9-3-2013

Preconditioners constructed from the interpolative decomposition for the variable coefficient Poisson problem

Ambrose Quintana

Ambrose Quintana

Candidate

Mathematics and Statistics

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Stephen Lau , Chairperson

Daniel Appelo

Evangelos Coutsias

Preconditioners Constructed from the Interpolative Decomposition for the Variable Coefficient Poisson Problem

by

Ambrose David Quintana

B.S., Mathematics, University of New Mexico, 2004

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Mathematics

The University of New Mexico

Albuquerque, New Mexico

July, 2013

Dedication

To my Dad. He was a great father and the best friend a guy could ever ask for.. He always encouraged and pushed me to reach my goals. May he rest in peace..

"Can do!" – Felix Quintana

Acknowledgments

I would like to thank my advisor, Professor Stephen Lau, for his support and guidance. Thanks to my committee members as well. Some of this work was supported by NSF grant No. PHY 0855678. I would also like to thank my Uncle John for believing in me and helping me when I needed it most. Thank you Monica, for hanging in there with me through trying times. Thank you Mom, for the guidance and encouragement as well.

Preconditioners Constructed from the Interpolative Decomposition for the Variable Coefficient Poisson Problem

by

Ambrose David Quintana

B.S., Mathematics, University of New Mexico, 2004

M.S., Mathematics, University of New Mexico, 2013

Abstract

When trying to solve elliptical problems such as the Poisson problem on complicated domains, one procedure is to split the domain into a union of simpler subdomains. When solving these problems iteratively, it becomes important to be able to precondition the coupling between the subdomains. Using the Poisson problem as a test case, this thesis explores one idea for preconditioning this coupling, an idea based on interpolative decomposition and random matrices. We find that this procedure does create an efficient preconditioner to get at the coupling between subdomains.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objective

When trying to solve elliptical problems such as the Poisson problem on complicated domains, one procedure is to split the domain into a union of simpler subdomains. These subdomains may overlap or share common boundaries. We shall be looking at the non-overlapping case. When solving these problems iteratively, it becomes important to be able to precondition the coupling between the subdomains. Using the Poisson problem as a test case, this thesis explores one idea for preconditioning this coupling, an idea based on interpolative decomposition and random matrices.

1.2 Preliminaries

1.2.1 Linear systems

Finding solutions to nonsingular linear systems of the form

$$Ax = b, \tag{1.1}$$

where A is a large $m \times m$ matrix and b is a size m vector, is an ongoing issue today in many real world problems. There are basically two methods to solving these systems, a direct method or an iterative method.

In exact arithmetic, direct methods recover the exact solution $x = A^{-1}b$ in a finite number of operations. For very large matrices however, they tend to run into problems with time as well as storage. Gaussian Elimination is an example of the direct method.

Iterative methods create a sequence of improving approximations to the solution until this solution meets a satisfactory tolerance. Examples of iterative methods include Jacobi's Method, Gauss-Seidel, and Successive Over-Relaxation method. Another popular method, applicable to the case where A is symmetric positive definite, is the Conjugate Gradient Method which is direct in principle. For a size N system, it is direct in exact arithmetic if N iterations are done, however in application it is desirable to perform far fewer iterations. One reason these iterative methods have gained such popularity lately is that in some problems, the number of unknowns are a million or more, making direct methods unusable. In our study, we will use iterative methods combined with preconditioning to solve the variable coefficient Poisson problem.

1.2.2 Preconditioning

The use of preconditioning is an important ongoing area of study in today's technology fields. Given a linear system like (1.1), sometimes it is better to change the structure of the system by introducing a preconditioner in order to solve it more efficiently. Preconditioning is a way for improving the condition number of a matrix, which renders an associated linear system more amendable to solution by iterative methods.

We can solve (1.1) indirectly by solving

$$M^{-1}Ax = M^{-1}b. \tag{1.2}$$

M is chosen such that $k(M^{-1}A) \ll k(A)$, where k is the condition number. Therefore, with M chosen appropriately, (1.2) can be solved faster than the original problem.

Ideally, a good choice for M would be something close to A, yet also easy to construct, invert, and apply. Also, this preconditioned system (1.2) is not actually formed in practice, rather it has the ability (in code) to apply A and apply M^{-1}.

1.3 Outline

The outline for this thesis is as follows. In our first main chapter, we will go into detail about various factorization techniques, e.g., the singular value decomposition and pivoted-QR factorization. We will discuss different approximations which stem from these factorizations. We are interested in these approximations because they will reduce the computational cost involved in eventually finding satisfactory solutions to our problem. We will then show a technique incorporating random matrices as well as finding the interpolative decomposition of matrices. These concepts will be used in later chapters to test certain proposed methods for preconditioning using the Poisson problem. We will also look at a specific algorithm created by G. W. Stewart for finding the pivoted-QR factorization in a creative way. This algorithm will be used because of it is unique properties.

Chapter 3 will involve solving the variable coefficient Poisson problem using the finite element method. We will go into detail of what the finite element method is and how it is derived. We will describe the procedure for creating a mesh grid to use as our domain. We will create a $[0, 1] \times [0, 1]$ grid of randomly chosen interior points to use as our domain, but also try a $[0, 1] \times [0, 1]$ domain with uniformly spaced points for comparison as well. We will state our problem's details and define the solution as well so that we can test the error estimates in the end. We will also describe the set up for implementing the procedure for repeatable tests. To make sure our algorithm is set up correctly and converges correctly, we will create an FEM convergence test plot, which will also be used in comparison of different values of the variable coefficient.

The final chapter will entail putting concepts from the previous chapters to work for our ultimate goal of constructing and testing our preconditioners. First, we will split our previous domain up into two subdomains. They will share a boundary side which often times is not easy to account for. We will describe a way to account for this conforming piece using techniques from the earlier chapters. Then, using it along with the two subdomain pieces together, should work as a good preconditioner to solve the Poisson problem we wish to solve. To test the validity of our preconditioner, we will solve the problem using other preconditioners too, and we will show the results in a table. We can then compare and contrast the efficiencies of the different methods. We will also discuss the cost of constructing our preconditioner and finally, possibilities for applications in the future.

Chapter 2

Low-rank approximation of matrices

In this chapter we describe different procedures for finding low-rank approximations of matrices.

2.1 Singular value decomposition

The singular value decomposition, or SVD, is a matrix factorization which has great value in linear algebra problems. From [6], given $A \in \mathbb{R}^{m \times n}$ where $m \geq n$ and A has full column rank n, the reduced SVD of A is the factorization

$$A = \hat{U}\hat{\Sigma}V^T, \tag{2.1}$$

where

$$\hat{U} \in \mathbb{R}^{m \times n} \text{ has orthonormal columns,}$$
$$\hat{\Sigma} \in \mathbb{R}^{n \times n} \text{ is diagonal,}$$
$$V \in \mathbb{R}^{n \times n} \text{ is orthogonal.}$$

The full or "thick" SVD $A = U\Sigma V^T$ features a square $m \times m$ orthogonal matrix $U = [\hat{U}, \tilde{U}]$, with $\Sigma = \left[\begin{smallmatrix} \hat{\Sigma} \\ 0 \end{smallmatrix}\right] \in \mathbb{R}^{m \times n}$ padded with zeros. The diagonal entries σ_j of $\hat{\Sigma}$ are non-negative and in non-increasing order; $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. The SVD is one of the most reliable factorizations, but unfortunately it is also one of the most expensive. According to [7], the cost of computing the SVD could be as much as $\sim 4m^2n + 22n^3$.

The SVD also allows for low-rank approximations of a matrix. One way to do this is to get rid of "small" singular values. We choose a $k < n$ that we think (or hope) might give good results. Again, A's singular values are σ_i where $i = 1, ..., n$. Now we set $\sigma_{k+1:n}$ to zero "by hand". This defines a matrix

$$\hat{\Sigma}^{(k)} = \begin{pmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_k & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix} \tag{2.2}$$

with which we define

$$A^{(k)}_{\mathtt{SVD}} = \hat{U}\hat{\Sigma}^{(k)}V^T = \hat{U} \begin{pmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_k & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix} V^T. \tag{2.3}$$

$A^{(k)}_{\mathtt{SVD}}$ clearly has rank-k, since from (2.3) only k columns of $\hat{U}$ participate in the multiplication. In fact, $A^{(k)}_{\mathtt{SVD}}$ is the optimal rank-k approximation to A in the 2-norm (and also the Frobenius norm) [6]. That is

$$\|A - A^{(k)}_{\mathtt{SVD}}\|_2 = \min\{\|A - B\|_2 : B \text{ is rank } k\}. \tag{2.4}$$

While $A^{(k)}_{\mathtt{SVD}}$ is in this sense the <u>optimal</u> approximation of A, due to the high cost of the SVD, it is often prohibitively expensive to compute.

2.2 Pivoted-QR factorization

The QR-factorization can be computed in various ways. We have used an algorithm due to Stewart [4], described later in Section 2.2.4. For now we present the structure of the factorization, and show how it may be used to construct a low-rank approximation of a matrix A. From this point forward, we will be dealing with the case where A is a square matrix.

2.2.1 Basic structure

A "thin", or reduced, pivoted QR-factorization of a rank p matrix A is

$$A_{m\times m} \cdot P_{m\times m} = Q_{m\times p} \cdot R_{p\times m}. \tag{2.5}$$

P is a permutation matrix, Q is a matrix with orthonormal columns, and R is an upper triangular matrix. With some manipulation,

$$AP = QR = Q\left[R_{11}, R_{1,p+1}\right] = QR_{11}\left[I,\ R_{11}^{-1}R_{1,p+1}\right], \tag{2.6}$$

R_{11} is $p \times p$ and $R_{1,p+1}$ is $p \times m - p$. For better visualization, we use northwest indexing at times, for example with R_{11}, in which each block carries the indices of the element in the northwest corner. By assumption, (i) AP's first p columns are linearly independent, (ii) the columns of Q are orthonormal, and (iii) R_{11} is upper triangular with no zeros on the diagonal. From the last equation,

$$(AP)(:,1:p) = QR_{11}. \tag{2.7}$$

Due to condition (iii), R_{11} has linearly independent columns and is therefore invertible and therefore equation (2.6) makes sense. The cost of computing the QR factorization using Householder triangularization is $\sim \frac{4}{3}m^3$. [6]

2.2.2 Rank-k approximation

The pivoted-QR algorithm also defines a low-rank approximation to a matrix A. Although p is the rank of A, we would like to approximate A by a lower rank, say k.

Once again, we start with a thin pivoted QR-factorization of a rank p matrix A,

$$A_{m\times m}\cdot P_{m\times m}=Q_{m\times p}\cdot R_{p\times m}. \tag{2.8}$$

P is a permutation matrix, Q is a matrix with orthonormal columns, and R is an upper triangular matrix. We partition the equation $AP=Q_{m\times p}\cdot R_{p\times m}$ as

$$A=[Q_{m\times k},Q_{m\times(p-k)}]\cdot\begin{pmatrix}R_{11} & R_{1,k+1}\\ 0 & R_{k+1,k+1}\end{pmatrix}\cdot P^T_{m\times m}. \tag{2.9}$$

If A were exactly rank-k then $R_{k+1,k+1}=0$, so to get a low-rank approximation to A, we drop $R_{k+1,k+1}$ and write

$$A^{(k)}_{\mathtt{QR}}=Q_{m\times k}\cdot[R_{11},R_{1,k+1}]\cdot P^T_{m\times m} \tag{2.10}$$

as a rank-k approximation to A.

2.2.3 Errors

Now we shall calculate the error $\|A-A^{(k)}_{\mathtt{QR}}\|_2$. We find

$$\begin{aligned}A-A^{(k)}_{\mathtt{QR}}&=\Big[[Q_{m\times k},Q_{m\times(p-k)}]\cdot\begin{pmatrix}R_{11} & R_{1,k+1}\\ 0 & R_{k+1,k+1}\end{pmatrix}\cdot P^T\\ &\quad-[Q_{m\times k},Q_{m\times(p-k)}]\cdot\begin{pmatrix}R_{11} & R_{1,k+1}\\ 0 & 0\end{pmatrix}\cdot P^T\Big]\\ &=[Q_{m\times k},Q_{m\times(p-k)}]\cdot\begin{pmatrix}0 & 0\\ 0 & R_{k+1,k+1}\end{pmatrix}\cdot P^T.\end{aligned} \tag{2.11}$$

Realizing that $[Q_{m\times k},Q_{m\times(p-k)}]$ and P^T both have orthonormal columns and that $\|\cdot\|_2$ is invariant under orthogonal transformations [6], taking the 2-norm of both sides of the equation gives us

$$\begin{aligned}\|A-A^{(k)}_{\mathtt{QR}}\|_2&=\left\|[Q_{m\times k},Q_{m\times(p-k)}]\cdot\begin{pmatrix}0 & 0\\ 0 & R_{k+1,k+1}\end{pmatrix}\cdot P^T\right\|_2\\ &=\left\|\begin{pmatrix}0 & 0\\ 0 & R_{k+1,k+1}\end{pmatrix}\right\|_2\\ &=\|R_{k+1,k+1}\|_2\end{aligned} \tag{2.12}$$

Assuming we have chosen an appropriate k, $\|R_{k+1,k+1}\|_2$ will be small enough and $A^{(k)}_{\mathtt{QR}}$ will be a good approximation to A (assuming that a good approximation is even possible; it depends solely on the A as well as the k).

