Numerical Methods to Solve 2-D and 3-D Elliptic Partial Differential Equations Using Matlab on the Cluster maya

David Stonko, Samuel Khuvis, and Matthias K. Gobbert (gobbert@umbc.edu)

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

Technical Report HPCF–2014–9, `www.umbc.edu/hpcf` > Publications

Abstract

Discretizing the elliptic Poisson equation with homogeneous Dirichlet boundary conditions by the finite difference method results in a system of linear equations with a large, sparse, highly structured system matrix. It is a classical test problem for comparing the performance of direct and iterative linear solvers. We compare in this report Gaussian elimination applied to a dense system matrix, Gaussian elimination applied to a sparse system matrix, the classical iterative methods of Jacobi, Gauss-Seidel, and SOR, and finally, the conjugate gradient method without preconditioning, and the conjugate gradient method with SSOR preconditioning. The key conclusions are: (i) The comparison of dense and sparse storage shows the crucial importance of sparse storage mode to solve problems even of intermediate size. (ii) The conjugate gradient method outperforms the classical iterative methods in all cases. (iii) Preconditioning can speed up the conjugate gradient method by an order of magnitude. (iv) We find that in two dimensions Gaussian elimination of a sparse system matrix is the fastest method, but runs out of memory eventually, where iterative methods can still solve the problem, but at the price of possibly extremely long run times. (v) However, in three dimensions, the iterative methods can be significantly faster than Gaussian elimination and can solve significantly larger problems. This explains the importance of iterative methods for three-dimensional problems.

Key words. Poisson Equation, Finite Difference Method, Iterative Methods, Matlab.

AMS subject classifications (2010): 65Y20, 65F50, 65M06, 65M12.

1 Introduction

Finding numerical methods to solve partial differential equations is an important and highly active field of research. There are numerous ways to approximate such a solution. However, significant differences exist in the sophistication of these methods, the speed at which these methods converge, how much memory these methods utilize during their computation, and in the programmability of these methods across different numerical software programs. In order to quantify some these differences, Coman, Brewster, et al. [1] did work to compare several numerical computation programs, including Matlab (`www.mathworks.com`) using various test problems. They also provide their driver code for some of the methods we test in this article, which we use with modification in the computational described in this article. One of the test problems they looked at was finding a solution to the Poisson equation in two dimensions. They computed the solution using Gaussian elimination and the conjugate gradient method and compared the quality of these methods and their efficiency.

In this article, we utilize the same test problem, except we use both the two- and three-dimensional versions, and we compare not only the methods of Gaussian elimination and conjugate gradient, but also several additional direct and iterative numerical methods, all of which have been programmed in Matlab. Ultimately, we will draw conclusions about the accuracy and the convergence rates of these methods, which will equip us with the knowledge of how these methods compare to one another so that we can predict how they would perform with new problems in the future. Specifically, the numerical methods we use are (1) Gaussian elimination applied to both a dense system matrix and a sparse system matrix, (2) the classical iterative methods including the Jacobi method, the Gauss-Seidel method, and the SOR(ω_{opt}) method, and (3) the conjugate gradient method without and with SSOR(ω_{opt}) preconditioning. We will compare these methods on the test problem in both two and three dimensions.

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. The current machine in HPCF is the 240-node distributed-memory cluster maya. The newest part of the cluster are the 72 nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory that include 19 hybrid nodes with two state-of-the-art NVIDIA K20 GPUs (graphics processing units) designed for scientific computing and 19 hybrid nodes with two cutting-edge 60-core Intel Phi 5110P accelerators. All nodes are connected via InfiniBand to a central storage of more than 750 TB.

Throughout this analysis, we utilize Matlab R2014a (8.3.0.532). Unless otherwise stated, results throughout this report are run on one of the compute nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory. This investigation is an extension of previous work done on a commodity laptop [5].

This report is organized as follows: Sections 2 and 3 introduce the Poisson equation and its discretization by finite differences, respectively, that yields the system of linear equations used as test problem for the direct and iterative linear solvers in the following sections. The following Sections 4, 5, and 6 present the results for Gaussian elimination, the classical iterative methods, and the conjugate gradient method, respectively, in the two-dimensional case. Then, Section 7 collects the results for all methods in the three-dimensional case. Finally, Section 8 compares the way in which the iterative methods converge to the tolerance, and Section 9 collects the conclusions for all methods considered in this report.

2 The Poisson Equation

For our test problem, we consider the classical problem of solving the two-dimensional Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\triangle u &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{aligned} \tag{2.1}$$

where $\partial\Omega$ denotes the boundary of the domain of Ω, the Laplace operator is defined as $\triangle u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ in two dimensions with the right-hand side given by the function

$$f(x,y) = -2\pi^2 \cos(2\pi x)\sin^2(\pi y) - 2\pi^2 \sin^2(\pi x)\cos(2\pi y)$$

on the unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$.

This problem has been designed so that we can obtain a closed-form solution which we can compare to our computed numerical solutions. This solution is given by

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

in two dimensions. Figure 2.1 shows the true solution of the two-dimensional Poisson equation plotted over Ω.

In short, the problem is to solve an elliptical partial differential equation that is linear with constant coefficients, which we consider on a square domain in two dimensions. This problem is open to analytical solutions by using techniques such as separation of variables and Fourier expansions, as provided above. However, it is also a classical model problem for numerical methods research for partial differential equations, where you can use the true solution in (2.2) to check our numerical solutions for accuracy and to compute its error.

