PHY905 Project 1 - Gauss Elimination

Alaina Ross

(Dated: February 3, 2017)

Background: To solve a differential equation computationally the derivative is approximated and a system of linear equations is solved instead. Such systems can be solved via matrix equations and Gauss elimination.
Purpose: The goal of this work is to implement various Gauss elimination algorithms and study the accuracy and performance of each when applied to a system of equations which result from the Poisson equation with a given example from electromagnetism.
Method: We use a general Gauss elimination algorithm as well as two for tridiagonal matrixes and one which uses LU decomposition. We inspect the time to solution for varying matrix sizes and compare to the analytic solution to evaluate the accuracy of the algorithms.
Results: We find the most accurate matrix size is 10^5 and the fastest algorithm for this matrix size is the tridiagonal algorithm.
Conclusions: Our results demonstrate the importance of choosing the right algorithm for the given physical situation Our results demonstrate the importance of choosing the right form of parallelism for the given algorithm, in this case shared memory parallelism is the right choice.

I. INTRODUCTION

Often in science problems can be written as a linear second order differential equation such as:

$$\frac{d^2y}{dx^2} + k^2(x)y = f(x). \tag{1}$$

One specific example of this is the Poisson equation in electromagnetism, which is used to find the electrostatic potential, Φ, from a spherically symmetric charge distribution, $\rho(r)$ as:

$$\nabla^2\Phi = \frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = -4\pi\rho(r) \tag{2}$$

using the substitution $\Phi = \phi/r$ and simplifying gives:

$$-\frac{d^2\phi}{dr^2} = 4\pi r\rho(r). \tag{3}$$

In general terms, the equation we aim to solve is $-u''(x) = f(x)$. Since this is a second order differential equation, it requires two boundary conditions. Here we use Dirchilet conditions, namely $u(0) = u(1) = 0$.

Turning these continuous functions in discrete ones we can approximate the second derivative as:

$$-\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i \quad i = 1, 2...n \tag{4}$$

where h is the step size defined as $h = 1/(n+1)$ and n is the number of grid points. This set of equations then can be rearranged as a matrix equation $\hat{A}\vec{u} = \vec{b}$ where:

$$\hat{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,n} \\ a_{n,1} & a_{n,2} & a_{n,n} \end{bmatrix} \quad \vec{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_n \end{bmatrix} \quad \vec{b} = \begin{bmatrix} h^2 f_1 \\ h^2 f_2 \\ h^2 f_n \end{bmatrix} \tag{5}$$

To solve this matrix equation the first goal is to get the matrix in row echelon form (shown below), where the only nonzero elements form the upper right triangle of the matrix. This step is called forward elimination and is done analytically via elementary row operations i.e. addition, interchanging rows, and multiplication by a constant.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,n} \\ 0 & \tilde{a}_{2,2} & \tilde{a}_{2,n} \\ 0 & 0 & \tilde{a}_{n,n} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_n \end{bmatrix} \tag{6}$$

Algorithmically, this can be calculated two ways: the simpler way which only uses addition and multiplication by a constant but will fail if $a_{i,i} = 0$, or the more sophisticated way where rows are interchanged to ensure a solution even when $a_{i,i} = 0$ [1]. In this work we will use the former method and simply ensure that for all matrices $a_{i,i} \neq 0$.

In this work we implement gauss elimination for a general matrix as well as for a tridiagonal matrix and study the performance characteristics and numerical accuracy of each for matrix sizes of $n = 10, 100, 1000$. In addition, we compare the performance of our solutions to that of a more sophisticated algorithm known as LU decomposition in order to determine the most efficient algorithm for the problem at hand. In Section II, the implementation of the various algorithms are described. In Section III the performance and accuracy of the code are analyzed. Finally, in Section IV we give a summary and our conclusions.

