jacobi preconditioner is in Appendix C.

Table 2.1. Diagonality

	Monolithic	Trace	Displacements
Sum of |Diagonal Entries|	342,056	7,956	171,277
Sum of |Off-Diagonal Entries|	519,372	10,513	291,281
Ratio	.66	.76	.59

2.2.2 Incomplete Cholesky Preconditioning

The next preconditioner we consider is the *incomplete Cholesky* (IC) preconditioner. The idea behind IC is to build an approximation to the ideal preconditioner $\mathbf{M} = \mathbf{A}$ (to satisfy condition 3) by computing an approximation to the Cholesky factorization of $\mathbf{A}$. In more detail, if we have $\mathbf{A} = \mathbf{H}\mathbf{H}^T$ in a Cholesky decomposition, we build $\mathbf{M}$ as $\mathbf{H}'\mathbf{H}'^T$, where $\mathbf{H}'$ is an approximation to $\mathbf{H}$, constructed by exactly matching the sparsity of $\mathbf{A}$; we accomplish this by building a Cholesky reduction while setting h_{ij} to zero wherever a_{ij} is zero, as shown in Algorithm 2.1, taken from [16]. There are other variants of IC (e.g., one replaces very small elements of $\mathbf{H}$ with zeros).

If we were to use the exact Cholesky factor $\mathbf{H}$, $\mathbf{H}\mathbf{H}^T$ would be exactly equal to $\mathbf{A}$ and preconditioned CG would converge in one step. However, as $\mathbf{A}$ becomes large, building the Cholesky factor becomes extremely memory intensive and can be impractical. Table 2.2 compares the sparsity of the original system with that of the Cholesky and IC factorizations. Therefore we resort to the less memory intensive IC. The code we wrote to implement this algorithm and build the IC preconditioner is in Appendix C.

Algorithm 2.3: The Incomplete Cholesky Factorization

Input: **A**, empty preconditioner **H** (same size as **A**)

Output: Incomplete Cholesky preconditioner–Lower Triangular Portion–($\mathbf{M} = \mathbf{H}\mathbf{H}^T$)

```
for k = 1 : n do
    H(k, k) = sqrt(A(k, k))
    for i = k + 1 : n do
        if A(i, k) ≠ 0 then
            H(i, k) = A(i, k) / A(k, k)
    for j = k + 1 : n do
        for i = j : n do
            if A(i, j) ≠ 0 then
                H(i, j) = A(i, j) − A(i, k)A(j, k)
```

Table 2.2. Non-zero entry comparison

Method	Level	Original Matrix	Cholesky	IC
Trace	1	1.89E+05	2.14E+05	9.55E+04
	2	7.15E+05	8.09E+05	3.59E+05
	3	2.78E+06	3.14E+06	1.39E+06
	4	1.10E+07	1.24E+07	5.49E+06
Displacements	1	1.28E+06	6.38E+06	6.49E+05
	2	4.14E+06	2.49E+07	2.11E+06
	3	1.46E+07	1.10E+08	7.45E+06
	4	5.46E+07	4.92E+08	2.78E+07
Monolithic	1	1.06E+06	6.22E+06	5.42E+05
	2	3.70E+06	2.54E+07	1.89E+06
	3	1.37E+07	1.09E+08	7.01E+06
	4	5.28E+07	5.06E+08	2.70E+07

2.2.3 Algebraic Multigrid Preconditioning

Multigrid is a technique that combines smoothing iterations and iterative refinement via coarse-problem corrections to generate optimal, or near-optimal, preconditioners for certain types of matrices. It is particularly useful for matrices arising from discretizations of elliptic partial differential equations (PDEs), such as the one we described in Chapter 1. Restricting ourselves to that context, multigrid methods are very powerful but are also challenging to implement, as traditional approaches require information about the discretization geometry and/or the differential operator.

Algebraic multigrid (AMG) is a variant of multigrid, that relies solely on the structure of the system matrix itself for constructing its hierarchy of coarse problems, without considering other system attributes. This "black box" nature of the method has made it a popular alternative to traditional geometric multigrid, as it can be applied to many PDEs with minimal or no input. A detailed description of AMG is beyond of the scope of this thesis; those interested can find more information in [17], and [18].

We did not write our own implementation of this preconditioner. Rather, we used the AlgebraicMultigrid.jl package publicly available on GitHub [19], which contains an implementation of the popular *Ruge-Stüben* variant of this method [20].

CHAPTER 3:
Numerical Experiments and Results

3.1 Experimental Setup

For each of the trace, displacements, and monolithic methods, a finite difference grid was used with four levels of resolution within each block (of the global mesh) to achieve higher accuracy. This does not increase the number of blocks in the global mesh, rather, the number of finite difference grid points within each block. Table 3.1 details the resulting size of each of these systems for each level of resolution. Julia code for generating these matrices and solving the attendant linear systems is detailed in Appendix A.

Table 3.1. System sizes (n)

	Monolithic	Trace	Displacements
Level 1 (17×17 grid)	19,872	1,728	18,144
Level 2 (34×34 grid)	71,960	3,360	68,600
Level 3 (68×68 grid)	273,240	6,624	266,616
Level 4 (136×136 grid)	1,064,216	13,152	1,051,064

We then measured the number of iterations it took for convergence for each of the three methods (monolithic, trace, displacements) using non-preconditioned CG and CG preconditioned with Jacobi, IC, and AMG. We also measured the amount of time it took for convergence using Julia's built-in direct solver, non-preconditioned CG, and CG preconditioned with Jacobi, IC, and AMG. We stopped a given run of CG when the 2-norm of the preconditioned residual was less than our chosen tolerance of 10^{-11}. We used a *Dell XPS 15 9510* with 16 GB of RAM and an *Intel* processor clocked at 2.50 GHz to do this. The Juila code used to develop these data can be found on NPS GitLab at [21]. This code builds on the original work which can be found on GitHub at [22].

3.2 Iterations

Figure 3.1 shows the number of iterations it took for each preconditioning method to converge to a solution for each solution method (trace, displacements, and monolithic). Table 3.2 shows the iteration reductions for each preconditioner across all three methods and four levels.

- For the trace method, Jacobi reduced iterations by a factor of nearly 2 and both IC and AMG reduced iterations by a factor of over 25 (i.e., IC and AMG achieved an approximately 96% reduction in iterations). The results for IC and AMG are nearly ideal, as the number of iterations does not significantly increase with the grid size.
- For the displacements method, Jacobi reduced iterations by a factor of nearly 3 (63% reduction), IC reduced iterations by a factor of over 40 (98% reduction), and AMG reduced iterations by a factor of over 30 (97% reduction).
- For the monolithic method, Jacobi reduced iterations by a factor of over 4 (77% reduction), IC reduced iterations by a factor of over 35 (97% reduction), and AMG reduced iterations by a factor of over 75 (99% reduction).

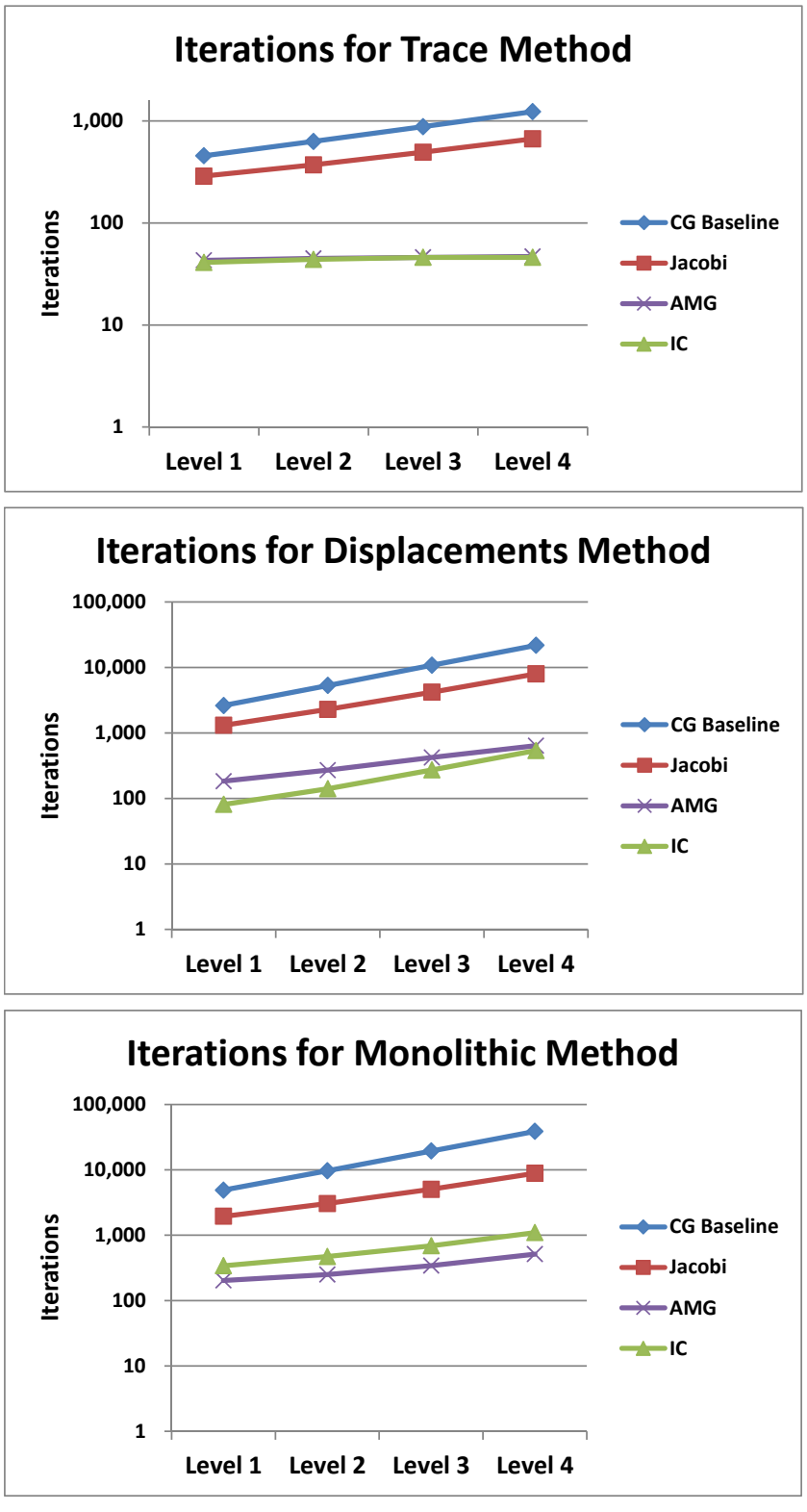


Figure 3.1. Number of iterations required for convergence

Table 3.2. Iteration reduction [factor (%)]

Method	Level	Jacobi	IC	AMG
Trace	1	1.6 (37%)	11.1 (91%)	10.6 (91%)
	2	1.7 (41%)	14.3 (93%)	14.0 (93%)
	3	1.8 (44%)	19.1 (95%)	19.1 (95%)
	4	1.8 (47%)	26.9 (96%)	26.3 (96%)
Displacements	1	2.0 (50%)	32.5 (97%)	14.3 (93%)
	2	2.3 (57%)	37.6 (97%)	19.6 (95%)
	3	2.6 (61%)	39.7 (97%)	25.6 (96%)
	4	2.7 (63%)	40.4 (98%)	34.0 (97%)
Monolithic	1	2.5 (60%)	14.4 (93%)	24.2 (96%)
	2	3.2 (68%)	20.6 (95%)	38.7 (97%)
	3	3.8 (74%)	28.3 (96%)	56.8 (98%)
	4	4.4 (77%)	35.5 (97%)	75.5 (99%)

3.3 Eigenvalues

We can explain the differences in the iteration counts for each preconditioner by examining the eigenvalues of the preconditioned matrices $\mathbf{M}^{-1}\mathbf{A}$. We only did this for level 1 (coarsest) grid refinement, as at the finer levels, the matrices are too large for us to compute all of their eigenvalues in a reasonable amount of time.

