Implementing Conjugate Gradients with Incomplete Cholesky Preconditioning in Playa

by

Kimberly R. Kennedy, B.S.

A Thesis
in
Mathematics and Statistics
Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for the Degree of
Master of Science

Approved

Kevin Long
Committee chair

Victoria E. Howle

Lourdes Juan

Dominick J. Casadonte, Jr.
Dean of the Graduate School

December, 2012

Acknowledgements

My accomplishments throughout my education and my research into the implementation of Conjugate Gradients with Incomplete Cholesky Preconditioning would not have been possible without the support of many people. I would like to express my deepest gratitude to my advisor, Associate Professor Kevin Long who was abundantly helpful and offered invaluable assistance, support and guidance. I would like to thank the members of my thesis committee, Assistant Professor Dr. Victoria E. Howle and Associate Professor Dr. Lourdes Juan without whose knowledge and assistance this thesis would not have been successful. Special thanks to Associate Professor Dr. Ram Iyer and the Department of Mathematics and Statistics for finding research opportunities, financial assistance, and well-balanced academic family. To the Texas Tech University Provost Office, I cannot thank you enough for the Provost Fellowship that I was awarded my first year as a graduate student. Lastly, I would like to express my love and gratitude to my beloved family especially my husband Justin Yost; for their understanding and endless love, through the duration of my studies.

Table of Contents

Abstract

In computational sciences, solving an immensely large sparse system of linear equations is a commonly recurring problem. One of the most effective iterative solvers used for these systems is the method of Conjugate Gradients (CG). CG solves the system $Ax = b$ where A is a NxN symmetric positive definite (SPD), b is a known vector, and x is the unknown vector to be determined. CG can be adapted to solve non-symmetric nonsingular matrices but it involves transforming the system $Ax = b$ into $A^T Ax = A^T b$.

Though CG must converge within N iterations in exact arithmetic, its performance can be vastly improved when a preconditioner is applied. There are several preconditioners that one can use for CG. However, Incomplete Cholesky factorization (ICC) was the preconditioner used in this research thus it will be the the primary focus.

The implementation of both CG and Incomplete Cholesky Conjugate Gradients (ICCG) can vary in complexity depending on the software used to formulate the code. In Playa, the solver and preconditioners are abstracted so that the end user is able to use a general method or specific method for solving their problem. However, the implementation of ICCG relied on another subpackage Ifpack for the creation of the preconditioners which led to several test matrices having poor preconditioners.

List of Tables

List of Figures

List of Nomenclature

CG The method of Conjugate Gradients

CGN The method of Conjugate Gradients for Normal Equations.

HCL Hilbert Class Libraries

ICC Incomplete Cholesky factorization

ICCG Incomplete Cholesky Conjugate Gradients

LOF Level of Fill

PCG Preconditioned Conjugate Gradients

SPD Symmetric Positive Definite

Chapter 1

Introduction

1.1 Algorithms

The method of Conjugate Gradients takes an initial vector and uses orthogonal directions to minimize the quadratic form $f(x) = \frac{1}{2}x^T Ax - x^T b$. The matrix A in the quadratic form is a symmetric positive definite matrix and the vector x is a solution to $Ax = b$. A symmetric positive definite matrix is a matrix that for all nonzero x, $x^T Ax > 0$, and $A^T = A$. The convergence of CG is dependent on the condition number κ and can be expressed by the inequality $\|e_n\|_A \leq 2(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^n \|e_0\|_A$ where e_0 and e_n are the initial and nth computed errors respectively [13, 14]. Each i-th error is defined as $e_i = x_i - x^*$ or the approximate solution in the i-th iteration minus the exact solution. The condition number for a SPD matrix A is defined as $\kappa(A) = |\frac{\lambda_{max}}{\lambda_{min}}|$. The use of a preconditioner, Incomplete Cholesky factorization preconditioner, can improve the convergence rate since the preconditioner lowers the condition number and clusters the eigenvalues close to 1 [8, 11, 14] .

1.2 Software

The implementation of the algorithms mentioned in this paper were done through Playa in the Trilinos software library developed by Sandia National Laboratories. Trilinos is a software framework that is designed around a collection of packages [6]. Most packages within Trilinos are fully capable of being independent from the library while interacting among the other packages.

Playa is a Trilinos package that uses the inoperability of Trilinos in order to construct more robust algorithms for many physics and engineering type problems. One of the many features of Playa is that the data structures and algorithms of Playa are constructed so that their storage details are hidden from the user. This package uses abstract classes to provide flexibility between generic problem definitions and specific user-defined problems. In order to provide high-performace overloaded operators,

Playa relies on an expression template mechanism [7]. Overloading operators allows for more flexibility for operations already defined by the programming language, C++, by letting the input determine the functionality of the operator. For algebraic preconditioners, Playa uses Ifpack and ML defined operators.

The Ifpack package contains a collection of object-oriented incomplete factorization preconditioners that can be used to precondition iterative solvers created in other packages of Trilinos [3]. Ifpack provides the end-user control over the creation of the preconditioner by allowing for several preconditioner parameters to be modified. For the implementation of ICCG, Ifpack's `ICT` (Incomplete Cholesky Factorization with drop tolerance) class will be utilized in order to create the ICC preconditioner.

For constructing CG and ICCG, Playa allows more control and flexibility for the user and the algorithms are able to scale for larger problems due to the memory management and parallelism done through Epetra and other packages. MATLAB already has a function created for CG and Preconditioned Conjugate Gradients (PCG) called `pcg` that has a limited set of attributes that the user can specify. Thus for sheer flexibility, end-user control, and scalability the implementation was done in Playa.

1.3 Similar Approaches in software [9, 5, 7, 4, 15]

There are several software packages that structure their operators and methods similarly to Playa. LLANO and Blitz++ use templating to determine structure and operator overloading to limit temporaries during computation. However, LLANO and Blitz++ are implemented for serial programming only. Hilbert Class Libraries (HCL) and Thyra [2], another Trilinos package, use polymorphism for several key data structures: vectors, operators, and functions. Polymorphism uses abstract base classes to define a structure for the derived classes. The derived classes are able to redefine functions given in the abstract class or not depending on the virtuality of the function in the base class. Since HCL breaks the code apart into base classes and derived classes, the end user can work with generic and application-specific code. This abstraction allows for more flexibility and a higher level style of code that is understandable [4]. Unfortunately, HCL and Thyra do not have operator overloading. Thus Playa is a far better choice than the above mentioned software packages since it utilizes operator overloading and polymorphism.

1.4 Software Issues

During the implementation of ICCG, there were a series a frustrations encountered with Ifpack. By far the largest issue was that the preconditioner seems to have a bug in computation. In testing, the preconditioner of several matrices contained elements with values of infinity or the absolute of the values were extremely large. This result should not have occured in Ifpack. Ifpack lacks readability and the steps to define a preconditioner are unclear. These issues cause the end user many headaches since they have to search through code in order to figure out how to set up the preconditioner and the preconditioner parameters.

Chapter 2

Conjugate Gradients

The Method of Conjugate Gradients is one of many common iterative methods used for solving large systems of linear equations in which traditional factorization and triangular solve is not efficient. The most common of these systems would be the standard $Ax = b$ problem where A is a sparse symmetric positive definite matrix, b is a known vector, and x is an unknown vector. A sparse matrix is a matrix that consists primarily of zeroes.