2.2.4 Stewart's algorithm

In the previous sections, we have described how a low-rank approximation comes from a factorization. In Section 2.1 we began with an SVD factorization, and then we threw out singular values to get our approximation. In Section 2.2.2 we started with a QR factorization, then we partitioned it to get rid of unwanted terms to get our approximation.

In this section we describe an algorithm by Stewart [4] to approximate a matrix A by a rank-k matrix $A^{(k)}$. This algorithm is based on a pivoted QR factorization, but it is halted at an appropriate time during the factorization. Unlike the previous approximations, one advantage this algorithm gives us is, that it allows us to create the approximation without actually doing the full factorization first.

Table 2.1: Truncated pivoted QR factorization algorithm from Stewart [4]

Given an $m \times n$ ($m \geq n$) matrix A this algorithm returns a truncated pivoted QR decomposition of A. Initially, the matrices Q and R are void.

```
 1.  $\nu_j = \|A[:,j]\|^2, \quad j = 1, \ldots, n$
 2.  Determine an index $n_1$ such that $\nu_{n_1}$ is maximal
 3.  For $q = 1$ to $n$
 4.    $A[:,q] \leftrightarrow A[:,n_q]$
 5.    $R[1:q-1,q] \leftrightarrow R[1:k-1,n_q]$
 6.    $Q[:,q] = A[:,q] - Q[:,1:q-1] * R[1:q-1,q]$
 7.    $R[q,q] = \|Q[:,q]\|$
 8.    $Q[:,q] = Q[:,q]/R[q,q]$
 9.    If necessary reorthogonalize $Q[:,q]$ and adjust $R[1:q-1,q]$
10.    $R[q,q+1:n] = Q[:,q]^T * A[:,q+1:n]$
11.    $\nu_j = \nu_j - R[q,j]^2, \quad j = q+1, \ldots, m$
12.    Determine an index $n_{q+1} \geq q+1$ such that $\nu_{n_{q+1}}$ is maximal
13.    If ($\nu_{n_{q+1}}$ is sufficiently small) leave $q$ fi
14.  end for $q$.
```

Table 2.1 shows the steps in Stewart's algorithm. To find the complexity of the algorithm, we must examine each line thoroughly. The cost of line 6 is $\sim mn^2 - 4mn$. Line 7 calculates a matrix norm which costs $\sim 2mn - n$. Line 8 costs $\sim mn$. The cost of line 10 is $\sim mn^2 - mn - n^2/2 + n/2$. And line 11 is $\sim 2mn - n^2 + n$. The overall cost for this procedure when not halted is roughly $\sim 2mn^2$. If we replace line 3 with "$q = 1$ to k" and remove line 13, then it becomes a rank k approximation. This would drop the cost down to $\sim 2mk^2$ which could be quite significant (as stated previously, the cost of computing the QR factorization for a square matrix using Householder triangularization is $\sim \frac{4}{3}m^3$).

This algorithm is also particularly useful for sparse matrices because it has the extra benefit of not destroying the sparsity of A, if present. The price paid here is that it relies on classical Gram-Schmidt which is known to be numerically unstable. Although one would think to use modified Gram-Schmidt to help with stability, this would be unwise. Modified Gram-Schmidt does projections onto all of the columns as it goes along, yet in this algorithm most likely, we will end up not even using many of those columns in the end. So, not only would this be doing unnecessary work, it would also destroy the sparsity of a sparse matrix. To combat this stability problem, we use the algorithm with reorthogonalization (see line 9).

2.3 Randomized algorithm for the interpolative decomposition

2.3.1 Interpolative decomposition

The interpolative decomposition of a matrix A involves choosing specific columns of the matrix and creating a "column skeleton" matrix B of the original matrix. Let matrix A be size $m \times m$ and have rank p. The decomposition is

$$A_{m\times m} = B_{m\times p} \cdot C_{p\times m}. \tag{2.13}$$

Matrix B's columns consist of a subset of the columns of matrix A. Matrix C has the form where some subset of its columns are the canonical basis vectors e_j where $j = 1, ..., p$.

A chosen $k < p$, where the $(k+1)^{st}$ singular value of A is sufficiently small, from [3] it turns out, will give us a good approximation,

$$B_{m\times k} \cdot C_{k\times m} \approx A_{m\times m}. \tag{2.14}$$

We now use results from Section 2.2 to find the B and C we need. Using equation (2.10), we have $A^{(k)}_{\mathtt{QR}} = QR_{11} \cdot [I, R_{11}^{-1}R_{1,k+1}] \cdot P^T$. The QR_{11} is the matrix B we are striving for, while $\left[I \; R_{11}^{-1}R_{1,k+1}\right] P^T$ is our C. The algorithm from Table 2.1 will be used to collect all these important pieces and have ready at hand for use in the upcoming section.

2.3.2 The randomized algorithm

We will now be describing a technique using random matrices to construct an interpolative decomposition using a chosen rank as in Section 2.2.2 together with concepts from previous sections. This follows closely from [3]. As stated previously, the Stewart algorithm is especially good for sparse matrices and for retaining the matrix's sparsity. This algorithm will be used, although here the issue of sparsity plays no role.

First we form a new $\ell \times m$ matrix Y by multiplying our original matrix A by a randomly generated $\ell \times m$ matrix N. Here ℓ is our guess for k, perhaps a little larger, $p > \ell \gtrsim k$. The matrix N's entries are each distributed as a Gaussian random variable of zero mean and unit variance.

$$Y_{\ell\times m} = N_{\ell\times m} \cdot A_{m\times m}. \tag{2.15}$$

From general matrix multiplication, Y's rows are linear combinations of matrix A's rows. The coefficients in these linear expansions are random variables, and we therefore assume that such a recombination of the rows leaves the linear dependencies of the column space intact. We can then apply the algorithm in Table 2.1 to this new Y to find its interpolative decomposition.

Much like (2.6) in Section 2.2, we now seek a factorization

$$Y_{\ell\times m} \cdot P_{m\times m} = \tilde{Q}_{\ell\times k} \cdot R_{k\times m}. \tag{2.16}$$

The R we get here should be the same one as the one in (2.6) in theory but in practice, it may be a little off due to round-off error. The algorithm will identify the important k chosen columns of Y. We then use those k chosen columns but instead of Y, we choose them from our original matrix A. This becomes our column skeleton matrix B that we were striving for. The algorithm also returns for us its R_{11}, $R_{1,k+1}$, and P which we will need to create the matrix

$$C = \left[I \; R_{11}^{-1} R_{1,k+1}\right] P^T. \tag{2.17}$$

We now have the pieces to make A,

$$B_{m\times k} \cdot C_{k\times m} \approx A_{m\times m}. \tag{2.18}$$

Use of random matrices allows us to compute the relevant factorization on a smaller problem. The randomized algorithm used here is displayed in Appendix B. This method has a high probability of approximating the matrix A within reason, while taking less operations in doing so.

2.3.3 Testing the algorithm

In order to know if the algorithm in Table 2.1 is indeed in working order, we must test it out to see if the results are reasonable. We will repeat an experiment demonstrated in [3].

We apply the algorithm to matrices A given by the formula

$$A = \frac{\Delta^{100}}{||\Delta^{100}||_2} + \frac{1}{\nu^2}\, c \cdot c^T, \tag{2.19}$$

where c is the $\nu^2 \times 1$ column vector whose entries are all ones, and Δ is the standard five-point discretization of the Laplacian on a $\nu \times \nu$ uniform grid (all of the diagonal entries of Δ are equal to -4, $\Delta_{p,q} = 1$ if the grid points p and q are neighbors, and all other entries of Δ are zeros.) A is an $m \times m$ matrix, and $m = \nu^2$. The Laplacian to such a high order features high-order derivatives and therefore Δ^{100} has a very large null space.

The results of the set of tests are shown in Table 2.2. Table 2.2 's columns are:

- m is the dimensionality of the $m \times m$ matrix A.
- k is the rank of the matrix approximating A.
- ℓ is the first dimension of the $\ell \times m$ matrix N.
- σ_{k+1} is the $(k+1)st$ greatest singular value of A, that is, the spectral norm of the difference between A and the $\underline{\text{optimal}}$ rank-k approximation to A.
- δ is the spectral norm of the difference between the original matrix A and its approximation obtained via the algorithm of appendix A. We denote the spectral (ℓ^2 - operator) norm of A by $\|A\|_2$. $\|A\|_2$ is the greatest singular value of A.

The singular values σ_i measure the errors in the 2-norm. Again, σ_{k+1} gives the optimal rank-k approximation in the 2-norm, although computing this requires computing the SVD of the matrix which scales like $O(m^3)$ for an $m \times m$ matrix. The computed δ's come at a cost of $O(\ell \cdot m^2)$ and yet are acceptably close to the optimal σ_{k+1} values.

The δ's from Table 2.2 were chosen as a representative of many; trial runs, ran with different random matrix Ns per trial. The results in Table 2.2 are consistent with those obtained in [3].

Table 2.2: Interpolative Decomposition of the $m \times m$ matrix A.

m	k	ℓ	σ_{k+1}	δ
400	48	56	.277E-08	.361E-07
1600	192	200	.449E-08	.234E-06
3600	432	440	.457E-08	.610E-06
6400	768	776	.553E-08	.218E-05
10000	1200	1208	.590E-08	.367E-05

Chapter 3

Finite element method - 2D Poisson problem

The finite element method (FEM) is a numerical method used in computational analysis, mechanical and structural engineering problems, as well as problems involving liquid flow and heat loss. As this presentation follows [2] closely, our focus is on using the FEM to solve partial differential equations (PDE).

3.1 Variable coefficient Poisson problem

Let u and f be functions of x and y. The standard two-dimensional Dirichlet-Poisson problem is

$$-\Delta u = f \text{ on } \Omega, \quad \text{where } \Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \text{ and } u = 0 \text{ on } \partial\Omega. \tag{3.1}$$

Here, Δ is the Laplace operator. Written another way,

$$\Delta u = \nabla \cdot \nabla u, \tag{3.2}$$

where ∇u is the *gradient* of u, i.e. $\nabla u = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right)$.

We will work with a variable coefficient version of (3.1). Generalizing (3.2) we consider

$$-\nabla \cdot \rho \nabla u = f \text{ in } \Omega, \quad u = 0 \text{ on } \partial\Omega, \tag{3.3}$$

where our domain Ω is the $[0,1]\times[0,1]$ square in $\mathbb{R}^2$ with boundary $\partial\Omega$. The functions f and ρ are given, and ρ is smooth and strictly positive on Ω.

We seek a weak formulation of (3.3). This weak formulation will then be discretized in a finite dimensional space, which will result in a linear problem whose solution approximately solves the original problem.

We now multiply (3.3) by a test function ϕ and integrate over Ω, thereby finding

$$-\int_\Omega \phi \nabla \cdot \rho \nabla u \, dA = \int_\Omega \phi f \, dA, \qquad \text{where } \phi \in V.$$

The space V is defined as follows:

$$V = \big\{\phi : \ \phi \text{ is continuous on } \Omega, \ \partial\phi/\partial x \text{ and } \partial\phi/\partial y \text{ are piecewise continuous on } \Omega, \text{ and } \phi = 0 \text{ on } \partial\Omega\big\}.$$

Integration by parts shows that

$$\begin{aligned} -\int_\Omega \phi \nabla \cdot \rho \nabla u \, dA &= -\int_\Omega \Big(\nabla \cdot \big(\phi \rho \nabla u\big) - \rho \nabla \phi \cdot \nabla u\Big) \, dA \\ &= -\int_{\partial\Omega} \phi \rho \, \hat{n} \cdot \nabla u \, ds + \int_\Omega \rho \nabla \phi \cdot \nabla u \, dA. \end{aligned}$$

Since $\phi \in V$, we know $\phi\big|_{\partial\Omega} = 0$, and so

$$-\int_{\partial\Omega} \hat{n} \cdot \phi \rho \nabla u \, ds = 0.$$

Therefore, for any such test function

$$-\int_\Omega \phi \nabla \cdot \rho \nabla u \, dA = -\int_\Omega \rho\big(\nabla\phi \cdot \nabla u\big) \, dA.$$

The weak formulation of (3.3) is

$$\int_\Omega \phi f \, dA = -\int_\Omega \rho\big(\nabla\phi \cdot \nabla u\big) \, dA \quad \forall \quad \phi \in V. \tag{3.4}$$

We now have a weak formulation of (3.3). We shall now discretize (3.4) using the Finite Element Method.

