Study of Preconditioners for Edge FEM Applied to Frequency Domain in Maxwell's Equations

University of Arizona / VIGRE
Summer 2009 Research Program in Computational Photonics

Team Leader: Robert Muth
Team Members: Matthew Hom and Daniel Rorabaugh

Program Director: Moysey Brio
Project Managers: Paul Dostert and Jinjie Liu

1 Introduction

Preconditioning is a method of modifying a linear system in order to reduce the costs associated with solving the system numerically. By applying a specifically-designed preconditioning algorithm before or during solving, it is possible to speed convergence and improve solution accuracy. Finding a maximally-effective, minimally-taxing preconditioning algorithm for a given iterative solving process is extremely important in application.

Our team's particular area of research is in physics-based preconditioners designed for the edge finite element method (FEM) of approximating solutions to Maxwell equations of electromagnetics. Krishna Gundu, former UA graduate student, reports encountering significant problems in attempting to solve systems that result from edge FEM discretization. He relates that a variety of iterative solvers (LSQR, GMRES, BICGSTAB), combined with extant preconditioning schemes (ILU, GMM, scaling) are only successful at solving for relatively small (less than 1500×1500) matrices. MUMPS, a direct solver, is successful at more reasonable $10{,}000 \times 10{,}000$ matrices, but involves potentially prohibitive 40 GB memory costs [1].

Comparatively, numerous recent papers report success in preconditioning FEM systems [3]. In light of these papers, our team attempted to verify Krishna Gundu's results, comparing the results of available preconditioners, and analyzing the properties of these specific matrices which cause the poor behavior when iterative solvers are applied. The goal was to find, adapt, or design an effective preconditioner that allows robust convergence and requires reasonable memory and CPU usage.

2 Background

Jinjie Liu and Paul Dostert, postdoctoral fellows at UA, produced for study a number of matrices using Krishna Gundu's Edge FEM code. The discretizations vary in coarseness so as to produce matrices of varying sizes, ranging from 2756 $\times$ 2756 to 33668 $\times$ 33668. All the matrices are sparse, square, complex, and symmetric (non-Hermitian), with complex positive and negative eigenvalues.

Before delving into the practical methods of handling large FEM matrices via matrix preconditioning, our group had to become familiar with the theoretical underpinnings of matrix norms, condition numbers, and direct and iterative solution methods. Given a linear system

$$Ax = b,$$

and the system that results from slight perturbations or uncertainty in the entries of the b vector,

$$A(x + \Delta x) = (b + \Delta b),$$

it is possible to derive the following inequality via matrix norms:

$$\frac{||\Delta x||}{||x||} \leq ||A|| \cdot ||A^{-1}|| \frac{||\Delta b||}{||b||}.$$

The quantity $||A|| \cdot ||A^{-1}||$ is known as the matrix condition number. From a practical standpoint, the condition number represents an upper bound on the relative error introduced into the solution vector [8]. This is of particular importance when solving large "ill-conditioned" matrices — matrices in which a small perturbation in the input vector, b, yields a large change in the solution vector, x. Consideration of condition is of great importance in numerical analysis of physical systems. Precise measurement of input elements can be for naught if the condition of the system is poor, since in general, systems with a condition number κ may, in the worst cases, expect a loss of $\log_{10} \kappa$ digits of solution accuracy, regardless of method. The condition number must be considered when truncating solution data to reliable values. MATLAB's condition number estimator *condest* places the condition number of the matrices under study (up through 7868 $\times$ 7868, as the larger matrices could not be calculated by MATLAB) in the range of 10^9 [3]. One might expect a staggering loss of 9 digits in solution accuracy, but this is an upper bound, and in practice, the loss is not so steep. The effective relative error magnification our group saw in practice, by repeatedly perturbing the data vector, are in the range of 10^4, much less than the worst-case scenario provided by the *condest* condition number. The loss of four digits of accuracy is still undesired, and an effective preconditioner would reduce the condition number and the error magnification in practice.

When dealing with large matrices of the order of magnitude $10^6 \times 10^6$ it is important to distinguish between direct methods and iterative methods of solving such systems. Direct methods, which generally involve some form of

Gaussian Elimination, are computationally intensive. For an $n \times n$ matrix, the number of required computations has an order of magnitude of n^3, and requires vast amounts of memory storage. Direct methods lie in stark contrast with iterative methods of solving systems of equations, which find solutions within a given tolerance via a converging series of approximations. These methods take advantage of the matrix-vector products which can be performed relatively quickly in sparse systems. In Robert Muth's preliminary research [3], only the general iterative solver GMRES was found to converge to a solution for the matrices under study. The fact that the matrices are non-Hermitian and not positive definite perhaps makes the system untenable for the other solvers. The GMRES convergence is very slow however, and for every $n \times n$ matrix tested, nearly n iterations were needed in order to converge to a modest relative residual of 10^{-6}. For GMRES to be effective in solving the large systems that result from sufficiently fine FEM discretizations, the number of iterations should be far less than n. Our research therefore additionally focuses on accelerating GMRES convergence through preconditioning.

