Physics 905-Computational Physics
Project 1

Crispin Contreras

February 16, 2016

Abstract

This paper discusses the numerical solution to Poisson's equation which is solved in one dimension and with Dirichlet boundary conditions. The second derivative is approximated with the three point formula and the equations are discretized. From this a set of linear equations are obtained and are solved to get the solution. Two different methods were implemented to solve the set of linear equations. One was a simplified version of Gaussian elimination and the other was using LU decomposition. The two methods were compared by the computation time and the number of floating point operations.

1 Introduction

We begin with the Poisson's equation in three dimensions with a known source. After some manipulation we get to look like a simple second order ordinary differential equation. The second derivative of the equation is approximated by the three point formula and we obtain a set of linear equations. We put them in matrix form which give us the equation $\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}$. By writing it in this form we find that $\mathbf{A}$ is a tridiagonal matrix and this leads us to develop a simplified algorithm for Gaussian elimination. The algorithm is implemented in C++ and we look at different size of the triadiagonal matrix such as 10, 100, and 1000. I then compute the relative error for the different sizes and also take the time that it takes to carry out this algorithm. I compare these parameters with the LU decomposition which I use from the armadillo library and solve the linear equations with lapack. I also compare the number of floating point operations.

2 Theory

We start with Poisson's equation in three dimensions, it reads

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r}).$$

With a spherically symmetric Φ and $\rho(\mathbf{r})$ the equations simplifies to a one-dimensional equation in r, namely

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

which can be rewritten via a substitution $\Phi(r) = \phi(r)/r$ as

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

The inhomogeneous term f or source term is given by the charge distribution ρ multiplied by r and the constant -4π. We will rewrite this equation by letting $\phi \to u$ and $r \to x$. The general one-dimensional Poisson equation reads then $u''(x) = f(x)$.

We then solve the one dimensional Poisson's Equation with Dirichlet boundary conditions by rewriting it as a set of linear equations. To be more explicit we will solve the equation

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0.$$

and we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We have then the boundary conditions $v_0 = v_{n+1} = 0$. We approximate the second derivative of u with the three point formula

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \dots, n, \tag{1}$$

where $f_i = f(x_i)$. Rewriting equation (1) with indeces we have

$$-v_{(i)(i+1)} - v_{(i+1)(i)} + 2v_{ii} = f_i h^2 \tag{2}$$

and set any of the indeces i that are greater than n to zero. These equations lead to the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}, \tag{3}$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & \dots & 0 & -1 & 2 \end{pmatrix} \tag{4}$$

and $\tilde{b}_i = h^2 f_i$.

In our case we will assume that the source term is $f(x) = 100e^{-10x}$, and keep the same interval and boundary conditions. Then the above differential equation has a closed-form solution given by $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$.

3 Methods

Two methods were implemented to solve this equation. One which follows from theory section and is similar to Gaussian elimination and the other is using LU decomposition.

3.1 Method 1: Gaussian Elimination

From equation (2) we set up the set of linear equations. I started by looking at a 4 by 4 matrix and used forward substitution and backward substitution to obtain the solutions. By using looking at a 4 by 4 matrix and using forward and backward substitution I noticed a pattern which come from the fact that our matrix is tridiagonal. I start with a general 4 by 4 matrix

$$\left[\begin{array}{cccc|c} d_1 & e & 0 & 0 & b_1 \\ e & d_2 & e & 0 & b_2 \\ 0 & e & d_3 & e & b_3 \\ 0 & 0 & e & d_4 & b_4 \end{array}\right] \tag{5}$$

and for simplicity set all the off diagonal terms to e since our matrix they are all equal to -1. I multiply the first row by e/d_1 and subtract from the second row to obtain

$$\left[\begin{array}{cccc|c} d_1 & e & 0 & 0 & b_1 \\ 0 & \tilde{d}_2 & e & 0 & \tilde{b}_2 \\ 0 & e & d_3 & e & b_3 \\ 0 & 0 & e & d_4 & b_4 \end{array}\right] \tag{6}$$

I then multiply the second row by $e/\tilde{d}_2$ and subtract it from the third row. Finally I multiply the third row by $e/\tilde{d}_3$ and subtract it from the fourth row. I finally obtain the following matrix

$$\left[\begin{array}{cccc|c} d_1 & e & 0 & 0 & b_1 \\ 0 & \tilde{d}_2 & e & 0 & \tilde{b}_2 \\ 0 & 0 & \tilde{d}_3 & e & \tilde{b}_3 \\ 0 & 0 & 0 & \tilde{d}_4 & \tilde{b}_4 \end{array}\right] \tag{7}$$