For Jacobi and IC we used the routines we wrote in Julia to produce $\mathbf{M}^{-1}$. For AMG, we built $\mathbf{M}^{-1}$ by brute force, creating the columns of $\mathbf{M}^{-1}$ by passing the standard basis vectors (1 to n) to the AlgebraicMultigrid.jl package publicly available on GitHub.

As Figures 3.2, 3.3, and 3.4 show, the eigenvalues of our systems were caused to cluster by Jacobi, IC, and AMG preconditioning, just as we said we would need in Section 2.2. Code for producing these graphs can be found in Appendix E.

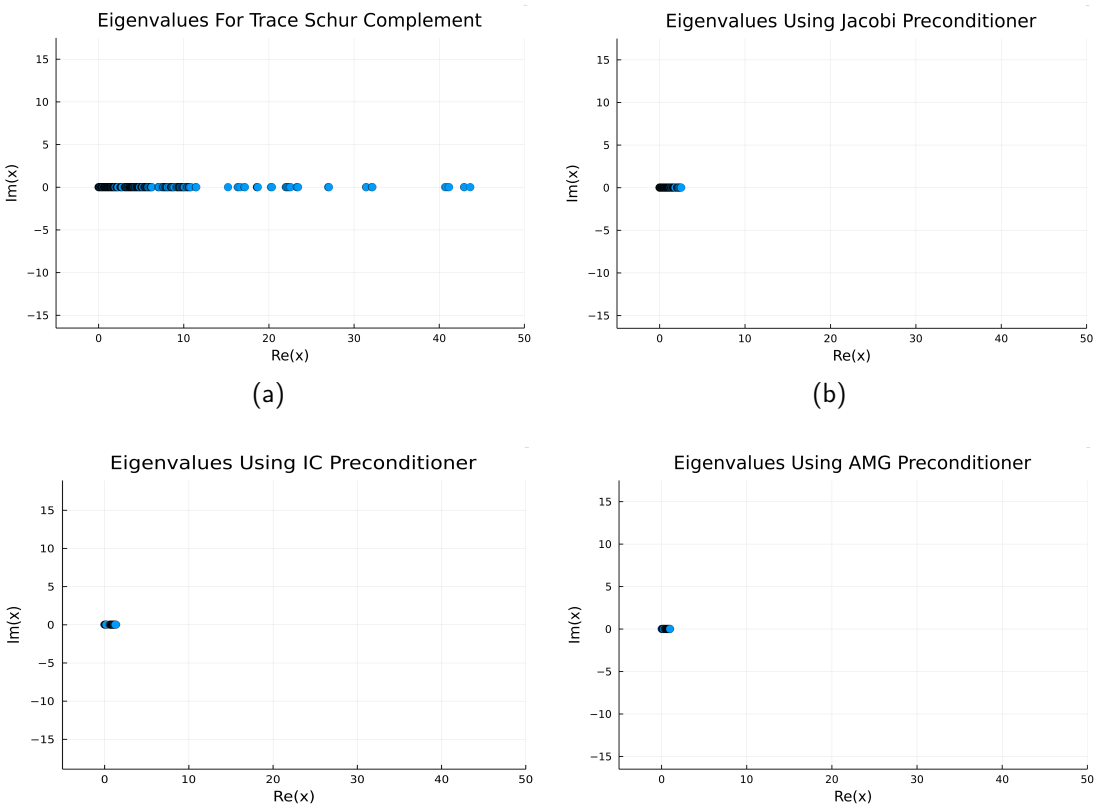
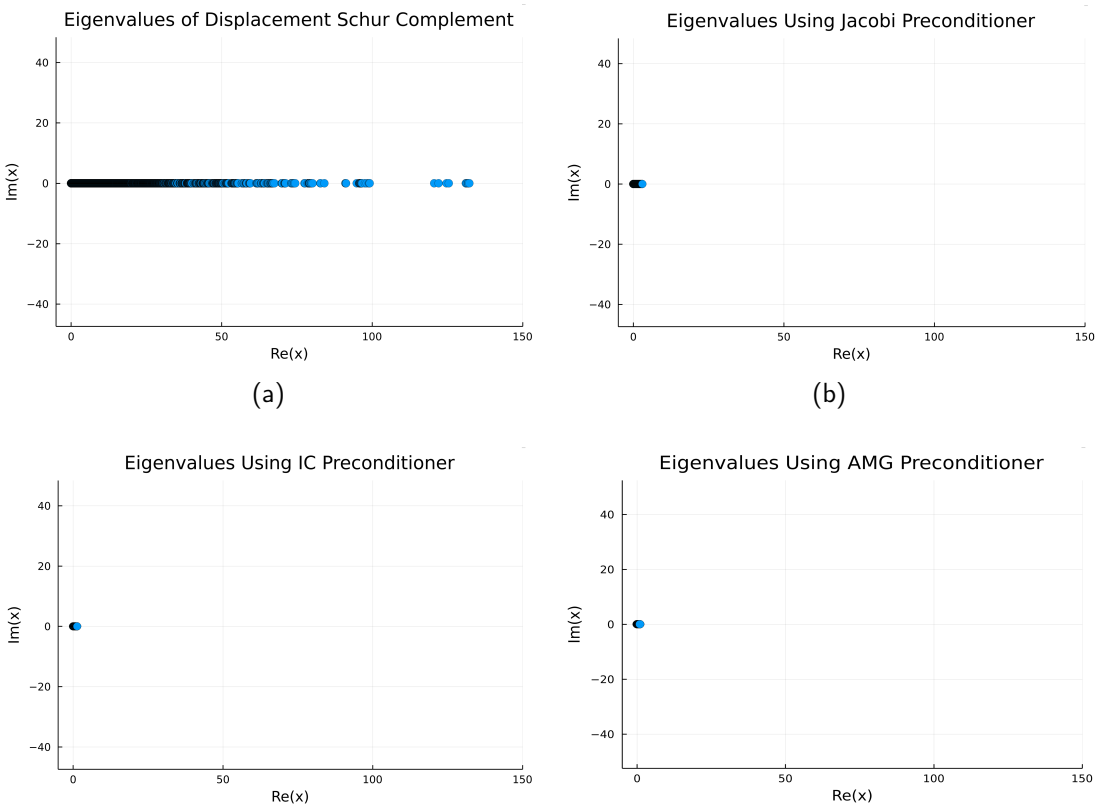


Figure 3.2. Eigenvalues of the trace Schur complement for unpreconditioned (a) and preconditioned with Jacobi (b), IC (c), and AMG (d)



(a) (b)

(c) (d)

Figure 3.3. Eigenvalues of the displacements Schur complement for unpreconditioned (a) and preconditioned with Jacobi (b), IC (c), and AMG (d)

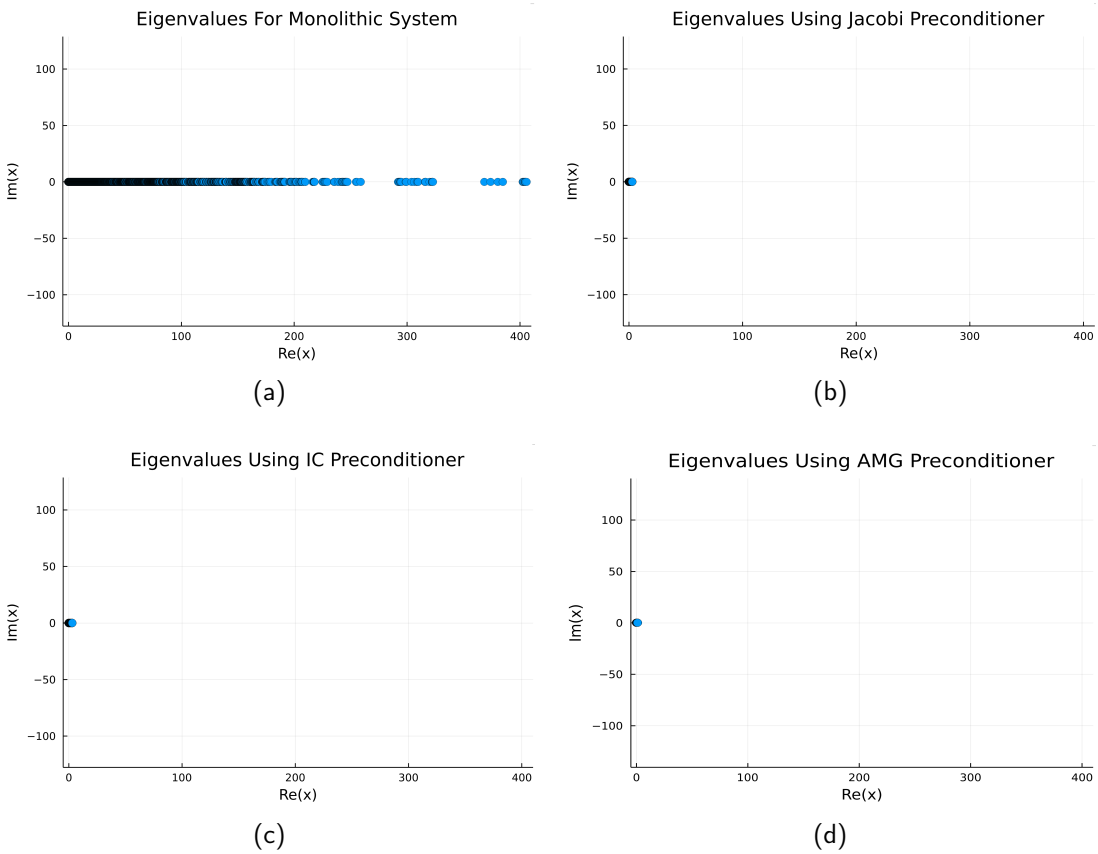


Figure 3.4. Eigenvalues of the monolithic system for unpreconditioned (a) and preconditioned with Jacobi (b), IC (c), and AMG (d)

Table 3.3 shows, for each method, the effect of each preconditioner on the condition number:

- Trace: Jacobi reduced κ by a factor of 3.6 (72% reduction), IC reduced κ by a factor of 215 (99.6% reduction), and AMG reduced κ by a factor of 222 (99.6% reduction).
- Displacements: Jacobi reduce κ by a factor of 4.6 (78.5% reduction), IC reduced κ by a factor of 1,405 (99.9% reduction), and AMG reduced κ by a factor of 260 (99.6% reduction).
- Monolithic: Jacobi reduced κ by a factor of 7.4 (86.6% reduction), IC reduced κ by a factor of 249 (99.6% reduction), and AMG reduced κ by a factor of 771 (99.9% reduction).

Table 3.3. Preconditioning effects on κ (for Level 1)

Preconditioner	Trace κ	Displacements κ	Monolithic κ
unpreconditioned	8.6E+03	2.5E+05	8.4E+05
Jacobi	2.4E+03	5.3E+04	1.1E+05
IC	4.0E+01	1.8E+02	3.4E+03
AMG	3.8E+01	9.5E+02	1.1E+03

These results explain why IC and AMG were more effective in reducing the number of iterations required to converge compared to Jacobi preconditioning.

3.4 Runtime

Figure 3.6 shows how long, in seconds, it took for each preconditioning method to converge to a solution for each solution method (trace, displacements, and monolithic). This is the time to run the iterative CG method and does not include the time to build the preconditioners, nor does it include the time to solve for the variables eliminated when using the trace and displacements methods, however, owing to the (block) diagonality of $\mathbf{M}$ and $\mathbf{D}$, it is clear the runtime for those methods will be dominated by the Schur complement solve (see Table 3.4). We only present these results for the trace method as the trace variable solve related to the displacement Schur solve is a diagonal system, and thus trivial. We also compare with Julia's built-in direct solver, which is based on *CHOLMOD* [23].

