Solving a Two-Dimensional Elliptic Model Problem with the Conjugate Gradient Method Using Matrix-Free SSOR Preconditioning in Matlab

Amanda K. Gassman and Matthias K. Gobbert

Department of Mathematics and Statistics, University of Maryland, Baltimore County

{amandag2,gobbert}@umbc.edu

Abstract. The existing Preconditioned Conjugate Gradient method in Matlab can be optimized in terms of wall clock time and, more importantly, required storage space. The developed optimized algorithm was tested repeatedly on a two dimensional Poisson problem to ensure that it produced the same numerical solution as the original Matlab function. The algorithm was optimized in several stages: first a function was created that reused some vectors in the algorithm, next we developed a matrix-free method of computing matrix-vector multiplications with the system matrix, then a matrix-free method that implements preconditioning on the system matrix was derived, and finally all memory-saving techniques were combined in order to create a matrix-free Preconditioned Conjugate Gradient method. This superior algorithm computes the same numerical solution to our problem as Matlab's original method, but requires less memory and less time. Ultimately, the goal of optimizing the algorithm was achieved and convergence results are presented to confirm the accuracy of the new method and demonstrate its superiority.

1 Introduction

A classical model problem is the Poisson equation with homogeneous Dirichlet boundary conditions, which is given by

$$-\Delta u = f(x,y) \quad \text{in } \Omega, \tag{1.1}$$

$$u = 0 \quad \text{on } \partial\Omega. \tag{1.2}$$

This problem is considered on the unit square $\Omega \subset \mathbb{R}^2$. Using the finite difference method with a standard five point stencil yields a highly structured system matrix that is symmetric positive definite. This paper focuses on deriving a matrix-free optimal Preconditioned Conjugate Gradient method and applying it to the model problem over the stated domain. We first present in Table 1 the results of the various optimization techniques for Matlab's Conjugate Gradient method (CG) and the optimal Conjugate Gradient method (CGopt) on our model problem for a system size of $N = 2048$ with a tolerance of 10^{-6}. The tolerance determines when the method has found a solution which when multiplied by the system matrix A yields a results within 10^{-6} of the right-hand side vector b. Table 2 presents results of the various optimization techniques for Matlab's Preconditioned Conjugate Gradient method (PCG) and the optimal Preconditioned Conjugate Gradient method (PCGopt) on our model problem using the same system size and initial parameters.

Table 1 displays the convergence results of implementing Matlab's CG method and the optimal CG method with A as a matrix and then matrix-free. In all cases, the error ratio is approximately 4 (which agrees with the theory), the relative residual is smaller than the tolerance, the conjugate gradient method required 3192 iterations to converge, the time increases when A is implemented matrix-free, and the observed memory usage decreases when A is implemented matrix-free. Each row of results in Table 1 is exactly the $N = 2048$ row of results in Tables 3 and 4 which are presented later in Section 3.

Table 2 presents the convergence data associated with the PCG method: the first portion show the results of using Matlab's PCG method while the second portion are the results for the possible combinations of "matrix", "matrix-free", and "transp" for PCGopt. Thus each row of results presented in Table 2 is exactly the $N = 2048$ row of results in Tables 5, 6, 7, 8, and 9 making Table 2 truly a summary table of the results presented later in Sections 3 and 4 where detailed studies and explanations of the various memory-saving techniques can be found.

Method	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's CG:						
$\cdot\, A$ =matrix	7.8019e-07	4.0075	9.8757e-07	3192	2,195.67	1,104
$\cdot\, A$ =matrix-free	7.8019e-07	4.0075	9.8769e-07	3192	3,226.87	808
CGopt:						
$\cdot\, A$ =matrix	7.8019e-07	4.0075	9.8757e-07	3192	1,632.51	1,104
$\cdot\, A$ =matrix-free	7.8019e-07	4.0075	9.8769e-07	3192	2,134.06	740

Table 1: Convergence Results for CG for $N = 2048$

Method	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's PCG:						
$\cdot\, A, M_1, M_2$ =matrix	7.8394e-07	3.9953	9.0626e-07	176	185.03	1,564
$\cdot\, M_1$ =matrix-free	7.8394e-07	3.9953	9.0626e-07	176	227.24	1,298
$\cdot\, M_1, M_2$ =matrix-free	7.8394e-07	3.9953	9.0626e-07	176	259.85	1,122
$\cdot\, A, M_1, M_2$ =matrix-free	7.8394e-07	3.9953	9.0626e-07	176	313.59	930
PCGopt:						
$\cdot\, A, M_1, M_2$=matrix	7.8394e-07	3.9953	9.0626e-07	176	154.64	1,432
$\cdot\, M_2$=transp	7.8394e-07	3.9953	9.0626e-07	176	235.45	1,356
$\cdot\, M_1$=matrix-free	7.8394e-07	3.9953	9.0626e-07	176	194.31	1,276
$\cdot\, M_1$,M_2=matrix-free	7.8394e-07	3.9953	9.0626e-07	176	230.08	1,118
$\cdot\, A, M_1, M_2$=matrix-free	7.8394e-07	3.9953	9.0626e-07	176	254.50	910

Table 2: Convergence Results for PCG for $N = 2048$

An initial glance at Tables 1 and 2 reveals that as the memory-savings techniques get more sophisticated the amount of memory used decreases; this meets with expectations since the techniques were specifically designed to save memory. Interestingly, comparison within Tables 1 and 2 shows that the wall clock time increases when the system matrix A is implemented matrix-free.

