

# Project 1

Jane Kim\*

Physics 480: Computational Physics

(Dated: February 5, 2018)

accurate and informative, state main findings and specific results. 5 pts

## I. INTRODUCTION

motivate reader, give a status of the problem and major objectives. state what you have done and give outline of report. 10 pts

The goal of this project was to develop and compare several methods of solving the one-dimensional Poisson equation with Dirichlet boundary conditions. More explicitly, we aimed to solve

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

for some given function  $f(x)$ . We first rewrote this problem as a set of linear equations by approximating the second derivative of  $u(x)$  using the discretized approximation to  $u(x)$ , namely  $v_i = u(x_i)$ , where  $x_i = ih$  is defined for  $i = 0, \dots, n+1$  and  $h = 1/(n+1)$  is the step size. The Poisson equation then becomes

$$-\frac{v_{i-1} - 2v_i + v_{i+1}}{h^2} = f(x_i), \quad i = 1, \dots, n. \quad (2)$$

Hence we obtained a set of  $n$  linear equations given by

$$A\vec{v} = \vec{b},$$
$$\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ f(x_n) \end{bmatrix}. \quad (3)$$

There exists many different methods for solving Equation (3) for  $\vec{v}$ , but the efficiency of the methods discussed in this report were mainly dependent on the initial assumptions of the form of  $A$ . Four different algorithms were implemented according to the following assumptions: (i)  $A$  is dense, (ii)  $A$  is tridiagonal, (iii)  $A$  is symmetric and tridiagonal, and (iv)  $A$  has the form shown in Equation (3).

In Section II, the four different algorithms are presented. In Section III, FINISH

## II. METHODS

discuss the methods used and their basis/suitability. give a presentation of algorithm and possible errors and flops! discuss assumptions of algorithms and titles of subsections below (correspond to programs in src). 20 pts

### A. LU Decomposition with Armadillo

This method uses standard LU decomposition to solve Equation (3) with no prior assumptions about the form of  $A$ . While this algorithm is simple to write (especially with Armadillo) and is more efficient than Gaussian elimination, the fact that  $A$  has more zeros than non-zero elements results in a relatively long computation time and storage issues for large  $n$ . For instance, running the program with  $n = 10^5$  resulted in an early termination due to lack of available memory. The number of floating point operations (FLOPS) for LU decomposition is  $\sim \frac{2}{3}n^3$  for large  $n$ . CITE PROF

### B. General Tridiagonal Matrix

By assuming  $A$  is tridiagonal, we can store the three main bands as  $n$ -dimensional vectors instead of storing the entire matrix. The main diagonal vector was denoted as  $\vec{d}$ , and the lower and upper diagonals were denoted as  $\vec{c}$  and  $\vec{e}$ , respectively. The algorithm was a simplified form of Gaussian elimination and consisted of two steps. In the forward substitution step, the lower diagonal elements  $c_i$  were eliminated for  $i = 2, \dots, n$  by Gaussian elimination, leaving 1's along the diagonals. With this construction, only  $\vec{e}$  and  $\vec{b}$  needed to be changed:

$$e_i = \begin{cases} \frac{e_i}{d_i}, & i = 1, \\ \frac{e_i - c_{i-1}e_{i-1}}{d_i - c_{i-1}e_{i-1}}, & i = 2, \dots, n, \end{cases} \quad (4)$$

$$b_i = \begin{cases} \frac{b_i}{d_i}, & i = 1, \\ \frac{b_i - b_{i-1}c_{i-1}}{d_i - c_{i-1}e_{i-1}}, & i = 2, \dots, n. \end{cases} \quad (5)$$

Notice that the denominators for the  $i > 1$  case is the same for both  $e_i$  and  $b_i$ . So this value was precalculated to reduce the number of floating point operations. The forward substitution step, therefore, consists of  $\sim 6n$  FLOPS for large  $n$ .

The backward substitution step solves for  $\vec{v}$  using the new values of  $\vec{e}$  and  $\vec{b}$ :

$$v_i = \begin{cases} b_i, & i = n, \\ b_i - e_i v_{i+1}, & i = n-1, \dots, 1, \end{cases} \quad (6)$$

thus it uses  $\sim 2n$  FLOPS. So the algorithm employs  $\sim 8n$  FLOPS all together.