The first row is unchanged and I obtain the equations

$$\begin{gathered} \tilde{d}_2 = d_2 - \frac{e}{d_1} \qquad \tilde{d}_3 = d_3 - \frac{e}{\tilde{d}_2} \qquad \tilde{d}_4 = d_4 - \frac{e}{\tilde{d}_3} \\ \tilde{b}_2 = b_2 - \frac{e}{d_1} b_1 \qquad \tilde{b}_3 = b_3 - \frac{e}{\tilde{d}_2} \tilde{b}_2 \qquad \tilde{b}_4 = b_4 - \frac{e}{\tilde{d}_3} \tilde{b}_3 \end{gathered} \tag{8}$$

It's clear to see from these equations the following pattern

$$\tilde{d}_i = d_i + \frac{1}{\tilde{d}_i} \qquad \tilde{b}_i = b_i + \frac{1}{\tilde{d}_{i-1}} \tilde{b}_{i-1} \tag{9}$$

where I have put in the value for e which is equal to -1 and changed d_1 to $\tilde{d}_1$ and similarly for b. From equations (7) and (3) I then obtain the following solutions for v which is down by backward substitution

$$v_4 = \frac{\tilde{b}_4}{\tilde{d}_4} \qquad v_3 = \frac{\tilde{b}_3 - ev_4}{\tilde{d}_3} \qquad v_2 = \frac{\tilde{b}_2 - ev_3}{\tilde{d}_2} \qquad v_1 = \frac{\tilde{b}_1 - ev_2}{\tilde{d}_1} \tag{10}$$

For these I also obtain a general formula which holds for i less than 4

$$v_i = \frac{\tilde{b}_i + v_{i+1}}{\tilde{d}_i} \tag{11}$$

where again I put in the value for for e. In C++ I implemented the forward and backward substitution this way

Project1

```
        //FORWARD SUBSTITUTION
        diagonal_element[0] = 2.0; // Defines First Diagonal element
```

```
for(int i=0; i <= ( size_matrix+1); i++)
{
        //Calculate diagonal terms
        if(i < size_matrix)
        {
                diagonal_element[i+1] = 2.0-(1.0/diagonal_element
                    [i]);

        }//end if statement


        //Calculate known functions
        if( (i>0) && (i<size_matrix))
        {
                known_fun[i] += known_fun[i-1]/diagonal_element[i
                    -1] ;

        }


}//End of Loop

//BACKWARD SUBSTITUTION
numerical_f[size_matrix]= known_fun[size_matrix-1]/
    diagonal_element[size_matrix -1];
numerical_f[0]=0.0;
numerical_f[size_matrix+1]=0;


for(int m = (size_matrix-1); m > 0; m-- )
{
                numerical_f[m] = (known_fun[m-1] + numerical_f[m
                    +1])/diagonal_element[m-1];

}//End of Loop
```

Figure 1: This shows sample code to calculate the forward and backward substitution.

These algorithms solve the equations and give the solution v_i. The number of floating point operations are counted by how many +, -, *, and / are in the equations. For the forward substitution there are a total of 4 and $(n-1)$ equations to solve and for the backward substitution there are $2(n-1)$. This give us a total of $6(n-1)$ floating point operations which is of the order of $\mathcal{O}(6n)$. Comparing this to the full Gaussian elimination which is given by $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ [1] we see that our method is much faster.

3.2 Method 2: LU Decomposition

We start with equation (3) and decompose **A** into a lower triangular matrix **L** and an upper triangular matrix **U**. This gives

$$A = \begin{bmatrix} 1 & 0 & 0 & \dots & \dots & 0 \\ l_{21} & 1 & 0 & \dots & \dots & 0 \\ l_{31} & l_{32} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & \dots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & \dots & u_{2n} \\ 0 & 0 & u_{33} & u_{34} & \dots & u_{3n} \\ \vdots & \vdots & 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & 0 & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix} \tag{12}$$

We can now rewrite (3) as $\mathbf{L}(\mathbf{U}\mathbf{v}) = \tilde{\mathbf{b}}$ and this leads to two equations

$$\mathbf{U}\mathbf{v} = \mathbf{y} \qquad \mathbf{L}\mathbf{y} = \tilde{\mathbf{b}} \tag{13}$$

