Numerical Solution Algorithms using the Finite Difference Method

Nicolas Dronchi*
Michigan State University
Dronchin@msu.edu
Git Hub link: https://github.com/dronchin/Project1
(Dated: February 5, 2018)

Three different algorithms for the 2 point Dirichlet boundary condition problem were created using LU decomposition, the general algorithm for solving a tridiagonal matrix, and a simplified algorithm for the special case of a tridiagonal Toeplitz matrix. The run times for matrices of 10^8 x 10^8 were on the order of seconds for the simple and general matrix. The error of the algorithm was minimized at a size of 10^6 x 10^6

Usage: Numerical solutions to second derivative 2 boundary condition problems like Poisson's equation and Laplace equations. Tridiagonal matricies can also solve problems including cubic spline problems.

I. INTRODUCTION

Having an accurate and quick two point boundary problem solver is important for many real like applications. A few of the applications include solving for Poisson's equation, solutions to Laplace equation, and can even include uses in cubic spline algorithms. While there are occasionally definite, analytical solutions to these problems, they only arise if you feed them perfect conditions. For most scenarios, these perfect conditions aren't realistic and the analysis to the problem requires an numerical solution with high accuracy.

A Dirichlet boundary condition, also known as a fixed boundary condition, is used problems where both ends of the solution are known and fixed in place. In the case of this project, the boundary conditions are held at 0 for both ends. This is chosen for simplification though as any number could be selected for the boundary condition on either side.

The first application mentioned was Poisson's equation[1]. With charge distribution $\rho(r)$ and electrostatic potential Φ defined by:

$$\Delta^2\Phi = -4\pi\rho(r)$$

In the radial dimension, this can be simplified using $\Phi(r) = \phi(r)/r$ and you get the following:

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

With this we can make one more simplification of $\phi \to u$ and $r \to x$ to define the problem as a second derivative and a function.

$$-u''(x) = f(x)$$

For the application to Laplace's equation, we only need to change the $f(x)$ to zero so that we get the following:

$$-u''(x) = f(x)$$

For this project I focus on using the finite difference method to get a numerical solution to equations of the form

$$-u''(x) = f(x)$$

using three different methods. The first method is a brute force solving of the problem using LU decomposition on the order of $O(\frac{2}{3}n^3)$. The second method is algorithm set up to solve any tridiagonal matrix on the order of $O(8n)$. The third method is a algorithm specifically set up to use the second order finite difference method to quickly solve the problem on $O(4n)$. The examination of run times and associated error to the solution are examined in this paper.

II. METHODS

The first order finite difference method is defined as

$$u'(x) = \frac{u(x+h) - u(x)}{h} + O(h)$$

where the error $O(h)$ is scaled with the size of the step size h. [2,3]

Here we use the second order finite difference method, which is defined as

$$u''(x) = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + O(h^2)$$

where the error $O(h^2)$ is scaled with the square of the step size h.

* Also at Computational Physics - PHY480

The solution of this problem begins with picking the number of steps you would like to take between $x \in (0,1)$ so that discretize x as $x_i = x_0, x_1, x_2, ..., x_n$. With x discretized, you discretize the finite differece method along with it so that you get

$$u_i'' = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} \qquad for \qquad i = 1, ..., n$$

Because of boundary conditions are defined as $u(0) = u(1) = 0$, we ignore them when setting up the system of linear equations of the form $Ax = b$ where A is an nxn matrix equal to the following:

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ 0 & \dots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & \dots & -1 & 2 \end{bmatrix}$$

b is a matrix equal where $b_i = h^2 f_i$ and x is the solutions to the second order derivative to f_i at the discretized values of x_i. This system of equations is the goal of all three solvers presented in this project.

The LUdecompsolver solves the system of equations treating each as the presented matrix, and then uses the armadillo library to solve the system using LU decomposition within the command solve(). To analyze this method, the solve() command was timed for different sizes of n x n matrices. The error of the numerical solution was analyzed by using $f(x) = 100e^{-10x}$ because the analytical solution is know to be
$f''(x) = 1 - (1 - e^{-10})x - e^{-10x}$

The GeneralTridiag solver creates a new system which connects the project to uses other than the finite difference method. This algorithm can still calculate the numerical solution of the finite difference method but it isn't tied down to only using the finite difference method. This program is used for comparing the efficiency of calculating the general case verses calculating the specific case of finite difference.

