Project 1

Parker Brue[1]

[1] *Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48823*

We present our Ferrari algorithm for solving linear equations. We wrote the one-dimensional Poisson equation, utilizing Dirichlet boundary conditions, as a linear set of equations and as a tridiagonal matrix. We compared a specialized algorithm for solving the tridiagonal matrix to an LU-decomposition of said matrix. Our best algorithm, the specialized solver, runs as $4n$ FLOPS with n the dimensionality of the matrix.

INTRODUCTION

As an introduction to the central ideas of the class, we studied the one-dimensional Poisson equation with Dirichlet boundary conditions. Namely, transforming the differential equation into a set of linear equations, and consequently, a matrix. We implemented a general algorithm, a specialized algorithm, and a LU-decomposition algorithm. In the course of this report, we introduce the theoretical model and the different algorithms we developed, then discuss the results of the different methods. Important points of comparison lie with relative error and relative speed of the calculation due to FLOPS.

THEORY

Theoretical solution of the one dimensional Poisson equation

In general, the one dimensional Poisson equation reads as follows:

$$-u^{''}(x) = f(x) \tag{1}$$

Through discretized approximation of u, we can solve for f using a set of linear equations:

$$f(x) = -\frac{v_{i+1} + u_{i-1} - 2u_i}{h^2}; i = 1, ..., n \tag{2}$$

Using Dirichlet boundary conditions, $u_0 = u_{n+1} = 0$, We can then rewrite this as a set of linear equations in the form of a tridiagonal matrix:

$$\hat{A} \cdot \hat{u} = \hat{f} \tag{3}$$

Consequently, we can solve these linear equations through forward and backward substitution.

Specific Poisson equation

For our purposes of solving the Poisson equation, we assume a function

$$f(x) = 100e^{-10x} \tag{4}$$

and a closed form solution with the Dirichlet boundary conditions:

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{5}$$

Solving a Tridiagonal Matrix

This is a general tridiagonal matrix, in an equation similar to (3):

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_{i-n} \\ 0 & 0 & e_{i-n} & d_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_n \end{bmatrix}$$

where d_i are the diagonal matrix elements and e_i are the off diagonal matrix elements. We can reduce this generalized matrix into an upper triangular matrix by first using forward substitution to yield:

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_{n-1} \\ 0 & 0 & 0 & \tilde{d}_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_n \end{bmatrix}$$

The off-diagonal elements e_i are unchanged. The other elements are given by:

$$\tilde{f}_i = f_i - \frac{\tilde{f}_{i-1} e_{i-1}}{\tilde{d}_{i-1}}; \tilde{d}_i = d_i - e_{i-1}^2 / \tilde{d}_{i-1} \tag{6}$$

After back substitution, the elements of solution vector u are given by:

$$u_i = (\tilde{f}_i - e_i u_{i+1}) / \tilde{d}_i \tag{7}$$

We were able to use the knowledge of these substitutions to create a special program to solve for our specific set of linear equations. However, a tridiagonal matrix can be solved using LU decomposition as well.

LU-decomposition

$$\mathbf{A} = \mathbf{LU} \tag{8}$$

A, the matrix can be decomposed into the product of **U**, upper triangular matrix and **L**, a lower triangular matrix

$$\mathbf{LUx} = \mathbf{b} \tag{9}$$

Which allows you to replace **A** and then use either triangular matrix to solve the problem.

$$\mathbf{Ly} = \mathbf{b} \tag{10}$$

$$\mathbf{Ux} = \mathbf{x} \tag{11}$$

ALGORITHMS

The matrix that represents the set of linear equations referred to in equation (2) appears here as a 4x4 matrix:

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

This form can be generalized to any size nxn matrix, where the diagonal elements d_i each equal 2 and the off diagonal elements e_i each equal -1.

Equation (6) can be generalized for this specific matrix, yielding:

$$\tilde{d}_i = \frac{i+1}{i}; \tilde{f}_i = f_i + \tilde{d}_{i-1}\tilde{f}_{i-1} \tag{12}$$

Equation (7) can be reduced to:

$$u_i = (\tilde{f}_i + u_{i+1})/\tilde{d}_i \tag{13}$$

By computing d_i once per iteration in a loop saves one operation in the forward substitution. Replacing the off diagonal elements with -1 when possible, and calculating the diagonal elements using the pattern seen in equation (12) also reduces the number of FLOPs. In this way, the specialized application of the forward and backward substitution algorithms are reduced from approximately 9n operations down to 4n.

METHODS

Relative error

In order to properly test the effectiveness of our algorithm, we are measuring the closeness, or relative error of our analytic solution:

$$\epsilon_i = log_{10}(|\frac{v_i - u_i}{u_i}|); i = 1, ..., n \tag{14}$$

u_i is the analytic solution, and v_i is the numerical solution.

RESULTS AND DISCUSSIONS

We ran the program with the step sizes of n = 1, 2 , and 3, corresponding to a 10x10, 100x100, and 1000x1000 matrix. We also optimized our program through specialization. Diagonal matrix elements were set to 2 and off-diagonal elements were set to -1. FLOPS were reduced from 9 in the general case to 4 in the specialized case. The program was much more efficient, and faster as a consequence to the precalculation.

Analyzing step size, there is a clear difference from N=10^1 and N=10^2, while N=10^2 to N=10^3 does not seem to superficially change as much as is demonstrated in the graphs below.

FIG. 1. n = 1. The x axis is the x values in the range [0,1] and the y axis is the resulting u values.

FIG. 2. n = 2. The x axis is the x values in the range [0,1] and the y axis is the resulting u values.

FIG. 3. n =3. The x axis is the x values in the range [0,1] and the y axis is the resulting u values.

Further investigation shows that the overabundance of data points does give us the full curve, it also results in a much larger error, as shown in the table below. There seems to be a saddle point situated around N=10^2, wherein the calculation is optimized in regard to completeness of the graph and uncertainty from the numerical solution.

Step Size (n)	Max Relative Error ($\lvert\epsilon\rvert$)
1	1.101
2	3.079
3	5.079
4	7.079
5	9.079
6	11.50
7	12.27

The above table shows us the absolute valued relative error results from varying step sizes of $N = 10^n$ from 1 to 7. The step size error for n=7 seems to be the breaking

point, the error starts to widely fluctuate. Therefore, the largest step size we would reccomend for an accurate result is n=6.

For further analysis, we included a timer in both the specialized program and the LU program, and compared the timing values for, for as large square matrices as either could reasonably calculate.

Column Size	Specialized Time (s)	LU Time (s)
10^1	0.000430	0.00127
10^2	0.001830	0.09897
10^3	0.009720	31.4228
10^4	0.0750	–
10^5	0.7340	–
10^6	7.820	–
10^7	92.2239	–

We found that specialized time was vastly more efficient over the LU time, as expected. We were unable to compute a LU matrix at $N = 10^4$, initializing the program with this input would immediately sieze the program up. For the specialized solver, $N = 10^7$ was the most reasonable limit, guessing at the size of the output file for $N = 10^7$, 400MB, we can assume that $N = 10^8$ would not only take several hundred seconds to compute, but would result in a file size well over a GB, if not several.

CONCLUSIONS

We found that yes, LU is bad

[1] G. A. Miller, A. K. Opper, and E. J. Stephenson, Annu. Rev. Nucl. Sci. **56**, 253 (2006).