The first is used to get the solution v and the second for to solve for y. This gives then

$$\begin{aligned} y_1 &= \tilde{b}_1 \\ l_{21}y_1 + y_2 &= \tilde{b}_2 \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + \dots + y_n &= \tilde{b}_n \end{aligned} \tag{14}$$

$$\begin{aligned} u_{11}v_1 + u_{12}v_2 + \dots + u_{n1} &= y_1 \\ &\vdots \\ u_{nn}v_n &= y_n \end{aligned}$$

The LU decomposition(12) and the solution to the set of equations (14) are found by using the armadillo library with lapack. The function to LU decompose is called lu and the one to solve the set of equations is called solve. The following code was used

Project1

```
#include "armadillo"

     lu(L,U,A);
     colvec Y = solve(L, W);
     V = solve(U, Y);
```

Figure 2: This shows a part of the code to find the LU decomposition and solve the set of equations using armadillo.

The number of floating point operations for this algorithm is given by n^2 iterations for forward and backward substitution and $\frac{2}{3}n^3$ for the LU decomposition. This gives a total of $n^2 + \frac{2}{3}n^3$ which is also slower than method 1.

4 Results

I implemented the code(attached in section 8) for the simplified Gaussian elimination and obtained plots to see how accurate the estimates for the size of n=10, 100, and 1000. To make the plots I used gnu plot. I did not make plots for the LU decomposition since they give the same values. I printed the results to 3 different data files Ten.dat, Hundred.dat, and Thousand.dat. The results of the plots are shown in figures 3, 4, and 5 for n=10, 100, and 100 respectively. I also looked at the relative error of the numerical solution with the formula

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

I look at each point but I find that it's the same for all of them. I then compare the errors for each of the steps sizes of n. I also made a plot of $\log_{10}\epsilon_{max}$ vs $\log_{10} h$ to verify that the slope of (1) is 2 since the truncation error is $\mathcal{O}(h^2)$. This can be seen in figure 6 and I summarize these results in table 1.

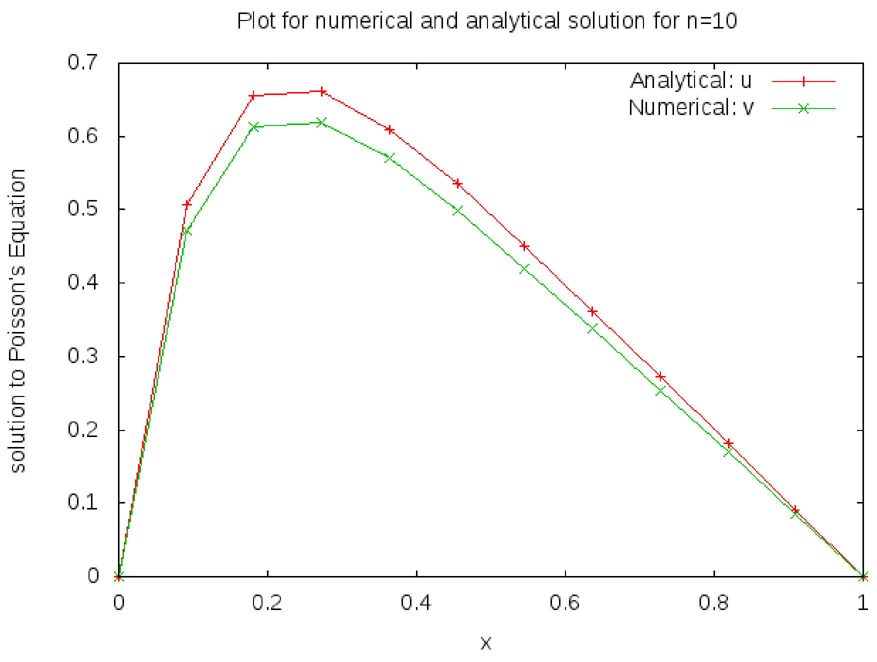

Figure 1: This shows the numerical and analytical solutions for n =10. The results show that the numerical solution is not very accurate.