The SimpleTridiag solver calculates the finite difference solution for the the speficic case of $f(x) = 100e^{-10x}$ as noted above. The importance of this algorithm is in the speed of calculating the answer to the numerical solution.

For both the GeneralTridiag and the SimpleTridiag, the solution is calculated using both Gaussian Elimination and the back substitution method. For the general case of A

$$A = \begin{bmatrix} d_1 & e_1 & 0 & \dots & \dots & 0 \\ c_1 & d_2 & e_2 & 0 & \dots & \dots \\ 0 & c_2 & d_3 & e_3 & 0 & \dots \\ 0 & \dots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & c_{n-2} & d_{n-1} & e_{n-1} \\ 0 & \dots & \dots & \dots & c_{n-1} & d_n \end{bmatrix}$$

Gaussian elimination of this matrix produces the following row echelon form matrix $\widetilde{A}$ with the answer x and matrix b also being augmented.

$$\begin{bmatrix} d_1 & e_1 & 0 & \dots & \dots & 0 \\ 0 & \widetilde{d_2} & e_2 & 0 & \dots & \dots \\ 0 & 0 & \widetilde{d_3} & e_3 & 0 & \dots \\ 0 & \dots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & \widetilde{d_{n-1}} & e_{n-1} \\ 0 & \dots & \dots & 0 & 0 & \widetilde{d_n} \end{bmatrix} x = \begin{bmatrix} b_1 \\ \widetilde{b_2} \\ \vdots \\ \vdots \\ \vdots \\ \widetilde{b_n} \end{bmatrix}$$

The augmented diagonal $\widetilde{d}$ and the matrix $\widetilde{b}$ for $i = 2, ..., n$ can easily be calculated with the following:

$$\widetilde{d_i} = d_i - \frac{e_{i-1}^2}{\widetilde{d_{i-1}}}$$

$$\widetilde{b_i} = b_i - \frac{e_{i-1} b_{i-1}}{\widetilde{d_{i-1}}}$$

Once forward substitution is solved for, you can then use backwards substitution. Back wards substitution starts with $x_n = \frac{\widetilde{b_n}}{\widetilde{d_n}}$ and works backwards going to $n-1, n-2, ..., 1$. The following is used for calculating x_i for $i = n-1, n-2, ..., 1$:

$$x_i = \widetilde{b_i} - \frac{c_i \widetilde{x_{i+1}}}{d_i}$$

III. RESULTS AND ANALYSIS

Two measures of the algorithm were recorded and compared between each other. The first measure of the algorithms was the timing to run the algorithm at different sizes of n x n matrices. The results of this can be seen in figure 1 where the different algorithms run times for only the algorithm are compared to the log10() of the size.

The LU decomposition was only able to run up to the size of about 10000 x 10000. Anything past that created run times that weren't feasible on my current set up. As you can see, this dat backs up that the LU

timing.png

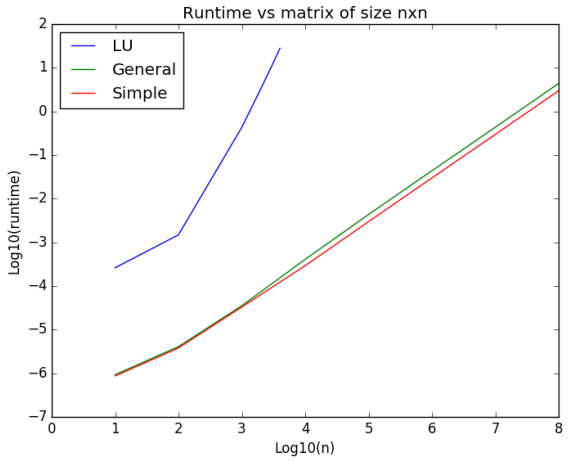

FIG. 1. Run times of the different algorithms with the LU decomposition solver in blue, the General solver in green, and the Simple solver in red