2.1 Method of Conjugate Gradients

In order to understand CG, one must first understand the method of steepest descent and the method of conjugate directions. The method of steepest descent finds a minimum for a function f by taking steps along the residual from an inital guess that has the largest decrease in direction, $-\nabla f(x_n)$. This method repeats this step until a minimum is found. Unfortunately steepest descent can repeat descent directions thus converging slowly. The convergence of the method of steepest descent is defined as $\|e_n\|_A \leq (\frac{\kappa-1}{\kappa+1})^n \|e_0\|_A$ [13]. Therefore, the method of conjugate directions takes this method and tweaks it by taking a sequence of orthogonal search directions in order to reach the minimum.

CG takes the method of conjugate directions further by requiring that all the search directions are A-orthogonal. The residuals are orthogonal to all previous residuals and search directions computed [13, 14]. Two vectors are said to be A-orthogonal if the inner product $(x, y)_A = x^T Ay$ is equal to zero. The residuals are defined as $r_n = b - Ax_n$, where x_n is the nth approximate solution and r_n is the residual calculated at the nth iteration. Another significant attribute to using conjugate gradients is that the sequence of iterates computed in the method make up a Krylov subspace. A Krylov subspace is a collection of vectors such that each consecutive vector is the previous vector multiplied by the matrix A or $(b, Ab, A^2b, \ldots, A^nb)$. This unique sequence also leads to the A-norm of the error to be minimized in that space, thus leading to the

solution of the system of equations. The A-norm is defined as $\|x\|_A = \sqrt{x^T Ax}$. The error is defined as $x_n - x^*$ or the approximate solution at the nth iteration minus the actual solution. Recall the function f mentioned in steepest descent, CG minimizes the function $f(x) = \frac{1}{2}x^T Ax - x^T b$. From CG, we also get the following identities:

1. $\langle x_1, x_2, \ldots, x_n \rangle = \langle p_0, p_1, \ldots, p_{n-1} \rangle$
2. $\langle p_0, p_1, \ldots, p_{n-1} \rangle = \langle r_0, r_1, \ldots, r_{n-1} \rangle$
3. $\langle r_0, r_1, \ldots, r_{n-1} \rangle = \langle b, Ab, \ldots, A^{n-1}b \rangle$.

These equalities can be clearly seen in the algorithm stated in 2.2 when $x_0 = 0$.

Though, CG is a very effective iterative method, dense and ill-conditioned matrices can be solved just as effectively through factoring and backsolve. Matrices that are not SPD can be solved by using the method of Conjugate Gradients for Normal Equations () by changing $Ax = b$ to $A^T Ax = A^T b$ [8, 14, 11]. However, these systems are difficult to precondition. Since this work is centered around SPD matrices, non-SPD matrices will not be addressed further.

2.2 Algorithm

In this section, the formal algorithm for CG is stated and explained. For CG, a SPD matrix A, a vector b, and initial guess x_0 are specified by the end user. The stopping condition of τ and the maximum number of iterations are also specified. From this user-specified information, an initial residual, r_0, is computed and is used for the initial search direction p. The algorithm then enters into a `for` loop that is limited by the maximum iterations. The first step in the loop is the calculation of the next vector in the Krylov subspace which is then used to calculate the step length in v, which ensures A-orthogonality. Next, the solution is approximated by adding the previous solution to the updated search direction. The residual is approximated using the previous residual approximation and the step length applied to the Krylov subspace. The step direction is then computed by using the previous residual and the current residual. Finally, the search direction is updated with the step direction. This cycle continues until either the maximum iterations is reached or the algorithm has converged based on the stopping condition.

Algorithm 2.1 The method of Conjugate Gradients

Given: matrix A

Given: right-hand side b

Given: initial guess x_0

Given: stopping condition τ and maximum iterations N_{max}

$r_0 \leftarrow b - Ax_0$

$p \leftarrow r_0$

for $n = 1 \rightarrow N_{max}$ **do**

$z \leftarrow A * p$

$v \leftarrow \frac{(r_{n-1}, r_{n-1})}{(p,z)}$

$x_n \leftarrow x_{n-1} + v * p$

$r_n \leftarrow r_{n-1} - v * z$

$\mu \leftarrow \frac{(r_n, r_n)}{(r_{n-1}, r_{n-1})}$

$p \leftarrow r_n + \mu * p$

if converged(r_n, A, b, τ) **then**

return x_n

end if

end for

Failure: maximum iterations exceeded

Chapter 3

Conjugate Gradients with Incomplete Cholesky Preconditioning

In order to improve CG, there are several preconditioners that can speed up convergence. A few of these preconditioners are Jacobi preconditioning, incomplete factorization preconditioning, and polynomial preconditioning. Jacobi preconditioning uses the inverse of the diagonal of A or a diagonal matrix formed by a vector c such that the condition number of $M^{-1}A$ is minimized, where M is the Jacobi Preconditioner [8, 11, 13, 14]. In most cases, minimizing the condition number is not enough. Polynomial preconditioning uses an estimation on the spectral radius of A by finding a polynomial p such that $1 - zp(z)$ is small on the spectral radius. If a p is found then $p(A)$ is used as the preconditioner. This method can significantly improve performance of CG and the implementation is matrix free, only matrix-vector computation. However, finding the polynomial p can be quite difficult to obtain depending on A. Incomplete factorization preconditioners transform $Ax = b$ by using another SPD system that obtains the same solution [8]. Let M be a SPD matrix that is close to A^{-1} then the eigenvalues of MA will be clustered near one. Unfortunately, MA is unlikely to be SPD and therefore CG cannot be applied to the system. Therefore for CG, we need to use a dual-sided preconditioner instead of a single-sided preconditoner. The Incomplete Cholesky factorization, $M = LL^T$ can be used to preserve the SPD properties by splitting the preconditioner such that $L^{-1}AL^{-T}u = L^{-1}b$ where $x = L^{-T}u$. This will lead to the eigenvalues clustering near one. The factorization is incomplete since the small elements of the factorization are dropped and/or a fixed number of non-zeros are allowed in the factors for storage [8]. Therefore, for this paper CG will be conditioned using Incomplete Cholesky factorization and referred to as ICCG for the remainder of this paper.

3.1 ICCG

From the introduction of this section, ICC was chosen to be the preconditioner used for CG. The algorithm for ICCG is much the same as CG with a few variations that will be discussed in the algorithm subsection. In order to implement ICC, the Ifpack package in Trilinos was utilized in order to create the preconditioner. In Ifpack [12, 3], the preconditioner has five parameters that determine how the preconditioner is computed. These five parameters are level of fill, absolute threshold, relative threshold, relaxation value, and drop tolerance. Level of fill is the ratio of non-zeros per row or column in the preconditioner to non-zeros from the upper triangular portion of A. Absolute threshold defines the value to add to each diagonal element in the original matrix. Relative threshold defines what value to multiply each diagonal element of the original matrix by before the contribution of absolute threshold. Drop tolerance defines the threshold value for dropping elements in the factor. These dropped elements are scaled by the relaxation value and added to the diagonal term unless the relaxation value is zero.

3.2 Algorithm

As stated before, ICCG has only a few changes from CG. One is that the initial search direction has the inverse of the ICC applied to the initial residual. Another change is on the sequence of Krylov subspaces: the subspaces are generated by $L^{-1}AL^{-T}$ as opposed to A. The final change is that the final solution x_n must have L^{-T} applied to it in order to have x be the solution of $Ax = b$ again.