All computations associated with this project were performed in the spring of 2009 in Matlab version R2008b on a quad core processor machine with four 2.83 GHz and 8 GB of memory. The memory observation was executed in Linux using RedHat EL5 using the Linux command "top", which shows in real time the CPU utilization details (i.e., what applications are using how much of the available resources). The investigation focused on the "VIRT" column (as it shows the memory used by a certain application) and recorded the highest value reached during each of the methods tested for each dimension N.

Possessing this knowledge is crucial when analyzing the results especially as it allows for a determination as why to a numerical solution could not be found for a larger dimension. In all cases, the resounding answer is that the system ran out of memory. Accordingly we seek to minimize the required storage of the Conjugate Gradient method without impacting its manner of functioning or final solution. Overall, this study helped us to understand how memory quickly becomes a limiting factor in this type of data processing and which sort of techniques are successful in limiting storage requirements.

This paper is organized as follows. Section 2 introduces the specific problem and its discretization by the finite difference method which results in particular system matrices. Section 3 explains the optimal Conjugate Gradient algorithm and why the system matrix A need not be formally stored in order to be used. Thus we explore all possible combinations of CG and CGopt where A is stored as a matrix and implemented matrix-free and present the resultant convergence tables. Section 4 describes the idea of preconditioning, defines (in detail) the algorithms which implement matrix-free Symmetric Successive Overrelaxation (SSOR(ω)) as the preconditioner, and presents convergence tables for the various combinations of PCG and PCGopt using preconditioning matrices and matrix-free techniques.

2 The Model Problem and Linear System Setup

The classical model problem is the Poisson equation considered over an open, bounded, simply connected, convex region in the two-dimensional plane $\Omega = (0,1) \times (0,1)$ with homogeneous Dirichlet boundary and where the function $f(x,y)$ is

$$f(x,y) = -2\pi^2 \cos(2\pi x)\sin^2(\pi y) - 2\pi^2 \sin^2(\pi x)\cos(2\pi y). \tag{2.1}$$

The solution to the problem, which was specifically designed to have a closed-form solution, takes the form

$$u^{true}(x,y) = \sin^2(\pi x)\sin^2(\pi y). \tag{2.2}$$

The specific problem can be stated as

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x,y) \qquad \text{for} \quad 0 < x < 1, \quad 0 < y < 1, \tag{2.3}$$

$$u(0,y) = u(x,0) = u(1,y) = u(x,1) = 0 \quad \text{for} \quad 0 \le x \le 1, \quad 0 \le y \le 1, \tag{2.4}$$

with the function $f(x,y)$ from (2.1). On this domain, a grid is defined such that

$$\Omega_h = \{(x_i, y_j) : x_i = ih, \quad i = 0, \ldots, N+1, \quad y_j = jh, \quad j = 0, \ldots, N+1\},$$

where $h = \frac{1}{N+1} > 0$. We will let $N = 2^\nu$ where $\nu = 1, 2, \ldots, 13$.

The previously defined grid of x and y values enabled an application of a second-order finite difference approximation at all interior points of Ω_h to the x-derivative and the y-derivative, respectively, and thus we obtain

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) \approx \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2}, \tag{2.5}$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) \approx \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2}, \tag{2.6}$$

where $i = 1, \ldots, N, \quad j = 1, \ldots, N$. We can approximate $-\Delta u = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2}$ in equation (2.3) with the above formulas to yield

$$-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i,j+1} - u_{i+1,j} = h^2 f_{i,j} \quad i,j = 1, \ldots, N. \tag{2.7}$$

By using the fact that $u = 0$ on the boundary, a linear system of $n = N^2$ equations for the interior mesh points can be created where $Au = b$. If using a natural ordering of $u_k \equiv u_{i,j}$, $k = i + N(j-1)$, then A is of the form

$$A = \begin{bmatrix} S & T & & & \\ T & S & T & & \\ & \ddots & \ddots & \ddots & \\ & & T & S & T \\ & & & T & S \end{bmatrix}, \quad S = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix}, \quad T = \begin{bmatrix} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & -1 & \\ & & & & -1 \end{bmatrix},$$

where $A \in \mathbb{R}^{NxN}$ is a block-tridiagonal matrix consisting of $N \times N$ blocks of $N \times N$ matrices. The setup of the matrix A for any dimension N implements the Kronecker tensor product of an identify matrix and tridiagonal matrix based on the finite difference discretization [1]. The right-hand side vector b depends on the function $f(x,y)$ where each entry $b_k = h^2 f_{i,j}$. As h is halfed, the error is improved by a factor of four. All matrices (when stored) are stored in sparse storage, which conserves valuable memory and allows better management of larger dimension matrices.