We perform analogous analysis on the three-dimensional system with domain Ω given by the unit cube. Here we have that the right-hand size is given by the function

$$f(x,y,z) = -2\pi^2\big(\cos(2\pi x)\sin^2(\pi y)\sin^2(\pi z) + \sin^2(\pi x)\cos(2\pi y)\sin^2(\pi z)^2 + \sin^2(\pi x)\sin^2(\pi y)\cos(2\pi z)\big)$$

and that the true solution is given by

$$u(x,y,z) = \sin^2(\pi x)\, \sin^2(\pi y)\, \sin^2(\pi z).$$

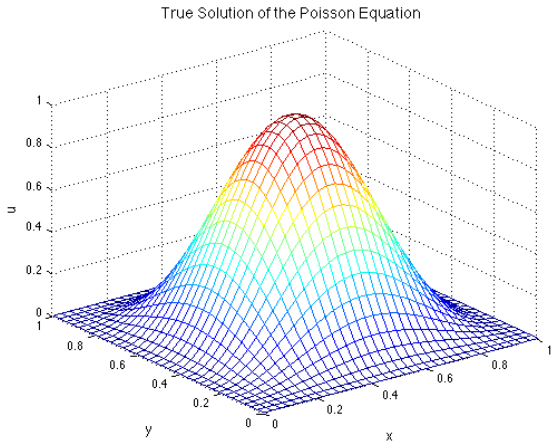


Figure 2.1: True solution of the two-dimensional Poisson equation (2.2).

3 The Finite Difference Method

The finite difference methods are numerical methods which are used to approximate the solutions of differential equations. On our test problem, this method results in a system of linear equations for the unknowns with a highly structured system matrix with size controlled by the number of mesh points. This property has made this problem a good test case for numerical methods which find the solution of linear systems of equations. This classical problem is also studied in [2–4, 6]. Moreover, the system matrix is sparse, which means it has many zero entries. This enables us to demonstrate the issue of storage requirements of scientific computing. It is symmetric positive definite, which allows us to use the conjugate gradient method, and it is also irreducibly diagonally dominant, which guarantees the convergence of our solution for all classical iterative methods that we employ.

In the next section, we develop the finite difference approximation for the two-dimensional system. With the previously given definitions for the three-dimensional problem and this development, the analogous setup for the three-dimensional test problem follows immediately.

3.1 The finite difference method discretization and error

First, we will divide our two-dimensional square domain Ω into a grid of mesh points $\Omega_h = \{(x_i, y_j) = (ih, jh)$ for $i, j = 0, \ldots, N+1\}$ with an uniform mesh width of $h = \frac{1}{N+1}$. Now we apply the second-order finite difference approximation to the x- and y-derivatives $\forall (x_i, y_j) \in \Omega_h$. Doing so, we obtain the approximations

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_i) \approx \frac{1}{h^2}\big(u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)\big), \tag{3.1}$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_i) \approx \frac{1}{h^2}\big(u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})\big). \tag{3.2}$$

By plugging these into (2.1), and using the boundary conditions defined on $\partial\Omega$ we find that

$$-\frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} - \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} = f_{i,j} \tag{3.3}$$

with short-hand notation $f_{i,j} = f(x_i, y_j)$. By simplifying and plugging back in we find that the problem becomes the following equations for the approximation $u_{i,j} \approx u(x_i, y_j)$

$$-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f_{i,j} \text{ for } i, j = 1, ..., N, \tag{3.4}$$

$$u_{0,j} = u_{i,0} = u_{N+1,j} = u_{i,N+1} = 0, \tag{3.5}$$

by the way that the mesh is defined. This is a set of linear equations in the unknowns $u_{i,j}$, which can be organized into a system of linear equations $Au = b$ with N^2 equations. Because we know the boundary values, we have precisely N^2 unknowns in the two-dimensional case. The system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ is defined as

$$A = \begin{bmatrix} S & -I & & & \\ -I & S & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & S & -I \\ & & & -I & S \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2}, \quad S = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix} \in \mathbb{R}^{N \times N},$$

where I is the identity matrix of size $N \times N$. In the three-dimensional case, it is possible to define the system matrix $A \in \mathbb{R}^{N^3 \times N^3}$ recursively in generalizing the above formulation, as specified in [7]. The right hand side vector $b \in \mathbb{R}^{N^3}$ is also defined analogously in three dimensions.

Since the aim of this project is to examine and compare numerical methods, we will also need a way to measure the distance between our numerical solution u_h, and the true solution u given in (2.2). We quantify this error as $||u - u_h||_{L^\infty(\Omega)}$, which is norm of the difference between the true solution and the numerical solution, as described previously. From analysis of the finite difference method, we expect that this method will converge so that $||u - u_h||_{L^\infty(\Omega)} \leq Ch^2$ for some constant C that is not dependent on the mesh width h, as h goes to 0 [6]. Moreover, for sufficiently small mesh sizes, we expect the ratio of the errors at step size $2h$ and h to be approximately 4. This can be shown easily:

$$\text{Ratio} = \frac{||u - u_{2h}||_{L^\infty(\Omega)}}{||u - u_h||_{L^\infty(\Omega)}} \approx \frac{C(2h)^2}{Ch^2} \approx 4. \tag{3.6}$$

Precisely, we utilize the function norm to calculate the distance between the computed solution and the true solution, which is given by $||u - u_h||_{L^\infty(\Omega)} = \sup_{(x,y)\in\Omega} |u(x,y) - u_h(x,y)|$. Throughout this article, we will use $||\cdot||$ to mean this norm. When using these methods to solve our test problem we will also compute this error, and this ratio to observe how this method works in practice. In future sections we will provide these data in the tables alongside the calculated solutions.