Preconditioning is a method of decreasing the condition number of the system under study and accelerating convergence. In general methods, this is achieved by transforming the original problem into

$$M^{-1}Ax \quad = \quad M^{-1}b,$$

which gives the same solution as $Ax = b$, but the new matrix $M^{-1}A$ is engineered to converge faster and be better conditioned. An M "close" to A is sought, as preconditioning by the exact inverse of A would have optimal qualities of convergence and condition. Since working with the inverse of A is computationally impractical, a sparse approximation of A is sought in defining M, for which $My = x$ type problems can be solved inexpensively.

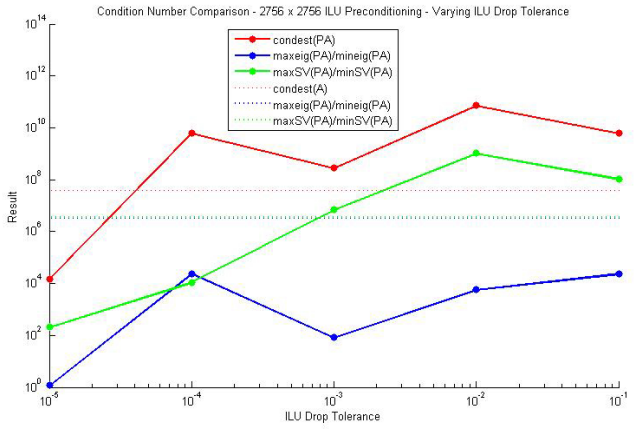
3 ILU Preconditioning

One method of direct solving is known as LU factorization, which involves factoring A, via Gaussian elimination, into the product of two triangular matrices LU, where L is lower triangular, and U is upper triangular. Thus the original $Ax = b$ problem is transformed into $(LU)x = b$, which can be solved via one step each of forward- and back-substitution. For use as a preconditioner, LU factorization is impractical and memory intensive, since the L and U matrices are in general much more dense than A. In order to cope with this, a variant of the method known as incomplete LU (ILU) was developed [5]. Designed to find approximate L and U factors that maintain some degree of sparsity, ILU has many variations and options, ranging from the "No-Fill" variation which mandates that L and U only have nonzero values in the same location as A, to ILUT (ILU with thresholding) in which small entries in L and U that are under a certain ratio (called the drop tolerance) with respect to the column and row norms are changed to zero in an attempt to increase sparsity, saving

computation steps and minimizing storage.

Preliminary research into ILUT showed that it may be promising in the case of edge FEM systems [3], particularly the "crout" variant of ILU available in MATLAB, which returns a unit upper triangular matrix. Preliminary results also showed that symmetric reverse Cuthill-McKee permutation of elements, which reorders the matrix so that elements are closer to the diagonal, is extremely effective in combination with ILUT techniques, since it leads to less fill-in, even under full LU factorization. So our group experimented with ILUT, exploring the variety of available options in MATLAB. We focused primarily on varying the drop tolerance parameter, which controls the amount of sparsity and the closeness to the full LU factorization, and also investigated pivoting and modified ILU options which attempt to add stability and preserve row or column sums. The effect these parameters had on condition numbers, as well as required memory storage, time, and convergence in GMRES were recorded for the 2756×2756 through 7868×7868 size matrices.

For the matrices studied, the introduction of dense ILU preconditioning (corresponding to a low drop tolerance and close approximation to the full LU factorization) greatly improves the condition number of the matrix, whereas the introduction of very sparse ILU preconditioning (corresponding to a high drop tolerance) adversely affects the condition number of the matrix. This trend is best captured graphically in the chart below, which shows condition numbers for the preconditioned system with various drop tolerance levels, for the representative 2756×2756 matrix. The *condest* approximations are shown, along with the ratio of the largest and smallest singular values of A, which constitutes the actual 2-norm condition number, and the ratio of the largest and smallest eigenvalues of A, which presents another rough approximation of the condition number.



As was shown in the preliminary report, ILU preconditioning can be used to arbitrarily lower the number of required GMRES iterations, so we sought to find an optimal drop tolerance which weighs the memory needs of ILU versus GMRES, and minimizes the amount of computation time. In order to do this,

our group researched the GMRES algorithm, with an eye towards storage requirements.