II. METHODS

To begin the Gauss elimination we implement the arrays using the matrix and vector classes in the armadillo library [3]. We then start with the first row and column of the matrix as what is called the pivot; this is the column that will be updated to be zeros based on the first pivot

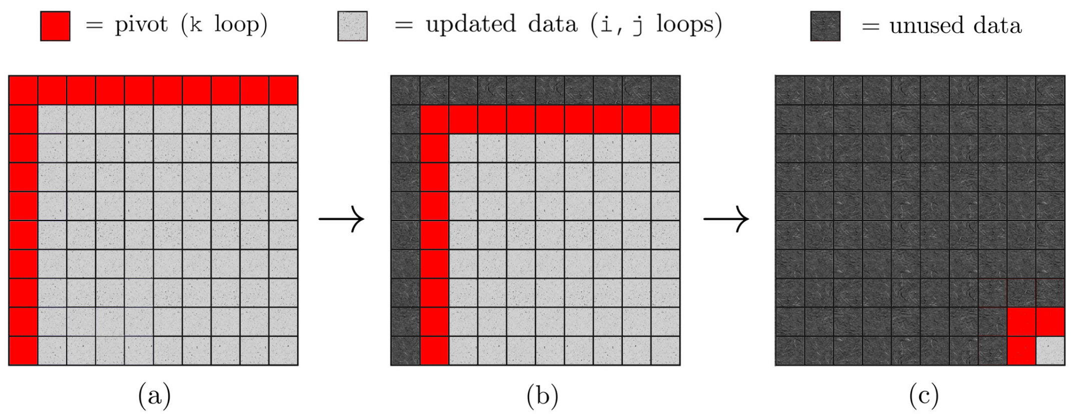

FIG. 1: Graphical representation of forward elimination algorithm based on [2] for (a) the first iteration, (b) the second iteration, and (c) the final iteration.

element (the first element in the row/the corner piece). Then, all values in the rest of the matrix are updated via the same operations that were used in the corresponding element of the pivot column. In the next iteration, the pivot moves to the next row/column and the old pivot becomes unused data for the rest of the iteration process, meaning that each iteration the subspace to update becomes smaller. This process can be viewed graphically in Fig. 1. The algorithm is given by:

$$a_{j,k}^{m+1} = a_{j,k}^{m} - \frac{a_{j,m}^{m} a_{m,k}^{m}}{a_{m,m}^{m}} \qquad j,k = m+1...n \tag{7}$$

$$b_{j}^{m+1} = b_{j}^{m} - \frac{a_{j,m}^{m} b_{m}^{m}}{a_{m,m}^{m}} \qquad j = m+1...n \tag{8}$$

Once the elimination process is complete the next step is called backwards substitution. First, the solution for u_n is obtained trivially from Eq. 6 as $\tilde{b}_n/\tilde{a}_{n,n}$. Next, this solution for u_n is substituted into the $n-1$ equation to solve for u_{n-1}. This is repeated until the whole vector $\vec{u}$ has been solved for. Algorithmically this corresponds to:

$$u_i = \frac{1}{\tilde{a}_{i,i}} \left(\tilde{b}_i + \sum_{j=i+1}^{n} \tilde{a}_{i,j} u_j \right) \tag{9}$$

In addition, we compare the performance of the general algorithm describes above to that of a general tridiagonal matrix as well as the specific tridiagonal matrix that results from Eq. 4 both shown below:

$$\hat{A} = \begin{bmatrix} d_1 & c_1 & 0 & 0 \\ a_1 & d_2 & c_2 & 0 \\ 0 & a_{n-2} & d_{n-1} & c_{n-1} \\ 0 & 0 & a_{n-1} & d_n \end{bmatrix} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \tag{10}$$

In the tridiagonal case we can now use three vectors to store the matrix $\hat{A}$ which greatly decreases the amount of memory needed for a given problem as well as the number of floating point operations (FLOPs). Furthermore, for the specific tridiagonal case only one vector is needed as the $\vec{c}$ vector will never be updated and the $\vec{a}$ vector will become zeroes.

Finally, we compare the performance of our solutions to that of the LU decomposition algorithm from the armadillo library [3]. In LU decomposition the matrix $\hat{A}$ is decomposed as $\hat{A} = \hat{L}\hat{U}$ where $\hat{L}$ and $\hat{U}$ are lower and upper triangular matrices respectively:

$$\hat{L} = \begin{bmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{n,1} & L_{n,2} & L_{n,n} \end{bmatrix} \qquad \hat{U} = \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,n} \\ 0 & U_{2,2} & U_{2,n} \\ 0 & 0 & U_{n,n} \end{bmatrix} \tag{11}$$

The matrix equation $\hat{A}\vec{u} = \vec{b}$ then becomes:

$$\hat{A}\vec{u} = \hat{L}(\hat{U}\vec{u}) = \hat{L}\vec{y} = \vec{b}. \tag{12}$$

This is then solved via forward substitution for $\vec{y}$ and then backwards substitution for $\vec{u}$ which allows for fewer FLOPs but the same amount of memory as the general gauss elimination.