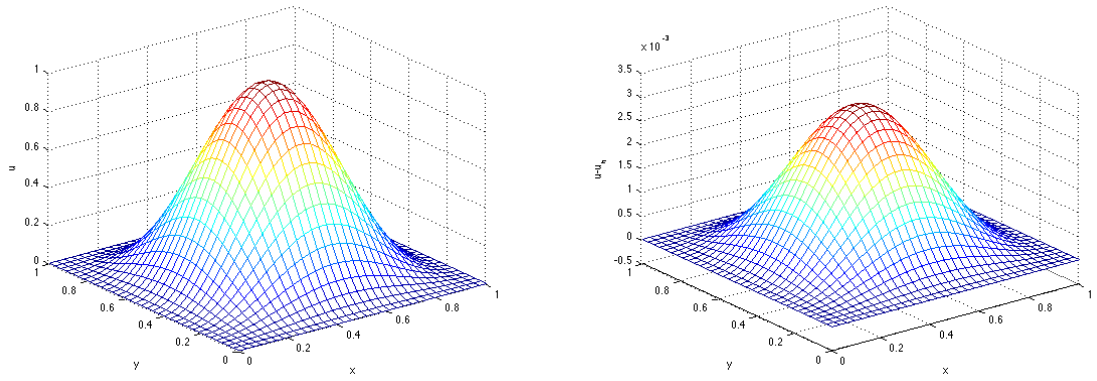
4 Gaussian elimination

The first numerical method we implement here is Gaussian elimination. We apply this method to a dense system matrix and sparse system matrix. In Section 3 we described the setup of this system, which we now use Gaussian elimination to solve. In the first case our system is dense and requires a lot of memory. In the second case our system is sparse and requires significantly less memory to store it. Here, we will investigate how this difference in memory requirement impacts numerical performance, and will ultimately see that the sparse system matrix has a much better capacity to handle system of larger values of N.

4.1 Gaussian elimination applied to a dense system matrix

As we have previously discussed, the system matrix corresponding to this problem is sparse. We set up the system matrix A and keep it sparse. Then, once A is set up, we use `A=full(A)` to make the system matrix dense. We then use code similar to that provided with the technical report on the HPCF webpage from Coman et al. [1]. The result of this system being solved with Gaussian elimination with incrementally larger systems is compiled into Table 4.1, and a plot of the numerical solution and corresponding error, is shown in Figure 4.1. This figure is specifically for $N = 32$ in the two-dimensional system.

In the top half of Table 4.1 we provide the results for this with increasing values of N in two dimensions. The first column in this table is N, which is the number of mesh points. The second column is the degrees of freedom, which is N^2 in two dimensions. The third column is the error of the solution that was computed at the corresponding N compared to the true solution to the equation, which was provided and plotted in Section 2. As expected, this error decreases as N increases. In fact, the ratio at which the error of each



(a) Numerical Solution

(b) Numerical Error

Figure 4.1: (a) The numerical solution and (b) the error of the numerical solution to the Poisson equation with $N = 32$.

successive run decreases is given in column four of Table 4.1. As expected, this ratio is nearly 4, which is the analytical solution derived previously in (3.6). The fifth column, C, is the error divided by h^2. The last column is the observed wall clock time that it took for the method to converge. As expected, the runtime increases each time N is doubled. Additionally, we see that increasing N past 256 causes the computer to run out of memory in the two-dimensional case.

4.2 Gaussian elimination applied to a sparse system matrix

We now aim to solve the exact same problem that we did in Section 4.1, but now we use a sparse system matrix so that we can compare the performance and the memory usage between the two options. To do this, we use the setup of A as described in the previous section, except that we do not make A full (i.e., we do not change it, and it remains sparse, because it is sparse upon setup).

The results of this method are provided in the second half of Table 4.1. Here, we observe much better results than we observed in Section 4.1 for the dense system matrix. More specifically, we obtain the same level of accuracy (and obtain the exact same solutions) as previously, but we do so in a fraction of the time. Also, we can solve much larger systems. Before, we ran out of memory past $N = 256$, where here we obtain results through $N = 8192$, which is more than an order of magnitude larger.

For $N = 8192$, 74.6 GB of memory are required. A compute node only has 64 GB of physical memory. If a job exceeds the 64 GB of physical memory then the node will use swap space. Swap space is a space on the hard drive in which chunks of memory are copied from physical memory. Since it is located on the hard drive swap space performs slower than physical memory. This results in slower runtimes when swap space is used. By running it on a user node with 128 GB of physical memory we can avoid using swap space. Moreover, the runtime of the sparse system at $N = 8192$ was about the same as the runtime for $N = 256$ with the dense system on the compute node and half the runtime on the user node, and hence substantially better.

We have demonstrated that when using Gaussian elimination, the memory requirement can be a limiting factor in the viability of the method. We have also shown that a main factor in the speed of these methods is memory usage. The only difference between these calculations was the sparsity of the matrix and, in turn, the memory that the system required. Indeed, the sparse system exhibited substantially better performance due to a smaller memory requirement.