3 Conjugate Gradient Method

3.1 Optimal Conjugate Gradient Method

A part of the focus of the project was to study an optimal Conjugate Gradient algorithm (referred to in the following as CGopt) with capabilities identical to the built-in Matlab Conjugate Gradient `pcg` function (referred to as CG). CGopt smartly reuses four vectors and therefore requires only six as compared to ten in Matlab's original algorithm. Additionally, CGopt requires only one matrix-vector product as compared to two in the standard Conjugate Gradient algorithm. For a comparison between the two Conjugate Gradient methods, we present full convergence results in Table 3 for Matlab's CG and CGopt where A is stored as a matrix.

For both CG and CGopt, the error decreases and the iteration count, time, and observed memory usage increase as the system size increases. For any case that converged in less than several seconds, the memory usage could not be observed and thus only a dash is shown. By comparing the results for CG with CGopt, we can both ensure accuracy of the optimized method and demonstrate improvement over the standard one. It should be noted that as CGopt is nearly identical to Matlab's CG, the convergence results are also nearly identical. In Table 3, a comparison of the error, error ratio, relative residual, and iteration count for each dimension N between the two methods reveals that both have the same convergence results, but that CGopt simply improves the time required. Finally, both methods ran out of memory for system sizes larger than $N = 2048$.

N	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's CG:						
4	1.1673e-01	N/A	4.0646e-16	3	0.00	-
8	3.9152e-02	2.9813	1.2237e-15	10	0.00	-
16	1.1267e-02	3.4748	6.6499e-07	24	0.00	-
32	3.0128e-03	3.7399	5.5637e-07	48	0.01	-
64	7.7811e-04	3.8719	7.0189e-07	96	0.06	-
128	1.9765e-04	3.9368	9.3340e-07	192	0.35	-
256	4.9797e-05	3.9690	8.9244e-07	387	3.53	427
512	1.2494e-05	3.9857	9.0693e-07	783	31.65	459
1024	3.1266e-06	3.9961	9.3989e-07	1581	273.93	579
2048	7.8019e-07	4.0075	9.8757e-07	3192	2,195.67	1,104
4096			Out of Memory			
8192			Out of Memory			
CGopt:						
4	1.1673e-01	N/A	1.5020e-16	3	0.00	-
8	3.9152e-02	2.9813	3.1421e-16	10	0.00	-
16	1.1267e-02	3.4748	6.6499e-07	24	0.00	-
32	3.0128e-03	3.7399	5.5637e-07	48	0.01	-
64	7.7811e-04	3.8719	7.0189e-07	96	0.05	-
128	1.9765e-04	3.9368	9.3340e-07	192	0.28	-
256	4.9797e-05	3.9690	8.9244e-07	387	2.65	426
512	1.2494e-05	3.9857	9.0693e-07	783	24.46	457
1024	3.1266e-06	3.9961	9.3989e-07	1581	203.18	574
2048	7.8019e-07	4.0075	9.8757e-07	3192	1,632.51	1,104
4096			Out of Memory			
8192			Out of Memory			

Table 3: Convergence Results for CG (A = matrix)

3.2 Matrix-Free A

It is important to note that the CG method requires only matrix-vector multiplications with the system matrix A and not the matrix itself [2]. Therefore when the CG method needs to do the multiplication $v = Au$, we instead call our matrix-free function with the vector u as input and receive as output the resultant vector v without ever creating or storing the system matrix A. Knowledge of the structure of A allows us to create a function that performs matrix-free matrix-vector products between A and a given vector. By programming this directly, there is no longer a need to store A and thus we save valuable memory.

Based on (2.6), we construct an algorithm that takes the vector u as input. This vector is immediately reshaped into an $N \times N$ matrix U using column-wise ordering as shown in the $N = 4$ example below. The algorithm then creates a matrix V such that $V = 4U$, which is derived from the entries along the main diagonal of A. Next in four deliberate steps we alter: all columns except the first, all rows except the first, all rows except the last, and all columns except the last. The following example for a dimension of $N = 4$ makes the results of each step clearer.

<u>EXAMPLE:</u> Matrix-Free Matrix-Vector Multiplication for $N = 4$

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = Au$$

$$u = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \rightarrow U = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \rightarrow V = 4U = \begin{bmatrix} 4a & 4c \\ 4b & 4d \end{bmatrix}$$

$$V = \begin{bmatrix} 4a & 4c-a \\ 4b & 4d-b \end{bmatrix} \rightarrow \begin{bmatrix} 4a & 4c-a \\ 4b-c & 4d-b-c \end{bmatrix} \rightarrow \begin{bmatrix} 4a-b & 4c-a-d \\ 4b-a & 4d-b-c \end{bmatrix} \rightarrow \begin{bmatrix} 4a-b-c & 4c-a-d \\ 4b-a-d & 4d-b-c \end{bmatrix}$$

$$V = \begin{bmatrix} 4a-b-c & 4c-a-d \\ 4b-a-d & 4d-b-c \end{bmatrix} \rightarrow v = \begin{bmatrix} 4a-b-c \\ 4b-a-d \\ 4c-a-d \\ 4d-b-c \end{bmatrix}$$