Algorithm 3.1 Incomplete Cholesky Conjugate Gradients

Given: matrix A

Given: right-hand side b

Given: initial guess x_0

Given: stopping condition τ and maximum iterations N_{max}

Compute: L is the incomplete Cholesky factorization

$r_0 \leftarrow b - Ax_0$

$p \leftarrow L^{-1} * r_0$

for $n = 1 \rightarrow N_{max}$ **do**

$z \leftarrow L^{-1} * A * L^{-T} p$

$v \leftarrow \frac{(r_{n-1}, r_{n-1})}{(p,z)}$

$x_n \leftarrow x_{n-1} + v * p$

$r_n \leftarrow r_{n-1} - v * z$

$\mu \leftarrow \frac{(r_n, r_n)}{(r_{n-1}, r_{n-1})}$

$p \leftarrow r_n + \mu * p$

if converged(r_n, A, b, τ) **then**

$x_n = L^{-T} x_n$

return x_n

end if

end for

Failure: maximum iterations exceeded

Chapter 4

Implementation of Conjugate Gradients and Incomplete Cholesky Conjugate Gradients

CG and ICCG are implemented in three parts: the main program, the solver, and the preconditioner. Only the main program is visible to the end user. The solver and preconditioner are implemented such that the end user never needs to interact with either. The main program contains all the user specified data which is the matrix, vector, preconditioner, and solver of the user's choosing. This user specified data is communicated to the solver and preconditioner classes to compute the solution of the system without the end user having to worry about the setup of the preconditioner or the solver. Thus, the user can focus more on the task at hand of solving their system of equations.

4.1 Main Program

In the main program the user defines the matrix as a linear operator and a vector b for the problem $Ax = b$. For the test program, two options were used for the matrix: a matrix market file or by constructing the matrix using pre-existing functions in Playa. The vector b in the test program was constructed by the multiplication of a random vector x and one of the matrices specified earlier. The vector was constructed this way so that the error could be tested after computation of the solution. Following the declaration of the matrix and vector, the user specifies the stopping conditions for CG. In order to focus on the overall implementation of CG and ICCG, the stopping conditions of the two methods are based off of a max iteration and a limit on the relative residual norm. For industrial use, the code will need to be modified to include better error handling, notification, and provide support for different stopping conditions. The preconditioner is specified by declaring a new `ICCPreconditionerFactory` and the solver is called with a return value of `SolverState`. The explanation of these will come later in the implementation explanation. Overall, this part of the implementation functions as the root location for all the user provided information and as

a interface to the solver and preconditioner.

4.2 Solver

The CG solver extends from an abstract base class `KrylovSolver` which defines the overall structure of all derived Krylov solvers. `KrylovSolver` contains two main virtual functions: `solve` and `solveUnprec`. The `solveUnprec` is a pure virtual function. Pure virtual functions allow for the implementation of the function to be specified later through the derived classes [10].These virtual functions provide a foundation for the specific Krylov solvers and can be modified for a specific use case. As an example, `CGSolvers` automatically contains the functions `solve` and `solveUnprec` and can define a different implementation for both of these functions. However, `CGSolvers` only provides a different implementation for `solveUnprec`. The code for `solveUnprec` follows from the algorithm stated in the CG section of this paper.

For the function, solve, the specification of a preconditioner is checked and if not present the work is passed onto the particular classes implementation of `solveUnprec`. If a preconditioner is specified then `KrylovSolver` calls for the creation of the preconditioner which is handled through the preconditioner factories. The preconditioner is constructed indirectly through factories since there is no information about the matrix until the `solve` function is called. Once the preconditioner has been created, the function `solve` determines the next action to occur based upon one of two preconditioner types: single-sided(left or right) or dual-sided. The explanation both of these was addressed in the section about ICC. Since the ICC preconditioner is being used for CG, the preconditioner is dual-sided.

The remaining implementation of this solve method follows from the algorithm stated in Conjugate Gradients with Incomplete Cholesky Conditioning. The return of both solve methods in all subclasses of `KrylovSolver` is a class that contains four attributes: convergence status, convergence status message, final iteration count, and final residual. Convergence status has three possible values: SolveCrashed, SolveFailedToConverge, and SolveConverged. The convergence status message is a programmer defined message regarding convergence. Final iteration count is the total number of iterations it took for the system to meet stopping conditions. Final Residual is the final residual computed when stopping conditions were met.

4.3 Preconditioner

PreconditionerFactory, a template class, takes an abstract specification and builds an implementation specification for `ICCPreconditionerFactory`. `ICCPreconditionerFactory` sends the preconditioner parameters of fillLevels, relaxation value, relative threshold, absolute threshold, and drop tolerance to `ICCFactorizableOp` through `getICCPreconditioner`. In `ICCFactorizableOp`, the matrix is checked for compatibility with ICC factorization before the parameters mentioned before are sent to `EpetraMatrix`. `EpetraMatrix` creates a new `ICCOperator` by creating a new instance of the `ICCOperator` with the parameters passed from `ICCFactorizableOp` and defines the two sides of the preconditioner (the ICC operator and its transpose). The definition for the two sides is then passed to `GenericTwoSidedPreconditioner`. `GenericTwoSidedPreconditioner` uses the definition to create the `left` and `right` functions for the ICC operator where `left` is the ICC operator and `right` is the transpose of the operator. `ICCOperator` takes the matrix from `EpetraMatrix` and the parameters passed down from `ICCPreconditionerFactory` to create the ICC preconditioner using Ifpack's `ICT` operator. The ICC Preconditioner is now available to be used throughout the remaining of the program by solver and used by the references of `left`, `right` and `apply`. The `apply` function applies the inverse of the preconditioner to a vector specified to get a new vector.

4.4 UML Sequence Diagram

The diagram below depicts the sequence of events that ICCG goes through in order to compute a solution to $Ax = b$. The boxes across the top represent the classes or the client in the diagram. The classes below are the following: `CGSolvers`, `ICCPreconditionerFactory`, `EpetraMatrix`, `IfpackICCOperator`, `GenericTwoSidedPreconditioner`, `ICCFactorizableOp`, `SolverState`, and `LinearOperator`. The lines extending down from the classes represent time and the diagram is read from left to right.