At each successive iteration k, GMRES approximates the solution x as a vector x_k in the orthogonalized Krylov subspace

$$\mathcal{K}_k \quad = \quad \text{span}\{b, Ab, A^2b, ..., A^{k-1}b\}.$$

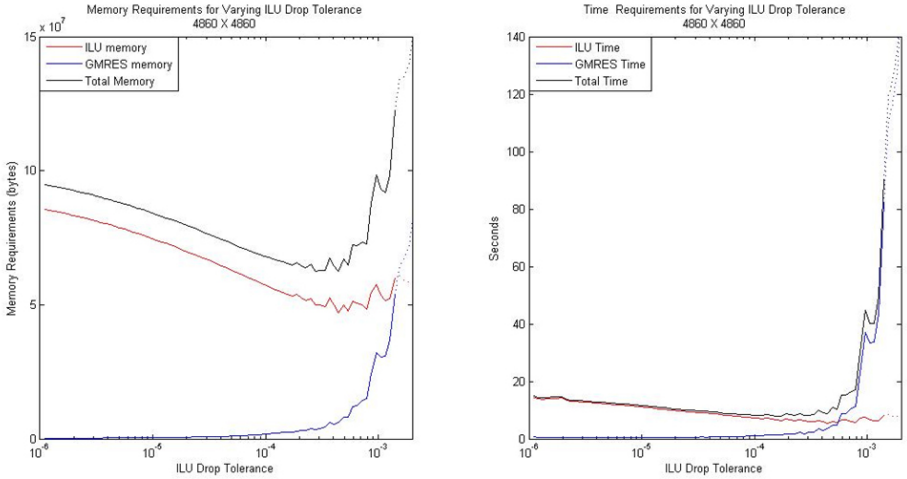
Unlike other Krylov-based methods, this entire basis must be saved in memory, so at the kth iteration, an $n \times k$ dense matrix must be stored. Additionally, an upper Hessenberg matrix (a by-product of the orthogonalization process) of size $k \times (k+1)$, which is used in solving the least squares problem

$$||r_k|| \quad = \quad \min_{x_k \epsilon \mathcal{K}_k} ||Ax_k - b||$$

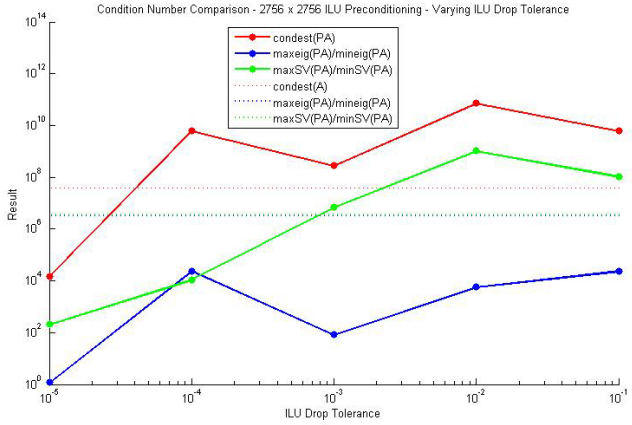
must also be stored. If GMRES is allowed to take (as in the un-preconditioned case) n iterations to solve, two dense matrices — one with n^2 elements and one with $\frac{1}{2}n^2$ elements – must exist in memory. This is not feasible for application.

So, we weigh these storage needs against those of ILUT. Complex valued elements in double precision in a dense matrix require 16 bytes each. If GMRES takes k iterations to converge (without restarting), the dense Krylov subspace matrix of dimension $n \times k$ and the dense upper Hessenberg $k \times (k+1)$ matrix must be stored, requiring $16 * [nk + \frac{1}{2}(k^2 + 3k)]$ bytes. Complex valued nonzero elements in MATLAB's A_{ij}, i, j sparse format require 20 bytes. A, L, and U are stored in this format, requiring $20 * [nnz(A) + nnz(L) + nnz(U)]$ bytes. The sum of these values gives a very rough idea of the memory required to solve via GMRES for given ILU settings on a single processor.

For the 2756×2756 through 7868×7868 size matrices tested, small drop tolerances were seen to effectively lower GMRES iterations to very small numbers. As the drop tolerance increased, up until around 10^{-3}, GMRES performance did not significantly worsen. When the drop tolerance grows past this point, GMRES performance rapidly degrades, requiring more iterations and subsequently more time and storage. Therefore, there exists an optimal ILUT drop tolerance which creates a preconditioner effective enough to minimize GMRES iterations, while maximizing sparsity. For the matrices we studied, the optimal drop tolerance fell between 10^{-3} and 10^{-4}. A representative graph of the 4860×4860 memory requriments is shown below, along with a graph of time requirements for each drop tolerance, which mirrors the same trend. Denser ILUT factors require more time to construct, and additional GMRES iterations require more time as well. There does not seem to be any significant time/memory trade off for the matrices tested, as the optimal memory drop tolerance corresponds nicely to minimal solve time.