Table 4 presents the convergence results found by running Matlab's CG and CGopt on our problem using matrix-free A implementation. In both cases, the error decreases and the iteration count, time, and observed memory usage increase as the system size increases. Before we can boast the memory savings, we must first confirm that the matrix-free A algorithm indeed solves the problem correctly. As in Table 3, a comparison of the error, $\|e\|_\infty$, error ratio, relative residual, $\|r\|_2/\|b\|_2$, and iteration count for each dimension N between the two methods reveals that they have the same convergence results thus confirming the accuracy of the matrix-free algorithm. However unlike Table 3, CGopt now improves on CG in terms of both the time required and memory usage observed. Also, both methods converge for system size $N = 4096$, but run out of memory for $N = 8192$.

By eliminating the need to store matrix A, we expect significant memory savings and this confirmed by a comparison of the observed memory usage for corresponding cases of N between Table 3 and Table 4. The only difference between the tables is the results in Table 3 originate from the storage of A as a matrix and the results in Table 4 from the implementation of matrix-free A. A comparison of CG for each dimension N in Table 3 to the same dimension N in Table 4 shows that by implementing matrix-free A the observed memory usage decreases. This confirms that not storing A as a matrix, but instead implementing the matrix-vector multiplications with a matrix-free function, conserves memory. However, it should be noted that the time required increases by approximately 50% when A is implemented matrix-free. A comparison of CGopt for each dimension N in Table 3 to the same dimension N in Table 4 also shows that the observed memory usage decreases. Again, the time required increases, but now by approximately 33% for each dimension N.

There is very little difference in the performances of Matlab's CG and CGopt where A is stored as a matrix as seen in Table 3. However, the complete convergence results for Matlab's CG and CGopt with matrix-free A displayed in Table 4 are useful because they demonstrates the superiority of the CGopt over Matlab's CG in terms of time required and observed memory usage. For all dimensions, CGopt requires less memory and less time. While the results in Table 4 reveal that CG and CGopt with matrix-free A successfully solve the problem for $N = 4096$, the wall clock time is now considerable. In fact, the time required by Matlab's CG with matrix-free A is significantly more than for CGopt for the same system size N and thus this motivates the further improvement of the optimal method using preconditioning in order to lessen total wall clock time.

N	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's CG:						
4	1.1673e-01	N/A	6.8238e-16	3	0.03	-
8	3.9152e-02	2.9813	1.8080e-15	10	0.01	-
16	1.1267e-02	3.4748	6.6499e-07	24	0.01	-
32	3.0128e-03	3.7399	5.5637e-07	48	0.03	-
64	7.7811e-04	3.8719	7.0189e-07	96	0.08	-
128	1.9765e-04	3.9368	9.3340e-07	192	0.42	-
256	4.9797e-05	3.9690	8.9244e-07	387	4.13	422
512	1.2494e-05	3.9857	9.0792e-07	783	46.25	445
1024	3.1266e-06	3.9961	9.3990e-07	1581	404.27	538
2048	7.8019e-07	4.0075	9.8769e-07	3192	3,226.87	808
4096	1.9366e-07	4.0287	9.8366e-07	6452	25,887.31	1,955
8192			Out of Memory			
CGopt:						
4	1.1673e-01	N/A	1.6953e-16	3	0.00	-
8	3.9152e-02	2.9813	3.8591e-16	10	0.00	-
16	1.1267e-02	3.4748	6.6499e-07	24	0.01	-
32	3.0128e-03	3.7399	5.5637e-07	48	0.01	-
64	7.7811e-04	3.8719	7.0189e-07	96	0.05	-
128	1.9765e-04	3.9368	9.3340e-07	192	0.29	-
256	4.9797e-05	3.9690	8.9244e-07	387	2.89	423
512	1.2494e-05	3.9857	9.0792e-07	783	31.45	441
1024	3.1266e-06	3.9961	9.3990e-07	1581	266.67	523
2048	7.8019e-07	4.0075	9.8769e-07	3192	2,134.06	740
4096	1.9366e-07	4.0287	9.8364e-07	6452	17,169.03	1,828
8192			Out of Memory			

Table 4: Convergence Results for CG (A = matrix-free)