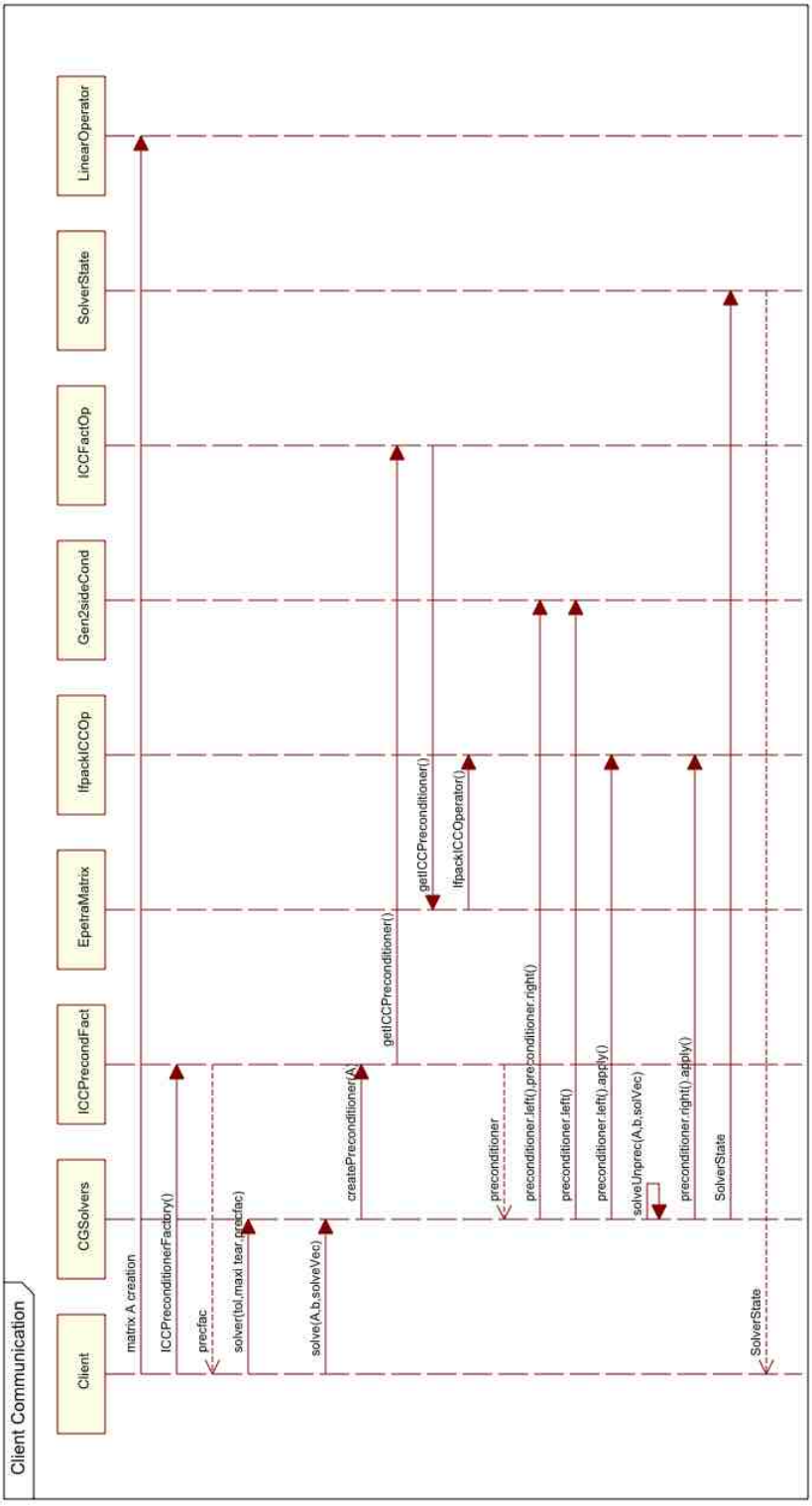

Figure 4.1. UML Sequence Diagram

Chapter 5

Results

Testing for CG and ICCG was done using several matrices, but I show the results for only three matrices. Two matrices, 494_bus and nos3, can be found in matrix market[1]. The third matrix, nut-stiffness-4, was generated by Dr. Kevin Long using a finite element program. The matrix 494_bus is a 494x494 matrix from a problem in power system networks. This matrix contains 1080 non-zeros and has a condition number 3.9e+6. The matrix nos3 is a 960x960 matrix from a finite element approximation to a biharmonic operator on a rectangular plate with one side fixed and the others free. This matrix contains 8402 nonzero elements and has a condition number of 7.3e+4. The matrix nut-stiffness-4 is a 22492x22492 thermal stiffness matrix for a meshed hex nut. This matrix contains 315412 nonzero elements and has a condition number of 777.8476. Below are the structure and city plots for each matrix. A structure plot depicts the nonzero elements in relation to the position in the matrix. The city plot shows the magnitiude of each nonzero in relation to the other nonzero elements. The top left corner represents the top left corner in the matrix.

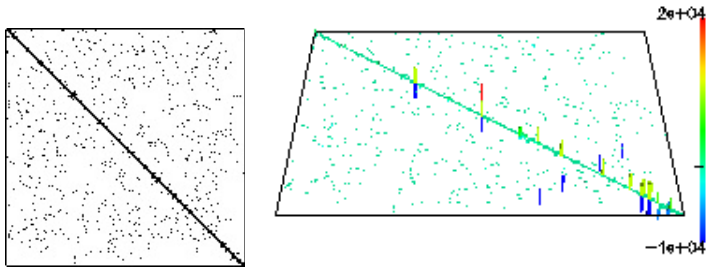

Figure 5.1. 494_bus Structure Plot and City Plot

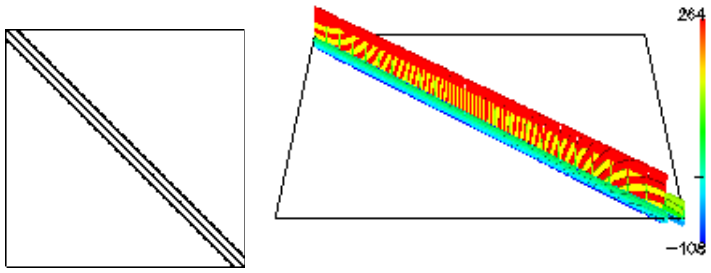

Figure 5.2. nos3 Structure Plot and City Plot

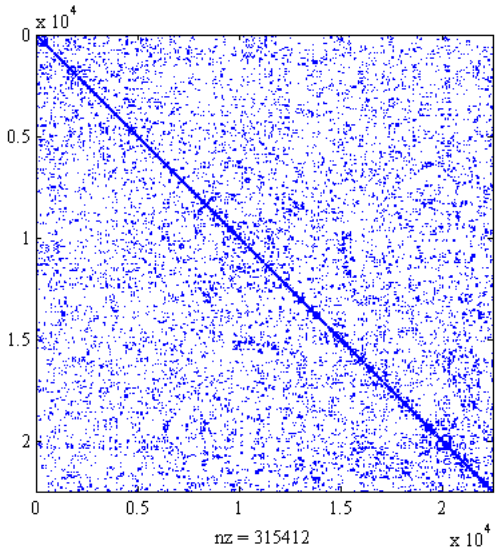

Figure 5.3. nut-stiffness-4 Structure Plot. City plot could not be shown for this matrix due to memory limitations.

5.1 Playa: CG vs ICCG

The results depicted in the subsections show convergence of the relative residual over iterations for Playa's CG and ICCG. The parameters for the preconditioner of ICCG are the following: level of fill(LOF) equals 0, drop tolerance of 1.0e-4, absolute threshold of 0.0, relative threshold of 1.0, and relaxation value of 0.0. The maximum iterations allowed for each algorithm was 20,000 and the tolerance on the relative residual for stopping was 1.0e-6.

5.1.1 494_bus Results

From the graph below, it is easy to see that the convergence of CG decreased in iterations by over a factor of 3 when a preconditioner was added.