For the source function, $f(x)$, we use $f(x) = 100e^{-10x}$. Analytically, this gives the solution for $u(x)$ as:

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}. \tag{13}$$

To examine the accuracy of our solution the error is defined as:

$$\epsilon_i = log_{10}\left(\left| \frac{u_i - u(x_i)}{u(x_i)} \right| \right) \tag{14}$$

where u_i are the solutions obtained from gauss elimination and $u(x_i)$ are the analytic solutions.

III. RESULTS

For all three algorithms that have been developed, we analyze the performance by the CPU time as well as the

matrix size (n)	10^2	10^3	10^4	10^5	10^6	
Algorithm	Time to solution (sec)					FLOPs
General Gauss	0.007	10.7	N/A	N/A	N/A	$\frac{2}{3}n^3$
Tridiagonal	4.6E-5	1.3E-4	9.6E-4	0.01	0.08	$8n$
Specific tridiag	3.8E-5	8.4E-5	0.001	0.006	0.065	$6n$
LU decomp	2.2E-4	0.02	1.21	N/A	N/A	$2n^2$

TABLE I: Comparison of CPU time for each algorithm with varying matrix size.

n	$log_{10}(h)$	$max(\epsilon_i)$
10	-1	-1.1797
10^2	-2	-3.08804
10^3	-3	-5.08005
10^4	-4	-7.07929
10^5	-5	-9.00487
10^6	-6	-6.77135
10^7	-7	-5.96231

TABLE II: Relative error as a function of step size

number of FLOPs required as a function of matrix size. In addition, we analyze the accuracy of the algorithms by investigating the error as described in the previous section. Finally, we compare the performance of our developed algorithms to that of the LU decomposition.

Using each of the algorithms, we record the timing for varying matrix sizes in Table I. For the general gauss algorithm and LU decomposition matrices are used rather than simply vectors, because of this n values of 10^5 and larger are not computed as the armadillo package does not support such large matrices. In general, the fastest algorithm was that for the specific tridiagonal matrix of this case, followed closely by the general tridiagonal algorithm, then LU decomposition and finally the general gauss elimination. Thinking generally about the problem this is the result we expect, the case that makes the most assumptions about the specific problem takes the least amount of time while the case the assumes the least takes the greatest amount of time.

The ordering of the time to solution for the various solutions is supported even more when the number of FLOPs for each algorithm is investigated as seen in the last column of Table I. The lowest number of operations are performed in the specific tridiagonal algorithm as the values above and below the diagonal in the matrix are known to be -1, so rather than fetching these values from memory and then performing operations (as with the tridiagonal algorithm), we can just use their values. The LU decomposition algorithm as described in Section II skips the forward elimination algorithm (used in the general gauss elimination) which is very computationally heavy and so it has fewer FLOPs.

Finally, we wish to determine the accuracy of the developed algorithms. In Fig. 2 the final solution $u(x_i)$ is plotted for various matrix dimensions along with the analytic solution $u(x)$. From this plot it is clear that a matrix dimension of 100 is plenty large enough to qualitatively match the analytic solution. Quantitatively, we use Equation 14 to determine the error of each algorithm. The maximum error for varying step sizes are given in Table II. As expected, as we increase the matrix dimension (and therefore have a more fine grid) the error decreases. This happens until we reach the overall minimum error at $n = 10^5$ after which the error increases again as we increase the matrix size. This can be seen in Fig 3.