Table 3.4. Trace Schur complement solve domination

	Level 1	Level 2	Level 3	Level 4
Time for Schur Complement Solve	3.50E-02	1.04E-01	4.40E-01	2.54E+00
Time for Volume Variable Solve	2.31E-03	9.54E-03	1.02E-01	5.23E-01
% Schur Solve	93.8%	91.6%	81.1%	82.9%

Runtime for IC was was on the same order of magnitude as the Julia built-in direct solver, with AMG taking second place, and Jacobi preconditioning last.

For ease of comparison, we used the same y-axis scale for the displacements and monolithic method results. A tabular form of these results is provided in Appendix D. We have also provided tabular results produced when stopping a given run of CG when the ℓ_∞ norm of the preconditioned residual was less than a tolerance of 10^{-16}.

We observe the Juila built-in direct solver is still faster than the iterative methods for solving these problems. This is because the problems we've considered are relatively small and, moreover, are derived from a 2-D geometry. The Julia built-in direct solver actually scales quite well for 2-D problems, as is indicated by Figure 3.5 which depicts the time taken to solve the problem for the monolithic method using the direct solver as a function of the matrix dimension n. The runtime for the direct solver scales nearly linearly with n which is the best one could hope to attain from any linear solver. This advantage would likely disappear for real-world problems which generally have 3-D geometries and for which direct solvers tend not to scale well [24].

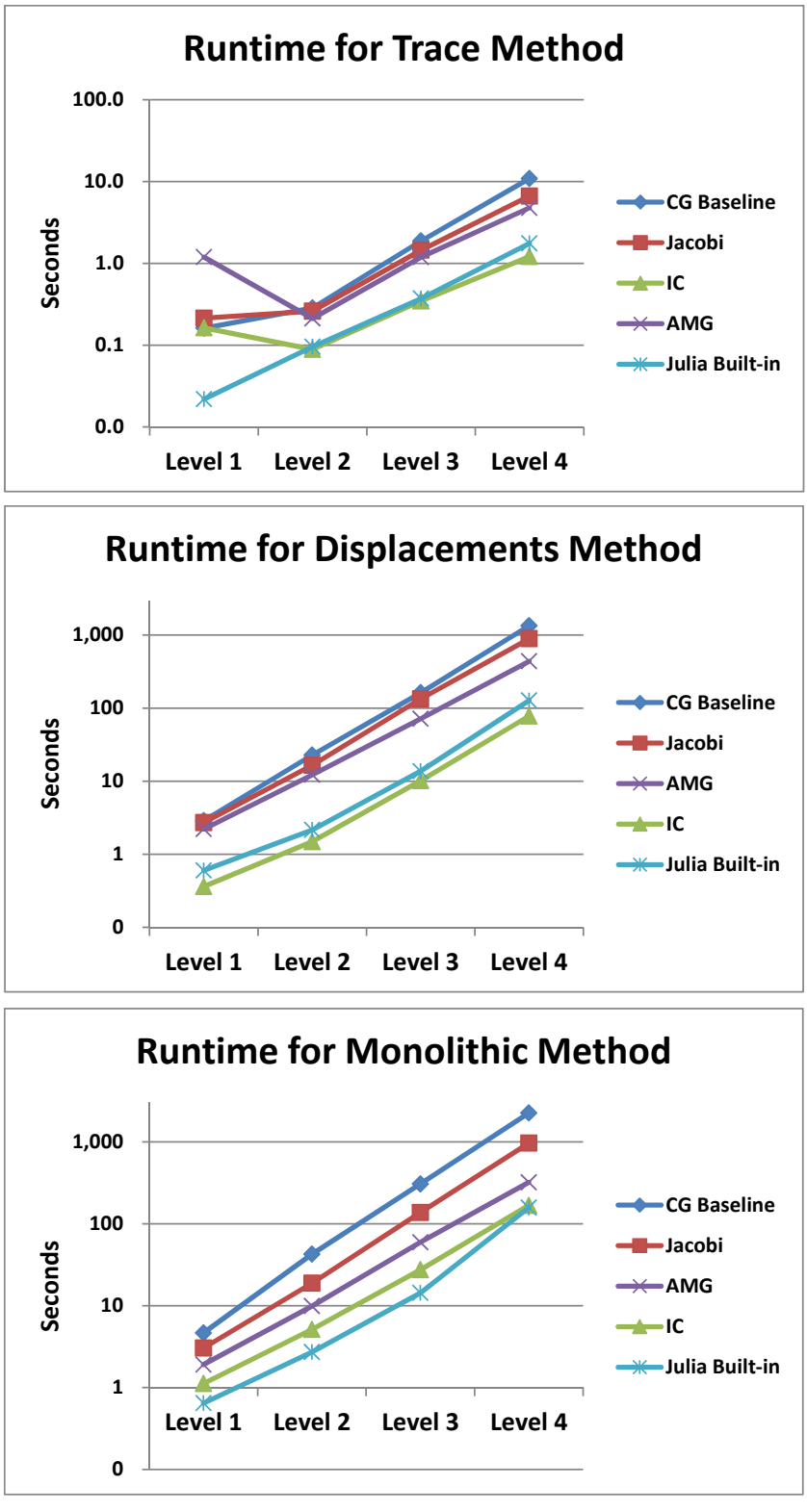


Figure 3.6. Seconds required for convergence

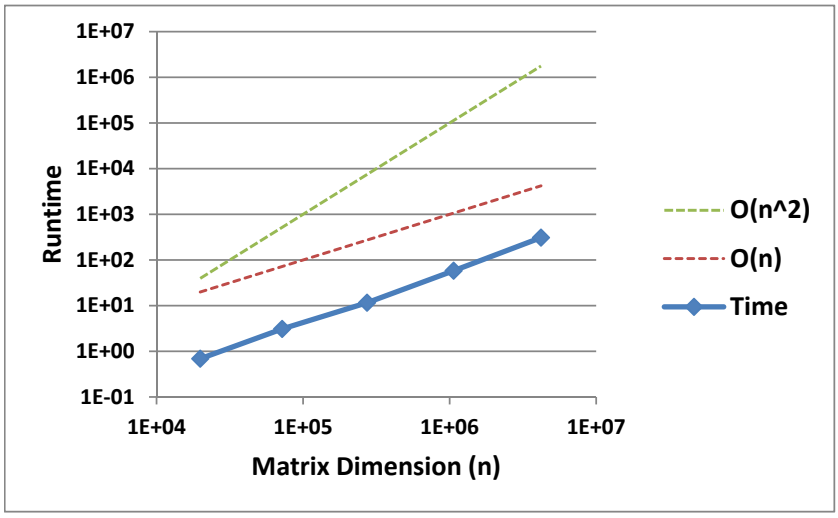


Figure 3.5. Comparison of runtime versus matrix dimension

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Conclusions / Future Research Areas

4.1 Conclusions

These results clearly show that all the preconditioning methods yielded iterations savings (especially IC and AMG) when compared to CG results without preconditioning. Furthermore, IC converged in roughly the same amount of time it took the built-in Julia direct solver to converge. Considering we did little to optimize our IC code, it is encouraging that it is competitive with the optimized Julia direct solver. In practice these simulations would be incorporated into a time loop and the same linear system would be solved many times with different right-hand side input. Thus, the cost of forming the preconditioner is amortized over many solves.

If IC can be further optimized for these type of problems, it could produce significant improvements over the Julia built-in direct solver and even be applied to real-world problems where direct solvers cannot be used.

4.2 Future Research Areas

We recommend exploring optimization of IC and AMG to see if additional savings can be realized. Further, it could prove fruitful to explore other preconditioners, especially methods that take advantage of the block structure of our problem. Among these, we recommend considering the approaches discussed in [13]. We also recommend exploring methods for using better initial guesses as another avenue to accelerate convergence, as discussed in [25], for example. Finally, we recommend extending our analysis of these preconditioners for problems posed in 3-D geometries, which can have considerably more degrees of freedom than problems in 2-D and for which direct solvers are less scalable.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Modified Kozdon et al. Julia Code