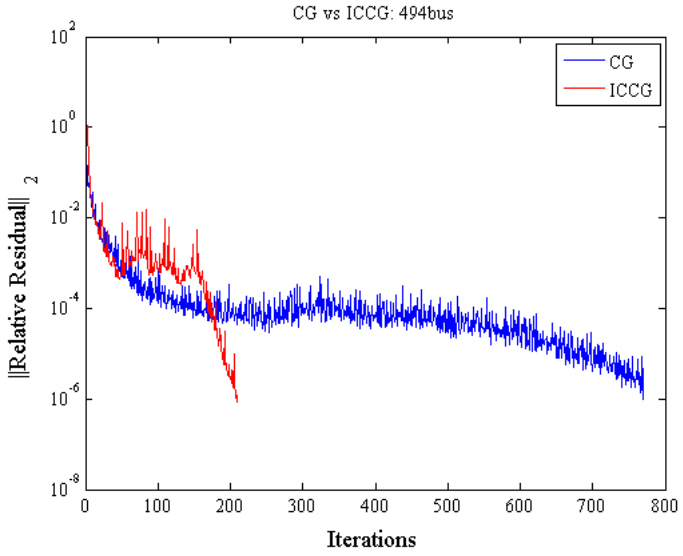

Figure 5.4. CG vs ICCG: 494_bus

5.1.2 nos3 Results

From the graph below it is easy to see that the convergence of CG decreased in iterations by about a factor of 2 when a preconditioner was added.

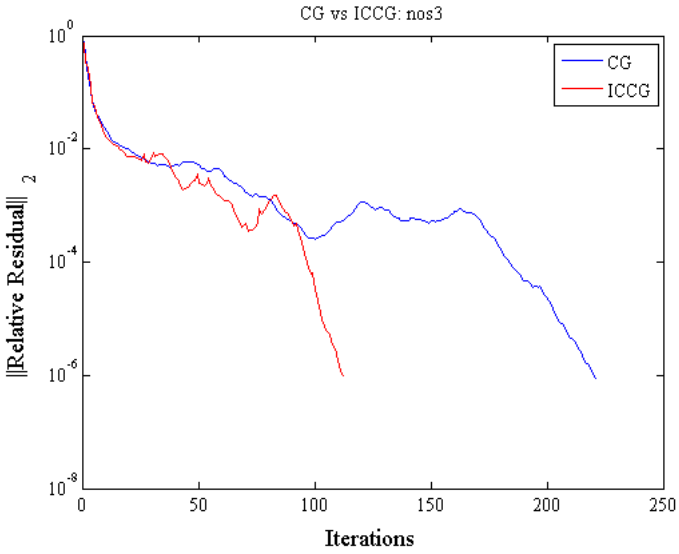

Figure 5.5. CG vs ICCG: nos3

5.1.3 nut-stiffness-4 Results

From the graph below it is easy to see that the convergence of CG decreased in iterations by about a factor of 3 when a preconditioner was added.

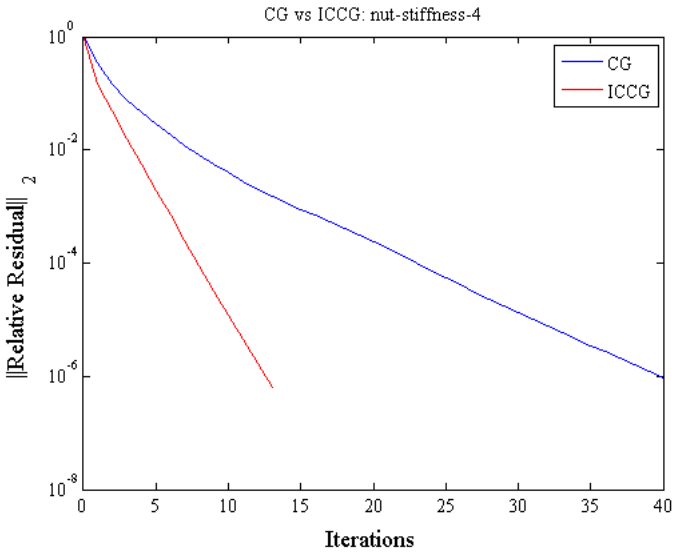

Figure 5.6. CG vs ICCG: nut-stiffness-4

5.2 Effect on performance of drop tolerance

In this section, the graphs depict the convergence of the relative residual over iterations for six preconditioners. The six preconditioners are the following: MATLAB `ICT`, MATLAB LOF=0, Ifpack LOF=0, Ifpack LOF=1, Ifpack LOF=2, and Ifpack LOF=3. The graphs were done for four different drop tolerances as well. The drop tolerances represented are 1.0e-1, 1.0e-2, 1.0e-4, and 1.0e-8.The maximum iterations allowed for each algorithm associated with the preconditioner was 20,000 and the tolerance on the relative residual for stopping was 1.0e-6. The graphs specify MATLAB and Playa since the convergence plots are with respect to the ICCG solvers of Playa and MATLAB. The Ifpack preconditioners were used with Playa and the MATLAB preconditioners were used with MATLAB.

5.2.1 494_bus Results

The graph below shows the convergence curves using the MATLAB preconditioners only since the data collected from Playa was useless due to the Ifpack preconditioners containing immensely small elements which led to no data for each iteration. This issue also led to the maximum iteration being met for all of the Ifpack preconditioners mentioned in the introduction of this section. MATLAB had a reduction of about 10 iterations by changing from a preconditioner with no fill to an Incomplete Cholesky preconditioner.

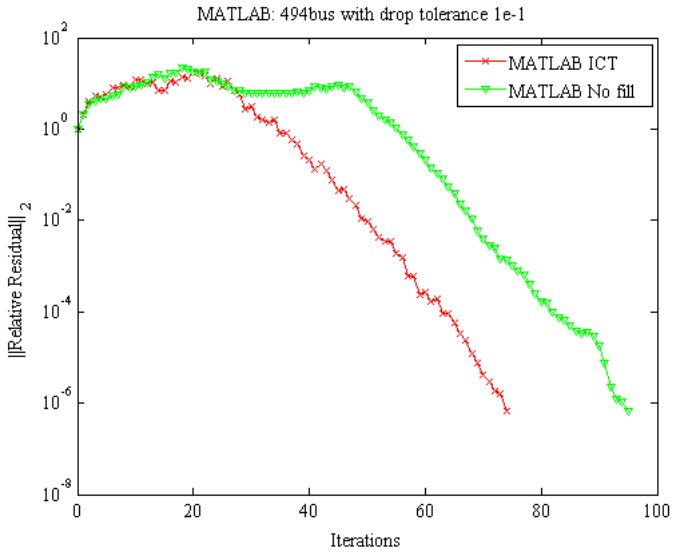

Figure 5.7. MATLAB: 494_bus with drop tolerance $1.0e-1$

In the graph depicted directly below we can see that changing the LOF from no fill to 1 in Ifpack shows an increase in convergence by about 80%. The rate of convergence is increased to about 56% from LOF=1 to LOF=2 and 40% from LOF=2 to LOF=3. MATLAB shows an increase in convergence by about 70% from no fill on the preconditioner to using `ICT` as the preconditioner.

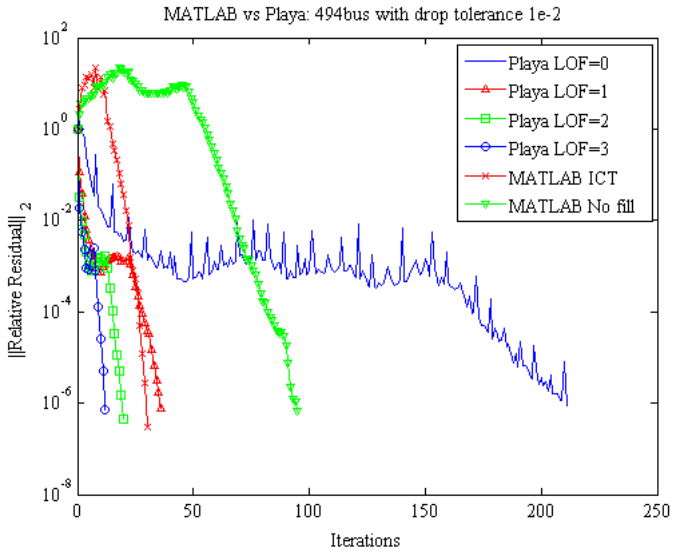

Figure 5.8. MATLAB vs Playa: 494_bus with drop tolerance $1.0e-2$

For Ifpack, the change of drop tolerance didn't effect the iteration count on any of the preconditioners when the drop tolerance was changed from 1.0e-2 to 1e-4. As for MATLAB, the `ICT` preconditioner showed a reduction of around 80%. MATLAB no-fill preconditioner showed no effect since the drop tolerance has no effect on the preconditioner.