2D - Gaussian Elimination applied to a dense system matrix:					
N	DOF $= N^2$	$\|\|u - u_h\|\|$	Ratio	C	Run time (sec.)
32	1024	3.0128e-03	3.7399	3.2809	0.06
64	4096	7.7812e-04	3.8719	3.2876	0.39
128	16384	1.9766e-04	3.9366	3.2890	9.95
256	65536	4.9807e-05	3.9685	3.2897	411.37
512	262144		Out of memory		
1024	1048576		Out of memory		
2048	4194304		Out of memory		
4096	16777216		Out of memory		
8192	67108864		Out of memory		
16384	268435456		Out of memory		
2D - Gaussian Elimination applied to a sparse system matrix:					
N	DOF $= N^2$	$\|\|u - u_h\|\|$	Ratio	C	Run time (s)
32	1024	3.0128e-03	3.7439	3.2809	<0.01
64	4096	7.7812e-04	3.8719	3.2876	0.01
128	16384	1.9766e-04	3.9366	3.2893	0.03
256	65536	4.9807e-05	3.9885	3.2897	0.14
512	262144	1.2500e-05	3.9843	3.2898	0.61
1024	1048576	3.1313e-06	3.9922	3.2899	3.04
2048	4194304	7.8362e-07	3.9961	3.2900	15.98
4096	16777216	1.9607e-07	3.9966	3.2912	70.60
8192	67108864	4.9325e-08	3.9751	3.3109	281.43
16384	268435456		Out of memory		

Table 4.1: Numerical results obtained from solving the Poisson equation using Gaussian elimination with a dense system matrix and a sparse system matrix for various values of N on maya. As N doubles, the error is approximately quartered, but the runtime increases exponentially. Clearly, Gaussian elimination with a sparse system matrix is much faster than this method on a dense system matrix. For $N = 8192$ using Gaussian Elimination with a sparse system matrix, the job was run on the user node since it requires 74.6 GB of memory. It is also possible to run this on a compute node by using swap space. However, this causes a significant increase in runtime to 414.93 seconds.

5 Classical Iterative Methods: Jacobi, Gauss-Seidel, and SOR

Another set of methods to solve linear systems of equations is referred to as the classical iterative methods and they include the Jacobi method, the Gauss-Seidel method, and the Successive Over Relaxation (SOR) method.

In order to use these iterative methods we developed a Matlab function called `classiter`. Since each of these functions can be written in the form as Watkins [6] writes as $Mx^{(k+1)} = Nx^{(k)} + b$ with the splitting matrix M depending on the method and $N = M - A$, we are able to set up the splitting matrix for the system depending on the choice of method and then use the same body of code to compute the iterations. This function takes the system matrix A, right hand side b, initial guess $x^{(0)}$, relaxation parameter ω, an error tolerance to determine convergence, a limit on the maximum number of iterations `maxit` and a parameter `imeth` which tells the function which method to use. Then, the splitting matrix is described as follows: for the Jacobi Method, M is defined to be the diagonal elements of the system matrix A, for the Gauss-Seidel method M is set to the lower triangular entries of A including the diagonal using the Matlab command `M=tril(A)` and for the SOR(ω_{opt}) method the splitting matrix is defined to

be `M=(1/w)*D+(tril(A)-D)`, where D is a matrix with the diagonal entries of A and zeros elsewhere. As indicated by the notation of ω_{opt} in SOR(ω_{opt}), we choose the optimal value of the relaxation parameter $\omega_{opt} = 2/(1 + \sin(\pi h))$, as defined by Yang and Gobbert [7]. The body of the function then computes the iterates until the method either converges within the error tolerance, or the maximum number of iterations is reached. This function returns the final iterate $x^{(k)}$, a flag, a number `relres` which is the final value of the relative residual $||r^{(k)}||_2/||b||_2$ for $r^{(k)} = b - Ax^{(k)}$, the number of iterations taken as `iter`, a vector `resvec`, which is the vector of residuals such that the $(k + 1)$-st component is the value of $||r^{(k)}||$ for k ranging for 0 to `iter`. Together, this gives us that the `classiter` functions can be controlled with a call to Matlab such as `[x,flag,relres,iter,resvec]=classiter(A,b,x0,0,0.05,1000,1)` which calls the function with `imeth=1` so it would use Jacobi method to solve the system matrix A with right hand side b and initial guess $x^{(0)}$.

Now we approach this problem in the same way that we approached it with the previous method using `classiter` to solve the same problem as before with each of the classical iterative methods. The top panel of Table 5.1 shows the results of the Jacobi method on this problem. As previously, the first column represents the number of mesh points and the second column is the degrees of freedom, $DOF = N^2$, and the third column is the error of the numerical method, the forth column is the ratio of the error as the mesh size halves and the fifth column is the constant $C = ||u-u_h||/h^2$. The sixth column shows the number of iterations taken by the method until convergence to the chosen tolerance of 10^{-6} in the relative residual. The last column is the wall clock time of each run. We observe that the Jacobi method is slower than previous methods, but fails to converge in a reasonable amount of time for even intermediate values of N. The second panel of this table shows the numerical results from the Gauss-Seidel method on solving the Poisson equation for the same values of N. The Gauss-Seidel method requires about half as many iterations and correspondingly less time than the Jacobi method. But for larger N, the required numbers of iterations is still unacceptable. The SOR(ω_{opt}) method has results shown in the third panel of Table 5.1. Here, we observe that this iterative method converges for higher choice of N and is dramatically faster than the other classical iterative methods. However, it is still not as fast as or suitable for large values of N as other methods are, such as Gaussian elimination or even the conjugate gradient method.

6 The Conjugate Gradient Method

We now aim to solve the Poisson equation using the conjugate gradient method. This method is another alternative to solve systems of linear equations, namely those whose system matrix is symmetric positive definite, as we have here. This method effectively solves the exact same problem as we examined in the previous sections, but has been reported to do so with much less computational cost. In this article, we examine two instances of this method. First, we examine how this method solves our test problem when implemented without preconditioning, and, second, with preconditioning. One benefit of using this method is that Matlab has a built in function `pcg` that implements the conjugate gradient method with preconditioners.