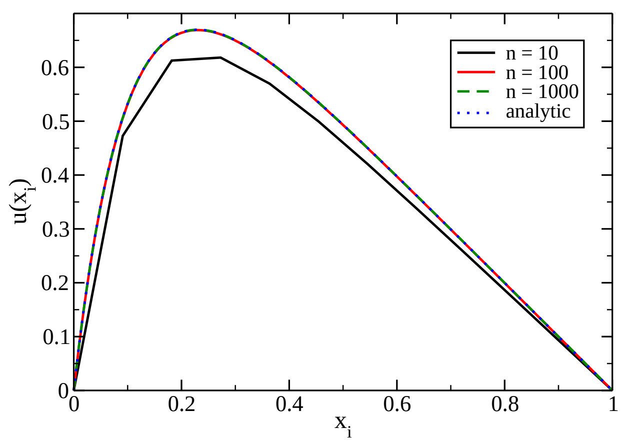

FIG. 2: Solution to diff eqn as a function of matrix size

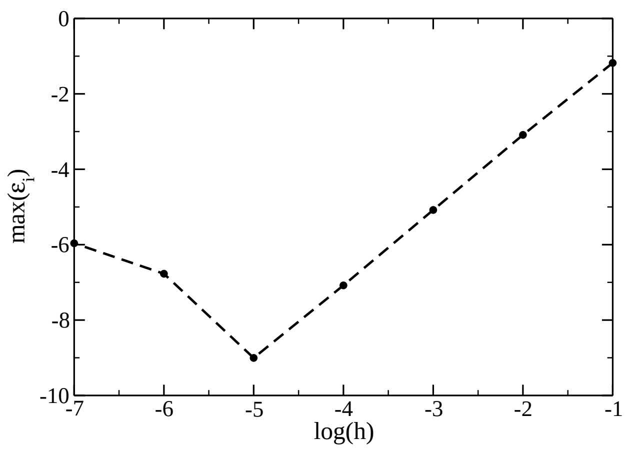

FIG. 3: Error of algorithms as a function of step size.

IV. CONCLUSIONS

In summary, the goal of this work was to implement Gauss elimination in order to solve the Poisson equation for a particular example with the most precision and fasted compute time. The second derivative in the Poisson equation is approximated using a three point formula, which turns the problem into a system of linear equations that can be modeled with a matrix equation.

The resulting matrix is a tridiagonal matrix with two's on the diagonal and negative one's in the off diagonal bands; this allows one to use algorithms that are only tridiagonal matrices as well as this specific tridiagonal matrix. We implemented four algorithms, one for general Gauss elimination, one for a general tridiagonal matrix, one for the specific matrix, and lastly one which used LU decomposition.

First, each of these algorithms was studied for various matrix sizes to determine which had the fastest compute time. In addition, the number of floating point operations for each algorithm was calculated to aid in the explanation for the comparative timing. As expected, the fastest algorithm was the one with the fewest FLOPs which was designed for the specific tridiagonal matrix in this problem. Finally, the error of the solution was explored as a function of step size to determine the optimal matrix size of $n = 10^5$.

There are a number of ways that these code can be improved upon, for example there are more sophisticated algorithms which employ pivoting and pipelining [4, 5]. In addition, because the matrices were implemented with armadillo, matrices larger than 10^4 were too large to use. While the general Gauss elimination algorithm and LU decomposition algorithms were the two slowest, if they were needed due to a dense matrix the arrays for the matrixes would need to be allocated without armadillo. Finally, the general Gauss elimination algorithm is trivially parallelized with OpenMP, which would lead to large increases in performance for large matrices.

Overall, the fastest and most accurate solutions were found using the algorithm specific to the tridiagonal matrix used with $n = 10^5$. This shows that sometimes an algorithm that is tailored to the specific problem can be much more powerful than a general algorithm, even with the loss of generality. In addition, it emphasizes that the smallest step size in a calculation is not always the best as very small step sizes can often lead to large errors.

[1] W. Cheney and D. Kincaid, *Numerical Mathematics and Computing* (Thompson Brooks/Cole, 2008), sixth ed.

[2] (2008), URL `http://riebecca.blogspot.com/2008/12/supercomputing-course-openmp-syntax-and_15.html`.

[3] C. Sanderson and R. Curtin, *Armadillo: a template-based C++ library for linear algebra* (????).

[4] J. A.-L. Goodman, Ph.D. thesis, University of Glasgow (2002).

[5] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, Tech. Rep., University of Tennessee and Inria Bordeaux Sud-Ouest (2013).