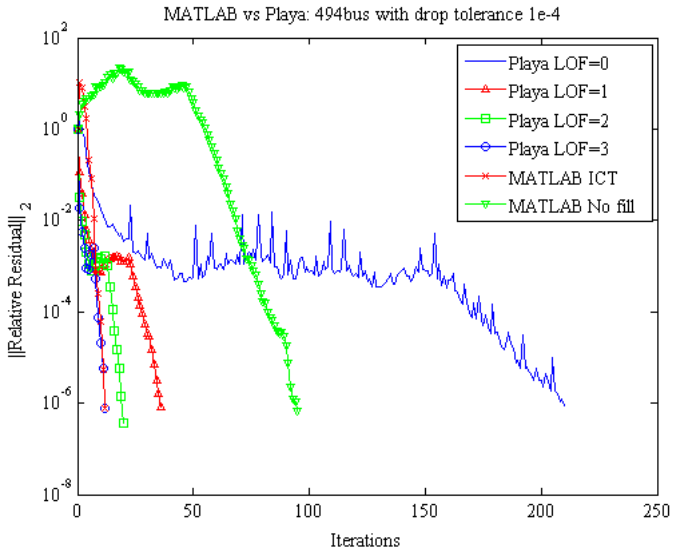

Figure 5.9. MATLAB vs Playa: 494_bus with drop tolerance $1.0e-4$

For Ifpack, the change of drop tolerance didn't effect the iteration count on any of the preconditioners when the drop tolerance was changed from 1.0e-4 to 1e-8. As for MATLAB, the `ICT` preconditioner showed a reduction around 80% . MATLAB no-fill preconditioner showed no effect since the drop tolerance has no effect on the preconditioner.

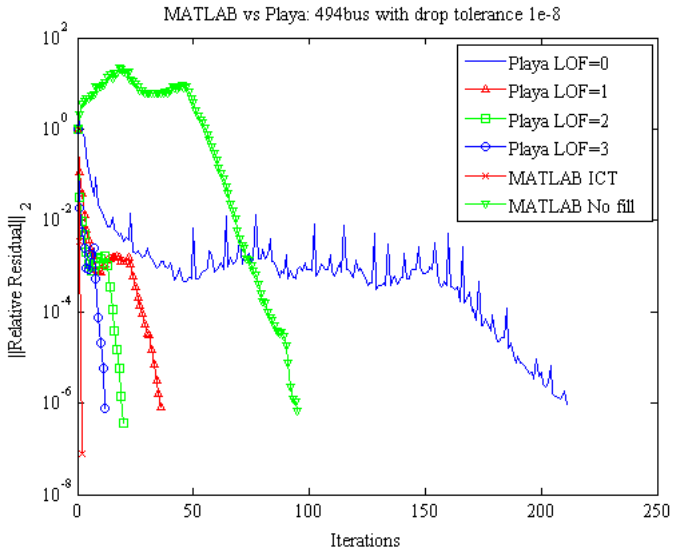

Figure 5.10. MATLAB vs Playa: 494_bus with drop tolerance $1.0e-8$

5.2.2 nos3 Results

For a drop tolerance of 1.0e-1, Ifpack preconditioners with a level of fill greater than zero seem to have the same convergence rate. MATLAB `ICT` actually has a slower convergence rate with this large of a drop tolerance as compared to MATLAB no-fill preconditioner. Ifpack preconditioners still converges in fewer iterations than MATLAB's preconditioners.

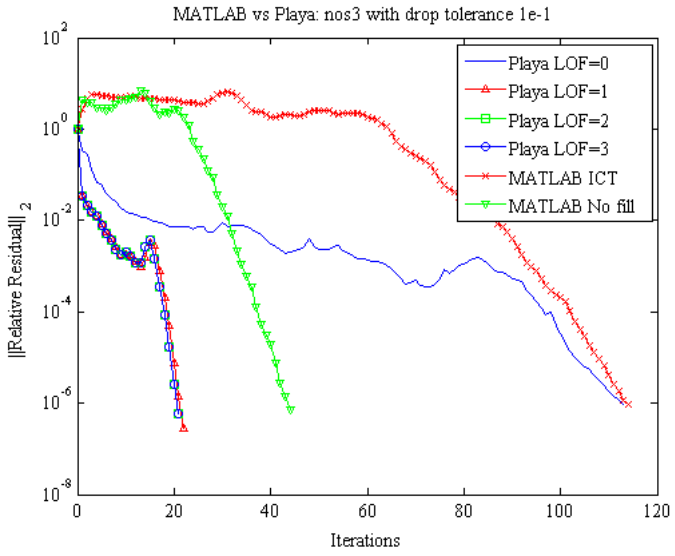

Figure 5.11. MATLAB vs Playa: nos3 with drop tolerance $1.0e-1$

For a drop tolerance of 1.0e-2, the Ifpack preconditioners with a level of fill greater than zero have shifted apart. The convergence rate for the Ifpack preconditioner as the LOF is increased and MATLAB `ICT` converges within fewer iterations than MATLAB Preconditioner with no fill. The convergence rate of `ICT` has improved by about 50%.

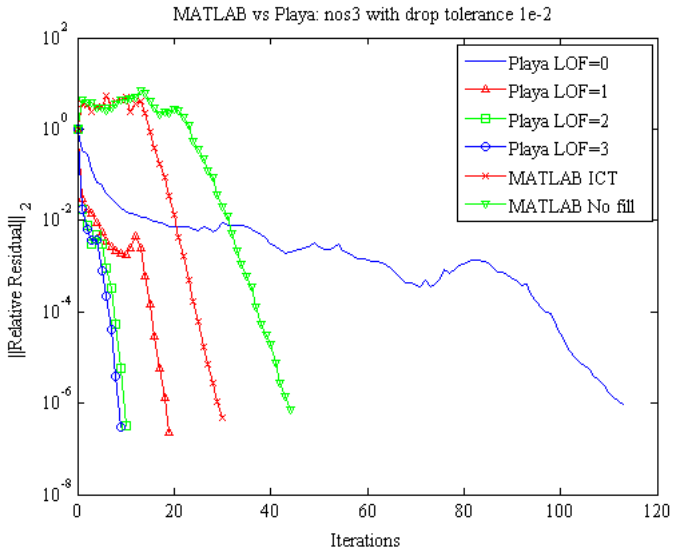

Figure 5.12. MATLAB vs Playa: nos3 with drop tolerance $1.0e-2$

For a drop tolerance of 1.0e-4, the Ifpack preconditioners with a level of fill greater than zero have shifted apart. The convergence rate for the Ifpack preconditioners do not show much of a change in convergence with the change in preconditioners. However, MATLAB `ICT`'s rate of convergence has increased by about 80%.