6.1 Conjugate gradient without preconditioning

Since we now aim to solve this system without preconditioning, we can pass Matlab's `pcg` function empty preconditioning matrices and the setup function provided with [1] on the HPCF webpage. The relevant lines of code in `driver_cg.m` are

```
A = setupA(N);
tol = 1.0e-6;
maxit = 99999;
u = zeros(N^2,1);
M1 = [];
M2 = [];
[u,flag,relres,iter,resvec] = pcg(A,b,tol,maxit,M1,M2,u);
```

2D - Jacobi Method:						
N	DOF $= N^2$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	1024	3.0114e-03	3.7411	3.2794	2902	0.13
64	4096	7.7673e-04	3.8771	3.2817	11258	1.46
128	16384	1.9626e-04	3.9577	3.2659	44331	17.19
256	65536	4.8397e-05	4.0551	3.1966	175921	235.52
512	262144	1.1089e-05	4.3646	2.9182	700881	4222.18
1024	1048576	1.7182e-06	6.4538	1.8051	2797915	61304.50
4028	4194304	Excessive time requirement				
8192	67108864	Excessive time requirement				
16384	268435456	Excessive time requirement				
2D - Gauss-Seidel Method:						
N	DOF $= N^2$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	1024	3.0114e-03	3.7411	3.2795	1452	0.07
64	4096	7.7673e-04	3.8771	3.2817	5630	0.86
128	16384	1.9626e-04	3.9577	3.2659	22166	10.74
256	65536	4.8398e-05	4.0551	3.1966	87962	152.01
512	262144	1.1089e-05	4.3645	2.9182	350442	2621.92
1024	1048576	1.7182e-06	6.4538	1.8052	1398959	41022.65
4028	4194304	Excessive time requirement				
8192	67108864	Excessive time requirement				
16384	268435456	Excessive time requirement				
2D - SOR(ω_{opt}) Method:						
N	DOF $= N^2$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	1024	3.0125e-03	3.7401	3.2807	92	0.01
64	4096	7.7785e-04	3.8729	3.2864	181	0.03
128	16384	1.9737e-04	3.9410	3.2845	359	0.20
256	65536	4.9522e-05	3.9856	3.2709	716	1.41
512	262144	1.2214e-05	4.0544	3.2145	1429	11.58
1024	1048576	2.8630e-06	4.2662	3.0080	2861	92.93
2048	4194304	5.5702e-07	5.1399	2.3386	5740	811.30
4096	16777216	4.6471e-07	1.1986	7.8004	11559	6601.23
8192	67108864	3.8580e-07	1.2045	2.5897	23420	53242.47
16384	268435456	Excessive time requirement				

Table 5.1: Numerical results obtained from solving the Poisson equation using the Jacobi, Gauss-Seidel, and SOR(ω_{opt}) in two dimensions on maya. The SOR(ω_{opt}) method converges faster than the Jacobi and Gauss-Seidel methods on corresponding systems. It also converges in fewer iterations. The Gauss-Seidel method converges faster than the Jacobi method in terms of wall clock run time and iteration count. Excessive time requirement corresponds to more than 24 hours wall clock time.

where `setupA.m` is given in the files on the HPCF webpage. We run these computations with tolerance of 10^{-6} and provide our results in Table 6.1. The columns here are analogous to those of the previous tables. It is clear that the conjugate gradient method is able to solve the same size of the problem as the best classical iterative method, the SOR method, but it did so about twice as fast in all cases.

6.2 Conjugate gradient with SSOR(ω_{opt}) preconditioning

Now we consider the test problem and the preconditioned conjugate gradient method with SSOR(ω_{opt}) as preconditioner, or PCG-SSOR(ω_{opt}) for short. We choose the optimal relaxation parameter ω_{opt} as defined

2D - Conjugate Gradient Method:						
N	DOF $= N^2$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	1024	3.0128e-03	3.7399	3.2809	48	0.01
64	4096	7.7812e-04	3.8719	3.2875	96	0.03
128	16384	1.9766e-04	3.9368	3.2891	192	0.13
256	65536	4.9807e-05	3.9899	3.2891	387	0.89
512	262144	1.2494e-05	3.9856	3.2881	783	7.08
1024	1048576	3.1266e-06	3.9961	3.2849	1581	52.73
2048	4194304	7.8019e-07	4.0075	3.2756	3192	441.59
4096	16777216	1.9366e-07	4.0290	3.2701	6452	3402.96
8192	67108864	4.7401e-08	4.0856	3.1818	13033	26633.88
16384	268435456	Excessive time requirement				
2D - Preconditioned Conjugate Gradient:						
N	DOF $= N^2$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	1024	3.0128e-03	3.7399	3.2809	19	0.01
64	4096	7.7812e-04	3.8719	3.2876	28	0.02
128	16384	1.9766e-04	3.9366	3.2893	40	0.06
256	65536	4.9811e-05	3.9683	3.2899	57	0.29
512	262144	1.2502e-05	3.9842	3.2902	83	1.75
1024	1048576	3.1321e-06	3.9917	3.2906	121	9.61
2048	4194304	7.8394e-07	3.9953	3.2913	176	59.52
4096	16777216	1.9620e-07	3.9961	3.2918	256	331.70
8192	67108864	4.9109e-08	3.9951	3.2964	375	1862.98
16384	268435456	1.2301e-08	3.9923	3.3025	548	11248.57

Table 6.1: Numerical results obtained from solving the Poisson equation with the conjugate gradient method on maya. For $N = 16384$ using PCG-SSOR, the job was run on the user node since it requires 87.5 GB of memory. It is also possible to run this on a compute node by using swap space however the runtime is over 24 hours. Excessive time requirement corresponds to more than 24 hours wall clock time.

