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

We can write a simple program that uses forward (6) and backward (7) substitutions to solve the set of linear equations.

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

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

where d_i are the diagonal matrix elements and e_i are the off diagonal matrix elements.

We were able to create another program that performed matrix mathematics to solve for the set of linear equations. Taking a tridiagonal matrix, we can solve it using LU decomposition.

FIG. 1. The code to find the inverse of a matrix utilizing LU decomposition

ALGORITHMS

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:

$$\epsilon_i = log_{10}(|\frac{v_i - u_i}{u_i}|); i = 1, ..., n \tag{8}$$

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

RESULTS AND DISCUSSIONS

By running the program for different step sizes with x in a range [0,1], we can see how the data changes with data points. We can see from the plot, that without enough data points we cannot produce the full curve, but with too many data points we recieve a larger error. This provides us with the knowledge that there is a saddle point in the step size where we optimize both uncertainty and completeness, in our case it appears to be the N=100 or N=10^2 step size.

We ran the program with the step sizes of n = 1, 2 , and 3, corresponding to a 10x10, 100x100, and 1000x1000 matrix.

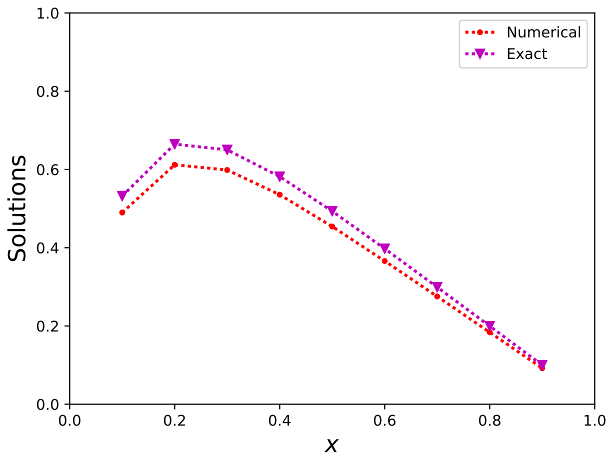

FIG. 2. Comparison between different step sizes. The x axis are the x values in the range [0,1] and the y axis are the resulting u values.

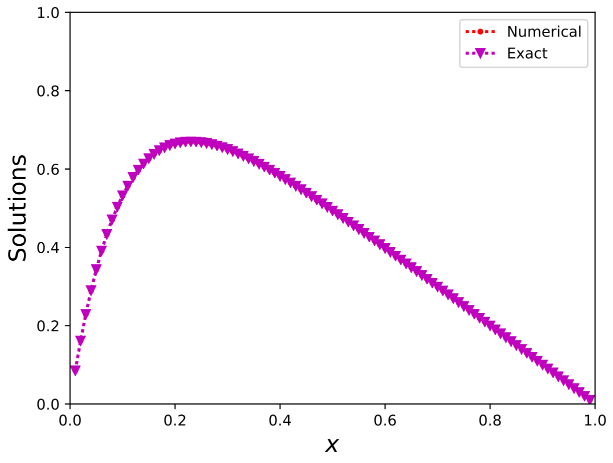

FIG. 3. Comparison between different step sizes. The x axis are the x values in the range [0,1] and the y axis are the resulting u values.

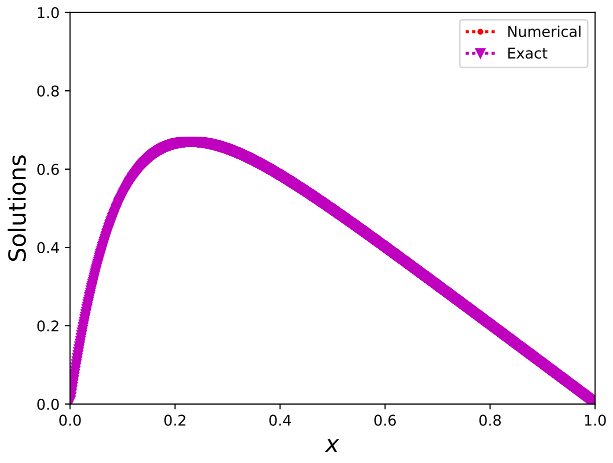

FIG. 4. Comparison between different step sizes. The x axis are the x values in the range [0,1] and the y axis are the resulting u values.

We can further optimize our program by solving it for a special case, in which the diagonal matrix elements are equal to 2 and the off diagonal matrix elements are equal to -1. This allows us to reduce the number of floating point operations per second (FLOPS) from an original 9 down to 4. This allows us to run our program much faster by precalculating coefficients.

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

By including equation in our code, we can calculate the relative error for a certain step size. The was done for multiple step sizes, $N = 10^n$ n=1,...,7 (Table above). It should be noted that these are the absolute values and averages of the errors. It should also be noted that it breaks down for n=7, as there is a large fluctuation in the error between 10 and infinity. It appears that n=6 is the largest order of step sizes that we can accurately use to describe the solutions to the function.

We may also include a timer to calculate the time required to compute the solutions. By adding a timing mechanism, we calculated the time required to solve square matrices with various column lengths.

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	–

FIG. 5. Exact and numerial solutions for $n = 10$ mesh points.

CONCLUSIONS

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