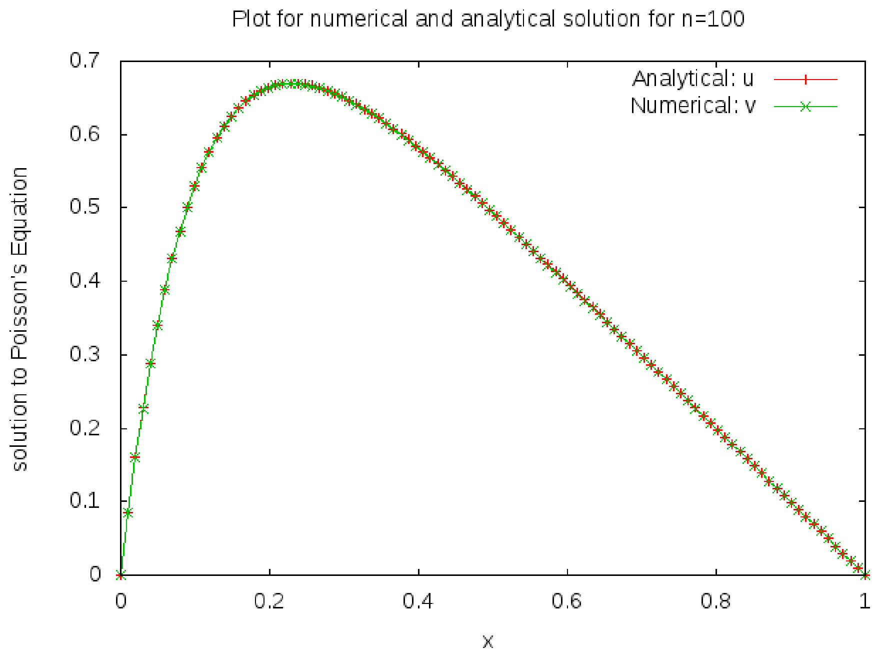

Figure 2: This shows the numerical and analytical solutions for n =100. The results show that the numerical solution is more accurate.

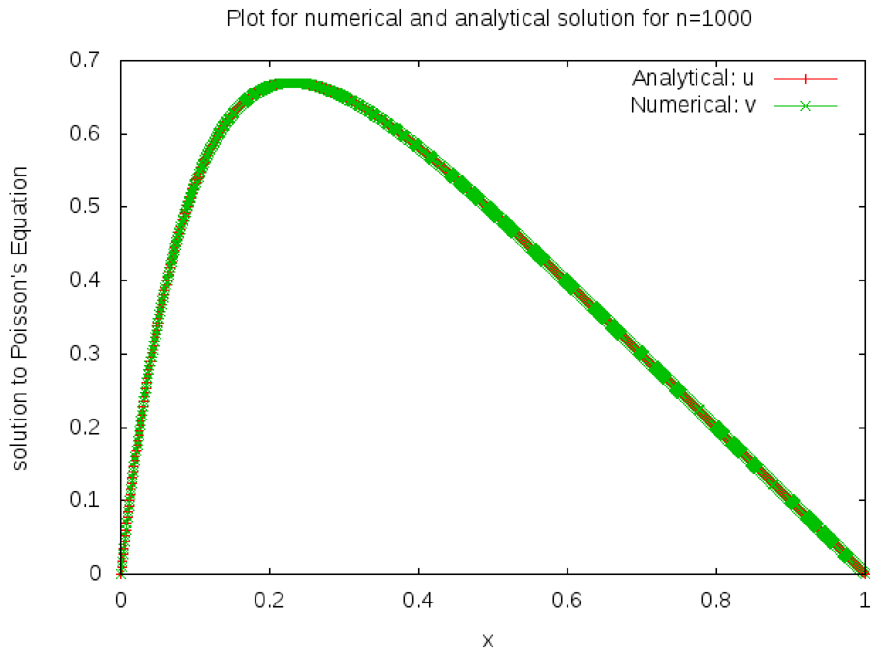

Figure 3: This shows the numerical and analytical solutions for n =1000. The results show that the numerical solution very accurate.