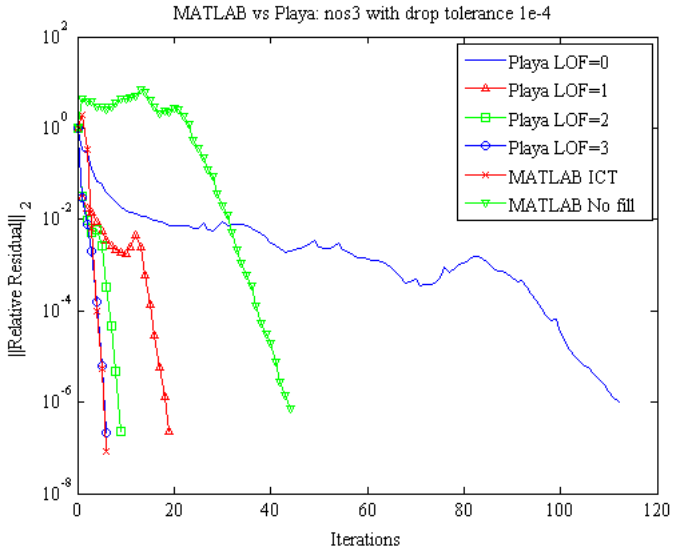

Figure 5.13. MATLAB vs Playa: nos3 with drop tolerance $1.0e-4$

For a drop tolerance of 1.0e-8, the convergence rates for the Ifpack preconditioners do not show much of a change in convergence with the change in preconditioners just like before. However, MATLAB `ICT`'s rate of convergence has increased by about 40%.

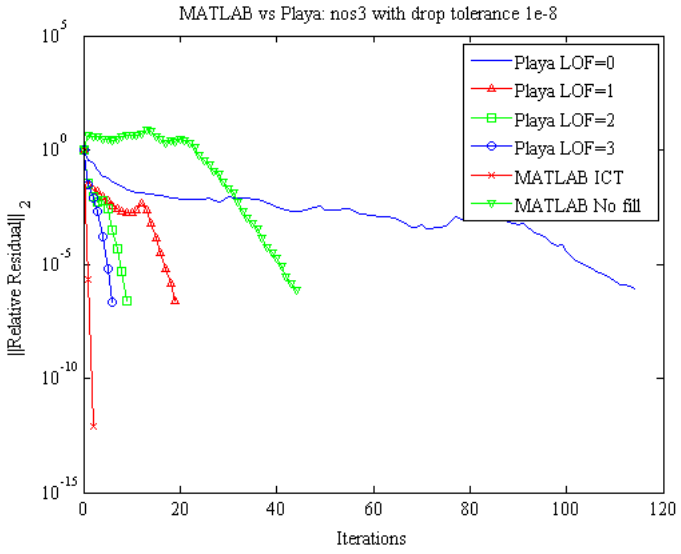

Figure 5.14. MATLAB vs Playa: nos3 with drop tolerance $1.0e-8$

5.2.3 nut-stiffness-4 Results

For a drop tolerance of 1.0e-1, MATLAB no-fill takes half as many iterations as MATLAB `ICT`. As for the Ifpack, the `ICT` preconditioners converge in the same number of iterations.

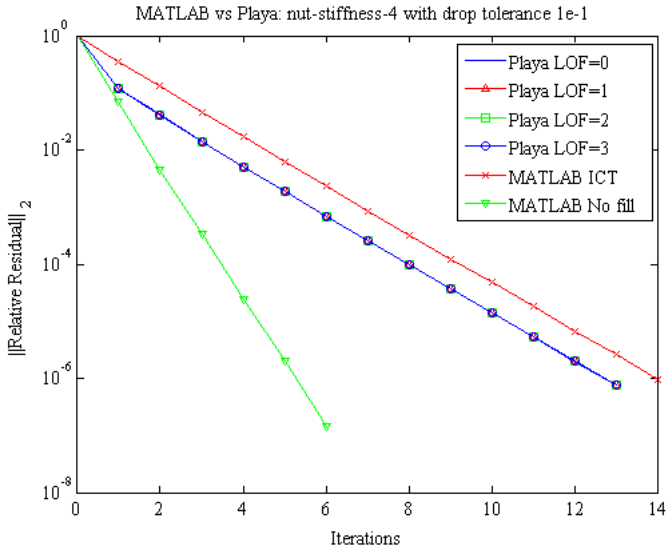

Figure 5.15. MATLAB vs Playa: nut-stiffness-4 with drop tolerance $1.0e-1$

When the drop tolerance is decreased to 1.0e-2, the Ifpack preconditioners show no change. The MATLAB `ICT` preconditioner converged in about a third of the iterations from the larger drop tolerance of 1e-1. The MATLAB no-fill preconditioner converged in the same number of iterations.

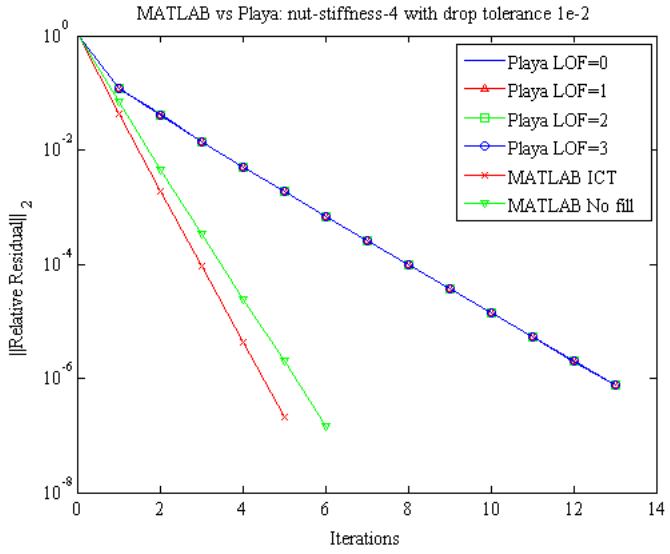

Figure 5.16. MATLAB vs Playa: nut-stiffness-4 with drop tolerance $1.0e-2$

For a drop tolerance of 1.0e-4, all the Ifpack preconditioners except the no-fill con-

verged in half the iterations as directly above. The MATLAB `ICT` preconditioner showed a decrease by over 50%. The no-fill preconditioners for MATLAB and Ifpack showed no decrease.

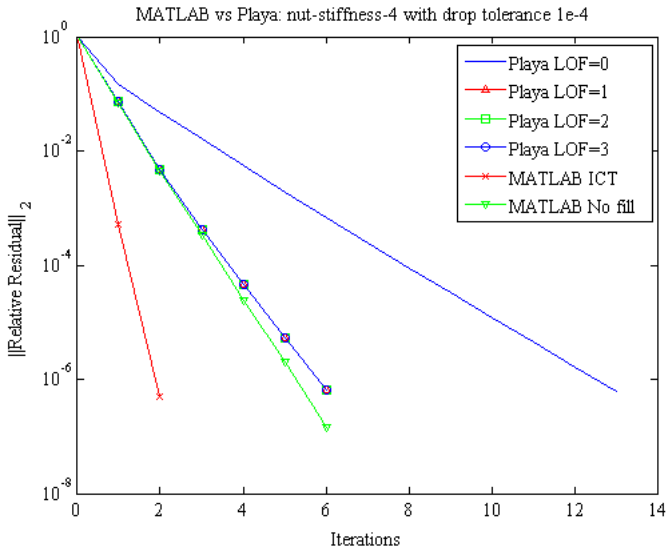

Figure 5.17. MATLAB vs Playa: nut-stiffness-4 with drop tolerance $1.0e-4$

The drop tolerance of 1.0e-8 causes the Ifpack preconditioners to finally separate into different converging patterns. The Ifpack LOF=3 decreases by 50% while Ifpack LOF=2 and Ifpack LOF=1 decrease by a third and a sixth respectively. MATLAB ICT decreases in iterations by 50%.