4 Preconditioned Conjugate Gradient Method

In this project we use the classic iterative method Symmetric Successive Overrelaxation (SSOR(ω_{opt})) as the preconditioner as it has been demonstrated that the Conjugate Gradient method when coupled with SSOR(ω_opt) yields faster convergence than with unpreconditioned CG. The splitting matrix M for SSOR(ω) is $M = \frac{\omega}{2-\omega}(\frac{1}{\omega}D - E)D^{-1}(\frac{1}{\omega}D - F)$. Here D is a diagonal matrix of the diagonal entries of the matrix A, $-E$ is the strictly lower triangular portion of matrix A, and $-F$ is the strictly upper triangular portion of matrix A such that $A = D - E - F$. In fact, this splitting matrix can be written as a product of a lower triangular matrix M_1 and an upper triangular matrix M_2 where $M_1 = \sqrt{\frac{\omega}{2-\omega}}(\frac{1}{\omega}D - E)D^{-1/2}$ and $M_2 = \sqrt{\frac{\omega}{2-\omega}}D^{-1/2}(\frac{1}{\omega}D - F)$. SSOR($\omega_opt$) preconditioning is implemented by adding linear solves with M_1 and M_2 which do not require many resources as these matrices are triangular. The speed of convergence depends largely on the relaxation parameter, ω. The optimal value, $\omega = \frac{2}{1+\sin \pi h}$, derived in the paper by Yang and Gobbert [3], is optimal and used here as it guarantees the best performance of the SSOR(ω) method in term of the mesh spacing h.

By using the optimal Conjugate Gradient code with our preconditioner we derive the optimal Preconditioned Conjugate Gradient method (PCGopt), whose capabilities are equivalent to the built-in Matlab PCG function. Both PCG and PCGopt can handle matrix-free implementation of the system matrix A (referred to as A=matrix-free) and matrix-free implementation of the preconditioning matrices M_1 and M_2 (M_1=matrix-free and M_2=matrix-free, respectively). Only PCGopt can handle implementation of the preconditioning matrix M_2 as the transpose of matrix M_1 as it is specifically designed to recognize the flag $M_2 =$"transp".

Table 5 presents full convergence results for Matlab's Preconditioned Conjugate Gradient method (PCG) and for the PCGopt method on our problem where all matrices are stored. In both cases, the error decreases and the iteration count, time, and observed memory usage increase as the system size increases. A comparison of the error, $\|e\|_\infty$, error ratio, relative residual, $\|r\|_2/\|b\|_2$, and iteration count for each dimension N between the two methods reveals that they have the same convergence results. However, PCGopt improves on PCG in terms of both the time required and memory usage observed, but the memory savings is not significant in this case. Finally, both methods converged for system size $N = 2048$, but run out of memory for $N = 4096$. The other various memory savings techniques are described in further detail in the following subsections.

4.1 Preconditioning with Transpose M_2

As widely demonstrated, preconditioning assists in solving problems more efficiently; it reduces the condition number $\kappa(A)$. Therefore, we next turn our attention toward improving the manner in which the system matrix A is preconditioned. Because our system is symmetric, our selected preconditioner Symmetric Successive Overrelaxation (SSOR(ω)) can be factored using Cholesky into a lower triangular factor M_1 and upper triangular factor M_2 where $M = M_1M_2$ and in fact $M_2 = {M_1}^T$. Matlab's PCG function does not take advantage of this fact; our PCGopt utilizes a flag to note when M2 = "transp", which allows the use of $M_2 = M_1^T$ in this linear solve involving M_2 instead of the storage of M_2. Rather than solving $M_2z = q$ using the backslash operator directly, we substituted M_1^T for M_2.

$$\begin{aligned} q &= M_2 z \\ q &= M_1^T z \\ q^T &= z^T M_1 \\ z^T &= q^T M_1^{-1} \\ z &= (q^T M_1^{-1})^T \end{aligned}$$

In Matlab syntax, we will solve $z = (q^T/M_1)^T$.

Complete convergence results for PCGopt with $M_2 =$ "transp" are displayed in Table 6. A comparison of this table with the PCGopt results in Table 5 reveals that the technique of substituting M_1^T for M_2 yields the same convergence results and does not change the ultimate outcome. PCGopt with $M_2 =$"transp" is

N	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's PCG:						
4	1.1673e-01	N/A	1.1573e-08	7	0.00	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.00	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.01	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.01	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.04	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.16	-
256	4.9811e-05	3.9683	9.2508e-07	57	0.93	-
512	1.2502e-05	3.9842	9.0263e-07	83	5.71	482
1024	3.1321e-06	3.9916	8.9577e-07	121	33.69	699
2048	7.8394e-07	3.9953	9.0626e-07	176	185.03	1,475
4096			Out of Memory			
8192			Out of Memory			
PCGopt:						
4	1.1673e-01	N/A	1.1573e-08	7	0.01	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.00	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.00	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.01	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.03	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.13	-
256	4.9811e-05	3.9683	9.2508e-07	57	0.78	-
512	1.2502e-05	3.9842	9.0263e-07	83	4.74	481
1024	3.1321e-06	3.9916	8.9577e-07	121	27.96	648
2048	7.8394e-07	3.9953	9.0626e-07	176	154.64	1,432
4096			Out of Memory			
8192			Out of Memory			

Table 5: Convergence Results for PCG

accurate and a valid method. This technique is not related to the main focus of the project, matrix-free computing, but is regardless a useful technique when trying to solve a symmetric system more efficiently.