error.png

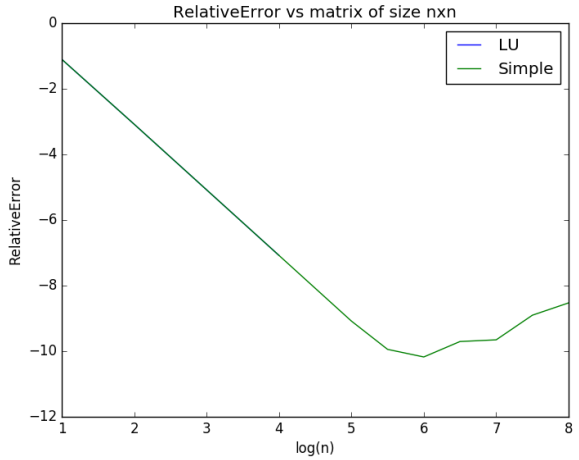

FIG. 2. Error of the associated algorithms. The error for LU decomposition solver overlaps the error of the simple. The error of the general solver is the same as the simple.

decomposition algorithm is on the order of $O(\frac{2}{3}n^3)$. This is one of the main motivations to create algorithms for general and the simplified tridiagonal solvers. The brute force LU decomposition way will only work up to a certain size of matrix.

The run times of the General and Simple algorithm were linear after log10(n) gets past 2. These algorithms were also able to easily get to larger sizes of matrices, even up to 10^8 x 10^8. The simple solver stayed true to being faster than the general solver but only by a small margin. This effected the times at the largest matrices but wasn't a large difference at smaller values of n. Overall this lends support to the statement that the General solver was on the order of $O(8n)$ while the Simple solver was on the order of $O(4n)$.

The error of the associated algorithms were all the same between programs. The numbers being manipulated are all the same and are being manipulated in generally the same way to produce the same end results. This is the cause of the overlapping of the LU error and the Simple error in 2 as well as the error for the general case not being plotted.

The error followed a linear trend with a slope of -2. This slope of the error can be attributed back to how the finite difference method was constructed to give an error on the order of $O(h^2)$. This relates to the slope of -2 because the log of $log10(h^2) = 2 * log10(h)$. This linear trend followed until the range of $log10(n) = 5.5$. Then the error broke down and started increasing again. This can be attributed to the rounding error of double floating point numbers. The computer can only reach a specific accuracy when the numbers get either too small or too large because it represents the numbers on a scale of base 2. Because of the error starting to play in past a $log10(n) = 5.5$, I would set the limit of usable matrix size to $10^{5.5}$ x $10^{5.5}$.

IV. CONCLUSION

The finite difference method provides an accurate numerical method for solving for the second derivative of a function. This is useful in 2 point Dirichlet boundary condition problems including the Poisson equation or Laplace equations. The general algorithm also has uses in calculating cubic splines for a set of data.

As shown in the paper, the finite difference method can be set up to solve for the second derivative of a function. These functions can be combined into a system of equations that can be solved to find all of the solutions to the equation at different values of x. The matrix equation $Ax = b$ is used where A is a n x n as follows [2]:

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ 0 & \dots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & \dots & -1 & 2 \end{bmatrix}$$

and b is created from values of $f(x)$ at different x values.

This system of equations was solved with three different methods. The first method is through LU decomposition through the armadillo solve() function. This was confirmed to be on the order of $O(\frac{2}{3}n^3)$ based upon the timing shown in 1. This algorithm shared the

trend of error being on the order of h^2. This algorithm is very effective for small systems that don't require extreme accuracy.

The other algorithms, the general tridiagonal solver and the simple tridiagonal solver, were solved using Gaussian elimination, also called forward substitution, and then using back substitution to solve for values of x. 1 gave supporting evedence that the general algorithm is $O(8n)$ and simple algorithm is $O(4n)$. These algorithms relative error was also on the order of h^2 just like the LU decomposition solver. This algorithm is very effective for matrices up to 10^6 x 10^6 where any larger produces floating point operation errors which increase the total error of the solution.

V. CITATIONS

1] J.S.C. Prentice (2012) Relative and absolute error control in a finite-difference method solution of Poisson's equation, International Journal of Mathematical Education in Science and Technology, 43:5, 684-694, DOI: 10.1080/0020739X.2011.622800

2] Hjorth, Morten (2018) Project1 description can be found on github at https://github.com/CompPhysics/ComputationalPhysicsMSU/tree/master/doc/Projects/2018/Project1/pdf

3] Urroz, Gilberto (2004) Numerical Solution to Ordinary Differential Equations `http://ocw.usu.edu/Civil_and_Environmental_Engineering/Numerical_Methods_in_Civil_Engineering/ODEsMatlab.pdf`