3.2 Triangularization of geometry and FEM approximation

In this section, we create a triangularization and a discretization in a finite dimensional space; as described in [2]. We must now construct a finite dimensional subspace V_h of V. For simplicity we shall assume Ω is a polygonal domain. Now, we create a triangulation of Ω by subdividing it into a set $\mathcal{T} = \{T_1, ..., T_m\}$ of non-overlapping triangles T_i,

$$\Omega = T_1 \cup T_2 \cup ... \cup T_m.$$

We define V_h as follows:

$$V_h = \big\{\phi : \ \phi \text{ is continuous on } \Omega,\ \phi\big|_T \text{ is linear for } T \in \mathcal{T}, \\ \text{and } \phi = 0 \text{ on } \partial\Omega\big\}.$$

This definition ensures that $V_h \subset V$. The space V_h consists of all the continuous functions that are linear on each triangle T and vanish on $\partial\Omega$.

We now must choose a basis corresponding to the triangularization for the set of test functions. We choose values $\phi(n_i)$ at the nodes $n_i, i = 1, ..., M$ of $\mathcal{T}$ but exclude the nodes on the boundary since $\phi = 0$ on $\partial\Omega$. The corresponding piecewise linear basis functions ϕ_j are defined by,

$$\phi_j(n_i) = \delta_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases} \quad \text{where } i, j = 1, ..., M.$$

We shall replace ϕ with ϕ_k for any k, and replace $u(x)$ with

$$u_h(x) = \sum_{j=1}^{M} u_j \phi_j(x), \ \text{for } x \in \Omega, \ \text{where } u_j = u_h(x_j).$$

We now enforce (3.4) for all $\phi \in V_h$, rather than $\phi \in V$. Since (3.4) is linear in ϕ, this restriction is equivalent to

$$\begin{aligned}\int_\Omega \phi_k f\, dA &= -\int_\Omega \rho\big(\nabla\phi_k \cdot \nabla u_h\big)\, dA \quad \forall\, k \\ &= -\int_\Omega \rho\big(\nabla\phi_k \cdot \nabla\big(\sum_{j=1}^{M} u_j\phi_j\big)\big)\, dA \\ &= -\sum_{j=1}^{M}\left[\int_\Omega \rho\nabla\phi_k \cdot \nabla\phi_j\, dA\right] u_j.\end{aligned}$$

Introduce the matrix S with elements

$$S_{kj} = \int \rho\nabla\phi_k \cdot \nabla\phi_j dA, \tag{3.5}$$

which obeys

$$\int_\Omega \rho\big(\nabla\phi_k \cdot \nabla u_h\big)\, dA = \sum_{j=1}^{M} S_{kj}u_j$$

and the right hand side of our linear system is

$$b_k = \int_\Omega \phi_k f\, dA. \tag{3.6}$$

Clearly, since the dot product is symmetric, this matrix S is symmetric. Also, we have

$$\begin{aligned}\langle v, Sv\rangle &= v^T S v \\ &= \sum_{i,j=1}^{M} v_i s_{ij} v_j \\ &= \sum_{i,j=1}^{M} v_i \left(\int \rho\nabla\phi_i \cdot \nabla\phi_j dA\right) v_j \\ &= \int \rho\left(\nabla v_h \cdot \nabla v_h\right) dA \qquad \text{where } v_h = \sum_{i=1}^{M} v_i\phi_i \\ \langle v, Sv\rangle &= \int \rho\left\|\nabla v_h\right\|^2 dA > 0 \quad \text{for } v \neq \vec{0}.\end{aligned}$$

Since $\rho > 0$, we see that S is positive definite. S is also a sparse matrix since most of its entries are zero ($S_{kj} = 0$ when n_j, n_k are not on the same triangle T).

Our problem is now discretized in a finite dimensional space which will allow us to proceed in finding an approximate solution.

3.3 Numerical implementation for $\Omega = [0,1] \times [0,1]$

To implement our algorithm, we must first input certain parameters. The number of edge points per side, n_{edge}, around our square mesh must be chosen and will dictate how many internal nodes, n_{int}, there will be; $n_{\text{int}} = (n_{\text{edge}} - 1)^2$. Also, we call the total number of boundary points $n_{\text{bnd}} = 4 \times n_{\text{edge}}$.

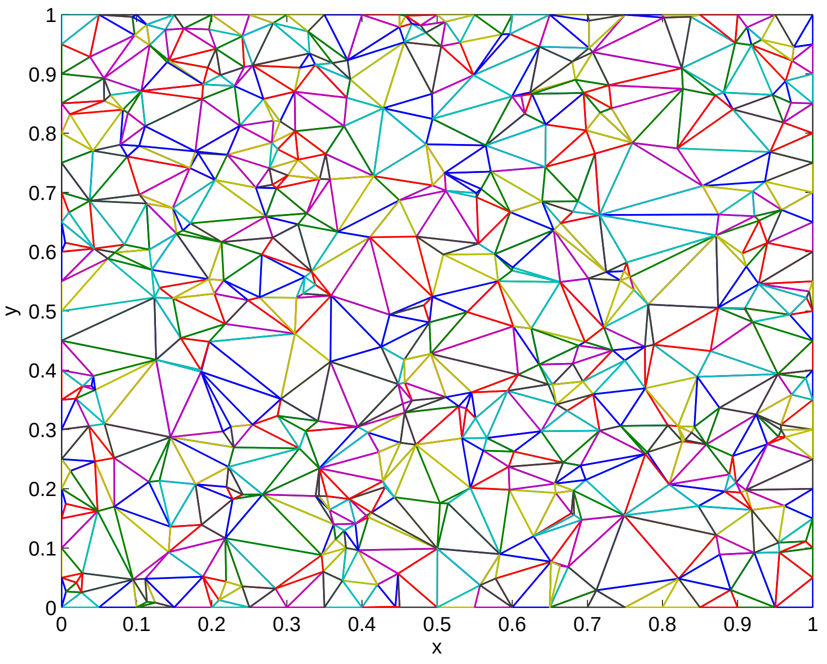

Figure 3.1: Triangle mesh on $[0,1] \times [0,1]$ grid. $n_{\text{edge}} = 20$ and $n_{\text{int}} = 361$.

The internal nodes placement is chosen at random which in turn creates random sized triangles. Matlab's built in function `delaunay` is used to create these triangles and the output is written in the form $G(k, \alpha)$ where k is the number of triangles and $\alpha = 1, 2, 3$. Once the grid is in place, the triangles are formed and the specific nodes from each triangle are recorded. Figure 3.1 shows this triangularization on the $[0,1] \times [0,1]$ grid for $n_{\text{int}} = 361$.

At times, we will also consider using a uniform grid. The uniform grid is comprised basically of right-angle isosceles triangles, rather than equilateral triangles. As in the random case, the triangularizations considered are easily generated with Matlab's `delaunay` function. Figure 3.2 shows an example of a uniform mesh, where the internal nodes are not chosen at random. We are considering both types of meshes to study the different effects of each. The randomized mesh is considered bad, while the uniform one is considered good.

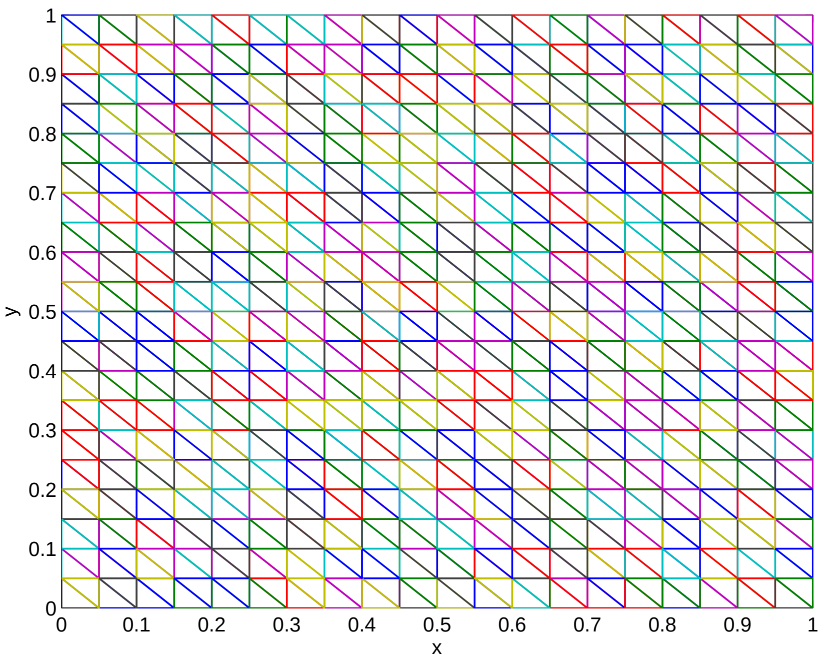

Figure 3.2: Triangle mesh on a uniform $[0,1]\times[0,1]$ grid. $n_{\text{edge}} = 20$ and $n_{\text{int}} = 361$.

As demonstrated in section 1.8 of [2], we will now build the triangularization linear system using piecewise linear finite elements. This is done by computing the local matrix elements of the system, S_{kj}, given by (3.5) along with the right hand side elements, b_k, given by (3.6). We know that $S_{kj} \neq 0$ only if n_k and n_j are of the same triangle $T^{(k)} \in G$. Then $G(\alpha, \beta)$, $\alpha = 1, 2, 3$, are the numbers of the vertices of $T^{(k)}$, and the x, y coordinates for these vertices are given. Knowing the vertices of $T^{(k)}$ we can now compute the local matrix elements $S^{(k)} = s^{(k)}_{\alpha\gamma}$, $\alpha, \beta = 1, 2, 3$ for element $T^{(k)}$

$$s^{(k)}_{\alpha\beta} = \int_{T^{(k)}} \rho \nabla\psi_\alpha \nabla\psi_\beta \, dA,$$

where ψ_α is the linear function on $T^{(k)}$ that takes the following values:

$$\psi^{(k)}_\alpha = \phi_j\big|_{T^{(k)}},$$

$$\psi_\alpha^{(k)}(n_{G(k,\beta)}) = \delta_{\alpha\beta} = \begin{cases} 1 \text{ if } \alpha = \beta \\ 0 \text{ if } \alpha \neq \beta \end{cases} \quad \text{where } \alpha, \beta = 1, 2, 3.$$

Now we create a matrix M that maps the triangles to a reference triangle T_{ref},

$$M^{(k)} = \begin{bmatrix} x(G(k,2)) - x(G(k,1)) & x(G(k,3)) - x(G(k,1)) \\ y(G(k,2)) - y(G(k,1)) & y(G(k,3)) - y(G(k,1)) \end{bmatrix}$$

and

$$T_{ref} = \{(s,t) : 0 \le s \le 1 \ , \ 0 \le t \le 1 - s\},$$

where

$$F^{(k)} : T_{ref} \longrightarrow T^{(k)},$$
$$\begin{bmatrix} s \\ t \end{bmatrix} \longmapsto \begin{bmatrix} x \\ y \end{bmatrix} = M^{(k)} \begin{bmatrix} s \\ t \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}.$$

The reference triangle basis functions become

$$\theta_\alpha(s,t) = \psi_\alpha\left(F^{(k)}(s,t)\right) = \psi_\alpha\left(F_1^{(k)}(s,t) \ , \ F_2^{(k)}(s,t)\right).$$

We use the transformation,

$$s_{\alpha\beta}^{(k)} = \left|det M\right| \int_{T_{ref}} \rho \nabla\theta_\alpha \cdot \nabla\theta_\beta \ ds \ dt$$

to better compute these integrals, which are computed using 3-point Gaussian quadrature. Our algorithm loops over all the elements $T^{(k)}$ and successively adds the contributions from each together to get the global matrix S.

$$S\left(G(k,\alpha), G(k,\beta)\right) = S\left(G(k,\alpha), G(k,\beta)\right) \ + \ s_{\alpha\beta}^{(k)}.$$

We can also compute

$$b_\alpha^{(k)} = \int_{T^{(k)}} f\psi_\alpha \ ds, \quad \alpha = 1, 2, 3$$

and similarly,

$$b\left(G(k,\alpha)\right)) = b\left(G(k,\alpha)\right)) \ + \ b_\alpha^{(k)}.$$

3.4 Numerical tests

3.4.1 Solver

For our testing purposes, we shall be using the method of manufactured solutions. We choose the exact solution as

$$u(x,y) = \sinh\big(Nx(1-x)\big)\sin\big(M\pi y\big), \tag{3.7}$$

which determines the right hand side to be

$$\begin{aligned} f\,(x,y) = \rho(x,y)\Big\{&2\nu\big(x-1/2\big)\big(1-2x\big)\cosh\big[Nx(1-x)\big]\sin(M\pi y) \\ &+ 2M\pi\nu\big(y-1/2\big)\sinh\big[Nx(1-x)\big]\cos(M\pi y) \\ &- \big[M^2\pi^2 - N^2(1-2x)^2\big]\sinh\big(Nx(1-x)\big)\sin(M\pi y) \\ &\quad + \ 2N\cosh\big(Nx(1-x)\big)\sin(M\pi y)\Big\}. \end{aligned}$$

and

$$\rho(x,y) = e^{-\nu\big[(x-1/2)^2+(y-1/2)^2\big]}. \tag{3.8}$$

We use constants, $M = 1$, $N = 0.5$, and $\nu = \{0.01, 0.001, 0.0001, 0.00001, 0\}$. Using smaller and smaller ν, our problem becomes nearly the ordinary Poisson problem.