by Yang and Gobbert [7], which is given by $\omega_{opt} = 2/(1+\sin(\pi h))$. We use a variation of the splitting matrix M for SSOR, listed in Watkins [6], namely factored multiplicatively as $M = M_1 M_2$ with triangular matrices

$$M_1 = \sqrt{\frac{\omega}{2-\omega}}\left(\frac{1}{\omega}D - E\right)D^{-1/2}, \quad M_2 = \sqrt{\frac{\omega}{2-\omega}}D^{-1/2}\left(\frac{1}{\omega}D - F\right),$$

where $-E$ is the strictly lower triangular portion of A, and $-F$ is the strictly upper triangular portion of A such that $A = D - E - F$. When actually coding this, we want to do it as inexpensively as possible, so instead of calculating M_2, we notice that $M_2 = M_1^\intercal$ for a symmetric matrix A, and we set M_2 by just taking the transpose of M_1. Moreover, we do all of these computations by maintaining the sparsity of the matrices to ensure that we achieve the best computational results. This method is implemented with the following code:

```
A = setupA(N);
tol = 1.0e-6;
maxit = 99999;
u = zeros(N^2,1);
w = 2/(1 + sin(pi*h)); % finds omega_opt
d = diag(A); % gets diagonal elements of system Matrix
```

```
D = spdiags(d,0,N^2,N^2); % puts the diag elements into sparse diag matrix
Ds = spdiags(1./sqrt(d),0,N^2,N^2); % finds D^(-1/2) and maintains sparsity
E = D-tril(A);
M1 = sqrt(w/(2-w))*((1/w)*D-E)*Ds; % calcuates the first precond.
M2 = M1'; % calculates the second, which is just the transpose of the first
[u,flag,relres,iter,resvec] = pcg(A,b,tol,maxit,M1,M2,u);
```

where `setupA(N)` sets up our system matrix with size N. Here, `pcg` performs a linear solve for each M_1 and M_2, but does each one cheaply because each is already a triangular matrix. We run this code and attempt to achieve the best N possible. The results are shown in the bottom panel of Table 6.1

From Table 6.1 we can compare the numerical performance of the conjugate gradient method with and without preconditioning in two dimensions. We see that the preconditioned conjugate gradient method has a much lower iteration count, and has a lower runtime than the method without preconditioning. Specifically, for $N = 8192$, the CG method took 13,033 iterations and 7 hours and 24 minutes versus the PCG method, which took only 375 iterations and just 31 minutes. Clearly the additional few lines of code necessary for preconditioning were worth it. These results also indicate that the conjugate gradient method is slower than Gaussian elimination for all cases where the latter one does not run out of memory. However, it shows that we can obtain results for larger N by using the preconditioned conjugate gradient method, even when compared to Gaussian elimination of a sparse system matrix. Ultimately, this means that while it may converge more slowly than Gaussian elimination, it may be an important option when looking at larger systems where we would encounter memory issues.

7 The Three-Dimensional Poisson Equation

Table 7.1 gives the results from solving the three-dimensional problem with Gaussian elimination on a dense system matrix in the top panel and a sparse system matrix in the bottom panel. These three-dimensional results are analogous to the data given in Table 4.1 for the two-dimensional system, but with degrees of freedom N^3. Even more so than in two dimensions, we observe better performance in terms of speed and memory with the sparse system compared to the dense system. Specifically, there was only sufficient memory to solve the system with $N = 32$ and it took approximately 1 minute and 3 seconds to do so. However, the sparse system solved this system in just 0.26 seconds. Additionally, the sparse system matrix was able to be solved with system size $N = 128$. Thus, we conclude that sparse storage mode is vital to solving problems of even intermediate size in three dimensions.

We also solve this system with the iterative methods. Table 7.2 contains the results from solving the system with the Jacobi, the Gauss-Seidel, and the SOR(ω_{opt}) methods. We observe here that, like in the two-dimensional case, the SOR(ω_{opt}) method can solve larger systems than Gaussian elimination, because memory is less of an issue. Moreover, it is actually faster than Gaussian elimination, for instance for $N = 128$, SOR(ω_{opt}) takes 29 seconds and Gaussian elimination 513 seconds. Here, SOR(ω_{opt}) solved the system up to $N = 512$; Beyond this, a solution took prohibitively long to compute.

From Table 7.3 we can compare the numerical performance of the conjugate gradient method without and with preconditioning in three dimensions. The first panel shows that already the unpreconditioned method is faster than SOR in all cases, just like in two dimensions. We see that the conjugate gradient method with preconditioning has a much lower iteration count, and has a lower runtime than without preconditioning. Specifically, for $N = 512$, the CG method took 999 iterations and 1.3 hours versus the PCG method, which took only 79 iterations and just 17 minutes.

In the three-dimensional case, the use of swap space is not an issue in the same way as in two dimensions. For Gaussian elimination, the largest N value possible on a compute node, $N = 128$, requires 65.8 GB of memory. This requires the use of swap space. However, by running it on the user node with 128 GB of memory we observe only a negligible improvement in runtime from 513.46 seconds to 467.28 seconds. For the PCG method the largest N value possible on a compute node, $N = 512$, requires 59.4 GB of memory so swap space is not an issue at all in this case and running on the user node will not result in any improvement

3D - Gaussian Elimination applied to a dense system matrix:					
N	DOF $= N^3$	$\|\|u - u_h\|\|$	Ratio	C	Run time (sec.)
32	32768	3.0060e-03	3.7164	3.2735	62.93
64	262144		Out of memory		
128	2097152		Out of memory		
256	16194277		Out of memory		
512	134217728		Out of memory		
3D - Gaussian Elimination applied to a sparse system matrix:					
N	DOF $= N^3$	$\|\|u - u_h\|\|$	Ratio	C	Run time (s)
32	32768	3.0060e-03	3.7164	3.2735	0.26
64	262144	7.7767e-04	3.8654	3.2856	8.84
128	2097152	1.9763e-04	3.9350	3.2888	513.46
256	16194277		Out of memory		
512	134217728		Out of memory		

Table 7.1: Numerical results obtained from solving the Poisson equation using Gaussian elimination with a dense system matrix and a sparse system matrix for various values of N in three dimensions on maya. Clearly, Gaussian elimination with a sparse system matrix is faster than this method on a dense system matrix.