Table 6 shows that the error decreases and the iteration count, time, and observed memory usage increase as the system size increases. When examining Table 6, it is necessary to remember that A and M_1 are stored as matrices and therefore only a savings of the M_2 is expected. Comparing the results of this method to the results of PCGopt in Table 5, we see a memory savings for dimensions $N = 1024$ and $N = 2048$, which represent an even larger total memory savings from Matlab's PCG (also in Table 5). Unfortunately, we also see that the wall clock time for PCGopt with $M_2 =$"transp" has increased slightly from regular PCGopt and from Matlab's PCG.

4.2 Matrix-Free M_1

To further optimize PCGopt, we must observed that preconditioning matrix M_1 is lower triangular and sparse. Consequently, in order to perform a matrix-vector multiplication with matrix M_1 and the r vector it not necessary to actually form M_1 in order to perform a matrix-vector multiplication; rather, we can do a matrix-free linear solve.

As previously noted, if we let D be a diagonal matrix whose entries are the diagonal entries of A, $-E$ be the strictly lower triangular portion of A and $-F$ be the strictly upper triangular portion of A, then $M_1 = \sqrt{\frac{\omega}{2-\omega}}(\frac{1}{\omega}D - E)D^{-1/2}$ [2] where $\omega = \frac{2}{1+\sin \pi h}$ [1, p.540]. To solve $r = M_1 q$ for vector q, we need to

N	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
4	1.1673e-01	N/A	1.1573e-08	7	0.01	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.00	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.00	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.01	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.04	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.18	-
256	4.9811e-05	3.9683	9.2508e-07	57	1.16	-
512	1.2502e-05	3.9842	9.0263e-07	83	7.03	481
1024	3.1321e-06	3.9917	8.9577e-07	121	41.47	661
2048	7.8394e-07	3.9953	9.0626e-07	176	235.45	1,356
4096			Out of Memory			
8192			Out of Memory			

Table 6: Convergence Results PCGopt (A=matrix, M_1=matrix, M_2=transp)

invert M_1 and then multiply it by vector r.

$$\begin{aligned} q &= M_1^{-1} r \\ q &= \left[\sqrt{\frac{\omega}{2-\omega}}\left(\frac{1}{\omega}D - E\right) D^{-1/2}\right]^{-1} r \\ q &= \sqrt{\frac{2-\omega}{\omega}} D^{1/2}\left(\frac{1}{\omega}D - E\right)^{-1} r \\ q &= 2\sqrt{\frac{2-\omega}{\omega}}\left(\frac{4}{\omega}I - E\right)^{-1} r \end{aligned}$$

Now we note that M_1 is a block $N^2 \times N^2$ matrix consisting of $N \times N$ blocks, and if we let $\tilde{M_1} = (\frac{4}{\omega}I - E)^{-1}$, then we get

$$\tilde{M_1} = \begin{bmatrix} M_{1N} & & & & \\ -I & M_{1N} & & & \\ & \ddots & \ddots & & \\ & & -I & M_{1N} & \\ & & & -1 & M_{1N} \end{bmatrix}, \quad M_{1N} = \begin{bmatrix} 4/\omega & & & & \\ -1 & 4/\omega & & & \\ & \ddots & \ddots & & \\ & & -1 & 4/\omega & \\ & & & -1 & 4/\omega \end{bmatrix}.$$

In this manner we can create one function to solve the block M_{1N} matrices as a part of a larger function that solves the $\tilde{M_1}$ matrix. Then, if we multiply the vector resultant from the previous step by $2\sqrt{\frac{2-\omega}{\omega}}$ we have found vector q without ever creating a matrix.

Table 7 presents the complete convergence results of using PCG and PCGopt with A and M_2 as matrices and with M_1 matrix-free. In both cases, the error decreases and the iteration count, time, and observed memory usage increase as the system size increases. Again, the convergence results for PCG and PCGopt for each dimension N are identical and so we are assured of the accuracy of the M_1 matrix-free method. Table 7 reports a lower observed memory usage for PCGopt M_1=matrix-free than for Matlab's PCG for each dimension N. However, even while the time required to compute the numerical solution for each dimension N for PCGopt is slightly more than for PCG, the decrease in observed memory usage supports the declaration of PCGopt with M_1=matrix-free superior to Matlab's PCG with M_1=matrix-free. Lastly, both methods converged for system size $N = 2048$, but run out of memory for $N = 4096$.

In this method we no longer store matrix M_1 and thus we expect to see a corresponding memory saving. In fact, a comparison of PCG and PCGopt in Table 7 with PCG and PCGopt in Table 5 where all matrices were stored shows that both methods used less memory for each dimension N. PCG also required less time when M_1=matrix-free. Therefore, PCG with M_1=matrix-free is an improvement to standard PCG.