We shall test our grid with an increasing amount of points: 80^2, 90^2, 100^2, ... As the number of points increases, the accuracy should also increase. The finite element method with hat functions is second order accurate.

To visualize the errors we are getting, we shall plot the error estimates for $n_{edge} = 80$. Figure 3.3 shows the point-wise error estimates for $n_{edge} = 80$ points on the random grid, and Figure 3.4 shows the point-wise error estimates for $n_{edge} = 80$ points on the uniform grid. Notice how much smoother the plot of the error estimates using the uniform grid is compared to results from the random grid.

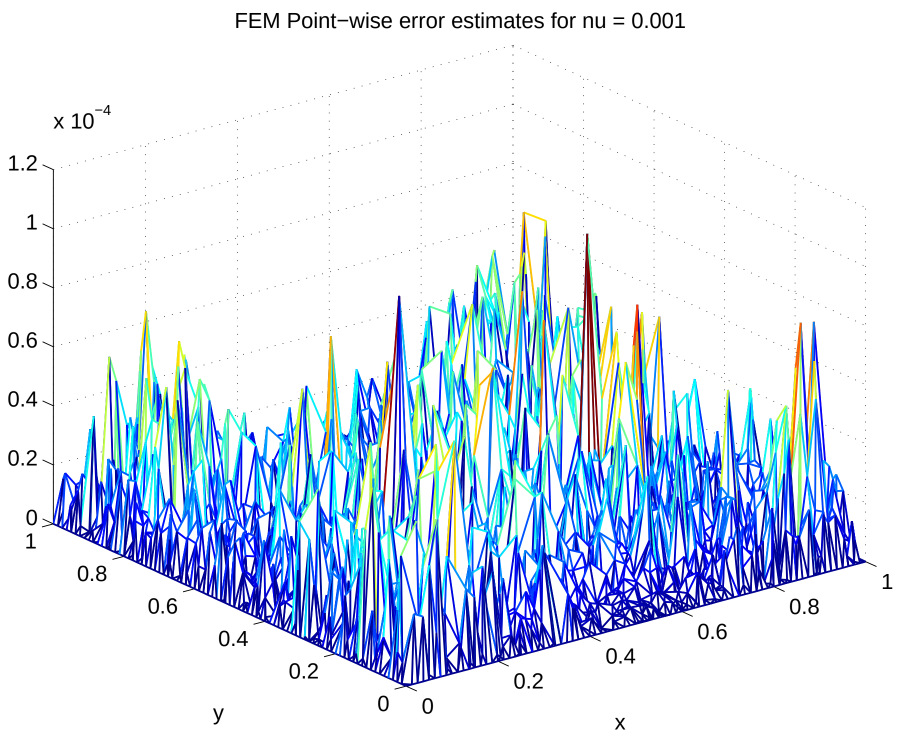

Figure 3.3: FEM POINT-WISE ERROR ESTIMATES. Graph of the point-wise error estimate for $n_{edge} = 80$ points.

3.4.2 Finite element convergence

The Finite Element Method quantifies the accuracy and reliability of a numerical solution by error estimates on the FEM error vs. the mesh spacing of the FEM mesh. We refer the reader to [5] for a more detailed description. These error estimates are

$$\|u(\cdot,\cdot) - u_h(\cdot,\cdot)\|_{L_2(\Omega)} = \left(\iint \left|u(x,y) - u_h(x,y)\right|^2 dxdy\right)^{1/2} \leq Ch^2, \tag{3.9}$$

as $h \to 0$. Here, $u(x,y)$ denotes the PDE solution of the problem and $u_h(x,y)$ the FEM solution. The mesh size is denoted by $h = 1/n_{edge}$ and C is a constant.

We would like to set up an easily repeated uniform mesh refinement to test our algorithm for correct convergence. One good test for reliability of a FEM solution is to refine the FEM mesh, compute the solution again on the refined mesh, and qualitatively compare the solutions on the different meshes. In solving the system, we use the conjugate gradient method preconditioned with the incomplete Cholesky

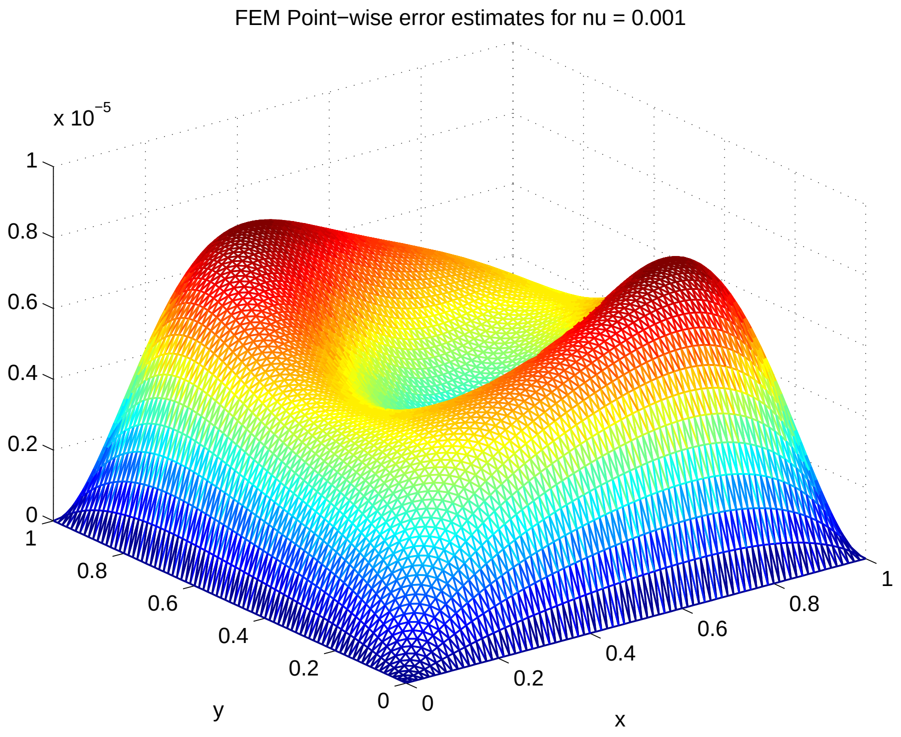

Figure 3.4: FEM Point-wise error estimates on a uniform grid. Graph of the point-wise error estimate for $n_{edge} = 80$ points.

factorization. For the conjugate gradient method we use a tolerance of 10^{-10}, and for the incomplete Cholesky factorization we use a drop tolerance of 10^{-7}.

Figure 3.5 shows a log-log plot of the error on the left hand side of (3.9) vs. the reciprocal of the mesh spacing, $1/h$. In this form, the estimate (3.9) plots as a line with its slope being equal to the negative of the convergence order 2. The predicted slope of -2 is shown as a dashed line in the figure.

We also perform the convergence test from the results gotten when using the uniform meshes. These are shown in Figure 3.6. These results follow the predicted slope of -2 even more precisely than the non-uniform cases which should be expected, and the errors themselves are considerably lower then the non-uniform cases. The errors are basically an order of magnitude smaller when using the uniform grid.

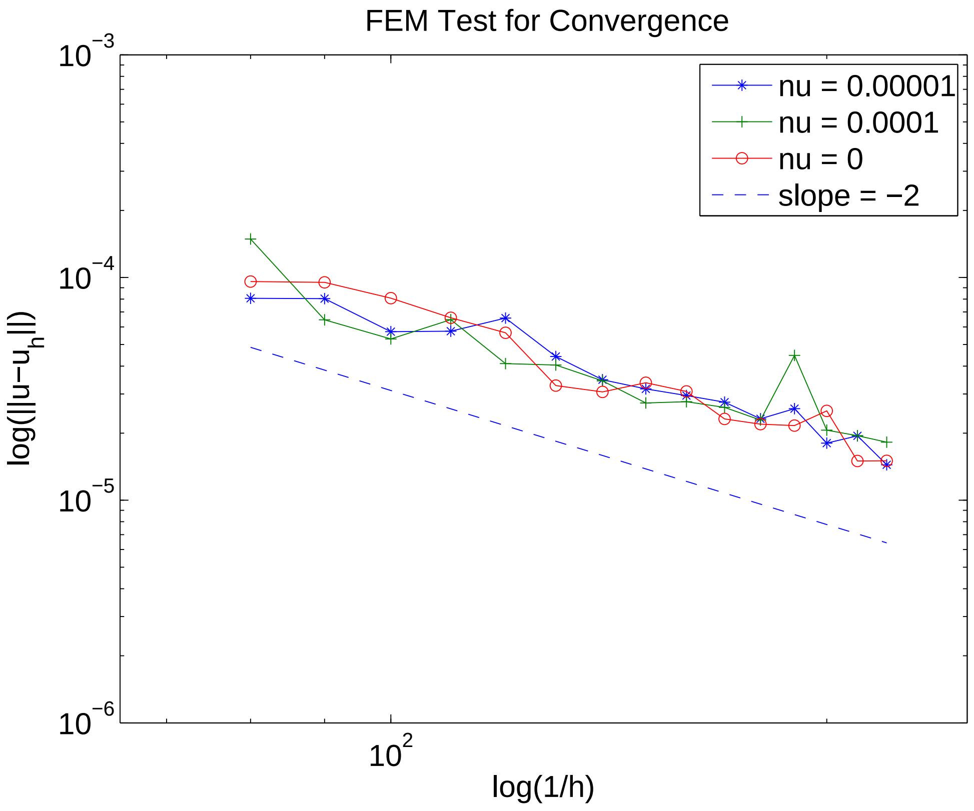

Figure 3.5: FEM Convergence test. log(error-norm) vs. log(1/h).

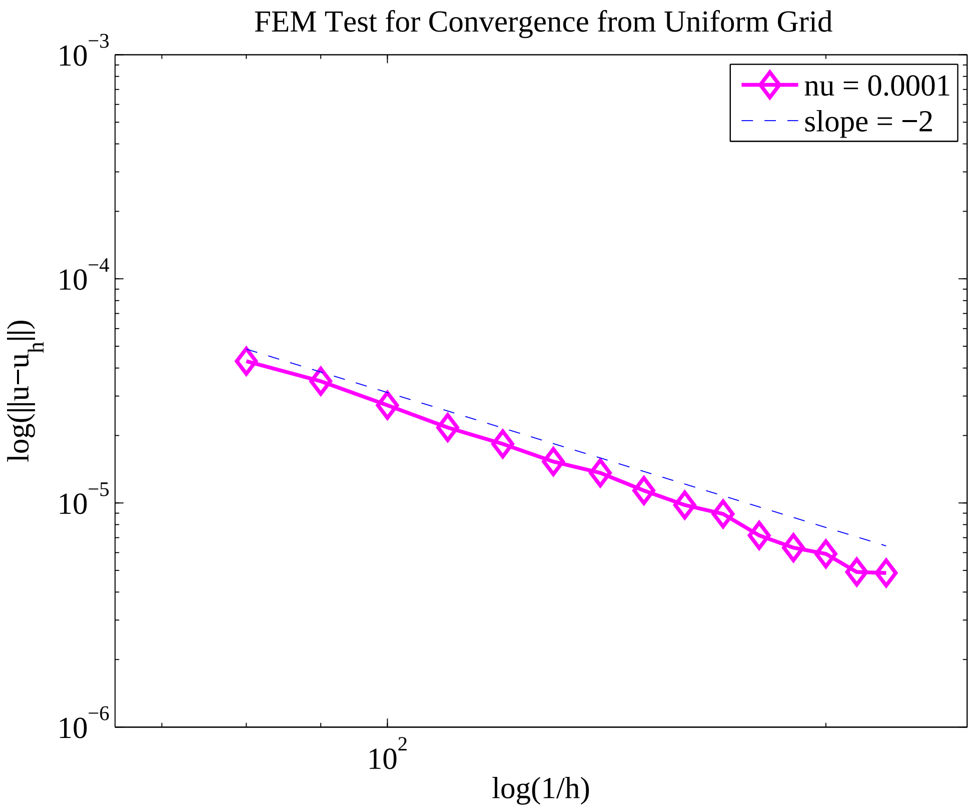

Figure 3.6: FEM Convergence test using a uniform grid. log(error-norm) vs. log(1/h).