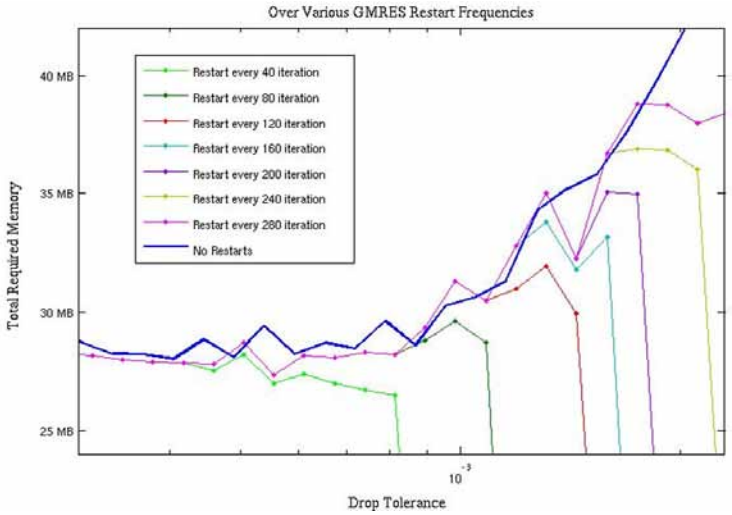
Trends established for smaller sized matrices (2000×2000 to 8000×8000) suggest that minimal ILUT memory requirements are roughly half of full LU (see below). This level of storage savings may not be enough when projected outwards for larger matrices. For instance, a sufficiently fine mesh for approaching Maxwell's equation would likely result in a $10^6 \times 10^6$ or larger matrix. Storing this matrix alone in sparse format would require around 20 GB. A reasonably effective ILU preconditioner which requires 100 to 200 GB would likely not be tenable on a single processor. So, the group sought ways to reduce memory demands.



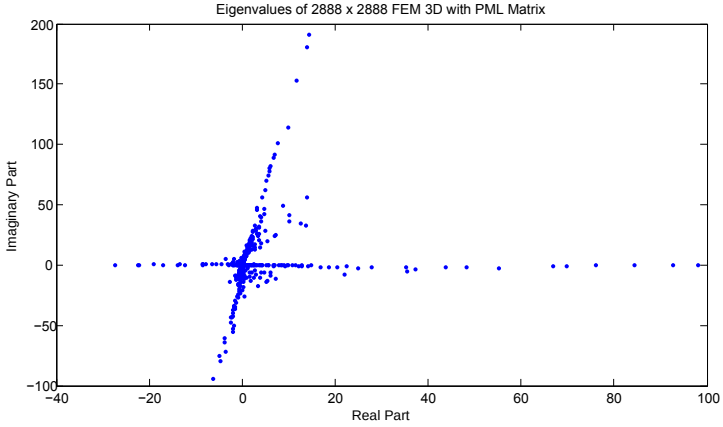
4 GMRES with Restarts

To lower memory requirements, we tested ILU preconditioning with restarted GMRES. After a set number of iterations, the Krylov subspace is cleared and the

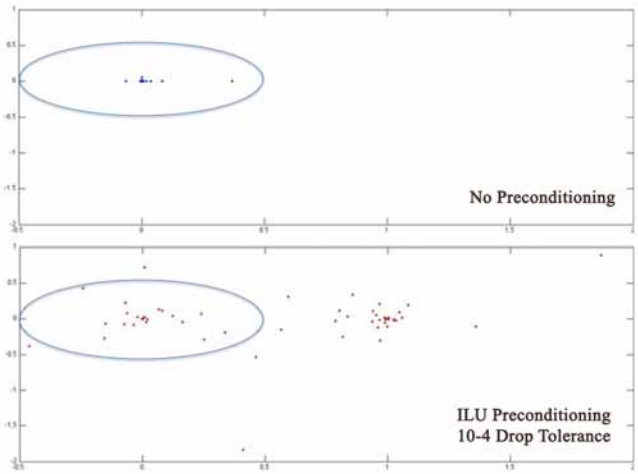
process builds a new subspace beginning with the last residual vector, repeating as many times as needed. This process requires only as much storage space as we allow, but the disadvantage to restarts is that more iterations are required for GMRES to converge to the solution, since there is no longer a guarantee that new vectors added to the subspace are orthogonal to all previous vectors, and (since our matrices are not symmetric positive definite) there is a chance it may never converge. We tested GMRES with restarts with our 2756×2756 matrix preconditioned with various ILUT drop tolerances. The goal is to save memory by operating with a sparse ILU preconditioner that would lead to many GMRES iterations in the non-restarted case, but cap memory demands by restarting as many times as necessary. As the following graph shows, restarting saves some memory for a small range of drop tolerances but quickly loses the ability to converge.



The rate of GMRES convergence is closely related to eigenvalue distribution. Studies show that GMRES convergence is slow if many eigenvalues of A are near zero or scattered elsewhere [7]. This accurately describes the eigenvalue distribution of the matrices under study and explains the slow convergence of the un-preconditioned system, as we see in this image of the eigenvalues of our 2888×2888 matrix.