All files required to run this code can be found on GitHub:
https://gitlab.nps.edu/timothy.james/thesis-james

```
using LinearAlgebra
using AlgebraicMultigrid
include("global_curved.jl")
include("run_conjugate_gradient_precon.jl")
include("run_algebraic_multigrid3.jl")
include("run_incomplete_cholesky3.jl")
include("PlotEVs.jl")

import PGFPlots

let
  if @isdefined in_method
    method = in_method
  else
    method = :trace   # Choices are :monolithic, :trace, and :displacements
  end
  if @isdefined in_solve
    solve = in_solve
  else
    solve = :IC  # Choices are
    #           :Julia,
    #           :CG (Conjugate Gradient),
    #           :Jacobi (Conjugate Gradient w/ Jacobi Preconditioning),
    #           :AMG (Conjugate Gradient w/ Algebraic Multigrid Preconditioning),
    #           and
    #           :IC (Conjugate Gradient w/ Incomplete Cholesky Preconditioning)
  end

  # Graph Eigenvalues?
  EVgraph = :No  # Choices are :Yes or :no

  # SBP interior order
  SBPp   = 6

  # mesh file side set type to actually boundary condition type
  bc_map = [BC_DIRICHLET, BC_DIRICHLET, BC_NEUMANN, BC_NEUMANN,
            BC_JUMP_INTERFACE]
  (verts, EToV, EToF, FToB, EToDomain) = read_inp_2d("meshes/square_circle.inp";
                                                     bc_map = bc_map)
```

```
# EToV defines the element by its vertices
# EToF defines element by its four faces, in global face number
# FToB defines whether face is Dirichlet (1), Neumann (2), interior jump (7)
#      or just an interior interface (0)
# EToDomain is 1 if element is inside circle; 2 otherwise

# number of elements and faces
(nelems, nfaces) = (size(EToV, 2), size(FToB, 1))
@show (nelems, nfaces)

# This is needed to fix up points that should be on the boundary of the
# circle, but the mesh didn't quite put them there
for v in 1:size(verts, 2)
  x, y = verts[1, v], verts[2, v]
  if abs(hypot(x,y) - 1) < 1e-5
    Q = atan(y, x)
    verts[1, v], verts[2, v] = cos(Q), sin(Q)
  end
end


# Plot the original connectivity before mesh warping
# plot_connectivity(verts, EToV)

# This is the base mesh size in each dimension
N1 = N0 = 17

# EToN0 is the base mesh size (e.g., before refinement)
EToN0 = zeros(Int64, 2, nelems)
EToN0[1, :] .= N0
EToN0[2, :] .= N1

@assert typeof(EToV) == Array{Int, 2} && size(EToV) == (4, nelems)
@assert typeof(EToF) == Array{Int, 2} && size(EToF) == (4, nelems)
@assert maximum(maximum(EToF)) == nfaces

# Determine secondary arrays
# FToE : Unique Global Face to Element Number
#        (the i'th column of this stores the element numbers that share the
#        global face number i)
# FToLF: Unique Global Face to Element local face number
#        (the i'th column of this stores the element local face numbers that
#        shares the global face number i)
# EToO : Element to Unique Global Faces Orientation
#        (the i'th column of this stores the whether the element and global
#        face are oriented in the same way in physical memory or need to be
#        rotated)
# EToS : Element to Unique Global Face Side
#        (the i'th column of this stores whether an element face is on the
#        plus side or minus side of the global face)
```

```
(FToE, FToLF, EToO, EToS) = connectivityarrays(EToV, EToF)

pgf_axis = PGFPlots.Axis(style="width=5cm, height=5cm, ticks=none",
                         xlabel=PGFPlots.L"$x$",
                         ylabel=PGFPlots.L"$y$",
                         xmin = -2, xmax = 2,
                         ymin = -2, ymax = 2)
for f in 1:nfaces
  if FToB[f] != BC_JUMP_INTERFACE
    (e, lf) = FToE[1,f], FToLF[1,f]
    if lf == 1
      v1, v2 = EToV[1, e], EToV[3, e]
    elseif lf == 2
      v1, v2 = EToV[2, e], EToV[4, e]
    elseif lf == 3
      v1, v2 = EToV[1, e], EToV[2, e]
    else
      v1, v2 = EToV[3, e], EToV[4, e]
    end
    x = verts[1, [v1 v2]][:]
    y = verts[2, [v1 v2]][:]
    push!(pgf_axis, PGFPlots.Linear(x, y, style="no marks, solid, black"))
  end
end
push!(pgf_axis, PGFPlots.Circle(0,0,1, style = "very thick, red"))
PGFPlots.save("square_circle.tikz", pgf_axis)

# Exact solution
Lx = maximum(verts[1,:])
Ly = maximum(abs.(verts[2,:]))



c = exp(1)/(1 + exp(1))

vinside(x,y,e) = begin
  r = sqrt.(x .^ 2 + y .^ 2)
  theta = atan.(y,x)
  return c .* (1 .- exp.(-1 .* r .^ 2)) .* r .* sin.(theta)
end

voutside(x,y,e) = begin
  r = sqrt.(x .^ 2 + y .^ 2)
  theta = atan.(y,x)
  return (r .- 1) .^ 2 .* cos.(theta) .+ (r .- 1) .* sin.(theta)
end


vinside_x(x,y,e) = begin
  r = sqrt.(x .^ 2 + y .^ 2)
```

```
  theta = atan.(y,x)
  dtheta_dx = -1 .* sin.(theta) ./ r
  dr_dx = cos.(theta)
  dv_dr = c * (2 .* r .^ 2 .* exp.(-1 .* r .^ 2) .+ 1 .- exp.(-1 .* r .^ 2))
          .* sin.(theta)
  dv_dtheta = c .* (1 .- exp.(-1 .* r .^ 2)) .* r .* cos.(theta)
  return dv_dr .* dr_dx + dv_dtheta .* dtheta_dx
end


vinside_y(x,y,e) = begin
  r = sqrt.(x .^ 2 + y .^ 2)
  theta = atan.(y,x)
  dtheta_dy = cos.(theta) ./ r
  dr_dy = sin.(theta)
  dv_dr = c * (2 .* r .^ 2 .* exp.(-1 .* r .^ 2) .+ 1 .- exp.(-1 .* r .^ 2))
          .* sin.(theta)
  dv_dtheta = c .* (1 .- exp.(-1 .* r .^ 2)) .* r .* cos.(theta)
  return dv_dr .* dr_dy + dv_dtheta .* dtheta_dy
end

voutside_x(x,y,e) = begin
  r = sqrt.(x .^ 2 + y .^ 2)
  theta = atan.(y,x)
  dtheta_dx = -1 .* sin.(theta) ./ r
  dr_dx = cos.(theta)
  dv_dr =  2 .* (r .- 1) .* cos.(theta) .+ sin.(theta)
  dv_dtheta = -1 .* (r .- 1) .^ 2 .* sin.(theta) .+ (r .- 1) .* cos.(theta)
  return dv_dr .* dr_dx + dv_dtheta .* dtheta_dx
end


voutside_y(x,y,e) = begin
  r = sqrt.(x .^ 2 + y .^ 2)
  theta = atan.(y,x)
  dtheta_dy = cos.(theta) ./ r
  dr_dy = sin.(theta)
  dv_dr = 2 .* (r .- 1) .* cos.(theta) .+ sin.(theta)
  dv_dtheta = -1 .*  (r .- 1) .^ 2 .* sin.(theta) .+ (r .- 1) .* cos.(theta)
  return dv_dr .* dr_dy + dv_dtheta .* dtheta_dy
end


polar_laplace(x,y,e) = begin #u_rr + (1/r)*u_r + (1/r^2)*u_theta,theta
  r = sqrt.(x .^ 2 + y .^ 2)
  theta = atan.(y,x)
  if EToDomain[e] == 1
    u_r = c .* (2 .* r .^ 2 .* exp.(-1 .* r .^ 2) .+ 1 .- exp.(-1 .* r .^ 2))
          .* sin.(theta)
    u_rr = c .* exp.(-1 .* r .^ 2) .* (6 .* r .- 4 .* r .^ 3) .* sin.(theta)
```

```
    return u_rr .+ (1 ./ r) .* u_r .- (c ./ r .^ 2) .* (1 .- exp.(-1 .* r .^ 2))
          .* r .* sin.(theta)
    return dv_dr .* dr_dy .+ (cos.(r .- 1) .* cos.(theta)) .* (cos.(theta))
  elseif EToDomain[e] == 2
    return 2 * cos.(theta) .+ (1 ./ r) .* (2 .* (r .- 1) .* cos.(theta) .+ sin.(theta))
          .+ (1 ./ r .^ 2) .* (-1 .* (r .- 1) .^ 2 .* cos.(theta) .- (r .- 1)
          .* sin.(theta))
  else
    error("invalid block")
  end
end



vex(x,y,e) = begin
  if EToDomain[e] == 1
    return vinside(x,y,e)
  elseif EToDomain[e] == 2
    return voutside(x,y,e)
  else
    error("invalid block")
  end
end

vex_x(x,y,e) = begin
  if EToDomain[e] == 1
    return vinside_x(x,y,e)
  elseif EToDomain[e] == 2
    return voutside_x(x,y,e)
  else
    error("invalid block")
  end
end

vex_y(x,y,e) = begin
  if EToDomain[e] == 1
    return vinside_y(x,y,e)
  elseif EToDomain[e] == 2
    return voutside_y(x,y,e)
  else
    error("invalid block")
  end
end


ϵ = zeros(4)  # NUMBER OF LEVELS

for lvl = 1:length(ϵ)
  flush(stdout)
  # Set up the local grid dimensions
```

```
Nr = EToN0[1, :] * (2^(lvl-1))
Ns = EToN0[2, :] * (2^(lvl-1))

#
# Build the local volume operators
#

# Dictionary to store the operators
OPTYPE = typeof(locoperator(2, 16, 16))
lop = Dict{Int64, OPTYPE}()

# Loop over blocks and create local operators
for e = 1:nelems
  # Get the element corners
  (x1, x2, x3, x4) = verts[1, EToV[:, e]]
  (y1, y2, y3, y4) = verts[2, EToV[:, e]]

  # Initialize the block transformations as transfinite between the corners
  ex = [(α) -> x1 * (1 .- α) / 2 + x3 * (1 .+ α) / 2,
        (α) -> x2 * (1 .- α) / 2 + x4 * (1 .+ α) / 2,
        (α) -> x1 * (1 .- α) / 2 + x2 * (1 .+ α) / 2,
        (α) -> x3 * (1 .- α) / 2 + x4 * (1 .+ α) / 2]
  exα = [(α) -> -x1 / 2 + x3 / 2,
         (α) -> -x2 / 2 + x4 / 2,
         (α) -> -x1 / 2 + x2 / 2,
         (α) -> -x3 / 2 + x4 / 2]
  ey = [(α) -> y1 * (1 .- α) / 2 + y3 * (1 .+ α) / 2,
        (α) -> y2 * (1 .- α) / 2 + y4 * (1 .+ α) / 2,
        (α) -> y1 * (1 .- α) / 2 + y2 * (1 .+ α) / 2,
        (α) -> y3 * (1 .- α) / 2 + y4 * (1 .+ α) / 2]
  eyα = [(α) -> -y1 / 2 + y3 / 2,
         (α) -> -y2 / 2 + y4 / 2,
         (α) -> -y1 / 2 + y2 / 2,
         (α) -> -y3 / 2 + y4 / 2]

  # For blocks on the circle, put in the curved edge transform
  if FToB[EToF[1, e]] == BC_JUMP_INTERFACE
    error("curved face 1 not implemented yet")
  end
  if FToB[EToF[2, e]] == BC_JUMP_INTERFACE
    error("curved face 2 not implemented yet")
  end
  if FToB[EToF[3, e]] == BC_JUMP_INTERFACE
    Q1 = atan(y1, x1)
    Q2 = atan(y2, x2)
    if !(-π/2 < Q1 - Q2 < π/2)
      Q2 -= sign(Q2) * 2 * π
    end
    ex[3] = (α) -> cos.(Q1 * (1 .- α) / 2 + Q2 * (1 .+ α) / 2)
    ey[3] = (α) -> sin.(Q1 * (1 .- α) / 2 + Q2 * (1 .+ α) / 2)
```

```
    β3 = (Q2 - Q1) / 2
    exα[3] = (α) -> -β3 .* sin.(Q1 * (1 .- α) / 2 + Q2 * (1 .+ α) / 2)
    eyα[3] = (α) -> +β3 .* cos.(Q1 * (1 .- α) / 2 + Q2 * (1 .+ α) / 2)
  end
  if FToB[EToF[4, e]] == BC_JUMP_INTERFACE
    Q3 = atan(y3, x3)
    Q4 = atan(y4, x4)
    if !(-π/2 < Q3 - Q4 < π/2)
      error("curved face 4 angle correction not implemented yet")
    end
    ex[4] = (α) -> cos.(Q3 * (1 .- α) / 2 + Q4 * (1 .+ α) / 2)
    ey[4] = (α) -> sin.(Q3 * (1 .- α) / 2 + Q4 * (1 .+ α) / 2)
    β4 = (Q4 - Q3) / 2
    exα[4] = (α) -> -β4 .* sin.(Q3 * (1 .- α) / 2 + Q4 * (1 .+ α) / 2)
    eyα[4] = (α) -> +β4 .* cos.(Q3 * (1 .- α) / 2 + Q4 * (1 .+ α) / 2)
  end

  # Create the volume transform as the transfinite blending of the edge
  # transformations
  xt(r,s) = transfinite_blend(ex[1], ex[2], ex[3], ex[4],
                              exα[1], exα[2], exα[3], exα[4],
                              r, s)
  yt(r,s) = transfinite_blend(ey[1], ey[2], ey[3], ey[4],
                              eyα[1], eyα[2], eyα[3], eyα[4],
                              r, s)

  metrics = create_metrics(SBPp, Nr[e], Ns[e], xt, yt)

  # Build local operators
  lop[e] = locoperator(SBPp, Nr[e], Ns[e], metrics, FToB[EToF[:, e]])
end

# If this is the first mesh level plot the mesh
lvl == 1 && plot_blocks(lop)

#
# Do some assemble of the global volume operators
#
(M, FbarT, D, vstarts, FToλstarts) =
LocalGlobalOperators(lop, Nr, Ns, FToB, FToE, FToLF, EToO, EToS,
                     (x) -> cholesky(Symmetric(x)))
@show lvl
locfactors = M.F

# Get a unique array indexes for the face to jumps map
FToδstarts = bcstarts(FToB, FToE, FToLF, BC_JUMP_INTERFACE, Nr, Ns)

# Compute the number of volume, trace (λ), and jump (δ) points
VNp = vstarts[nelems+1]-1
λNp = FToλstarts[nfaces+1]-1
```

```
δNp = FToδstarts[nfaces+1]-1