Chapter 4

Solving the Poisson problem using preconditioning

In this chapter we will solve the variable coefficient Poisson problem on a $[0,1]\times[0,1]$ grid using finite element discretization. In order to examine the preconditioning of conforming subdomains through the techniques introduced in previous chapters, we here (somewhat artificially) split our rectangular domain into two subdomains.

4.1 Problem set up

We will be solving the variable-coefficient Poisson problem

$$-\nabla\cdot\rho\nabla u = f \text{ in } \Omega, \quad u = 0 \text{ on } \partial\Omega,$$

where $\Omega = [0,1]\times[0,1]$.

We refer the reader to Section 3.4, as we choose the same $\rho(x,y)$, $u(x,y)$, and $f(x,y)$. We pick our constants to be $M = 1$, $N = 0.5$, and $\nu = 10^{-3}$. Now we move on to the details and implementation of our procedure.

4.2 Obtaining a preconditioner

Often the Poisson or a similar elliptic problem needs to be solved on a geometrically complicated domain Ω. One numerical approach towards solving such a problem splits Ω into a union of simpler subdomains which cover all of Ω. These constituent subdomains may overlap or share common boundaries. In any case, the presence of multiple subdomains often slows down the convergence of iterative solvers of the discretized equations (which enforce both the bulk PDE and the coupling between the subdomains). The practical use of multiple subdomains therefore often requires extra preconditioning, beyond any preconditioning of the bulk equations.

Often the need for multiple subdomains arises when the basic domain Ω is geometrically complicated. Here we consider the simplest possible decomposed geometry: a rectangle split into two subrectangles.

$$\Omega = \Omega_1 \cup \Omega_2 \text{ where } \Omega_1 = [0, 0.5] \times [0, 1],\ \Omega_2 = [0.5, 1] \times [0, 1] \tag{4.1}$$

Figure 4.1 shows this triangularization on the $[0, 1] \times [0, 1]$ grid. Notice the evenly spaced line of points in the center which become a boundary side for each subdomain.

n_{int} represents all of the interior points of the main domain (including those points on the center line) and n_{edge} is the number of points per edge on the boundary. Introduce n_{c} as the number of uniformly spaced interior points that run along the vertical center line of the domain; $n_{\text{c}} = n_{\text{edge}} - 1$. The total number of boundary points around the domain is $n_{\text{bnd}} = 4 \times n_{\text{edge}}$.

Due to the formation of the subdomains, our n_{int} here differs slightly from the n_{int} of Section 3.3. Here, $n_{\text{int}} \neq (n_{\text{edge}} - 1)^2$, rather $n_{\text{int}} = n_{\text{int1}} + n_{\text{int2}} + n_c$ where $n_{\text{int1}} = n_{\text{int2}} = (n_{\text{edge}} - 1)(n_{\text{edge}}/2 - 1)$ is the number of interior points in each subdomain (and therefore, we must always choose n_{edge} even).

Based on the setup presented earlier, the matrix L represents the linear system for the interior nodes of the domain. As shown in Section 3.2, the matrix L is also symmetric positive definite. Splitting the domain into the two subdomains, we see

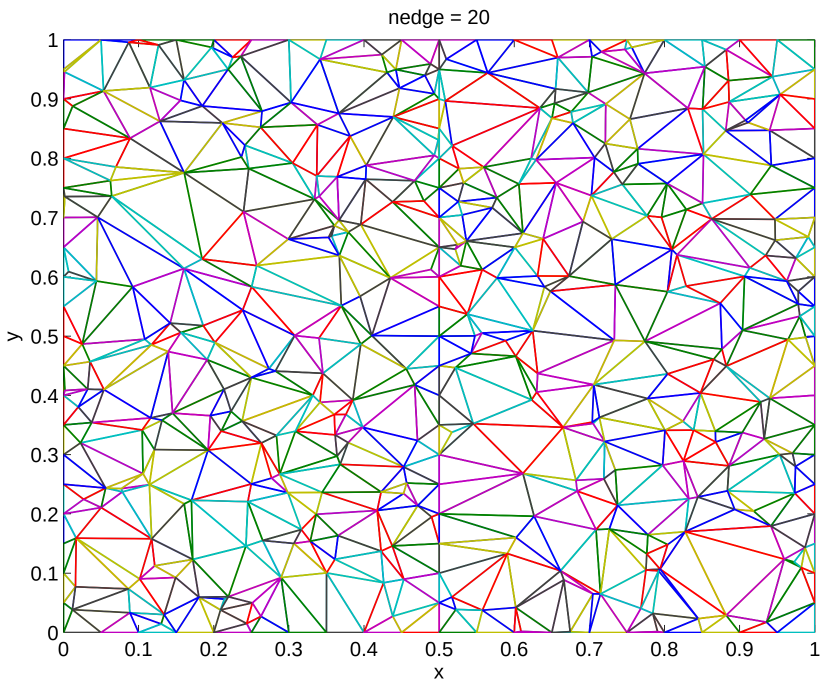

Figure 4.1: TRIANGLE MESH ON $[0,1] \times [0,1]$ GRID. $n_{\text{bnd}} = 80$ and $n_{\text{int}} = 380$. In splitting the domain up into 2 subdomains, the center line of points becomes a boundary for each subdomain.

that the matrix L takes the form

$$L = \begin{pmatrix} L_{11} & L_{1J} & \\ L_{J1} & L_{JJ} & L_{J2} \\ & L_{2J} & L_{22} \end{pmatrix} \tag{4.2}$$

where the inner $L_{\alpha\beta}$ are block matrices, $\alpha, \beta = 1, 2, J$. To represent a "joining" of the two subdomains, the variable J is used here. L_{11} and L_{22} contain the interior points of each subdomain; and thus, $L_{11}, L_{22} \in \mathbb{R}^{n_{\text{int1}} \times n_{\text{int1}}}$. The other block matrices $L_{1J}, L_{J1}, L_{2J}, L_{J2}$, and L_{JJ} are part of the coupling between the two subdomains. $L_{JJ} \in \mathbb{R}^{n_c \times n_c}$, $L_{1J}, L_{2J} \in \mathbb{R}^{n_{\text{int1}} \times n_c}$, $L_{J2}, L_{J1} \in \mathbb{R}^{n_c \times n_{\text{int1}}}$ (and since L is symmetric, $L_{1J} = L_{J1}^T$ and $L_{2J} = L_{J2}^T$).

For a coefficient matrix L with the structure above, a standard choice of preconditioner is the block-Jacobi preconditioner. When solving on each subdomain, we assume Dirichlet boundary conditions and ignore the coupling between the subdomains. We set $L_{1J}, L_{J1}, L_{2J}, L_{J2}$ all to zero and let L_{JJ} become the $n_c \times n_c$ identity matrix. Call G, the approximate inverse of L, the block-Jacobi preconditioner

$$G = \begin{pmatrix} L_{11}^{-1} & & \\ & I & \\ & & L_{22}^{-1} \end{pmatrix}. \tag{4.3}$$

We are given G, or rather it is easily constructed, and our goal is to improve upon it by incorporating information about the matching of the two subdomains.

For L^{-1} being the exact inverse of L and G our approximate inverse, the difference is the matrix A.

$$A = L^{-1} - G \tag{4.4}$$

Because the inverse of a symmetric matrix is also symmetric, L^{-1}, G, and consequently A are each symmetric. We propose to approximate the correction A using techniques from chapter 2, and then adding it on to G for an even more capable preconditioner G'.

4.3 Correction of the "rough preconditioner" G

4.3.1 Matlab code: PerformSolve

The main matlab code we use is displayed in Appendix B. This program defines the functions and sets the parameters we decide on. It then defines the exact solution, $u(x, y)$, which will be needed in the end to compare the accuracy. Two sets of random points plus hand selected uniformly spaced n_c points on the center line are then formed, assuming a uniform grid along the boundary, with each edge divided into $1/n_{\text{edge}}$ equally spaced subintervals. Again, using the setup from Section 3.3, the matlab function delaunay is used to create the triangles. The result is a $[0, 1] \times [0, 1]$ grid of random triangles.

Once the triangles are known, along with the positions of their vertices, the linear system can be built. The function GetLinearSystem loops over all the triangles to produce the sparse matrix, L, and the right hand side vector, b, which define the linear system $Lu = b$. This is also done for the subdomains, producing L_{11}, L_{22} and b_1, b_2 respectfully, as in Section 4.2.

The matrices L_{11} and L_{22} are important for the construction of the block-Jacobi preconditioner, G. This matrix G is defined by the action of L_{11}^{-1} and L_{22}^{-1}, which we can implement by solving $L_{11}z_1 = b_1$ and $L_{22}z_2 = b_2$ for z_1 and z_2 given the sources b_1 and b_2. These solves are performed using the conjugate gradient method, in which they are themselves preconditioned with the incomplete Cholesky factorization.

4.3.2 Building the correction of the "rough preconditioner" G

To produce the correction we are looking for, we must go through several steps. L is size $n_{\text{int}} \times n_{\text{int}}$. First, we introduce the random $4n_{\text{c}} \times n_{\text{int}}$ matrix K whose entries are each distributed as a Gaussian random variable of zero mean and unit variance. Next, define

$$N^T = LK^T \tag{4.5}$$

Here we are using N as if it is the original random matrix, but it actually comes from applying L to the random matrix K^T. We seek a smaller, approximate version

$A \equiv L^{-1} - G$, and compute the following.

$$
\begin{aligned}
A &= L^{-1} - G \\
AN^T &= L^{-1}N^T - GN^T \\
AN^T &= L^{-1}(LK^T) - GN^T \\
AN^T &= K^T - GN^T \\
\left(AN^T\right)^T &= \left(K^T - GN^T\right)^T \\
NA &= K - \left(GN^T\right)^T \\
Y &= K - \left(GN^T\right)^T
\end{aligned}
\tag{4.6}
$$

By construction, the rows of $Y \equiv NA$ are linear combinations of the rows of A, and importantly, Y is smaller than A. Therefore performing a matrix decomposition (here pivoted-QR) on Y will be cheaper than on A.

We know K by construction, but we need GN^T. Using the function ApplyBlock-JacobiPCiterative in Appendix D, we apply G to each of the columns of N^T. With these pieces, we can then form the matrix Y.

Now we use Stewart's algorithm from Appendix A on the matrix Y. Using Stewart's pivoted-QR algorithm allows for an early exit once a desired tolerance is met but this feature is not exploited here. Here, it runs to the end, yet it could possibly be faster if it were to exit earlier (the use of Stewart's algorithm allows for exploration of this possibility later). The algorithm is needed to find the positions of the columns we want to use in creating the desired interpolative decomposition:

$$\left[Q^{(Y)}, R^{(Y)}, R_{11}^{(Y)}, R_{12}^{(Y)}, k^{(Y)}, pivots^{(Y)}, P^{(Y)}\right] = \text{PivotedQR}\left(Y, tol\right) \tag{4.7}$$

Similar to Section 2.3.1, the outputs determine the equation

$$Y = Q^{(Y)}R_{11}^{(Y)} \cdot \left[I_{k^{(Y)} \times k^{(Y)}}, R_{11}^{(Y)} \backslash R_{12}^{(Y)}\right] \cdot \left(P^{(Y)}\right)^T \tag{4.8}$$

The output $pivots^{(Y)}$ is a vector which describes the reordering of matrix Y's columns. We will use the vector $pivots^{(Y)}$ to reorder the columns of the matrix A in an implicit manner.

Define a mapping

$$\Pi : \{1, 2, ..., k\} \longrightarrow \{1, 2, ..., k\} \tag{4.9}$$

$$j \to \Pi(j) \tag{4.10}$$

$$pivots^{(Y)} = \big[\Pi(1), \Pi(2), ..., \Pi(k)\big] \tag{4.11}$$

Now we have $e_{\Pi(j)}$ where e_j is the j^{th} canonical basis vector.