N	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's PCG:						
4	1.1673e-01	N/A	1.1573e-08	7	0.00	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.01	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.01	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.02	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.07	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.21	-
256	4.9811e-05	3.9683	9.2508e-07	57	1.00	-
512	1.2502e-05	3.9842	9.0263e-07	83	5.24	486
1024	3.1321e-06	3.9916	8.9577e-07	121	28.61	634
2048	7.8394e-07	3.9953	9.0626e-07	176	156.12	1,427
4096			Out of Memory			
8192			Out of Memory			
PCGopt:						
4	1.1673e-01	N/A	1.1573e-08	7	0.00	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.00	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.01	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.02	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.06	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.22	-
256	4.9811e-05	3.9683	9.2508e-07	57	1.10	-
512	1.2502e-05	3.9842	9.0263e-07	83	6.22	473
1024	3.1321e-06	3.9916	8.9577e-07	121	35.56	620
2048	7.8394e-07	3.9953	9.0626e-07	176	194.31	1,276
4096			Out of Memory			
8192			Out of Memory			

Table 7: Convergence Results PCGopt (A=matrix, M_1=matrix-free, M_2=matrix)

4.3 Matrix-Free M_2

At this point, we have demonstrated that we can perform a matrix-free matrix-vector multiplication where A is used in the Conjugate Gradient method and have eliminated the need to store M_1 as a matrix by doing a matrix-free linear solve when M_1 is called. Similarly, to further optimize PCGopt, we can create another function that removes the need to store preconditioning matrix M_2 by doing another matrix-free linear solve; the results of combining this technique with M_1 matrix-free are shown in row four under PCGopt in Table 2 for $N = 1024$. In order to perform a matrix-vector multiplication with matrix M_2 and the q vector we must observed that preconditioning matrix M_2 is upper triangular and sparse. With matrices D, $-E$, and $-F$ as defined above, $M_2 = \sqrt{\frac{\omega}{2-\omega}} D^{-1/2}(\frac{1}{\omega}D - F)$. To solve $q = M_2 z$ for vector z we must multiply both sides by M_2^{-1}.

$$
\begin{aligned}
z &= M_2^{-1} q \\
z &= \left[\sqrt{\frac{\omega}{2-\omega}} D^{-1/2} \left(\frac{1}{\omega} D - F\right)\right]^{-1} q \\
z &= \sqrt{\frac{2-\omega}{\omega}} \left(\frac{1}{\omega} D - F\right)^{-1} D^{1/2} q \\
z &= 2\sqrt{\frac{2-\omega}{\omega}} \left(\frac{4}{\omega} I - F\right)^{-1} q
\end{aligned}
$$

N	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's PCG:						
4	1.1673e-01	N/A	1.1573e-08	7	0.00	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.01	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.01	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.03	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.08	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.29	-
256	4.9811e-05	3.9683	9.2508e-07	57	1.29	-
512	1.2502e-05	3.9842	9.0263e-07	83	6.32	484
1024	3.1321e-06	3.9916	8.9577e-07	121	34.94	608
2048	7.8394e-07	3.9953	9.0626e-07	176	189.44	1,134
4096			Out of Memory			
8192			Out of Memory			
PCGopt:						
4	1.1673e-01	N/A	1.1573e-08	7	0.04	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.01	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.01	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.03	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.08	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.31	-
256	4.9811e-05	3.9683	9.2508e-07	57	1.42	-
512	1.2502e-05	3.9842	9.0263e-07	83	7.57	462
1024	3.1321e-06	3.9916	8.9577e-07	121	41.16	582
2048	7.8394e-07	3.9953	9.0626e-07	176	230.08	1,118
4096			Out of Memory			
8192			Out of Memory			

Table 8: Convergence Results PCGopt (A=matrix, M_1=matrix-free, M_2=matrix-free)

Now we note that M_2 is a block $N^2 \times N^2$ matrix consisting of $N \times N$ blocks, and if we let $\tilde{M}_2 = (\frac{4}{\omega}I - F)^{-1}$, then we get

$$\tilde{M}_2 = \begin{bmatrix} M_{2N} & -I & & & \\ & M_{2N} & -I & & \\ & & \ddots & \ddots & \\ & & & M_{2N} & -I \\ & & & & M_{2N} \end{bmatrix}, \quad M_{2N} = \begin{bmatrix} 4/\omega & -1 & & & \\ & 4/\omega & -1 & & \\ & & \ddots & \ddots & \\ & & & 4/\omega & -1 \\ & & & & 4/\omega \end{bmatrix}.$$

In this manner we can create one function to solve the block M_{2N} matrices as a part of a larger function that solves the $\tilde{M}_2$ matrix. Then, if we multiply the vector resultant from the previous step by $2\sqrt{\frac{2-\omega}{\omega}}$ we have found vector z without ever creating a matrix.

Complete convergence results of using PCGopt with A as a matrix with M_1 and M_2 matrix-free are shown in Table 8. In both cases as expected, the error decreases and the iteration count, time, and observed memory usage increase as the system size increases. A comparison of the error, $\|e\|_\infty$, error ratio, relative residual, $\|r\|_2/\|b\|_2$, and iteration count for each dimension N between the two methods reveals that they have the same convergence results thus confirming the accuracy of the matrix-free algorithm. PCGopt improves slightly on PCG in terms of the observed memory usage, but unfortunately requires more time. Both methods converge for system size $N = 2048$, but run out of memory for $N = 4096$.