# @show (VNp, λNp, δNp)

# Build the (sparse) λ matrix using the schur complement and factor
B = assembleλmatrix(FToλstarts, vstarts, EToF, FToB, locfactors, D, FbarT)
BF = cholesky(Symmetric(B))

(bλ, λ, gδ) = (zeros(λNp), zeros(λNp), zeros(λNp))
(Δ, u, g) = (zeros(VNp), zeros(VNp), zeros(VNp))
δ = zeros(δNp)


for f = 1:nfaces
  if FToB[f] == BC_JUMP_INTERFACE
    (e1, e2) = FToE[:, f]
    (lf1, lf2) = FToLF[:, f]
    (xf, yf) = lop[e1].facecoord
    @views δ[FToδstarts[f]:(FToδstarts[f+1]-1)] =
    vex(xf[lf1], yf[lf1], e2) - vex(xf[lf1], yf[lf1], e1)
    #@show δ[FToδstarts[f]:(FToδstarts[f+1]-1)]
  end
end

bc_Dirichlet = (lf, x, y, e, δ) -> vex(x, y, e)
bc_Neumann   = (lf, x, y, nx, ny, e, δ) -> (nx .* vex_x(x, y, e)
                                            + ny .* vex_y(x, y, e))
in_jump      = (lf, x, y, e, δ) -> begin
  f = EToF[lf, e]
  if EToS[lf, e] == 1
    if EToO[lf, e]
      return -δ[FToδstarts[f]:(FToδstarts[f+1]-1)]
    else
      error("shouldn't get here")
    end
  else
    if EToO[lf, e]
      return  δ[FToδstarts[f]:(FToδstarts[f+1]-1)]
    else
      return  δ[(FToδstarts[f+1]-1):-1:FToδstarts[f]]
    end
  end
end

for e = 1:nelems
  gδe = ntuple(4) do lf
    f = EToF[lf, e]
    if EToO[lf, e]
      return @view gδ[FToλstarts[f]:(FToλstarts[f+1]-1)]
    else
      return  @view gδ[(FToλstarts[f+1]-1):-1:FToλstarts[f]]
```

```
    end
  end
  locbcarray!((@view g[vstarts[e]:vstarts[e+1]-1]), gδe, lop[e],
              FToB[EToF[:,e]], bc_Dirichlet, bc_Neumann, in_jump, (e, δ))

  source = (x, y, e) -> (-polar_laplace(x, y, e))
  locsourcearray!((@view g[vstarts[e]:vstarts[e+1]-1]), source, lop[e], e)
end

if method == :monolithic
  # Solve the monolithic system
  M = blockdiag(ntuple(i->lop[i].M̃, length(lop))...)
  A = [M FbarT'; FbarT Diagonal(D)]
    if solve == :CG
      println("Monolithic - Conjugate Gradient (baseline)")
      # Conjugate Gradient
      if EVgraph == :Yes   #Graph Eigenvalues?
        EV = copy(A)
        return plot_eigenvalues(EV, solve)
      end
      I_mono_precon!(Z, r) = Z .= r
      @time uλ =  run_conjugate_gradient_precon(rand(size([g;gδ])...),
                A , [g;gδ], I_mono_precon!, 10e-16)
    elseif solve == :Jacobi
     # Jacoian Preconditioning
      println("Monolithic - Jacobi Preconditioner")
      Minv = inv(Diagonal(A));
      if EVgraph == :Yes   #Graph Eigenvalues?
        EV = Minv*A
        return plot_eigenvalues(EV, solve)
      end
      jacobi_mono_precon!(z,r) = mul!(z, Minv,r)
      @time uλ =  run_conjugate_gradient_precon(rand(size([g;gδ])...),
                A, [g;gδ], jacobi_mono_precon!, 10e-16)
    elseif solve == :AMG
      # Algebraic Multigrid
      println("Monolithic - Algebraic Multigrid Preconditioner")
      ml = ruge_stuben(A)
      if EVgraph == :Yes   #Graph Eigenvalues?
        Minv = zeros(size(A))
        x = zeros(size(A,1),1)
        Z = zeros(size(A,1),1)
        for i in 1:size(A,1)
          x[i] = 1
          Z.=0
          Minv[:, i] = AlgebraicMultigrid.solve!(Z, ml, x, maxiter = 1,
                       calculate_residual = false)
          x[i] = 0
        end
        EV = Minv*A
```

```
        return plot_eigenvalues(EV, solve)
      end
      function AMG_mono_precon!(Z,r)
        Z.=0
        # Z is updated in place
        AlgebraicMultigrid.solve!(Z, ml, r, maxiter = 1,
                                  calculate_residual = false)
      end
      @time uλ = run_conjugate_gradient_precon(rand(size([g;gδ])...), A,
                 [g;gδ], AMG_mono_precon!, 10e-16)
    elseif solve == :IC
      # Incomplete Cholesky
      println("Monolithic - Incomplete Cholesky Preconditioner")
      n = size(A,1)
      H = copy(A)
      run_incomplete_cholesky3(H,n)
      H = LowerTriangular(H)
      HT = UpperTriangular(sparse(H'))
      if EVgraph == :Yes   #Graph Eigenvalues?
        invH = inv(H)
        EV = invH*A*invH'
        return plot_eigenvalues(EV, solve)
      end
      function chol_mono_precon!(Z,r)
        ldiv!(Z,H,r)
        ldiv!(HT,Z)
      end
      @time uλ = run_conjugate_gradient_precon(rand(size([g;gδ])...), A,
                 [g;gδ], chol_mono_precon!, 10e-16)
    elseif solve == :Julia
      # Julia \
      println("Monolithic - Julia built-in")
      if EVgraph == :Yes   #Graph Eigenvalues?
        EV = copy(A)
        return plot_eigenvalues(EV, solve)
      end
      @time uλ = A  \ [g;gδ]
    end
  u[:] .= uλ[1:length(u)]

elseif method == :trace
  # Solve for the trace variables then compute displacements
  LocalToGLobalRHS!(bλ, g, gδ, u, locfactors, FbarT, vstarts)
    if solve == :CG
      # Conjugate Gradient
      println("Trace - Conjugate Gradient (baseline)")
      if EVgraph == :Yes   #Graph Eigenvalues?
        EV = copy(B)
        return plot_eigenvalues(EV, solve)
      end
```

```
    I_trace_precon!(Z, r) = Z .= r
    @time λ = run_conjugate_gradient_precon(rand(size(bλ)...), B, bλ,
              I_trace_precon!, 10e-16)
  elseif solve == :Jacobi
    # Jacobi Preconditioning
    println("Trace - Jacobi Preconditioner")
    Minv = inv(Diagonal(B));
    if EVgraph == :Yes   #Graph Eigenvalues?
      EV = Minv*B
      return plot_eigenvalues(EV, solve)
    end
    jacobi_trace_precon!(z,r) = mul!(z, Minv, r)
    @time λ =  run_conjugate_gradient_precon(rand(size(bλ)...), B, bλ,
               jacobi_trace_precon!, 10e-16)
  elseif solve == :AMG
    # Algebraic Multigrid
    println("Trace - Algebraic Multigrid Preconditioner")
    ml = ruge_stuben(B)
    if EVgraph == :Yes   #Graph Eigenvalues?
      Minv = zeros(size(B))
      x = zeros(size(B,1),1)
      Z = zeros(size(B,1),1)
      for i in 1:size(B,1)
        x[i] = 1
        Z.=0
        Minv[:, i] = AlgebraicMultigrid.solve!(Z, ml, x, maxiter = 1,

                     calculate_residual = false)
        x[i] = 0
      end
      EV = Minv*B
      return plot_eigenvalues(EV, solve)
    end
    function AMG_trace_precon!(Z,r)
      Z.=0
      # Z is updated in place
      AlgebraicMultigrid.solve!(Z, ml, r, maxiter = 1, calculate_residual =
false)
    end
    @time λ = run_conjugate_gradient_precon(rand(size(bλ)...), B, bλ,
              AMG_trace_precon!, 10e-16)
  elseif solve == :IC
    # Incomplete Cholesky
    println("Trace - Incomplete Cholesky Preconditioner")
    n = size(B,1)
    H = copy(B)
    run_incomplete_cholesky3(H,n)
    H = LowerTriangular(H)
    HT = UpperTriangular(sparse(H'))
```

```
      if EVgraph == :Yes   #Graph Eigenvalues?
        invH = inv(H)
        EV = invH*A*invH'
        return plot_eigenvalues(EV, solve)
      end
      function chol_trace_precon!(Z,r)
        ldiv!(Z,H,r)
        ldiv!(HT,Z)
      end
      @time λ = run_conjugate_gradient_precon(rand(size(bλ)...), B, bλ,
                chol_trace_precon!, 10e-16)
    elseif solve == :Julia
      # Julia \
      println("Trace - Julia Built-in")
      if EVgraph == :Yes   #Graph Eigenvalues?
        EV = copy(B)
        return plot_eigenvalues(EV,solve)
      end
      @time λ[:] = B \ bλ
    end
  u[:] = -FbarT' * λ
  u[:] .= g .+ u
  for e = 1:nelems
    F = locfactors[e]
    @views u[vstarts[e]:(vstarts[e+1]-1)] = F \ u[vstarts[e]:(vstarts[e+1]-1)]
    #=
    ldiv!((@view u[vstarts[e]:(vstarts[e+1]-1)]), F,
          (@view u[vstarts[e]:(vstarts[e+1]-1)]))
    =#
  end

elseif method == :displacements
  # Solve for the discplacement directly
  M = blockdiag(ntuple(i->lop[i].M̃, length(lop))...)
  C = M - FbarT' * Diagonal(1 ./ D) * FbarT
    if solve == :CG
      # Conjugate Gradient
      println("Trace - Conjugate Gradient (baseline)")
      if EVgraph == :Yes   #Graph Eigenvalues?
        EV = copy(C)
        return plot_eigenvalues(EV, solve)
      end
      I_displace_precon!(Z, r) = Z .= r
      @time u =  run_conjugate_gradient_precon(rand(size(g
                - FbarT' * (gδ ./ D))...), C, g - FbarT' * (gδ ./ D),
                I_displace_precon!, 10e-16)
    elseif solve == :Jacobi
      # Jacobi Preconditioning Conjugate Gradient
      println("Displacement - Jacobi Preconditioner")
      Minv = inv(Diagonal(C));
```

```
  if EVgraph == :Yes   #Graph Eigenvalues?
    EV = Minv*C
    return plot_eigenvalues(EV,solve)
  end
  jacobi_displace_precon!(z,r) = mul!(z, Minv,r)
  @time u =  run_conjugate_gradient_precon(rand(size(g
            - FbarT' * (gδ ./ D))...), C, g - FbarT' * (gδ ./ D),
            jacobi_displace_precon!, 10e-16)
elseif solve == :AMG
  # Algebraic Multigrid
  println("Displacement - Algebraic Multigrid Preconditioner")
  ml = ruge_stuben(C)
  if EVgraph == :Yes   #Graph Eigenvalues?
    Minv = zeros(size(c))
    x = zeros(size(C,1),1)
    Z = zeros(size(C,1),1)
    for i in 1:size(C,1)
      x[i] = 1
      Z.=0
      Minv[:, i] = AlgebraicMultigrid.solve!(Z, ml, x, maxiter = 1,
                   calculate_residual = false)
      x[i] = 0
    end
    EV = Minv*B
    return plot_eigenvalues(EV, solve)
  end
  function AMG_displace_precon!(Z,r)
    Z.=0
    # Z is updated in place
    AlgebraicMultigrid.solve!(Z, ml, r, maxiter = 1,
                              calculate_residual = false)
  end
  @time u = run_conjugate_gradient_precon(rand(size(g
            - FbarT' * (gδ ./ D))...), C, g - FbarT' * (gδ ./ D),
            AMG_displace_precon!, 10e-16)
elseif solve == :IC
  # Incomplete Cholesky
  println("Displacement - Incomplete Cholesky Preconditioner")
  n = size(C,1)
  H = copy(C)
  run_incomplete_cholesky3(H,n)
  H = LowerTriangular(H)
  HT = UpperTriangular(sparse(H'))

  if EVgraph == :Yes   #Graph Eigenvalues?
    invH = inv(H)
    EV = invH*C*invH'
    return plot_eigenvalues(EV, solve)
  end
  function chol_displace_precon!(Z,r)
```

```
            ldiv!(Z,H,r)
            ldiv!(HT,Z)
          end
          @time u = run_conjugate_gradient_precon(rand(size(g
                    - FbarT' * (gδ ./ D))...), C, g - FbarT' * (gδ ./ D),
                    chol_displace_precon!, 10e-16)
        elseif solve == :Julia
          # Julia \
          println("Displacement - Julia Built-in")
          if EVgraph == :Yes   #Graph Eigenvalues?
            EV = copy(C)
            return plot_eigenvalues(EV, solve)
          end
          @time u[:] = C \ (g - FbarT' * (gδ ./ D))
        end
      end

      for e = 1:nelems
        (x, y) = lop[e].coord
        JH = lop[e].JH
        #@show vex(x[:],y[:],e)' * JH * vex(x[:],y[:],e)
        @views Δ[vstarts[e]:(vstarts[e+1]-1)] = (u[vstarts[e]:(vstarts[e+1]-1)] -
                                                 vex(x[:], y[:], e))
        ϵ[lvl] += (Δ[vstarts[e]:(vstarts[e+1]-1)]'
                  * JH * Δ[vstarts[e]:(vstarts[e+1]-1)])

      end
      ϵ[lvl] = sqrt(ϵ[lvl])
      @show (lvl, ϵ[lvl])
    end
    println((log.(ϵ[1:end-1]) - log.(ϵ[2:end])) / log(2))

    nothing
end
```