We can now apply G to $e_{\Pi(j)}$, once again using ApplyBlockJacobiPCiterative. Also, using L, $e_{\Pi(j)}$ and the preconditioned conjugate gradient function pcg, we can now construct $L^{-1}e_{\Pi(j)}$. From these formulations, we can now form $\tilde{B}$,

$$\tilde{B}(:, j) = Q^{(A)} R_{11}^{(Y)}(:, j) = L^{-1}e_{\Pi(j)} - Ge_{\Pi(j)} \tag{4.12}$$

Also, from equation (4.8), we can also form $\tilde{C}$,

$$\tilde{C} = \big[I_{k^{(Y)} \times k^{(Y)}}, R_{11}^{(Y)} \backslash R_{12}^{(Y)}\big] \cdot \big(P^{(Y)}\big)^T. \tag{4.13}$$

Now we have the pieces to formulate our correction

$$\tilde{A} = \tilde{B} \cdot \tilde{C}. \tag{4.14}$$

Addition of this correction $\tilde{A}$ to our approximate inverse G gives us our preconditioner G',

$$G' = G + \tilde{A}. \tag{4.15}$$

Using this final preconditioner G' involves first applying G and then applying $\tilde{A}$,

$$y = Gz \tag{4.16}$$

$$y = y + \tilde{A}z \tag{4.17}$$

The function pcg is now used to solve this new preconditioned system.

This procedure may be used as an alternate way of dealing with non-overlapping conforming subdomains when the purpose is for creating a preconditioner. Again, using multiple subdomains in this fashion is especially helpful when dealing with complicated geometries.

An added benefit of using multiple subdomains is that they have good parallelizing qualities. So, as long as the subdomains are of similar sizes, the block-Jacobi preconditioner should parallelize effectively. Also, once it is constructed, the application of the correction is also very parallelizable (for instance, using parallel-matrix multiplication).

4.4 Results

For comparison, we will solve the variable coefficient Poisson problem a variety of different ways. The first two ways we solve the system, there will be no dividing up of our domain. The subscript `noPC` is used when solving the system without using a preconditioner. We also solve the system with the incomplete Cholesky factorization as a preconditioner and represent this with the subscript `PC(InC)`. We found that a drop tolerance of 5×10^{-3} for the incomplete Cholesky was satisfactory in weighing the cost vs. the better approximation of the matrix.

The following factorizations will involve the use of domain decomposition. The subscript `PC(DD-Inv)` will be used for solving the system using domain decomposition along with directly applying the actual inverses of L_1 and L_2, `PC(DD-CG)` will be the subscripts for solving the system using domain decomposition along with the block-Jacobi preconditioner found by performing the conjugate gradient method on L_1 and L_2 computed via LU-factorization. The last subscript used, `PCA(DD-CG)`, is similar to the above subscript, but it also includes applying our constructed correction $\tilde{A}$.

We ran our algorithm many times, and the results in Table 4.1 are representative. Table 4.1's rows are as follows:

- n_{edge} is the number of edge points per side.

- $t_{\texttt{noPC}}$, $t_{\texttt{PC(InC)}}$, $t_{\texttt{PC(DD-Inv)}}$, $t_{\texttt{PC(DD-CG)}}$ and $t_{\texttt{PCA(DD-CG)}}$ are the computational times in seconds taken to solve the system.

- $i_{\texttt{noPC}}$, $i_{\texttt{PC(InC)}}$, $i_{\texttt{PC(DD-Inv)}}$, $i_{\texttt{PC(DD-CG)}}$ and $i_{\texttt{PCA(DD-CG)}}$ are the number of iterations required when solving the system.

- $res_{\texttt{noPC}}$, $res_{\texttt{PC(InC)}}$, $res_{\texttt{PC(DD-Inv)}}$, $res_{\texttt{PC(DD-CG)}}$ and $res_{\texttt{PCA(DD-CG)}}$ are the relative residuals when solving the system, $res = \|b - Ax\|/\|b\|$.

Table 4.1: Comparisons of the Poisson problem neglecting the costs of the construction of the preconditioners.

n_{edge}	10	20	30	40	50
$t_{\texttt{noPC}}$	0.0282	0.0525	0.0947	0.1554	0.2376
$t_{\texttt{PC(InC)}}$	0.0251	0.0278	0.0325	0.0362	0.0451
$t_{\texttt{PC(DD-Inv)}}$	0.0196	0.0930	0.1873	0.6993	1.3806
$t_{\texttt{PC(DD-CG)}}$	0.1767	1.2746	1.2170	3.0615	3.8832
$t_{\texttt{PCA(DD-CG)}}$	0.0140	0.0269	0.0435	0.0558	0.0760
$i_{\texttt{noPC}}$	67	189	340	468	624
$i_{\texttt{PC(InC)}}$	7	13	17	21	26
$i_{\texttt{PC(DD-Inv)}}$	36	80	120	230	221
$i_{\texttt{PC(DD-CG)}}$	40	107	141	284	272
$i_{\texttt{PCA(DD-CG)}}$	2	3	4	4	4
$res_{\texttt{noPC}}$	5.5574E-11	9.9022E-11	8.0423E-11	9.5044E-11	8.5179E-11
$res_{\texttt{PC(InC)}}$	3.2793E-11	8.4074E-12	2.2930E-11	7.6781E-11	4.5419E-11
$res_{\texttt{PC(DD-Inv)}}$	7.0380E-11	1.5619E-11	9.8126E-11	8.0093E-11	8.3283E-11
$res_{\texttt{PC(DD-CG)}}$	6.8307E-11	7.9035E-11	6.6010E-11	9.8751E-11	7.9270E-11
$res_{\texttt{PCA(DD-CG)}}$	1.5702E-11	2.0771E-12	5.2776E-13	4.2361E-14	1.5003E-13

The computational times when using the block-Jacobi preconditioners *without* the correction $\tilde{A}$, $t_{\texttt{PC(DD-Inv)}}$ and $t_{\texttt{PC(DD-CG)}}$, appear quite slow compared to the others. A main reason for this is that the cost of applying these block-Jacobi preconditioners is included in the computational times since each of the local solves for these are nested within each iteration. This is also true for $t_{\texttt{PCA(DD-CG)}}$, but with so fewer iterations, is not as apparent.

Solving the system with the incomplete Cholesky factorization as a preconditioner, `PC(InC)`, is a very efficient way to solve the system. Yet the results of

$t_{\texttt{PCA(DD-CG)}}$ are comparable, and in some cases show improvement in the computational time over the incomplete Cholesky (along with the other methods). Also, the relative residual for the `PCA(DD-CG)` method is almost always smaller than the other methods which indicates that its associated preconditioned linear system has better conditioning. However, the time in constructing this preconditioner for the `PCA(DD-CG)` is also something that should be considered. We return to this issue below.

Table 4.2: Comparisons of the Poisson problem on a uniform grid neglecting the costs of the construction of the preconditioners.

n_{edge}	10	20	30	40	50
$t_{\texttt{noPC}}$	0.0224	0.0288	0.0371	0.0495	0.0706
$t_{\texttt{PC(InC)}}$	0.0246	0.0256	0.0288	0.0328	0.0446
$t_{\texttt{PC(DD-Inv)}}$	0.0120	0.0231	0.0556	0.1368	0.3390
$t_{\texttt{PC(DD-CG)}}$	0.0846	0.1742	0.2676	0.4183	0.6283
$t_{\texttt{PCA(DD-CG)}}$	0.0128	0.0228	0.0380	0.0494	0.0656
$i_{\texttt{noPC}}$	32	64	94	121	146
$i_{\texttt{PC(InC)}}$	6	10	14	17	21
$i_{\texttt{PC(DD-Inv)}}$	18	29	37	43	49
$i_{\texttt{PC(DD-CG)}}$	20	31	37	44	49
$i_{\texttt{PCA(DD-CG)}}$	2	3	4	4	4
$res_{\texttt{noPC}}$	8.5341E-11	5.1537E-11	7.1472E-11	9.1416E-11	9.372E-11
$res_{\texttt{PC(InC)}}$	5.5046E-11	4.3825E-11	8.1638E-12	7.4034E-11	4.9169E-11
$res_{\texttt{PC(DD-Inv)}}$	1.0477E-11	3.0291E-11	6.3688E-11	9.6758E-11	6.5653E-11
$res_{\texttt{PC(DD-CG)}}$	1.1874E-11	1.8829E-11	7.0006E-11	6.8281E-11	6.6870E-11
$res_{\texttt{PCA(DD-CG)}}$	1.1139E-12	4.8106E-11	2.3436E-13	3.5745E-14	2.4468E-13

Table 4.2 shows results gotten by solving the problem just like before except this time using a uniform grid. With a uniform mesh, all the performances are slightly better. Solving with no preconditioner (`noPC`) shows substantial improvement using the uniform grid over the random grid. As before with the random grid, solving the system with the incomplete Cholesky factorization as a preconditioner, `PC(InC)`, is very efficient. And like before, the results of $t_{\texttt{PCA(DD-CG)}}$ are comparable here, and sometimes show improvement in the computational time over the incomplete Cholesky (along with the other methods). Here, the relative residual for the `PCA(DD-CG)` method is very good again, as it was when using the non-uniform mesh.

4.4.1 Vertically skewed domain

As an experiment, we changed up our simple $[0,1] \times [0,1]$ grid. We decided to move the whole center portion vertically up to add acute corners to the grid and see if the results would change (see Figure 4.2). But, the results we got here were consistent with the results we got with the $[0,1] \times [0,1]$ grid.

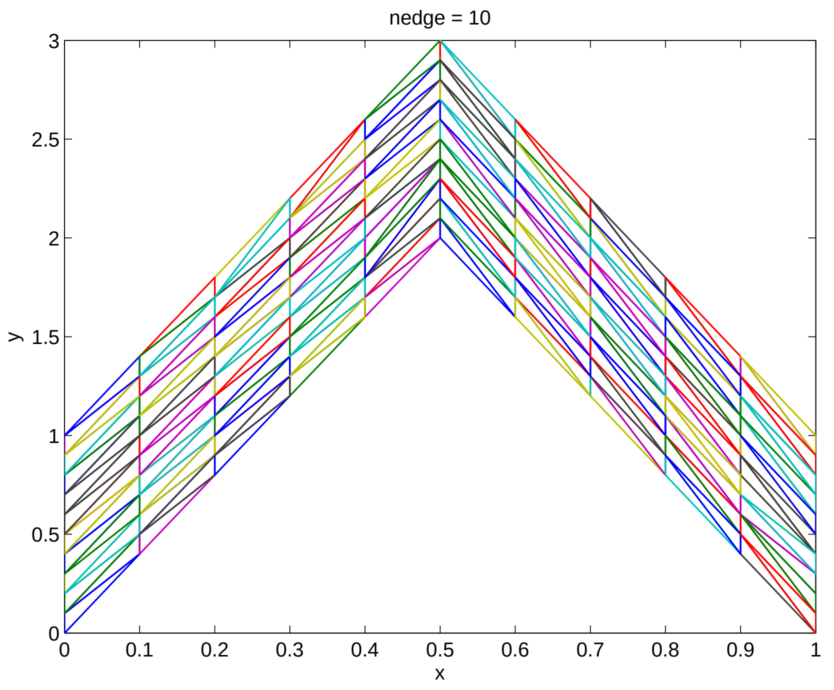

Figure 4.2: Triangle mesh performed on a vertically skewed grid. $n_{\text{bnd}} = 40$ and $n_{\text{int}} = 90$. This domain is split along the vertical center line resulting in two subdomains, similar as before.

4.5 Cost of constructing $\tilde{A}$

Although this preconditioner does indeed help with solving the system faster, we should also consider the cost of constructing the correction $\tilde{A}$. Again $t_{\texttt{PC(DD-CG)}}$ denotes the time taken to solve the two-domain problem using the block-Jacobi preconditioner *without* the correction $\tilde{A}$, and $t_{\texttt{PCA(DD-CG)}}$ is the time taken to solve it *with* the correction $\tilde{A}$. We will call T the time taken to construct $\tilde{A}$. Now $t_{\texttt{PCA(DD-CG)}} < t_{\texttt{PC(DD-CG)}}$

and presumably $t_{\texttt{PC(DD-CG)}} < T + t_{\texttt{PCA(DD-CG)}}$. There exists a break even point $k_{\texttt{LS}}$, LS represents linear solves, where approximately

$$k_{\texttt{LS}} \cdot t_{\texttt{PC(DD-CG)}} \simeq T + k_{\texttt{LS}} \cdot t_{\texttt{PCA(DD-CG)}} \tag{4.18}$$

or

$$k_{\texttt{LS}} \simeq T/(t_{\texttt{PC(DD-CG)}} - t_{\texttt{PCA(DD-CG)}}). \tag{4.19}$$

Clearly it is advantageous if one could construct $\tilde{A}$, and then use it multiple times in the same problem. In these cases, it would be cost effective if $\tilde{A}$ is used more than $k_{\texttt{LS}}$ times. An example might be a problem involving a time-stepping method, like say the backward Euler method. Solving the heat equation using the backward Euler method consists of doing a number of these solves over and over again for many different right hand sides. In the future we hope to explore the use of our preconditioning methods in this context.