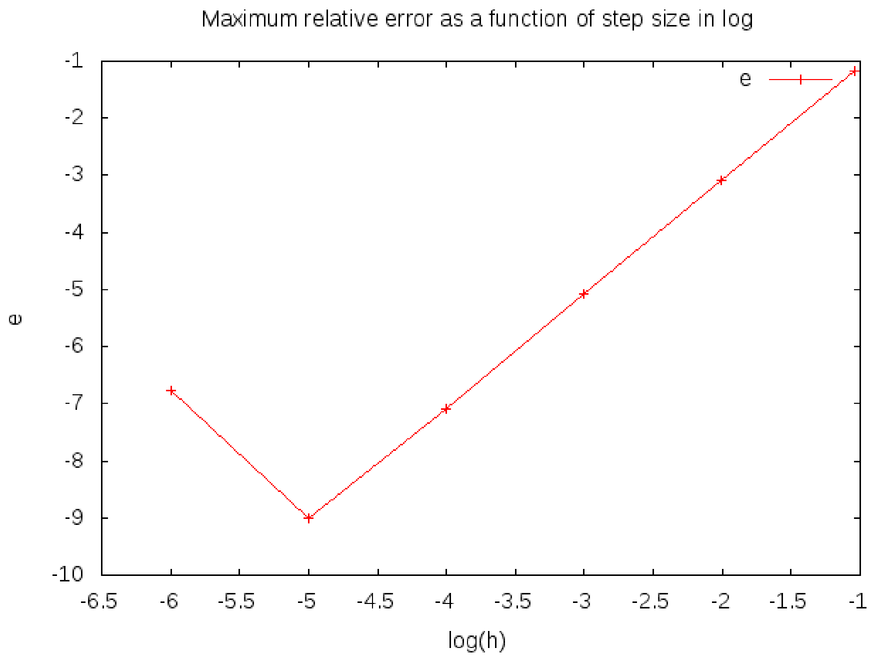

Figure 4: This shows the numerical and analytical solutions for n =1000. The results show that the numerical solution very accurate.

Number of grid points, n	Log of step size, $\log_{10}(h)$	Log of maximal relative error, $\log_{10}(\epsilon_i)$
10^1	-1.0413927	-1.1796978
10^2	-2.0043214	-3.0880368
10^3	-3.0004341	-5.0800516
10^4	-4.0000434	-7.0792852
10^5	-5.0000043	-9.0047052
10^6	-6.0000004	-6.7713552

Table 1: My caption

Number of grid points, n	Calculation time for tridiagonal method[s]	Calculation time for LU[s]
10^1	9.99×10^-7	6.79×10^-5
10^2	3.99×10^-6	3.53×10^-4
10^3	3.19×10^-5	3.73×10^-2
5000	1.91×10^-4	8.25×10^-1
10^5	3.88×10^-4	3.70
10^6	3.20×10^-2	–

Table 2: My caption

5 Discussion

Method 1 worked as I expected, the results are shown in Figures 1, 2, and 3. As the step size increased(n) the value improved until I go to $n = 10^6$ where the relative error started to decrease as shown in Table 1. Additionally from Figure 4 I see a slope of 2 from $n = 10^5 - 10$ which is what is expected from the three point formula with $\mathcal{O}(h^2)$.

6 Conclusion

lasdjflkasjdflk;sajdf askdlfjas;dfkjasldkfj asdlkfjasdl;fkjasdl;fj asdfklasjdflkjasdlfj asdklfjasdlf;jkasdlf;kj asdklfjasl;dfjasd;lk

7 References

[1] M. Hjorth-Jensen. Computational Physics, Lecture Notes Spring 2016.