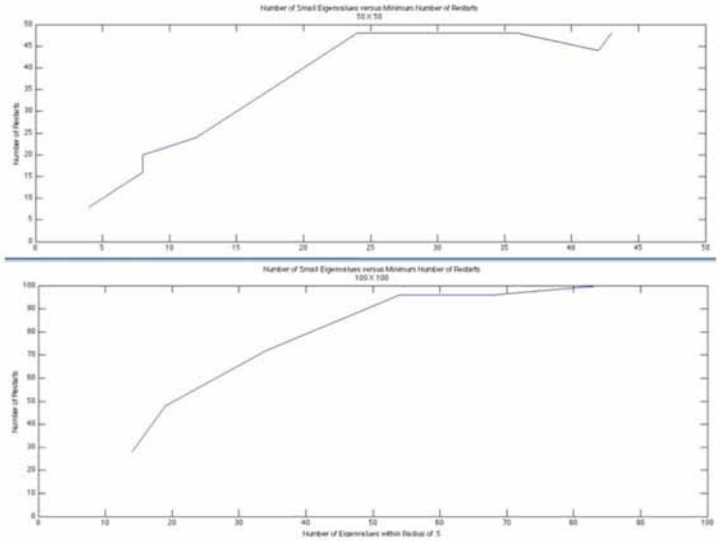


To better understand GMRES convergence with restarts, we examined the minimum number of iterations required before restarting for a matrix to converge compared to the number of eigenvalues that lie within some radius of zero. The next image shows the eigenvalues of a poorly conditioned 100×100 complex matrix within a circle of radius .5 before and after preconditioning the matrix.



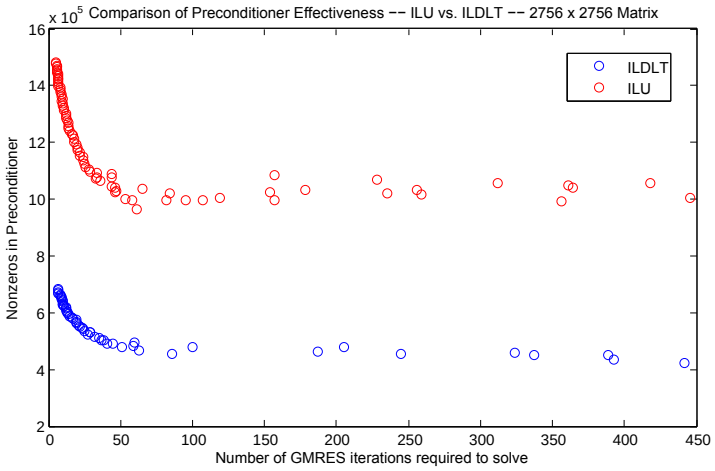
Testing on smaller matrices designed to mimic the eigenvalue structure of our larger matrices, we witnessed a general trend between the number of eigenvalues within a .5-radius circle (achieved via preconditioning with various drop tolerances) and the minimum restart number with which GMRES was able to converge to a solution. In the case of the small matrices tested, it appears that GMRES is not likely to converge with restarts if over half of the eigenvalues are "close" to zero (see below). Since GMRES is not guaranteed to converge with restarts, its application seems limited to a case-by-case basis. If operating at maximum memory capacity, restarts can be utilized as a memory cap. But for the purposes of optimizing all other aspects of solving our matrices, we avoided

that minimum-gain risk and proceeded without using restarts.



5 ILDLT Preconditioning

If an LU decomposition exists for a given matrix, then a unique LDU factorization does as well, where L and U are unit triangular matrices, and D is a diagonal matrix. If the matrix is symmetric, then $LDU = LDL^T$. In practical terms, this LDLT factorization requires storage of only one triangular matrix, equivalent to half of LU storage needs. MATLAB has a full LDLT factorization function, but it only works on Hermitian matrices, so we constructed our own MATLAB ILDLT function, using Yousef Saad's pseudocode algorithm [5] and adding a dropping step to maintain sparsity. For the 2756 $\times$ 2756 through 7868 $\times$ 7868 size matrices tested, ILDLT for a given drop tolerance was comparable to ILU in regards to GMRES performance, but required half as much storage (see below). However, the program is currently not as fully optimized and integrated as ILU in MATLAB, and still takes around 10 to 20 times as long to complete. Once optimized, ILDLT presents a more realizable option for large matrices than ILU, given the symmetric nature of our problem.



6 Block Jacobi Preconditioning

For larger matrices, where ILU preconditioning on a single processor is not feasible, we seek a preconditioner with better parallelization capability than ILU alone. Our group chose to research Block Jacobi preconditioning. The Block Jacobi preconditioner M consists of the union of j overlapping block submatrices A_1, A_2, ... A_j that exist along the diagonal of A (see below). We apply this matrix as a *right* preconditioner, so the condition and convergence properties depend upon AM^{-1} instead of A.