We ran our algorithm multiple times with different amounts of points and Table 4.3 shows the averaged results.

Table 4.3: Break even point $k_{\texttt{LS}}$, with the size of the system, n_{int}.

n_{edge}	n_{int}	$k_{\texttt{LS}}$
10	109	10.7
20	419	24.7
30	929	31.2
40	1639	38.4
50	2549	46.2

As an example, we ran the algorithm for $n_{\text{edge}} = 20$. We got back $t_{\texttt{PC(DD-CG)}} = .868$, $t_{\texttt{PCA(DD-CG)}} = .0342$, and $T = 20.614$. So,

$$k_{\texttt{LS}} \simeq \frac{T}{t_{\texttt{PC(DD-CG)}} - t_{\texttt{PCA(DD-CG)}}} \simeq \frac{20.614}{.868 - .0342} \simeq 24.7 \tag{4.20}$$

4.6 The Schur complement method

Another approach for using non-overlapping domain decomposition worth mentioning is the Schur complement method. As described in [8], this method involves breaking down our matrix L into the block LDU decomposition (for LDU; L represents a lower triangular matrix with ones along its diagonal, D represents a diagonal matrix, and U represents an upper triangular matrix with ones along its diagonal).

Again, we have our symmetric positive definite matrix

$$L = \begin{pmatrix} L_{11} & L_{1J} & 0 \\ L_{J1} & L_{JJ} & L_{J2} \\ 0 & L_{2J} & L_{22} \end{pmatrix}$$

The block LDU decomposition becomes

$$L = \begin{pmatrix} I & 0 & 0 \\ L_{J1}L_{11}^{-1} & I & 0 \\ 0 & L_{J2}L_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & S & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} L_{11} & L_{1J} & 0 \\ 0 & I & L_{J2} \\ 0 & 0 & L_{22} \end{pmatrix} \tag{4.21}$$

Where

$$S = L_{JJ} - L_{J1}L_{11}^{-1}L_{1J} - L_{J2}L_{22}^{-1}L_{2J} \tag{4.22}$$

is called the *Schur complement* of the leading principal submatrix containing L_{11} and L_{22}. Calculating L^{-1}, we get

(4.23)
$$L^{-1} = \begin{pmatrix} L_{11}^{-1} & -L_{11}^{-1}L_{1J} & 0 \\ 0 & I & -L_{22}^{-1}L_{2J} \\ 0 & 0 & L_{22}^{-1} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & S^{-1} & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ -L_{J1}L_{11}^{-1} & I & 0 \\ 0 & -L_{J2}L_{22}^{-1} & I \end{pmatrix}$$

Here, multiplying a vector by L^{-1} entails multiplying by the blocks in the entries of this factored form of L^{-1}; this includes L_{1J} and L_{2J} (and their transposes $L_{J1} = L_{1J}^T$ and $L_{J2} = L_{2J}^T$), L_{11}^{-1} and L_{22}^{-1}, and S^{-1}. Multiplying by L_{1J} and L_{2J} is cheap because they are very sparse. Multiplying by L_{11}^{-1} and L_{22}^{-1} should not be expensive if we choose a suitable fast method (i.e. fast Fourier transform, multigrid method,...).

We now explain multiplying by S^{-1}; as indicated by [8]. Because there are much fewer grid points on the boundary than in the subdomains, L_{JJ} and S have a much smaller dimension than L_{11} and L_{22} (this situation grows for finer grid spacings). S is symmetric positive definite and dense. To get S^{-1} explicitly, one would need to solve with each subdomain once per boundary grid point (from the $L_{11}^{-1}L_{1J}$ and $L_{22}^{-1}L_{2J}$ terms in equation (4.22)). This could be done, then afterward factor S using dense Cholesky and then continue to solve the system. But this would be expensive, much more so than just multiplying a vector by S. A better way would be by using the conjugate gradient method, which requires only multiplying a vector by S (requiring only one solve per subdomain using equation (4.22)).

With the conjugate gradient method, the number of matrix-vector multiplications depends on the condition number of S. The key to this method is that S is much better conditioned than the original matrix L. Therefore, using the conjugate gradient method would be ideal here and would result in fast convergence involving less iterations.

It should also be noted that a common preconditioner for use when two subdomains are involved with an overlap of their boundaries is the additive Schwartz preconditioner. For this procedure, you would solve on one side, then use the solution to set the boundary conditions for the other side iterating back and forth. Further research comparing and contrasting the different overlapping and non-overlapping methods may be worth while in the future.

Chapter 5

Conclusion

The Poisson equation is instrumental in a variety of different fields such as electrostatics, electromagnetics, acoustics, mechanical engineering, and theoretical physics. Any progress in better solving this important problem should only benefit the future of these fields among others.

This study was set out to explore a different procedure for creating an effective preconditioner to help solve the variable coefficient Poisson problem using the finite element method. Techniques were implored using random matrices while constructing the interpolative decomposition of matrices as well as the use of domain decomposition to establish this preconditioner. We looked closely at two questions regarding the Poisson problem:

- Can sampling a matrix with the interpolative decomposition (i.e., picking the most important columns of a matrix and having them represent the entire matrix) and also combining the use of random matrices, be an effective technique for creating a preconditioner?
- For a non-overlapping domain decomposition preconditioner method, can we find a good way to account for the conforming boundary area?

Using this interpolative decomposition to create the preconditioner with the correction does indeed reduce the number of iterations involved. It does also speed up the time involved. But it may be possible to figure out a way to apply it faster for even less time taken. This would of course be beneficial especially since the creation of the preconditioner takes a good amount of time by itself.

This preconditioner may also be beneficial for problems involving time-stepping, e.g., the heat equation. When solving problems of this sort, explicit methods encounter a stability limit, so implicit methods are favored. Although these methods take larger steps in their process, they must solve an equation at each step. This could be ideal for this kind of preconditioner since once it's created, it can be used over and over again, making up for the cost of its creation. One other possible application could be for cases involving more complicated geometrical domains. This procedure may be effective when others are not. Again, these concepts should be explored more in the future.

Interpolative decomposition using statistical sampling on a matrix seems to be a worthy endeavor. As far as building a preconditioner, employing this method can be useful, but to what extent though is the question. Going through all the trouble of it seems to be worth it in this situation and as stated, one may find more usable applications in the future.

Appendices

Appendix A

Stewart's Pivoted-QR Algorithm

```
%  Computes pivoted QR-decomposition A*P = Q*R of an m-by-n matrix
%  via method described in G. W. Stewart, "Two Algorithms for the
%  Efficient Computation of Truncated Pivoted QR Approximations to
%  a Sparse Matrix." Well suited for low-rank approximation of a
%  sparse matrix A. Inputs are A (possibly sparse), and desired
%  tolerance tol for the approximation. Outputs are as follows.
%  Matrices Q [m-by-k], R11 [k-by-k], R12 [k-by-(n-k)], P [n-by-n]
%  such that A*P \simeq Q*[R11 R12]. P is a permutation matrix, Q
%  is an orthonormal matrix, and R = [R11 R12] is upper triangular.
%  ncol is the (numerical) rank of matrix/approximation.
%  function [Q, R11, R12, ncol, pivots, P] = PivotedQR(A,tol,ncol)
   function [Q, R, R11, R12, ncol, pivots, P] = PivotedQR(A,tol,ncol)
[m, n] = size(A);
pivots = 1:n;
%ncol = min([m, n]);
for j=1:n
   nu(j) = norm(A(:,j))^2; % nu for squared 2-norms as in Stewart.
end
R = zeros(ncol,n);

for k=1:ncol  % Loop over columns of A.
   %-----------------------------------------------
   %  Determine the pivot column and swap it with
   %  column k. Since max taken over length-(n-k+1)
   %  vector, readjust index relative to length n.
   %-----------------------------------------------
   [maxnorm, jmax] = max(nu(k:n));
   jmax = jmax + (k-1);
```

```
%-----------------------------------------------
% Perform pivot.
%-----------------------------------------------
pivots([k jmax])  = pivots([jmax k]);
nu([k jmax]) = nu([jmax k]);
if k > 1;
   R(1:k-1,jmax) = R(1:k-1,k);
end

% Gram-Schmidt step with reorthogonalization.
a = A(:,pivots(k));
if k == 1
   R(1,1) = norm(a);
   q = a/R(1,1);
   Q = q;
else
   %-----------------------------------------------
   % With A_k-1 = A(:,pivots(1:k-1)) = Q_k-1 R_k-1,
   % Q_k-1 = A_k-1 inv(R_k-1) is an o.n. basis for
   % these columns of A_k-1. With a = A(:,pivots(k))
   % we have r_kj = a'*Q_k-1(:,j) as a row vector
   % r_k = [r_k1 r_k2 ... r_k,k-1]. Transpose it.
   %-----------------------------------------------
   r = a'*A(:,pivots(1:k-1))/R(1:k-1,1:k-1); r = r';

   %-----------------------------------------------
   % Subtract out proj. onto column space of Q_k-1.
   % A_k-1 (R_k-1\r) = A_k-1 inv(R_k-1) r = Q_k-1 r.
   %-----------------------------------------------
   q = a - A(:,pivots(1:k-1))*(R(1:k-1,1:k-1)\r);

   %-----------------------------------------------
   % Reorthogonalization for good measure
   %-----------------------------------------------
   Dr = q'*A(:,pivots(1:k-1))/R(1:k-1,1:k-1); Dr = Dr';
   q  = q - A(:,pivots(1:k-1))*(R(1:k-1,1:k-1)\Dr);

   %-----------------------------------------------
   % Update R.
   %-----------------------------------------------
   r = r + Dr; R(1:k-1,k) = r; R(k,k) = norm(q);

   %-----------------------------------------------
   % Compute the kth column of Q.
   %-----------------------------------------------
   q = (a-A(:,pivots(1:k-1))*(R(1:k-1,1:k-1)\r))/R(k,k);
   Q = [Q q];
   end

   if k+1<=n
         %  Compute the k-th row of R, Eq.(4) from Stewart.
         R(k,k+1:n) = q'*A(:,pivots(k+1:n));

         %  Update nu.
         nu(k+1:n) = max([nu(k+1:n) - R(k,k+1:n).^2; zeros(1,n-k)]);
         nu(k)     = sum(nu(k+1:n));
         if (sqrt(nu(k)) < tol) break; end
   else
      nu(k) = 0;
   end
end %  End loop over columns of A.

%  Finish up.
ncol = k;
R    =  R(1:ncol,:);
R11  =  R(:,1:k);
R12  =  R(:,k+1:n);
if (nargout == 7)
   P = zeros(n,n);
   for i = 1:n
       for j = 1:n
           if pivots(i)==j
              P(j,i) = 1;
           end
       end
   end
end
```