APPENDIX B:
Conjugate Gradient Implementation Julia Code

B.1 Conjugate Gradient Julia Code

```
function run_conjugate_gradient_precon(x0, A, b, precon!, tol)
    """
    Variable definitions:
    x0  =  initial guess
    A   =  matrix of interest
    b   =  right hand side (Ax = b)
    precon!   =  Preconditioner Function
    tol =  tolerance
    """

    # initial check
    # JEK Make copy of x0 so we can update x_new later
    x_new = copy(x0)        # current guess
    r_new = b - A * x0;     # current residual
    # JEK: Since Minv is a sparse diagonal matrix, the vector z_new was a
    #      sparse vector as well, which is slow in this case, so make
    #      it an array
    z_new = similar(r_new);
    precon!(z_new, r_new);  #applying inverse of preconditioner to r_new
                            #- produces z_new
    p_new = copy(z_new);    # current direction
    err = norm(r_new,Inf);  # Calculate error


    # return x0 if within tolerance
    if err < tol
        println("You guessed correctly!")
        return x0
    end

    i = 0

    # JEK: Preallocate these vectors
    x_old = similar(x_new)
    r_old = similar(r_new)
    z_old = similar(z_new)
    p_old = similar(p_new)

    # JEK: preallocate storage for the matrix vector product
```

```
Ap_old = similar(p_new)

# JEK: avoid repeatedly doing the dot product
#      (cache previous computation)
rT_z_new = r_new' * z_new

# further iterations
for outer i = 1:length(b)

    # JEK: Copy data into preallocated vectors
    x_old .= x_new;
    r_old .= r_new;
    z_old .= z_new;
    p_old .= p_new;

    # JEK: Cache previous dot product
    rT_z_old = rT_z_new

    #JEK: precompute the matrix vector product
    mul!(Ap_old, A, p_old)

    # calculate step size
    # JEK: Use precomputed Ap_old
    α = rT_z_old / (p_old' * Ap_old);

    # take a step in current conjugate direction
    # JEK: broadcast to make the code non-allocating
    @. x_new = x_old + α * p_old;

    # calculate new residual
    # JEK: Use precomputed Ap_old
    # JEK: broadcast to make the code non-allocating
    @. r_new = r_old - α * Ap_old;

    # compute absolute error and break if within tolerance
    # JEK: use inf norm to avoid allocation
    err = norm(r_new, Inf)

    if err < tol
        break
    end

    # Solve M*z_new = r_new
    # z_new = Minv * r_new
    # JEK: Use inplace matrix multiply
    precon!(z_new, r_new)

    # calculate new linear combination
    # JEK: Calculate new dot product
    rT_z_new = r_new' * z_new
```

```
        β =  rT_z_new / rT_z_old

        # calculate new conjugate direction
        # JEK: broadcast to make the code non-allocating
        @. p_new = z_new + β * p_old

    end

    println("Terminated in $i iterations")

    return x_new
end
```

B.2 Preconditioned Conjugate Gradient Julia Code

```
function run_conjugate_gradient_precon(x0, A, b, precon!, tol)
    """
    Variable definitions:
    x0  =  initial guess
    A   =  matrix of interest
    b   =  right hand side (Ax = b)
    precon!   =  Preconditioner Function
    tol =  tollerance
    """

    # initial check
    # JEK Make copy of x0 so we can update x_new later
    x_new = copy(x0)        # current guess
    #Minv = M^-1            # M inverse
    r_new = b - A * x0;     # current residual
    # JEK: Since Minv is a sparse diagonal matrix, the vector z_new was a
    #      sparse vector as well, which is slow in this case, so make
    #      it an array
    #z_new = Array(Minv * r_new)     # solve for z_new ***(using Minv)***
    z_new = similar(r_new);
    precon!(z_new, r_new);  #applying inverse of preconditioner to r_new
                            #- produces z_new
    p_new = copy(z_new);    # current direction
    err = norm(r_new,Inf);  # Calculate error


    # return x0 if within tolerance
    if err < tol
        println("You guessed correctly!")
        return x0
    end

    i = 0

    # JEK: Preallocate these vectors
    x_old = similar(x_new)
    r_old = similar(r_new)
    z_old = similar(z_new)
    p_old = similar(p_new)

    # JEK: preallocate storage for the matrix vector product
    Ap_old = similar(p_new)

    # JEK: avoid repeatedly doing the dot product
           (cache previous computation)
    rT_z_new = r_new' * z_new
```

```
# further iterations
for outer i = 1:length(b)

    # JEK: Copy data into preallocated vectors
    x_old .= x_new;
    r_old .= r_new;
    z_old .= z_new;
    p_old .= p_new;

    # JEK: Cache previous dot product
    rT_z_old = rT_z_new

    #JEK: precompute the matrix vector product
    mul!(Ap_old, A, p_old)

    # calculate step size
    # JEK: Use precomputed Ap_old
    α = rT_z_old / (p_old' * Ap_old);

    # take a step in current conjugate direction
    # JEK: broadcast to make the code non-allocating
    @. x_new = x_old + α * p_old;

    # calculate new residual
    # JEK: Use precomputed Ap_old
    # JEK: broadcast to make the code non-allocating
    @. r_new = r_old - α * Ap_old;

    # compute absolute error and break if within tolerance
    # JEK: use inf norm to avoid allocation
    err = norm(r_new, Inf)

    if err < tol
        break
    end

    # Solve M*z_new = r_new
    # z_new = Minv * r_new
    # JEK: Use inplace matrix multiply
    #mul!(z_new, Minv, r_new)  #***(using Minv)***
    precon!(z_new, r_new)

    # calculate new linear combination
    # JEK: Calculate new dot product
    rT_z_new = r_new' * z_new
    β =  rT_z_new / rT_z_old

    # calculate new conjugate direction
    # JEK: broadcast to make the code non-allocating
    @. p_new = z_new + β * p_old
```

```
    end

    println("Terminated in $i iterations")

    return x_new
end
```

APPENDIX C:
Julia Code Constructing Preconditioners

C.1 Jacobi Julia Code

```
Minv = inv(Diagonal(B))
```

C.2 Incomplete Cholesky Julia Code

```
function run_incomplete_cholesky3(A, n = size(A, 1))
    Anz = nonzeros(A)
    #incomplete cholesky
    @inbounds for k = 1:n
        # Get the row indices
        r1 = Int(SparseArrays.getcolptr(A)[k])
        r2 = Int(SparseArrays.getcolptr(A)[k+1]-1)
        r1 = searchsortedfirst(rowvals(A), k, r1, r2, Base.Order.Forward)
        # @assert r2 ≥ r1
        Anz[r1] = sqrt(Anz[r1])
        # Loop through non-zero elements and update the kth column
        for r = r1+1:r2
            i = rowvals(A)[r]
            A[i,k] = A[i,k]/A[k,k]
            A[k,i] = 0
        end
        # Loop through the remaining columns, k:n,
          and update them as needed
        # (that if [i,k] is non zero AND [i, j] is nonzero then we need to
        # update the value of [i, ]
        for r = r1+1:r2
            j = rowvals(A)[r]
            s1 = Int(SparseArrays.getcolptr(A)[j])
            s2 = Int(SparseArrays.getcolptr(A)[j+1]-1)
            s1 = searchsortedfirst(rowvals(A), j, s1, s2,
                 Base.Order.Forward)
            # @assert s2 ≥ s1
            for s = s1:s2
                i = rowvals(A)[s]
                A[i,j] = A[i,j] - A[i,k]*A[j,k]
                if i != j
                    A[j,i] = 0
                end
            end
        end
    end
    return A
end
```