With right preconditioning, at the kth iteration, the Krylov subspace at the root of the GMRES method consists of

$$\mathcal{K}_k \quad = \quad \text{span}\{v, (AM^{-1})v, (AM^{-1})^2v, ..., (AM^{-1})^{k-1}v\},$$

where at each iteration, a new subspace vector v_{k+1} is added to the Krylov subspace. It is formed by calculating

$$AM^{-1}v_k \quad = \quad v_{k+1}.$$

We avoid explicitly calculating the inverse of M, as doing so would be time- and memory-intensive. Instead we construct the inverse implicitly by letting $M^{-1}v_k = z$, so that $Az = v_{k+1}$. Dropping the subscript for simplicity, we then have a new system to solve in order to compute the next vector in our subspace:

$$Mz \quad = \quad v.$$

We seek an approximation to z via Block Jacobi. If we make a guess at a solution z_k, we will be left with some residual vector r:

$$r \quad = \quad v - Mz_k.$$

So we correct our subsequent guess by some vector q:

$$z_{k+1} = z_k + q$$

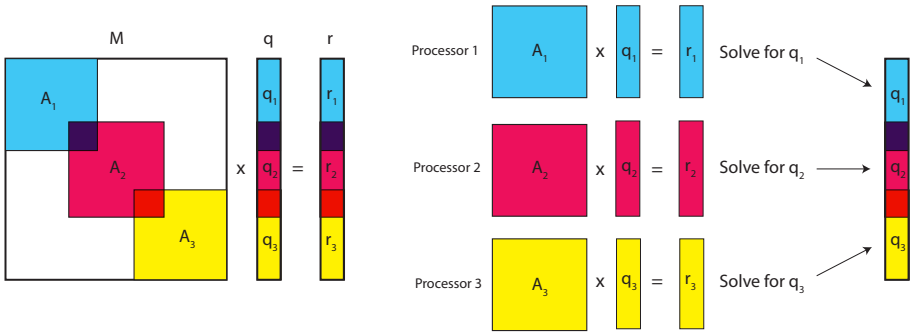
It follows that:

$$\begin{aligned} r &= v - M(z_{k+1} - q) \\ r &= v - Mz_{k+1} + Mq \end{aligned}$$

If $Mq = r$, then $Mz_{k+1} = v$, which gives our solution and the next entry in our Krylov subspace. So, we can rewrite our problem one more time as:

$$Mq = r$$

We construct an approximate q by solving the submatrix and subvector systems independently, either directly or via GMRES (see below). We then recombine, adding the resulting solutions, applying a weight factor $1/k$ for each element of the vector where k submatrix solutions overlap. Since the resulting q would only be exact if the submatrices were disjoint, we only seek an approximation for q. We use q as a correction vector to find a new approximate z and resulting r, and repeat the process as needed.[5]



We felt Block Jacobi represented a good fit for our problem. With the reordering scheme implemented in the ILU section, we were able to move the matrix elements very close to the diagonal, so that we could include all of the elements of A within our preconditioner without resorting to overly large submatrices or excess overlap. Additionally, since all the submatrices can be solved independently, this method presents a good opportunity for parallelization, as each submatrix system can be stored and solved on separate processors.

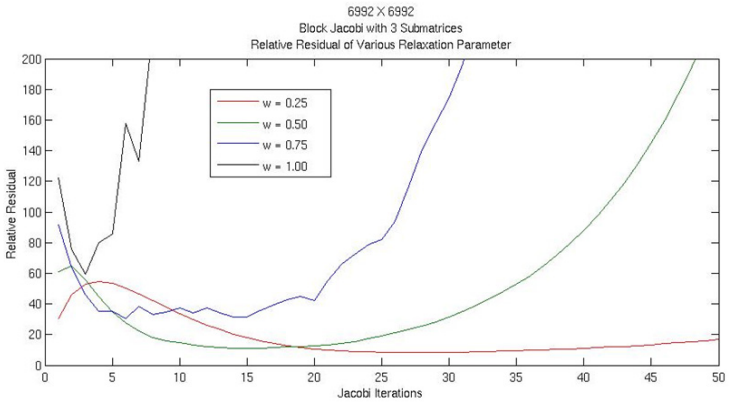
We wrote a script in MATLAB, which, given a certain number of submatrices in the desired splitting, chooses an optimal size for the submatrices relative to the bandwidth of the reordered matrix, so that all the elements of A are guaranteed to be included in M, with as little overlap as possible. These submatrices are then integrated into the Block Jacobi MATLAB script we created, following the methods described by Saad, which behaves as described above [5].