3D - Jacobi Method:						
N	DOF $= N^3$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	32768	3.0049e-03	3.7174	3.2723	2916	2.51
64	262144	7.7656e-04	3.8695	3.2810	11316	75.87
128	2097152	1.9652e-04	3.9516	3.2703	44566	2389.50
256	16194277		Excessive time requirement			
512	134217728		Excessive time requirement			
3D - Gauss-Seidel Method:						
N	DOF $= N^3$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	32768	3.0049e-03	3.7174	3.2723	1459	1.59
64	262144	7.7656e-04	3.8695	3.2810	5659	49.48
128	2097152	1.9652e-04	3.9516	3.2703	22284	1610.31
256	16194277		Excessive time requirement			
512	134217728		Excessive time requirement			
3D - SOR(ω_{opt}) Method:						
N	DOF $= N^3$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	32768	3.0049e-03	3.7174	3.2723	97	0.12
64	262144	7.7656e-04	3.8695	3.2818	196	1.79
128	2097152	1.9652e-04	3.9516	3.2747	388	28.78
256	16777216	4.9734e-05	3.9723	3.2849	773	476.97
512	134217728	1.2427e-05	4.0019	3.2705	1542	8431.63

Table 7.2: Numerical results obtained from solving the Poisson equation using the Jacobi, Gauss-Seidel, and SOR(ω_{opt}) in three dimensions on maya. Excessive time requirement corresponds to more than 24 hours wall clock time.

3D - Conjugate Gradient Method:						
N	DOF $= N^3$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	32768	3.0060e-03	3.7165	3.2735	61	0.08
64	262144	7.7765e-04	3.8654	3.2856	120	1.13
128	2097152	1.9763e-04	3.9349	3.2888	244	17.30
256	16194277	4.9807e-05	3.9679	3.2897	493	282.19
512	134217728	1.2503e-05	3.9836	3.2904	999	4705.95
3D - Preconditioned Conjugate Gradient:						
N	DOF $= N^3$	$\|\|u - u_h\|\|$	Ratio	C	Iterations	Run time (s)
32	32768	3.0060e-03	3.7165	3.2735	19	0.06
64	262144	7.7765e-04	3.8654	3.2857	27	0.71
128	2097152	1.9763e-04	3.9349	3.2889	38	7.66
256	16194277	4.9809e-05	3.9679	3.2898	55	81.82
512	134217728	1.2503e-05	3.9838	3.2905	79	1014.21

Table 7.3: Numerical results obtained from solving the Poisson equation in three dimensions with the conjugate gradient method on maya.

in runtime. By comparing Table 7.3 to Table 7.1 we are able to make a stronger conclusion than in the two-dimensional case. Not only is it possible to solver larger systems with CG and PCG than with GE, but CG and PCG are actually able to solve the problem with a lower runtime. In particular, PCG-SSOR(ω_{opt}) is faster than Gaussian elimination by nearly two orders of magnitude eventually for $N = 128$, and it is therefore clearly the best method to use in the three-dimensional case.

8 Comparison of Iterative Methods

Up to this point we have seen that the more sophisticated iterative methods have converged in fewer and fewer iterations and typically with a shorter wall clock run time. We now aim to further characterize the differences in convergence rate in terms of iterations. In order to do so we have taken the vector of residuals, which are supplied by the `pcg` and `classiter` functions return variable `resvec` from each method with $N = 32$ using the setup of our test problem. We then use this vector to calculate the relative residuals by dividing the residuals by the 2-norm of b. Then, we plot the relative residuals versus the iteration number and compare the results from each method, which is shown in Figure 8.1. Here we have two visualizations of this result. Figure 8.1(a) depicts the convergence of each method for the first 2000 iterations and (b) shows the same data, but only for the first 200 iterations, so that we can take a closer look at the methods that converge very fast. From Figure 8.1(a) we can see that the Jacobi and Gauss-Seidel methods are much worse than the other three methods. They clearly take many more iterations to converge. In Figure 8.1(b) we see that either instance of the conjugate gradient method is better than the SOR(ω_{opt}), and that the preconditioned conjugate gradient method converges the fastest (in terms of iteration count). This figure also lends to the observation that there are some maybe unexpected results for the first few iterations. Here we observe that SOR actually started off slightly worse than the other methods, but quickly improved over the Jacobi and Gauss-Seidel methods. We also see a bit of fluctuation in the conjugate gradient method, before it rapidly converges.

We also perform this analysis on the three-dimensional problem, and this is given in Figure 8.2. Here we observe the same general relationship between the iterative methods. In the right hand panel, which is the same as the left, but focused in on the lower iterations, we see that the convergence is similar for each method in the three-dimensional test problem, but that there are some small differences. Namely, that the conjugate gradient method remains above the SOR method for a bit longer.