With this technique, a total savings of two preconditioning matrices is expected as M_1 and M_2 are implemented matrix-free. Therefore, the results presented in Table 8, which reflect PCG and PCGopt with

N	$\|e\|_\infty$	Error Ratio	$\|r\|_2/\|b\|_2$	Iter	Time(sec)	Mem(MB)
Matlab's PCG:						
4	1.1673e-01	N/A	1.1573e-08	7	0.01	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.01	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.01	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.03	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.09	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.31	-
256	4.9811e-05	3.9683	9.2508e-07	57	1.27	-
512	1.2502e-05	3.9842	9.0263e-07	83	6.83	459
1024	3.1321e-06	3.9916	8.9577e-07	121	38.27	543
2048	7.8394e-07	3.9953	9.0626e-07	176	215.43	889
4096	1.9619e-07	3.9958	9.9989e-07	256	1,213.42	2,234
8192			Out of Memory			
PCGopt:						
4	1.1673e-01	N/A	1.1573e-08	7	0.01	-
8	3.9153e-02	2.9813	5.9428e-07	9	0.01	-
16	1.1267e-02	3.4748	2.4780e-07	14	0.01	-
32	3.0128e-03	3.7399	8.3630e-07	19	0.03	-
64	7.7812e-04	3.8719	5.7114e-07	28	0.08	-
128	1.9766e-04	3.9366	7.3653e-07	40	0.31	-
256	4.9811e-05	3.9683	9.2508e-07	57	1.43	-
512	1.2502e-05	3.9842	9.0263e-07	83	8.07	451
1024	3.1321e-06	3.9916	8.9577e-07	121	45.50	525
2048	7.8394e-07	3.9953	9.0626e-07	176	254.50	820
4096	1.9619e-07	3.9958	9.9989e-07	256	1,444.06	2,100
8192			Out of Memory			

Table 9: Convergence Results PCGopt (A=matrix-free, M_1=matrix-free, M_2=matrix-free)

M_1=matrix-free and M_2=matrix-free, should be better than the results in Table 7 where only M_1=matrix-free and they are. The observed memory usage was lower for both methods for each dimension when both preconditioning matrices were implemented matrix-free rather than just M_1=matrix-free. Finally, it should be noted that by implementing both preconditioning matrices with matrix-free methods, we are able to decrease total observed memory from Matlab's PCG and PCGopt in Table 5 where all matrices were stored.

4.4 Matrix-Free Implementation

Finally, Table 9 incorporates all matrix-free techniques for PCGopt in an effort to maximize efficiency (minimize memory usage for each dimension N). Entirely matrix-free implementation of PCGopt means that there are no matrices stored by the solver and therefore this method is expected to yield the best results in terms of memory storage. Convergence results of using PCG and PCGopt with A, M_1, and M_2 matrix-free are displayed in Table 9. In both cases, the error decreases and the iteration count, time, and observed memory usage increase as the system size increases. Also, the convergence results for PCG and PCGopt are exactly identical in terms of error, error ratio, relative residual, and iteration count and therefore we declare matrix-free implementation of the Preconditioned Conjugate Gradient method to be a success at least in terms of accuracy. Finally, it is important to observe that for each dimension N, PCGopt uses less memory than PCG, but requires slightly more time to converge. Both method are now able to converge for a system size of $N = 4096$, but run out of memory for $N = 8192$.

Table 9 shows the results of no longer storing any matrices and thus a memory savings is expected when compared to the results in Table 8 where A was stored. A comparison of PCG and PCGopt in Table 9 with PCG and PCGopt in Table 8 shows that both methods used less memory for each dimension N, but again required more time to converge. Therefore at least in terms of memory usage, the entirely matrix-free implementation of PCG is a success. When Table 9 is compared with Table 5 where all matrices are stored for each dimension N, both PCG and PCGopt improve in observed memory usage. For $N = 2048$, the memory usage of matrix-free PCG and PCGopt improve on standard PCG and PCGopt by approximately 40%. The time required for convergence for each dimension for both method does increase, but not to the point of doubling. As the matrix-free PCG and PCGopt methods use less memory than the previously described methods, the matrix-free techniques are a sucess and the value of the memory savings is indisputable.

Finally, by incorporating all matrix-free techniques that are described in detail in section 4, a matrix-free optimal Preconditioned Conjugate Gradient method is developed. This method performs the same task as Matlab's PCG function, but in an optimal way so as to use far less memory. An examination of the convergence results in Table 9 in comparison with the original convergence results for PCG in Table 5, demonstrates concretely that the objective was achieved. The matrix-free optimized method for $N = 2048$ uses less memory than Matlab's PCG function. More impressively, while standard PCG can solve the problem for system sizes up to $N = 2048$, the new all matrix-free method can successfully solve the problem for $N = 4096$! Though the time requirement is not inconsiderable, it cannot belie this remarkable accomplishment.

References

[1] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997.

[2] D. S. Watkins, *Fundamentals of Matrix Computations*, Wiley, second ed., 2002.

[3] S. Yang and M. K. Gobbert, *The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions*, Appl. Math. Lett., 22 (2009), pp. 325–331.