Preliminary testing results of the performance of the Block Jacobi preconditioner for larger matrices are shown below. The number of GMRES iterations

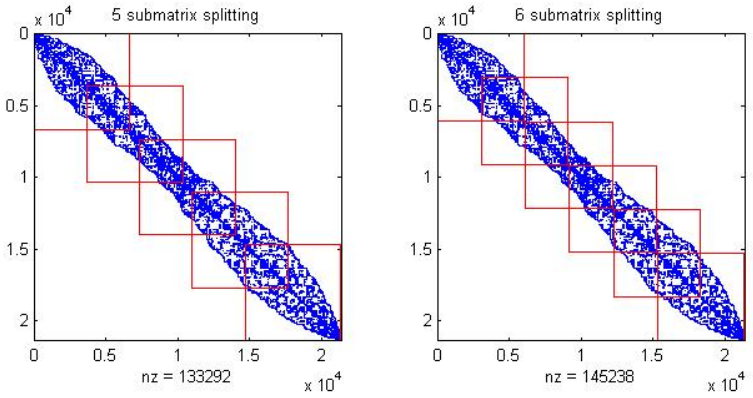
required for convergence is shown, for corresponding numbers of submatrices in the splitting. Performance tends to degrade somewhat for greater numbers of submatrices and hence more overlapping, but in general, since the number of GMRES iterations is around 1% of the matrix dimension, results are promising.

Number of Submatrices	3	4	5	6	7
21268 × 21268 Matrix	64	153	197	196	253
28424 × 28424 Matrix	135	210	275		
33668 × 33668 Matrix	131	284			

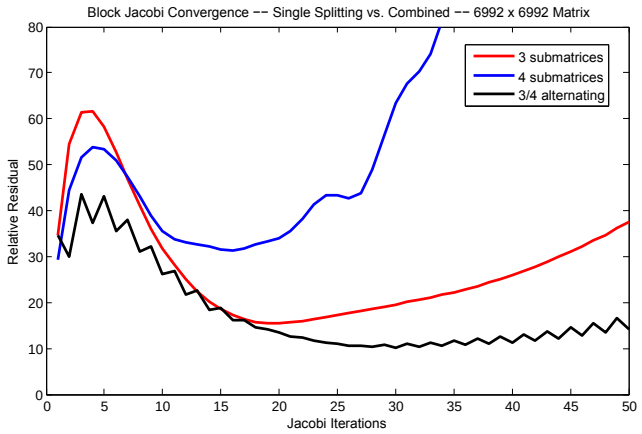
There are numerous parameters included within this code that can be optimized, and the group very briefly experimented with altering each of these. A relaxation parameter w was incorporated into the code, which allowed that the correction vector q could be weighted in order to smooth out convergence, and delay divergence during Jacobi iterations. Typical results of variations in this parameter are shown below. In general, lower q values tended to allow the Jacobi process to converge to a smaller residual, over a greater number of iterations, while higher q values caused quick, shallower convergence. So, the cost in time of performing more Jacobi iterations for low w must be weighed against the less effective preconditioning of high w, which results in more GMRES iterations and memory usage.



Additionally, experimentation showed that when we combine different submatrix splittings, alternating between them at each Jacobi step, we can increase convergence. We have not seen suggestion of this behavior in Block Jacobi literature, so we lack a theoretical basis for why this occurs, but the general idea is that inherent errors that result in solving via a certain splitting may be mitigated or smoothed out by solving with a different splitting, with a different set of inherent errors. As an example, the splitting of the 21368 × 21368 into five and six submatrices (shown below), alternating solving with each splitting, resulted in GMRES convergence in 85 iterations, whereas alone, each method required 197 and 196 iterations, respectively. Much experimentation remains to be done on optimizing the combination of splittings, perhaps combining three or more, or performing iterations in 'V' or 'W' cycles, à la multigrid techniques.



In general, for the matrix sizes tested, when splitting methods are alternated (only neighboring splittings were tested as yet, such as 5/6, 6/7, and so forth) up through 33668 $\times$ 33668, the residual at each Jacobi step is as small as, or smaller than either splitting method alone. A representative plot, for the 6992 $\times$ 6992 matrix, is shown below.



The results shown above are for a static number of Jacobi iterations. The residual tends to increase after a minimum has been reached, so additional Jacobi steps after that point are extraneous. Seeing this behavior, we added to our Jacobi script an option to proceed with more iterations if the process is still converging, or to cut the process short if the residual has increased for a certain number of iterations. All of the residual plots seemed to behave in the same manner, so once divergence (after the initial steps) starts occurring, there is little danger of losing later convergence. As is seen in the chart below, by dy-

namically changing the number of Jacobi iterations, we can decrease the solving time substantially, spending a few extra GMRES steps while greatly reducing the number of inner Jacobi iterations.