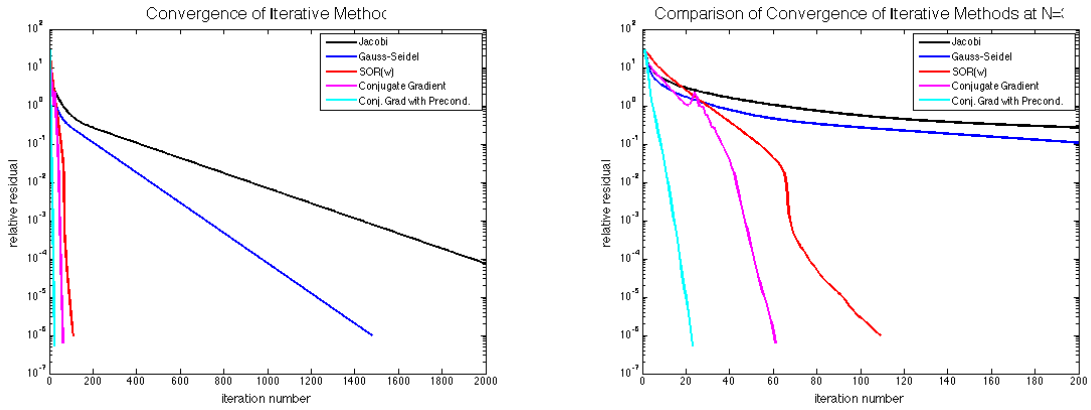


Figure 8.1: Comparison of iterative methods in two dimensions. (a) The relative residual versus the iteration number for each of the iterative methods with $N = 32$ on a semilog plot for 2000 iterations and (b) the same data as (a), but only the first 200 iterations.

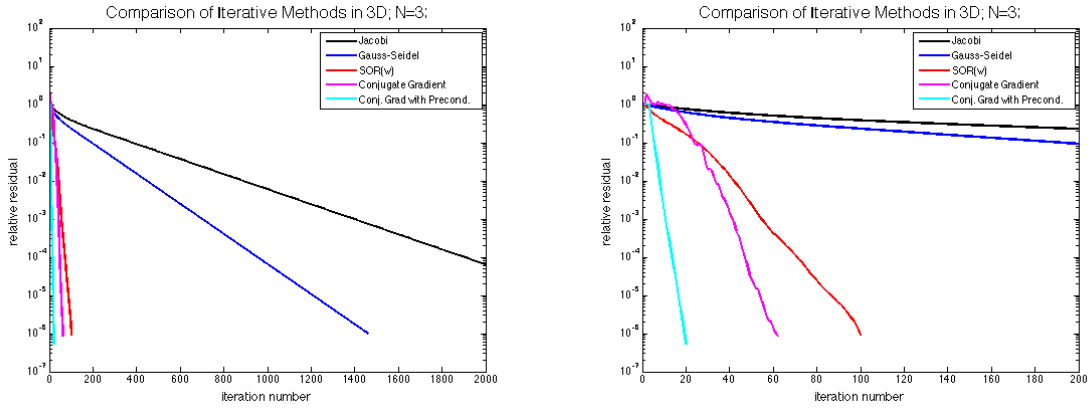


Figure 8.2: Comparison of iterative methods in three dimensions. (a) The relative residual versus the iteration number for each of the iterative methods with $N = 32$ on a semilog plot for 2000 iterations and (b) the same data as (a), but only the first 200 iterations.

9 Conclusions

Through this investigation, we have seen that in both the two- and three-dimensional case, using a sparse storage mode of the system matrix is needed to solve problems even of intermediate size. Also, we have shown that in both the two-dimensional and three-dimensional case, the SOR(ω_{opt}) is the best among the classical iterative methods, but already the (unpreconditioned) conjugate gradient method is twice as fast in all cases. This explains why modern iterative methods have supplanted the classical iterative methods in practice. In turn, the conjugate gradient method with SSOR(ω_{opt}) preconditioning is faster by an order of magnitude than without preconditioning. This explains the crucial importance of preconditioning to improve the performance of iterative methods.

As for Gaussian elimination, we find very different conclusions in two and three dimensions: In two dimensions, we observe that Gaussian elimination with sparse storage is the fastest solver, provided that it does not run out of memory; the iterative methods can eventually solve larger problems than Gaussian elimination, but not significantly larger ones and at the price of extremely long run times, even for the best

iterative method tests, PCG-SSOR(ω_{opt}). In three dimensions, the difference with respect to problem size solved is more significant, with iterative methods being able to handle significantly larger problems than Gaussian elimination, and moreover the iterative methods are faster than Gaussian elimination, with PCG-SSOR(ω_{opt}) nearly two orders of magnitude. This explains the importance of iterative methods to solve three-dimensional problems.

Acknowledgments

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources. This project began as the class project of the first author in Math 630 Numerical Linear Algebra in Spring 2014 at UMBC. It is a heavily modified version of that report [5], which considered the same test problem on a commodity laptop. The second author acknowledges financial support as HPCF RA.

References

[1] Ecaterina Coman, Matthew W. Brewster, Sai K. Popuri, Andrew M. Raim, and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, FreeMat, Scilab, R, and IDL on tara. Technical Report HPCF–2012–15, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2012.

[2] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[3] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, vol. 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.

[4] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 1996.

[5] David Stonko. Numerical methods to solve 2-D and 3-D elliptical partial differential equation using Matlab, 2014. Department of Mathematics and Statistics, University of Maryland, Baltimore County, `http://userpages.umbc.edu/~dstonko1/ClassProject_630.pdf`.

[6] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, third edition, 2010.

[7] Shiming Yang and Matthias K. Gobbert. The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions. *Appl. Math. Lett.*, vol. 22, pp. 325–331, 2009.