C.3 Algebraic Multigrid Julia Code

```
ml = ruge_stuben(B)
function AMG_trace_precon!(Z,r)
    Z.=0
    # Z is updated in place
    AlgebraicMultigrid.solve!(Z, ml, r, maxiter = 1,
                              calculate_residual = false)
end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D:
Tabular Results

Table D.1. Results for trace method

		Julia Built-in	Conjugate Gradient						
		Regular	Regular	Jacobi		Incomplete Cholesky		Algebraic Multigrid	
		Results	Results	Results	Savings	Results	Savings	Results	Savings
Level 1	Iterations	N/A	456	288	3.7E-01	41	9.1E-01	43	9.1E-01
	Time to Build Precon	N/A	N/A	2.9E-04	N/A	5.6E-01	N/A	1.3E+00	N/A
	Time to Solve	2.2E-02	1.6E-01	2.1E-01	-3.2E-01	1.6E-01	-9.7E-03	1.3E+00	-6.4E+00
	Total Runtime	2.2E-02	1.6E-01	2.1E-01	-3.3E-01	7.3E-01	-3.5E+00	2.5E+00	-1.5E+01
	ϵ	3.0E-07	3.0E-07	3.0E-07	-7.8E-08	3.0E-07	2.8E-08	3.0E-07	-1.7E-08
Level 2	Iterations	N/A	631	372	4.1E-01	44	9.3E-01	45	9.3E-01
	Time to Build Precon	N/A	N/A	8.4E-04	N/A	3.7E+00	N/A	2.5E-02	N/A
	Time to Solve	9.6E-02	2.9E-01	2.6E-01	8.5E-02	8.9E-02	6.9E-01	2.1E-01	2.6E-01
	Total Runtime	9.6E-02	2.9E-01	2.6E-01	8.2E-02	3.8E+00	-1.2E+01	2.4E-01	1.6E-01
	ϵ	1.2E-08	1.2E-08	1.2E-08	-1.6E-06	1.2E-08	5.3E-07	1.2E-08	2.1E-06
Level 3	Iterations	N/A	880	494	4.4E-01	46	9.5E-01	46	9.5E-01
	Time to Build Precon	N/A	N/A	1.8E-03	N/A	3.2E+01	N/A	1.2E-01	N/A
	Time to Solve	3.7E-01	1.9E+00	1.5E+00	2.3E-01	3.5E-01	8.2E-01	1.2E+00	3.7E-01
	Total Runtime	3.7E-01	1.9E+00	1.5E+00	2.3E-01	3.2E+01	-1.6E+01	1.3+00	3.0E-01
	ϵ	4.3E-10	4.3E-10	4.3E-10	-1.4E-03	4.3E-10	6.7E-04	4.3E-10	-2.4E-04
Level 4	Iterations	N/A	1,236	669	4.6E-01	46	9.6E-01	47	9.6E-01
	Time to Build Precon	N/A	N/A	4.0E-03	N/A	2.8E+02	N/A	1.4E+00	N/A
	Time to Solve	1.8E+00	1.1E+01	6.7E+00	3.9E-01	1.2E+00	8.9E-01	4.8E+00	5.6E-01
	Total Runtime	1.8E+00	1.1E+01	6.7E+00	3.9E-01	2.8E+02	-2.5E+01	6.1E+00	4.4E-01
	ϵ	1.4E-11	2.5E-11	3.1E-11	-2.2E-01	2.7E-11	-5.1E-02	2.3E-11	8.6E-02

Table D.2. Results for displacements method

		Julia Built-in	Conjugate Gradient						
		Regular	Regular	Jacobi		Incomplete Cholesky		Algebraic Multigrid	
		Results	Results	Results	Savings	Results	Savings	Results	Savings
Level 1	Iterations	N/A	2,630	1,307	5.0E-01	81	9.7E-01	184	9.3E-01
	Time to Build Precon	N/A	N/A	5.4E-03	N/A	2.2E+00	N/A	8.8E-02	N/A
	Time to Solve	6.0E-01	2.9E+00	2.7E+00	5.8E-02	3.6E-01	8.7E-01	2.2E+00	2.3E-01
	Total Runtime	6.0E-01	2.9E+00	2.7E+00	5.8E-02	2.5E+00	1.2E-01	2.3E+00	2.0E-01
	ϵ	3.0E-07	3.0E-07	3.0E-07	-1.4E-07	3.0E-07	-2.9E-09	3.0E-07	-3.1E-08
Level 2	Iterations	N/A	5,303	2,295	5.7E-01	141	9.7E-01	271	9.5E-01
	Time to Build Precon	N/A	N/A	1.4E-02	N/A	5.3E+00	N/A	2.7E-01	N/A
	Time to Solve	2.2E+00	2.3E+01	1.7E+01	2.8E-01	1.5E+00	9.4E-01	1.2E+01	4.6E-01
	Total Runtime	2.2E+00	2.3E+01	1.7E+01	2.8E-01	6.8E+00	7.0E-01	1.3+01	4.5E-01
	ϵ	1.2E-08	1.2E-08	1.2E-08	-2.4E-06	1.2E-08	1.9E-06	1.2E-08	-5.1E-06
Level 3	Iterations	N/A	10,802	4,194	6.1E-01	272	9.8E-01	422	9.6E-01
	Time to Build Precon	N/A	N/A	4.7E-02	N/A	1.4E+01	N/A	1.5E+00	N/A
	Time to Solve	1.4E+01	1.6E+02	1.3E+02	1.8E-01	1.0E+01	9.4E-01	7.2E+01	5.6E-01
	Total Runtime	1.4E+01	1.6E+02	1.3E+02	1.8E-01	2.4E+01	8.5E-01	7.3E+01	5.5E-01
	ϵ	4.3E-10	4.3E-10	4.3E-10	4.5E-04	4.3E-10	-8.2E-03	4.3E-10	6.4E-03
Level 4	Iterations	N/A	21,781	7,988	6.3E-01	539	9.8E-01	641	9.7E-01
	Time to Build Precon	N/A	N/A	3.7E-01	N/A	4.2E+01	N/A	9.0E+00	N/A
	Time to Solve	1.3E+02	1.3E+03	8.9E+02	3.4E-01	7.8E+01	9.4E-01	4.4E+02	6.7E-01
	Total Runtime	1.3E+02	1.3E+03	8.9E+02	3.4E-01	1.2E+02	9.1E-01	4.5E+02	6.7E-01
	ϵ	1.4E-11	8.5E-11	8.7E-11	-1.9E-02	1.0E-10	-1.9E-01	7.8E-11	8.5E-02

Table D.3. Results for monolithic method

		Julia Built-in	Conjugate Gradient						
		Regular	Regular	Jacobi		Incomplete Cholesky		Algebraic Multigrid	
		Results	Results	Results	Savings	Results	Savings	Results	Savings
Level 1	Iterations	N/A	4,904	1,943	6.0E-01	341	9.3E-01	203	9.6E-01
	Time to Build Precon	N/A	N/A	4.0E-03	N/A	1.7E+00	N/A	8.9E-02	N/A
	Time to Solve	6.4E-01	4.6E+00	3.0E+00	3.5E-01	1.1E+00	7.6E-01	1.9E+00	5.9E-01
	Total Runtime	6.4E-01	4.6E+00	3.0E+00	3.4E-01	2.6E+00	4.5E-01	2.0E+00	5.7E-01
	ϵ	3.0E-07	3.0E-07	3.0E-07	4.4E-07	3.0E-07	-3.2E-07	3.0E-07	6.3E-08
Level 2	Iterations	N/A	9,683	3,051	6.9E-01	471	9.5E-01	250	9.7E-01
	Time to Build Precon	N/A	N/A	1.3E-02	N/A	4.2E+00	N/A	2.4E-01	N/A
	Time to Solve	2.7E+00	4.2E+01	1.9E+01	5.6E-01	5.1E+00	8.8E-01	9.9E+00	7.7E-01
	Total Runtime	2.7E+00	4.2E+01	1.9E+01	5.6E-01	9.4E+00	7.8E-01	1.0E+01	7.6E-01
	ϵ	1.2E-08	1.2E-08	1.2E-08	-9.6E-06	1.2E-08	-1.0E-06	1.2E-08	-1.71E-06
Level 3	Iterations	N/A	19,409	5,050	7.4E-01	687	9.6E-01	342	9.8E-01
	Time to Build Precon	N/A	N/A	6.0E-02	N/A	1.2E+01	N/A	1.6E+00	N/A
	Time to Solve	1.44E+01	3.1E+02	1.4E+02	5.5E-01	2.7E+01	9.1E-01	6.0E+01	8.1E-01
	Total Runtime	1.44E+01	3.1E+02	1.4E+02	5.5E-01	4.0E+01	8.7E-01	6.1E+01	8.0E-01
	ϵ	4.3E-10	4.3E-10	4.3E-10	-3.1E-03	4.3E-10	7.0E-04	4.3E-10	6.3E-05
Level 4	Iterations	N/A	38,816	8,861	7.7E-01	1,092	9.7E-01	514	9.9E-01
	Time to Build Precon	N/A	N/A	2.8E-01	N/A	4.0E+01	N/A	7.8E+00	N/A
	Time to Solve	1.6E+02	2.3E+03	9.6E+02	5.7E-01	1.7E+01	9.2E-01	3.2E+02	8.6E-01
	Total Runtime	1.6E+02	2.3E+03	9.6E+02	5.7E-01	2.1E+02	9.0E-01	3.3E+02	8.5E-03
	ϵ	1.4E-11	7.2E-11	8.6E-11	-2.0E-01	5.9E-11	1.8E-01	8.6E-11	-2.0E-01

Table D.4. Results for trace method using tolerance = 10^{-16} and infinite-norm

		Julia Built-in	Conjugate Gradient						
		Regular	Regular	Jacobi		Incomplete Cholesky		Algebraic Multigrid	
		Results	Results	Results	Savings	Results	Savings	Results	Savings
Level 1	Iterations	N/A	549	353	3.6E-01	52	9.1E-01	56	9.0E-01
	Time to Build Precon	N/A	N/A	3.5E-02	N/A	1.5E-01	N/A	2.4E-02	N/A
	Time to Solve	2.2E-02	2.6E-01	2.5E-01	3.9E-02	1.5E-01	4.3E-01	1.9E-01	2.7E-01
	Total Runtime	2.2E-02	2.6E-01	2.8E-01	-9.7E-02	6.9E-01	1.7E+00	2.1E-01	1.8E-01
	ϵ	3.0E-07	3.0E-07	3.0E-07	-2.0E-10	3.0E-07	-4.0E-10	3.0E-07	-2.1E-10
Level 2	Iterations	N/A	761	454	4.0E-01	55	9.3E-01	58	9.2E-01
	Time to Build Precon	N/A	N/A	7.8E-04	N/A	3.6E+00	N/A	2.5E-02	N/A
	Time to Solve	9.6E-02	3.0E-01	2.3E-01	2.5E-01	8.4E-02	7.2E-01	2.9E-01	2.8E-02
	Total Runtime	9.6E-02	3.0E-01	2.3E-01	2.4E-01	3.6E+00	-1.1E+01	3.2E-01	5.7E-02
	ϵ	1.2E-08	1.2E-08	1.2E-08	3.4E-09	1.2E-08	1.3E-09	1.2E-08	-7.3E-10
Level 3	Iterations	N/A	1,056	594	4.4E-01	57	9.5E-01	60	9.4E-01
	Time to Build Precon	N/A	N/A	1.6E-03	N/A	2.7E+01	N/A	1.3E-01	N/A
	Time to Solve	3.7E-01	2.2E+00	1.4E+00	3.8E-01	3.6E-01	8.3E-01	1.6E-01	2.6E-01
	Total Runtime	3.7E-01	2.2E+00	2.3E+00	3.8E-01	2.8E+01	-1.2E+01	1.8+00	2.0E-01
	ϵ	4.3E-10	4.3E-10	4.3E-10	-1.5E-07	4.3E-10	-1.2E-07	4.3E-10	-4.1E-08
Level 4	Iterations	N/A	1,479	789	4.7E-01	60	9.6E-01	62	9.6E-01
	Time to Build Precon	N/A	N/A	3.8E-03	N/A	2.0E+02	N/A	1.2E+00	N/A
	Time to Solve	1.8E+00	1.2E+01	7.6E+00	3.8E-01	1.6E+00	8.7E-01	5.9E+00	5.2E-01
	Total Runtime	1.8E+00	1.2E+01	7.6E+00	3.8E-01	2.0E+02	-1.6E+01	7.1E+00	4.2E-01
	ϵ	1.4E-11	1.4E-11	1.4E-11	3.0E-04	1.4E-11	-1.5E-05	1.4E-11	1.8E-04

Table D.5. Results for displacements method using tolerance = 10^{-16} and infinite-norm

		Julia Built-in	Conjugate Gradient						
		Regular	Regular	Jacobi		Incomplete Cholesky		Algebraic Multigrid	
		Results	Results	Results	Savings	Results	Savings	Results	Savings
Level 1	Iterations	N/A	3,232	1,540	5.1E-01	99	9.7E-01	219	9.3E-01
	Time to Build Precon	N/A	N/A	4.5E-03	N/A	2.5E+00	N/A	1.0E-01	N/A
	Time to Solve	6.0E-01	3.1E+00	3.0E+00	3.4E-02	4.2E-01	8.6E-01	3.0E+00	3.0E-02
	Total Runtime	6.0E-01	3.1E+00	3.0E+00	3.2E-02	2.0E+00	7.6E-02	3.1E+00	-1.7E-03
	ϵ	3.0E-07	3.0E-07	3.0E-07	-2.4E-09	3.0E-07	2.1E-09	3.0E-07	1.6E-09
Level 2	Iterations	N/A	6,355	2,703	5.8E-01	171	9.8E-01	332	9.5E-01
	Time to Build Precon	N/A	N/A	1.2E-02	N/A	6.3E+00	N/A	3.5E-01	N/A
	Time to Solve	2.2E+00	2.4E+01	1.9E+01	2.1E-01	1.8E+00	9.3E-01	1.6E+01	3.4E-01
	Total Runtime	2.2E+00	2.4E+01	1.9E+01	2.1E-01	8.1E+00	6.7E-01	1.6E+01	3.3E-01
	ϵ	1.2E-08	1.2E-08	1.2E-08	2.7E-08	1.2E-08	3.4E-08	1.2E-08	1.1E-08
Level 3	Iterations	N/A	13,005	4,946	6.2E-01	330	9.8E-01	498	9.6E-01
	Time to Build Precon	N/A	N/A	5.2E-02	N/A	2.0E+01	N/A	9.8E-01	N/A
	Time to Solve	1.4E+01	1.2E+02	1.3E+02	3.1E-01	1.4E+01	9.3E-01	9.5E+01	5.1E-01
	Total Runtime	1.4E+01	1.2E+02	1.3E+02	3.1E-01	3.4E+01	8.3E-01	9.6E+01	5.1E-01
	ϵ	4.3E-10	4.3E-10	4.3E-10	6.8E-07	4.3E-10	3.6E-07	4.3E-10	3.2E-07
Level 4	Iterations	N/A	26,354	9,334	6.5E-01	635	9.8E-01	765	9.7E-01
	Time to Build Precon	N/A	N/A	3.3E-01	N/A	2.1E+02	N/A	9.6E+00	N/A
	Time to Solve	1.3E+02	1.6E+03	1.0E+03	3.5E-01	6.8E+01	9.6E-01	5.8E+02	6.4E-01
	Total Runtime	1.3E+02	1.6E+03	1.0E+03	3.5E-01	2.8E+02	8.3E-01	5.9E+02	6.3E-01
	ϵ	1.4E-11	1.4E-11	1.4E-11	2.9E-04	1.4E-11	7.5E-04	1.4E-11	1.5E-04

Table D.6. Results for monolithic method using tolerance = 10^{-16} and infinite-norm

		Julia Built-in	Conjugate Gradient						
		Regular	Regular	Jacobi		Incomplete Cholesky		Algebraic Multigrid	
		Results	Results	Results	Savings	Results	Savings	Results	Savings
Level 1	Iterations	N/A	5,876	2,290	6.1E-01	428	9.3E-01	245	9.6E-01
	Time to Build Precon	N/A	N/A	4.4E-03	N/A	1.7E+00	N/A	6.9E-02	N/A
	Time to Solve	6.4E-01	5.6E+00	3.8E+00	3.2E-01	1.4E+00	7.5E-01	2.3E+00	5.9E-01
	Total Runtime	6.4E-01	5.6E+00	3.8E+00	3.2E-01	3.1E+00	4.4E-01	2.4E+00	5.8E-01
	ϵ	3.0E-07	3.0E-07	3.0E-07	5.6E-10	3.0E-07	2.5E-09	3.0E-07	2.1E-09
Level 2	Iterations	N/A	11,561	3,607	6.9E-01	565	9.5E-01	304	9.7E-01
	Time to Build Precon	N/A	N/A	1.6E-02	N/A	6.3E+00	N/A	2.7E-01	N/A
	Time to Solve	2.7E+00	9.2E+01	2.5E+01	7.3E-01	5.1E+00	9.5E-01	1.2E+01	8.7E-01
	Total Runtime	2.7E+00	9.2E+01	2.5E+01	7.3E-01	1.1E+01	8.8E-01	1.2E+01	8.7E-01
	ϵ	1.2E-08	1.2E-08	1.2E-08	-5.2E-08	1.2E-08	-4.1E-08	1.2E-08	-5.21E-08
Level 3	Iterations	N/A	23,213	5,955	7.4E-01	827	9.6E-01	416	9.8E-01
	Time to Build Precon	N/A	N/A	5.1E-02	N/A	3.3E+01	N/A	1.4E+00	N/A
	Time to Solve	1.44E+01	5.8E+01	1.6E+02	7.3E-01	1.6E+01	9.7E-01	6.6E+01	8.9E-01
	Total Runtime	1.44E+01	5.8E+01	1.6E+02	7.3E-01	4.9E+01	9.2E-01	6.8E+01	8.8E-01
	ϵ	4.3E-10	4.3E-10	4.3E-10	2.0E-06	4.3E-10	9.0E-07	4.3E-10	2.3E-06
Level 4	Iterations	N/A	46,709	10,428	7.7E-01	1,298	9.7E-01	627	9.9E-01
	Time to Build Precon	N/A	N/A	1.1E+03	N/A	1.9E+02	N/A	7.7E+00	N/A
	Time to Solve	1.6E+02	2.6E+03	1.1E+03	5.7E-01	5.6E+01	9.8E-01	4.1E+02	8.4E-01
	Total Runtime	1.6E+02	2.6E+03	2.3E+03	1.4E-01	2.4E+02	9.1E-01	4.2E+02	8.4E-03
	ϵ	1.4E-11	1.4E-11	1.4E-11	8.6E-03	1.4E-11	8.3E-03	1.4E-11	2.2E-03

APPENDIX E:
Julia Plotting Code