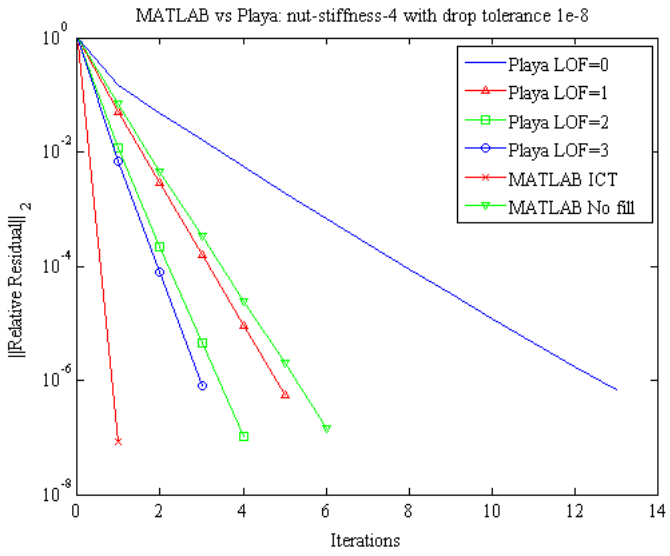

Figure 5.18. MATLAB vs Playa: nut-stiffness-4 with drop tolerance $1.0e-8$

5.3 Effect on fill of drop tolerance and LOF

The tables below represent the number of nonzero elements in the preconditioner for the preconditioner mentioned on the left and the different drop tolerances across the top of the table. For 494_bus and nos3 tables, the preconditioners that had no fill were not effected by drop tolerance as expected. However, the table for nut-stiffness-4 matrix showed an increase in non-zeros for the Ifpack no-fill. For all matrices the number of non-zeros in the preconditioner increased or stayed relatively the same going from a large to small drop tolerance. This appears to be the case when increasing the fill for each drop tolerance except for the MATLAB preconditioners with a drop tolerance of 1.0e-1. As the number of non-zeros increase in the preconditioner, the amount of storage and computation time increase.

5.3.1 494_bus

In the below table, the number of non-zeros for each preconditioner computed for the 494_bus is displayed.

Table 5.1. Number of non-zeros for the preconditioners of 494_bus

	1e-1	1e-2	1e-4	1e-8
Playa LOF0	849	849	849	849
Playa LOF1	1390	1408	1413	1413
Playa LOF2	1718	1866	1922	1931
Playa LOF3	1904	2165	2270	2308
MATLAB `ICT`	961	1857	3844	6561
MATLAB NF	1080	1080	1080	1080

5.3.2 nos3

In the below table, the number of non-zeros for each preconditioner computed for the nos3 is displayed.

Table 5.2. Number of non-zeros for the preconditioners of nos3

	1e-1	1e-2	1e-4	1e-8
Playa LOF0	1919	1919	1919	1919
Playa LOF1	13991	16406	16447	16448
Playa LOF2	14211	26477	31784	31843
Playa LOF3	14211	27279	38833	39088
MATLAB `ICT`	1626	9960	35193	39850
MATLAB NF	8402	8402	8402	8402

5.3.3 nut-stiffness-4

In the below table, the number of non-zeros for each preconditioner computed for the nut-stiffness-4 is displayed.

Table 5.3. Number of non-zeros for the preconditioners of nut-stiffness-4

	1e-1	1e-2	1e-4	1e-8
Playa LOF0	43048	43048	44619	44619
Playa LOF1	168952	168952	170862	277750
Playa LOF2	168952	168952	171116	483455
Playa LOF3	168952	168952	171116	666511
MATLAB `ICT`	30760	209691	917592	4527599
MATLAB NF	168952	168952	168952	168952

Chapter 6

Software Complications

While testing the implementation of CG and ICCG, several problems with Ifpack were encountered. Ifpack is very hard for the reader to understand since there are several function names that are unclear as to what they actually compute. The construction of preconditioners is not well defined and thus led to reading through several lines of code to see the necessary functions needed to build the preconditioners and its parameters. The main issue that Ifpack had was the reoccuring of elements of quantity inf or values that were really large or small present in the preconditioner. The presence of these values in the preconditioner caused a failure in the computation of the vector returned when applying the preconditioner which trickled down to the solvers not converging or having a negative residual norm.

Chapter 7

Conclusion

The implementation of CG and ICCG in Playa can be cumbersome but the separation of parts is necessary in both locating possible issues with the program and keeping specific operations as their own class. Ifpack had two major issues that were tremendously troublesome. The first of these issues was that the computation of the preconditioners was often unreliable. The other issue was that the code was not well documented for an end-user to determine the correct preconditioner to use or the creation process of the preconditioner. A bug report for Ifpack will be constructed in order to help the developers improve Ifpack for future users of ICC preconditioners. Future directions for Krylov solvers and Preconditioned Solvers include the implementation of Newton Krylov Solvers, Preconditioned Newton Krylov Solvers, and expansion to nonlinear equations.

Bibliography

[1] Matrix market. http://math.nist.gov/MatrixMarket/, May 2007.

[2] Thyra: Interfaces for abstract numerical algorithms. http://trilinos.sandia.gov/packages/thyra, May 2007.

[3] Trilinos/ifpack: Object-oriented algebraic preconditioner package. http://trilinos.sandia.gov/packages/ifpack, October 2012.

[4] Mark S Gockenbach, Matthew J. Petro, and William W Symes. C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software TOMS*, 25(2):191–212, June 1999.

[5] Mark S Gockenbach, Daniel R Reynolds, Peng Shen, and William W Symes. Efficient and automatic implementation of the adjoint state method. *ACM Transactions on Mathematical Software TOMS*, 28(1):22–44, 2002.

[6] Michael Heroux, Roscoe Bartlett, Victoria E. Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of trilinos. *ACM Transactions on Mathematical Software TOMS*, V(N):1–27, 2004.

[7] Victoria E. Howle, Robert C. Kirby, Kevin Long, Brian Brennan, and Kimberly Kennedy. Playa: High-performance programmable linear algebra. *Scientific Programming*, 20(3):257–273, 2012.

[8] C.T. Kelley. *Frontiers in Applied Mathematics: Iterative Methods for Linear and Nonlinear Equations*. SIAM,Philadelphia, 1995.

[9] Robert C. Kirby. A new look at expression templates for matrix computation. *Computing in Science and Engineering*, 5(3):66–70, 2003.

[10] Ray Lischner. *C++ in a nutshell*. O'Reilly, Sebastopol, 2003.

[11] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1996.

[12] Marzio Sala and Michael Heroux. Robust algebraic preconditioners using ifpack 3.0. Technical report, Sandia National Laboratories, 2005.

[13] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, 1994.

[14] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[15] Todd L. Veldhuizen. *Blitz++: The Library that Thinks it is a Compiler.* Springer-Verlag, Berlin, Heidelberg, New York, 2000.