21368 × 21368 Matrix

Preconditioner Type	GMRES Iter	Time
20 Iterations - Static, w=0.5	85	3.02 hrs
Varying # of Iterations, w=0.5	87	0.87 hrs
One step of each splitting, w=1	77	0.26 hrs

The third entry in the chart above points to another, more straightforward option. We can choose to only perform one iteration on each splitting, taking that result as "close enough" and moving onto the next GMRES iteration. By combining this with an increased relaxation parameter to speed convergence during those steps, we can save even more time, and in the case above, this speedy convergence actually performed better than the smoother, lower w cases above. In subsequent testing, this was our preferred option, as it seemed to perform much faster, and the increase in GMRES iterations, if any, was not substantial enough to cause memory concerns.

As it stands, the MATLAB Block Jacobi script we wrote merely imitates parallelization, as all of the submatrix solving steps are currently done sequentially. Our group wanted to verify the effectiveness of the procedure before delving into parallelization coding. So, the GMRES iteration data would still apply in the parallel case, but the times recorded would be substantially lower. For instance, the 33668×33668 matrix (solved via GMRES/Block Jacobi in 153 outer iterations, with the alternating 6/7 submatrix splitting) finished in 1.09 hours. Since the average submatrix solve time for 6-splitting was 1.72 seconds, and 1.69 seconds for the 7-splitting, if these solves were performed in parallel, the total process would be finished in 16.8 minutes (plus some additional time for passing data between processors). As the number of submatrices increases, the time saved by parallelizing increases accordingly.

It is rather difficult to extrapolate to the effectiveness of Block Jacobi on $10^6 \times 10^6$ size matrices, which might require 40+ submatrices. Experiments on smaller matrices point to it being a workable method, which is perhaps more than we can say for ILU, but we have no way of knowing whether convergence would take hours or weeks. Currently we lack the facilities for parallelizing on that scale in MATLAB, so a program would most likely have to be implemented in C, perhaps in conjunction with GMM, PETSc, or other linear algebra packages. Given more time, this would be the next step in testing Block Jacobi preconditioning.

7 Conclusion

Our group delineated the limits of MATLAB's ILU preconditioner as applied to the edge FEM matrices under study. For appropriately small matrices, ILU provides an effective option when direct solving strains memory limitations, and if the preconditioner is near enough to full LU, the condition number is accordingly lowered, placing a better bound on the amount of relative error introduced when solving.

In order to widen the application of the ILU-style preconditioner, our group created a MATLAB ILDLT script which demonstrated similar effectiveness in aiding convergence for symmetric matrices, at a cost of half the memory of ILU.

Lastly, our group created a parallelizable Block Jacobi script in MATLAB, for the purpose of solving systems larger than ILU/ILDLT could presently handle on a single processor. Reasonable effectiveness was demonstrated on matrices up through 33668×33668, and work remains to be done to determine its usefulness on very large systems.

8 Future Work

Should our group, or anyone else, continue work on this project, parallelization and implementation of the Block Jacobi script may be a worthwhile route. In addition, there are other, more sophisticated preconditioning methods, such as Helmholtz decomposition or multigrid [6] [4] [2], which remain to be tried. One burgeoning option, algebraic multigrid, is also parallelizable and has the advantage of working in a "black box" fashion, like ILU and Block Jacobi, as it requires little manipulation of the FEM code or understanding of the physics problem itself. Our group likely would have pursued this option next, but it required more in-depth understanding than time availed us during this month-long project.

9 Acknowledgments

Our group would like to thank the University of Arizona and VIGRE for providing and funding this valuable learning experience, as well as Program Director Moysey Brio, and Project Directors Paul Dostert and Jinjie Liu for lending their time and knowledge, helping us navigate this difficult problem and assisting in the technical aspects.

References

[1] Krishna Mohan Gundu. Solving Sparse Linear Systems. ACMS Group Meeting, October 2007. Presentation.

[2] Ralf Hiptmair and Jinchao Xu. Auxiliary Space Preconditioning for Edge Elements. *IEEE Transactions on Magnetics*, 44(6), June 2008.

[3] Robert Muth. A Preliminary Study of Preconditioners for Edge FEM Applied to Frequency Domain in Maxwell's Equations. Preliminary Report for UA VIGRE Summer Program, June 2009. Report.

[4] Ronan Perrussel, Laurent Nicolas, and Franois Musy. An Efficient Preconditioner for Linear Systems Issued from the Finite-Element Method for Scattering Problems. *IEEE Transactions on Magnetics*, 40(2), March 2004.

[5] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[6] Olaf Schenk, Michael Hagemann, and Stefan Rollin. Recent Advances in Sparse Linear Solver Technology for Semiconductor Device Simulation Matrices. *Simulation of Semiconductor Processes and Devices*, 2003.

[7] Josef Sifuentes. Preconditioning the Integral Formulation of the Helmholtz Equation via Deflation. Master's thesis, Rice University, Houston TX, 2006.

[8] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.