Appendix B

Perform Solve Algorithm

```
%-------------------------------------------------------
% Solution and compatible source for -div(psi grad u)=f.
% function PerformSolveDivGrad2_Alternate(nedge)
function PerformSolveDivGrad2_Alternate(nedge)

SubPCparams;
%-------------------------------------------------------
N = 1.6; M = 2;
u         =@(x,y)sinh(N*x*(1-x))*sin(M*pi*y);
%
ux        =@(x,y)N*(1-2*x)*cosh(N*x*(1-x))*sin(M*pi*y);
uxx       =@(x,y)(-2*N*cosh(N*x*(1-x)) ...
           +N*N*(1-2*x)^2*sinh(N*x*(1-x)))*sin(M*pi*y);
%
uy        =@(x,y)M*pi*sinh(N*x*(1-x))*cos(M*pi*y);
uyy       =@(x,y)( ...
           -M*M*pi*pi*sinh(N*x*(1-x))*sin(M*pi*y));
%
uxxPLUSuyy=@(x,y)(uxx(x,y)+uyy(x,y));
nu = 1e-8;
psi       =@(x,y)exp(-nu*((x-0.5)^2+(y-0.5)^2));
psix      =@(x,y)(-2*nu*(x-0.5)*psi(x,y));
psiy      =@(x,y)(-2*nu*(y-0.5)*psi(x,y));
%
f         =@(x,y)(-psix(x,y)*ux(x,y) ...
                  -psiy(x,y)*uy(x,y) ...
                  -psi(x,y)*uxxPLUSuyy(x,y));
f_Lap     =@(x,y)(-uxxPLUSuyy(x,y));
%-------------------------------------------------------
```

```
% Build triangularization for each block and then put together.
%
a = 0; b = 0.5; c = 0.0; d=1.0;
%[TRI1,x1,y1,nbdry] = BuildSquareGrid(nedge,a,b,c,d,'R');
[TRI1,x1,y1,nbdry] = BuildRomGrid(nedge,a,b,c,d,'R');
%
a = 0.5; b = 1.0; c = 0.0; d=1.0;
%[TRI2,x2,y2,nbdry] = BuildSquareGrid(nedge,a,b,c,d,'L');
[TRI2,x2,y2,nbdry] = BuildRomGrid(nedge,a,b,c,d,'L');
%
x = [x1;x2]; y = [y1;y2]; TRI = [TRI1; TRI2 + length(x1)];
%-----------------------------------------------------------------

%-----------------------------------------------------------------
extrimesh(TRI,x,y)
xlabel('x'); ylabel('y'); title('nedge = 20')
saveas(gcf,'grid30.pdf','pdf')
saveas(gcf,'grid30.eps','epsc')
% Fill up exact solution vector. Does not include boundary points.
npts = length(x1); nint = 2*npts-2*nbdry+2*(nedge-1); z = zeros(nint,1);

for k = 1:nint
    if k+nbdry-(nedge-1) <= npts
       kint = k + nbdry-(nedge-1);
    else
       kint = k + 2*nbdry-2*(nedge-1);
    end
    z(k) = u(x(kint),y(kint));
end

% Get the sparse matrix and righthand side which define linear system.
[L1, b1] = GetLinearSystemDivGrad(f,psi,TRI1,x1,y1,nbdry); W1=cholinc(L1,5e-3);
[L2, b2] = GetLinearSystemDivGrad(f,psi,TRI2,x2,y2,nbdry); W2=cholinc(L2,5e-3);
[L, b]   = GetLinearSystemDivGrad2(f,psi,TRI,x,y,nbdry,nedge); W=cholinc(L,5e-3);
%%%%%% Commented out various full matrix operations used for testing.
%%%%%%
       invL1 = inv(full(L1));  invL2 = inv(full(L2));
%%%%%% [L_L, L_U] = GE(L);
%%%%%%
%%%%%% Eye = eye(nedge-1);
%%%%%% G = blkdiag(Eye,invL1,Eye,invL2);
%%%%%% invL = inv(L);
%%%%%% A = invL-G;
%%%%%% N = randn(76,enn);
%
%
```

```
%
[enn, emm] = size(L);
ell = nedge*4 - 4;
%
%  Preallocation of memory.
%
Bapprox = zeros(enn,ell); Papprox = zeros(ell,enn);
GNt = zeros(enn,ell); Ge = zeros(enn,1);
e = zeros(enn,1);
invLe = zeros(enn,1);
%
%
%
tic
R = randn(ell,enn); Rt = transpose(R); Nt = L*Rt;
for j = 1:ell
    GNt(:,j) = ApplyBlockJacobiPC_iterative(Nt(:,j),L1,W1,L2,W2,Bapprox,Papprox,nbdry,nedge,nint);
end
Y = R - transpose(GNt);
tol = 1e-10;
[QY, R, R11Y, R12Y, kY, pivotsY, PY] = PivotedQR(Y,tol);
%Y_  = QY*R11Y;
ISY = [eye(kY) R11Y\R12Y];
%
%
%
pcgTOL=1e-10; maxiter = 2000;
subTOL=1e-6; subITER = 2000;
correct='n';
for q = 1:kY
    e = zeros(enn,1);
    e(pivotsY(q)) = 1;
    Ge = ApplyBlockJacobiPC_iterative(e,L1,W1,L2,W2,Bapprox,Papprox,nbdry,nedge,nint);
    [invLe,flag,relres,niter,resvec]=pcg(L,e,pcgTOL,maxiter,@(w)ApplyBlockJacobiPC_iterative
(w,L1,W1,L2,W2,Bapprox,Papprox,nbdry,nedge,nint),[],[]);
    Bapprox(:,q) = invLe - Ge;
end
Papprox = ISY*transpose(PY);
Aapprox = Bapprox*Papprox;
Tconstruct = toc;
%
%-----------------------------------------------------------------
%

% No Preconditioner used
tic
   [znum3,flag,relres,niter]=pcg(L,b,pcgTOL,maxiter,[],[],[]);
```

```
time = toc;

% Incomplete LU factorization (Cholesky) on entire "L"
tic
   [znumL,flag,relresL,niterL,resvec]=pcg(L,b,pcgTOL,maxiter,W',W,[]);
tocL = toc;

% Using Domain Decomposition along with the direct inverse of "L1" and direct inverse of h"L2"
tic
   correct='n';
   [znum2b,flag,relres0b,niter0b,resvec]=pcg(L,b,pcgTOL,maxiter,@(w)ApplyBlockJacobiPC_direct
(w,invL1,invL2,Bapprox,Papprox,nbdry,nedge,nint),[],[]);
t0b = toc;

% Using Domain Decomposition and Conjugate Gradient on "L1","L2"
tic
   correct='n';
   [znum2,flag,relres0,niter0,resvec]=pcg(L,b,pcgTOL,maxiter,@(w)ApplyBlockJacobiPC_iterative
(w,L1,W1,L2,W2,Bapprox,Papprox,nbdry,nedge,nint),[],[]);
t0 = toc;

% Using Domain Decomposition and Conjugate Gradient on "L1","L2" plus the Correction "A"
tic
   correct='y';
   [znum1,flag,relres1,niter1,resvec]=pcg(L,b,pcgTOL,maxiter,@(w)ApplyBlockJacobiPC_iterative
(w,L1,W1,L2,W2,Bapprox,Papprox,nbdry,nedge,nint),[],[]);
t1 = toc;

%--------------------------------------------------------------------

k = Tconstruct/(t0 - t1);
q = npts-nbdry+(nedge-1);
zgraph = [
          zeros(nbdry-(nedge-1),1);
          znum1(1:q)-z(1:q);
          zeros(nbdry-(nedge-1),1);
          znum1(q+1:nint)-z(q+1:nint)];
%extrimesh(TRI,x,y,abs(zgraph))
%xlabel('x'); ylabel('y'); title('Pointwise error')

disp(['no PC      computation time: ',num2str(time)])
disp(['PC(InC)    computation time: ',num2str(tocL)])
disp(['PC(DD-Inv)  computation time: ',num2str(t0b)])
disp(['PC(DD-CG)  computation time: ',num2str(t0)])
disp(['PCA(DD-CG) computation time: ',num2str(t1)])
disp(['------------------------------------'])
disp(['no PC      iteration number: ',num2str(niter)])
disp(['PC(InC)    iteration number: ',num2str(niterL)])
disp(['PC(DD-Inv)  iteration number: ',num2str(niter0b)])
disp(['PC(DD-CG)  iteration number: ',num2str(niter0)])
disp(['PCA(DD-CG) iteration number: ',num2str(niter1)])
disp(['------------------------------------'])
disp(['no PC      relative residual: ',num2str(relres)])
disp(['PC(InC)    relative residual: ',num2str(relresL)])
disp(['PC(DD-Inv)  relative residual: ',num2str(relres0b)])
disp(['PC(DD-CG)  relative residual: ',num2str(relres0)])
disp(['PCA(DD-CG) relative residual: ',num2str(relres1)])

clear all
```

Appendix C

Get Linear System

```
% Computes approximation Az = b of the Dirichlet problem
%
%  -Lap(u)=f , (x,y) in (0,1)X(0,1), u=0 on boundary
%
% using piecewise linear finite elements on the triangulation
% describe by (T,x,y). nbdry is the number of boundary nodes.
% They are assumed to come first.
% function [A, b]=GetLinearSystemDivGrad2(f,psi,T,x,y,nbdry,nedge)
  function [A, b]=GetLinearSystemDivGrad2(f,psi,T,x,y,nbdry,nedge)

npts = length(x)/2;
nint = 2*npts-2*nbdry+2*(nedge-1);
intRange=[nbdry-(nedge-1)+1,npts,npts+nbdry-(nedge-1)+1,2*npts];

[ntri d]=size(T);

A=spalloc(nint,nint,7*nint);  % Guessing on average < 7 nonzeros per row
b=zeros(nint,1);

% Now loop over the triangles and build the linear system
for k=1:ntri
  j1=T(k,1); j2=T(k,2); j3=T(k,3);
  [A_loc b_loc]=GetLinearSystemDivGrad_loc(x(j1),y(j1),x(j2),y(j2),x(j3),y(j3),f,psi);
  [j1test j1ref] = TestInRange(j1,intRange,nbdry,nedge);
  if (j1test == 1)
    k1=j1-j1ref;
    A(k1,k1)=A(k1,k1)+A_loc(1);
    b(k1)=b(k1)+b_loc(1);
    [j2test j2ref] = TestInRange(j2,intRange,nbdry,nedge);
    if (j2test == 1)
      k2=j2-j2ref;
      A(k1,k2)=A(k1,k2)+A_loc(2);
      A(k2,k1)=A(k1,k2);
    end
    [j3test j3ref] = TestInRange(j3,intRange,nbdry,nedge);
    if (j3test == 1)
      k3=j3-j3ref;
      A(k1,k3)=A(k1,k3)+A_loc(3);
      A(k3,k1)=A(k1,k3);
    end
  end
  [j2test j2ref] = TestInRange(j2,intRange,nbdry,nedge);
  if (j2test == 1)
    k2=j2-j2ref;
    A(k2,k2)=A(k2,k2)+A_loc(4);
    b(k2)=b(k2)+b_loc(2);
    [j3test j3ref] = TestInRange(j3,intRange,nbdry,nedge);
    if (j3test == 1)
      k3=j3-j3ref;
      A(k2,k3)=A(k2,k3)+A_loc(5);
      A(k3,k2)=A(k2,k3);
    end
  end
  [j3test j3ref] = TestInRange(j3,intRange,nbdry,nedge);
  if (j3test == 1)
    k3=j3-j3ref;
    A(k3,k3)=A(k3,k3)+A_loc(6);
    b(k3)=b(k3)+b_loc(3);
  end
end
```

Appendix D

Apply Block Jacobi Preconditioner

```
function z = ApplyBlockJacobiPC_iterative(x,L1,W1,L2,W2,B,P,nbdry,nedge,nint)
SubPCparams;
nint1 = nint/2;
z = x;
z(1:nedge-1) = x(1:nedge-1);
%
%
%
[z(nedge:nint1),flag,relres,niter]=pcg(L1,x(nedge:nint1),subTOL,subITER,W1',W1,[]);
%
%
%
z(nint1+1:nint1+nedge-1) = x(nint1+1:nint1+nedge-1);
%
%
%
[z(nint1+nedge:2*nint1),flag,relres,niter]=pcg(L2,x(nint1+nedge:2*nint1),subTOL,...
subITER,W2',W2,[]);
%
%
%
if correct == 'y'
   qqq = P*x;
   z = z + B*qqq;
end
```

References

[1] Cheng, H., Gimbutas, Z., Martinsson, P.G., and Rokhlin, V., *On the Compression of Low Rank Matrices.* SIAM J. Sci. Comput., vol.26, no.4, 2005, pp. 1389-1404.

[2] Johnson, Claes, *Numerical Solution of Partial Differential Equations by the Finite Element Method.* Dover Publications Inc. Mineola, New York 2009.

[3] Liberty, Edo, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rohklin, and Mark Tygert, *Randomized algorithms for the low-rank approximations of matrices.* Department of Computer Science and Program in Applied Math, Yale University Department of Applied Math, University of Colorado 2007.

[4] Stewart, G.W., *Two Algorithms for The Efficient Computation of Truncated Pivoted QR Approximations to a Sparse Matrix.* Department of Computer Science, University of Maryland <http://www.cs.umd.edu/stewart/>.

[5] Trott, David W., and Matthias K. Gobbert *Finite Element Convergence for Time-Dependent PDEs with a Point Source in COMSOL 4.2.* Department of Mathematics and Statistics, University of Maryland, Baltimore County, Baltimore, Maryland

[6] Trefethen, Lloyd N. and David Bau, III *Numerical Linear Algebra* SIAM (Society for Industrial and Applied Mathematics) Philadelphia, PA 1997

[7] Golub, Gene H. and Charles F. Van Loan *Matrix Computations* 3rd ed. The Johns Hopkins University Press Baltimore and London 1996.

[8] Demmel, James W. *Applied Numerical Linear Algebra* SIAM (Society for Industrial and Applied Mathematics) Philadelphia, PA 1997.