```
using LinearAlgebra, Plots; gr()
using Arpack

function plot_eigenvalues(H,solve)
#eigenvalues = eigvals(H)
eigenvalues = eigvals(Matrix(H))
display(size(H))
display(length(unique(eigenvalues)))
K = maximum(eigenvalues)/minimum(eigenvalues)
display(K)
ix = sortperm(abs.(eigenvalues))
if solve == :Julia
    scatter(real(eigenvalues), imag(eigenvalues), xlabel = "Re(x)",
    ylabel = "Im(x)", xlims=(-5, 50), ratio=1, ms=5, msw=0,label=false,
    title = "Eigenvalues for A")
else
    scatter(real(eigenvalues), imag(eigenvalues), xlabel = "Re(x)",
    ylabel = "Im(x)", xlims=(-5, 50), ratio=1, ms=5, msw=0,label=false,
    title = "Eigenvalues for A using $(solve) Preconditioner")
end
return(eigenvalues)
end
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] T. Bridges and M. J. King, "Supporting mission resilience through natural infrastructure," *The Military Engineer*, vol. 114, no. 737, pp. 46–48, Feb. 2022 [Online]. doi: https://samenews.org/tme-january-february-2022/.

[2] L. Wald and C. Collett, "The 2019 Ridgecrest, California, earthquake sequence," U.S. Geological Survey, Reston, VA, USA, Tech. Rep., 2021 [Online]. Available: https://earthquake.usgs.gov/storymap/index-ridgecrest.html

[3] Senator Dianne Feinstein (D-Calif.), "Feinstein announces $2.9 billion for China Lake reconstruction," Dec. 18, 2019 [Online]. Available: https://www.feinstein.senate.gov/public/index.cfm/press-releases?ID=F3A0ADB4-65EE-45B7-82E4-D405B2D3E5E9

[4] J. Jiang, B. A. Erickson, V. R. Lambert, J.-P. Ampuero, R. Ando, S. D. Barbot, C. Cattania, L. D. Zilio, B. Duan, E. M. Dunham, A.-A. Gabriel, N. Lapusta, D. Li, M. Li, D. Liu, Y. Liu, S. Ozawa, C. Pranger, and Y. van Dinther, "Community-driven code comparisons for three-dimensional dynamic modeling of sequences of earthquakes and aseismic slip," *Journal of Geophysical Research: Solid Earth*, vol. 127, no. 3, p. e2021JB023519, 2022, e2021JB023519 2021JB023519. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2021JB023519

[5] B. A. Erickson, J. Jiang, M. Barall, N. Lapusta, E. M. Dunham, R. Harris, L. S. Abrahams, K. L. Allison, J. Ampuero, S. Barbot, C. Cattania, A. Elbanna, Y. Fialko, B. Idini, J. E. Kozdon, V. Lambert, Y. Liu, Y. Luo, X. Ma, M. Best McKay, P. Segall, P. Shi, M. van den Ende, and M. Wei, "The Community Code Verification Exercise for Simulating Sequences of Earthquakes and Aseismic Slip (SEAS)," *Seismological Research Letters*, vol. 91, no. 2A, pp. 874–890, 01 2020. Available: https://doi.org/10.1785/0220190248

[6] B. A. Erickson, J. Jiang, V. Lambert *et al.*, "Incorporating full elastodynamic effects and dipping fault geometries in community code verification exercises for simulations of earthquake sequences and aseismic slip (SEAS)," EarthArXiv preprint, pp. 1–54, [Online] 2022. Available: https://eartharxiv.org/repository/view/3288/

[7] J. E. Kozdon, B. A. Erickson, and L. C. Wilcox, "Hybridized summation-by-parts finite difference methods," *Journal of Scientific Computing*, vol. 87, no. 3, pp. 1–28, 2021.

[8] G. J. Gassner, "A skew-symmetric discontinuous galerkin spectral element discretization and its relation to sbp-sat finite difference methods," *SIAM Journal on*

Scientific Computing, vol. 35, no. 3, pp. A1233–A1253, 2013. Available: https://doi.org/10.1137/120890144

[9] B. A. Erickson and S. M. Day, "Bimaterial effects in an earthquake cycle model using rate-and-state friction," *Journal of Geophysical Research: Solid Earth*, vol. 121, no. 4, pp. 2480–2506, 2016.

[10] L. Karlstrom and E. M. Dunham, "Excitation and resonance of acoustic-gravity waves in a column of stratified, bubbly magma," *Journal of Fluid Mechanics*, vol. 797, pp. 431–470, 2016.

[11] J. E. Kozdon, E. M. Dunham, and J. Nordström, "Interaction of waves with frictional interfaces using summation-by-parts difference operators: Weak enforcement of nonlinear boundary conditions," *Journal of Scientific Computing*, vol. 50, no. 2, pp. 341–367, 2012.

[12] G. C. Lotto and E. M. Dunham, "High-order finite difference modeling of tsunami generation in a compressible ocean from offshore earthquakes," *Computational Geosciences*, vol. 19, no. 2, pp. 327–340, 2015.

[13] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite Elements and Fast Iterative Solvers with Applications in Incompressible Fluid Dynamics*, 2nd ed. Oxford, UK: Oxford University Press, 2014.

[14] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," August 1994. Available: http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

[15] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, Pa: Society for Industrial and Applied Mathematics SIAM, 3600 Market Street, Floor 6, Philadelphia, PA, USA 19104, 2003.

[16] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, MD, USA 21218, 1996.

[17] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[18] K. Stüben, "A review of algebraic multigrid," *Journal of Computational and Applied Mathematics*, vol. 128, no. 1, pp. 281–309, 2001, numerical Analysis 2000. Vol. VII: Partial Differential Equations. Available: https://www.sciencedirect.com/science/article/pii/S0377042700005161

[19] Ranjanan, "AlgebraicMultigrid.jl," GitHub, Sep. 14, 2017 [Online]. Available: https://github.com/JuliaLinearAlgebra/AlgebraicMultigrid.jl

[20] J. W. Ruge and K. Stüben, "Algebraic multigrid," in *Multigrid Methods*. SIAM, 1987, pp. 73–130.

[21] James, "Thesis - James," NPS GitLab, May. 29, 2022 [Online]. Available: https://gitlab.nps.edu/timothy.james/thesis-james

[22] Kozdon, "HybridSBP," GitHub, Apr. 23, 2021 [Online]. Available: https://github.com/Thrase/HybridSBP

[23] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software*, vol. 35, no. 3, pp. 1–14, 2008.

[24] J. Poulson, "Fast parallel solution of heterogeneous 3d time-harmonic wave equations," Ph.D. dissertation, The University of Texas at Austin, Austin, TX, USA, 2012.

[25] A. P. Austin, N. Chalmers, and T. Warburton, "Initial guesses for sequences of linear systems in a gpu-accelerated incompressible flow solver," *SIAM Journal on Scientific Computing*, vol. 43, no. 4, pp. C259–C289, 2021. Available: https://doi.org/10.1137/20M1368677

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California