8 Code Attachment

```
/***************************************************************
***** Author: Crispin Contreras
***** Class:  Physics 905
***** Purpose: solve tridiagonal matrix with LU decomposition (Armadillo)
    and my own method.
***************************************************************/

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <stdio.h>
#include "time.h"
#include "armadillo"

using namespace arma;
using namespace std;
ofstream ofile;

int main()
{
        //DECLARE VARIABLES
        int size_matrix;
        double size_h;
        double *diagonal_element, *known_fun, *numerical_f, *x;
        double *analytic_f, *epsilon;
        char outfilename[30];

        //READ INPUT
        cout<<"Enter the size of the matrix: ";
        cin>>size_matrix;
```

```
cout<<"Enter the name of the output file: ";
cin>>outfilename;


//ALLOCATE MEMORY
size_h = 1.0/(1.0+ size_matrix);
diagonal_element = new double[size_matrix];
known_fun = new double[size_matrix+2];
numerical_f = new double[size_matrix + 2];
x = new double [size_matrix + 2];
analytic_f = new double [size_matrix + 2];



//CALCULATE h^2*f(X) AND x[i]
for (int i=0; i<= (size_matrix+1); i++)
{
        //Define x
        x[i]=i*size_h;

        //Calculate analytic function
        analytic_f[i] = 1.0 -(1-exp(-10))*x[i] - exp(-10*x[i]);

        //Calculating source of charge h^2*f
        if( (i>0) && (i< (size_matrix+1)))
        {
                known_fun [i-1] = size_h*size_h*(100.0*exp(-10.0*
                    x[i]));
        }
}


//Clock
clock_t start, finish;
start = clock();


//FORWARD SUBSTITUTION
diagonal_element[0] = 2.0; // Defines First Diagonal element
for(int i=0; i <= ( size_matrix+1); i++)
{
        //Calculate diagonal terms
        if(i < size_matrix)
        {
                diagonal_element[i+1] = 2.0-(1.0/diagonal_element
                    [i]);

        }//end if statement


        //Calculate known functions
        if( (i>0) && (i<size_matrix))
        {
```

```
                        known_fun[i] += known_fun[i-1]/diagonal_element[i
                            -1] ;

            }


}//End of Loop


//BACKWARD SUBSTITUTION
numerical_f[size_matrix]= known_fun[size_matrix-1]/
    diagonal_element[size_matrix  -1];
numerical_f[0]=0.0;
numerical_f[size_matrix+1]=0;


for(int m = (size_matrix-1); m > 0; m-- )
{
                    numerical_f[m] = (known_fun[m-1] + numerical_f[m
                        +1])/diagonal_element[m-1];

}//End of Loop

finish = clock(); // End clock
double elapsed_t = (double(finish-start)/CLOCKS_PER_SEC);



//OPENING, WRITING TO FILE, AND FINDIND THE RELATIVE ERROR
ofile.open(outfilename);
ofile<<setiosflags(ios::showpoint | ios::uppercase);
ofile<<"#    x:                     Analytic:                 Numerical
    :               Epsilon Largest:                  log(h)"<<endl;

//Define Variable to find relative error
epsilon = new double [size_matrix+2];
epsilon[0] = epsilon[size_matrix+2]=0;//set error here to 0 since
     it's undefined

for(int i=0; i<=(size_matrix+1); i++)
{
          if(i>0)
          {
          epsilon[i]=log10(fabs((numerical_f[i]-analytic_f[i])/
              analytic_f[i]));
          }

          //writing to file
          ofile<<setw(15)<<setprecision(8)<< x[i];
          ofile<<setw(20)<<setprecision(8)<<analytic_f[i];
          ofile<<setw(25)<<setprecision(8)<<numerical_f[i];
          ofile<<setw(28)<<setprecision(8)<<epsilon[i];
          ofile<<setw(29)<<setprecision(8)<<log10(size_h)<<endl;
```

```
}//End Of Loop


//USING LU DECOMPOSITION WITH ARMADILLO
mat A(size_matrix, size_matrix);
mat L(size_matrix, size_matrix);
mat U(size_matrix, size_matrix);
colvec V(size_matrix);
colvec  W(size_matrix);


A.zeros(size_matrix, size_matrix); //Filling the matrix with
    zeros
A(0, 0)=2.0;
W(0) =  size_h*size_h*(100.0*exp(-10.0*size_h));


//filling the rest of the matrix
for(int i=1; i<size_matrix; i++)
{
        A(i, i) = 2.0;
        A(i,i-1)= A(i-1, i) = -1;
        W(i) = size_h*size_h*(100.0*exp(-10.0*size_h*(i+1)));
}


//Clock
start = clock();

lu(L,U,A);
colvec Y = solve(L, W);
V = solve(U, Y);

finish = clock(); // End clock
double elap_tLU = (double(finish-start)/CLOCKS_PER_SEC);

cout << setiosflags(ios::showpoint | ios::uppercase);
cout << setprecision(50) << "Elapsed time mine: "<<elapsed_t<<"
    LU time: "<<elap_tLU<<endl
ofile << "# LU decomp (used to verify if the results are the same
    )" << endl;
for(int i=0; i<size_matrix; i++)
{
        epsilon[i]=log10(fabs((V[i]-analytic_f[i+1])/analytic_f[i
            +1]));
        //writing to file
        ofile<<setw(7)<<setprecision(8)<<"# "<<x[i+1];
        ofile<<setw(10)<<setprecision(8)<<"# "<<analytic_f[i+1];
        ofile<<setw(15)<<setprecision(8)<<"# "<<V[i];
        ofile<<setw(20)<<setprecision(8)<<"# "<<epsilon[i]<< endl
            ;
}//End Of Loop
```

```
        ofile.close();

        //free memory
        delete [] diagonal_element;
        delete [] known_fun;
        delete [] x;
        delete [] numerical_f;
        delete [] analytic_f;
        delete [] epsilon;
        return 0;

}//End main program
```