### C. Symmetric Tridiagonal Matrix

We can simplify the previous algorithm even further. If we assume that  $\vec{c} = \vec{c}$ , then the lower diagonal elements can be eliminated while keeping the upper diagonal elements fixed. Then the forward substitution step is given by

$$d_i = \begin{cases} d_i, & i = 1, \\ d_i - \frac{c_{i-1}^2}{d_{i-1}}, & i = 2, \dots, n, \end{cases} \quad (7)$$

$$b_i = \begin{cases} b_i, & i = 1, \\ b_i - \frac{c_{i-1}b_{i-1}}{d_{i-1}}, & i = 2, \dots, n. \end{cases} \quad (8)$$

Again, the factor  $\frac{c_{i-1}}{d_{i-1}}$  can be precalculated so that the forward substitution step uses  $\sim 5n$  floating point operations.

The backward substitution step is as follows:

$$v_i = \begin{cases} \frac{b_i}{d_i}, & i = n, \\ \frac{b_i - c_i v_{i+1}}{d_i}, & i = n-1, \dots, 1. \end{cases} \quad (9)$$

This step requires  $\sim 3n$  FLOPS so that the overall algorithm uses about the same number of FLOPS as the general tridiagonal case.

### D. Simple Tridiagonal Matrix

Finally, let  $d_i = 2$  and  $c_i = -1$  for  $i = 1, \dots, n$  in the previous algorithm. By inspection, we find that the forward substitution step for  $\vec{d}$  follows a predictable pattern, where  $d_i = \frac{i+1}{i}$ . So we can precalculate the entire vector  $\vec{d}$  in the set-up step, rather than the forward substitution step. Then the forward and backward substitution steps only consist of

$$b_i = \begin{cases} b_i, & i = 1, \\ b_i + \frac{b_{i-1}}{d_{i-1}}, & i = 2, \dots, n, \end{cases} \quad (10)$$

$$v_i = \begin{cases} \frac{b_i}{d_i}, & i = n, \\ \frac{b_i + v_{i+1}}{d_i}, & i = n-1, \dots, 1, \end{cases} \quad (11)$$

which together uses  $\sim 4n$  FLOPS. Clearly, this is the least number of floating point operations among the four methods discussed in this report, but this simple algorithm is the least versatile, as it solves Equation (3) only.

Since many programming languages use zero-based numbering, one may need to alter the equation for  $d_i$  slightly:  $d_i = \frac{i+2}{i+1}$ ,  $i = 0, \dots, n-1$ .

### III. IMPLEMENTATION

discuss how you implemented algorithms, tested algo, timings of algos, benchmark calculations. clarity/readability of code? 20 pts.

All four programs require a maximum power of 10 and a file name in the command line. Then the programs loop through successive powers of 10 for the number of grid points  $n$ . In each pass of the loop, a final file name is designated with the corresponding power of 10.

Each program was tested with  $f(x) = 100e^{-10x}$  so that Equation (1) had a known analytic solution  $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$  with which we could compare the discretized approximation  $v_i$ . It is important to note that the set up step was not included in the timed portion of the program. Only the algorithm portion, either forward and backward substitution or call to function, of each program was timed.

In addition, a vector  $e\vec{r}r$  was created to store the relative errors between  $u(x_i)$  and  $v_i$ . Armadillo allows for easy extraction of the maximum, minimum, and average relative error values. The three programs in which Armadillo was not used included explicit calculations of these values. Finally, the function inputs  $x_i$ , the approximate solution  $v_i$ , the exact solution  $u(x_i)$ , and the logarithm of the relative errors were printed to file. The number of grid points  $n$ , the time used for computation, and the maximum, minimum, and average relative error values were also printed to the beginning of each file.

For methods B-D, example file outputs for  $n = 10^1, \dots, 10^6$  can be found in the 'benchmark' folder. For method A,  $n > 10^3$  proved to be slow and problematic, so only three benchmark calculations were provided.

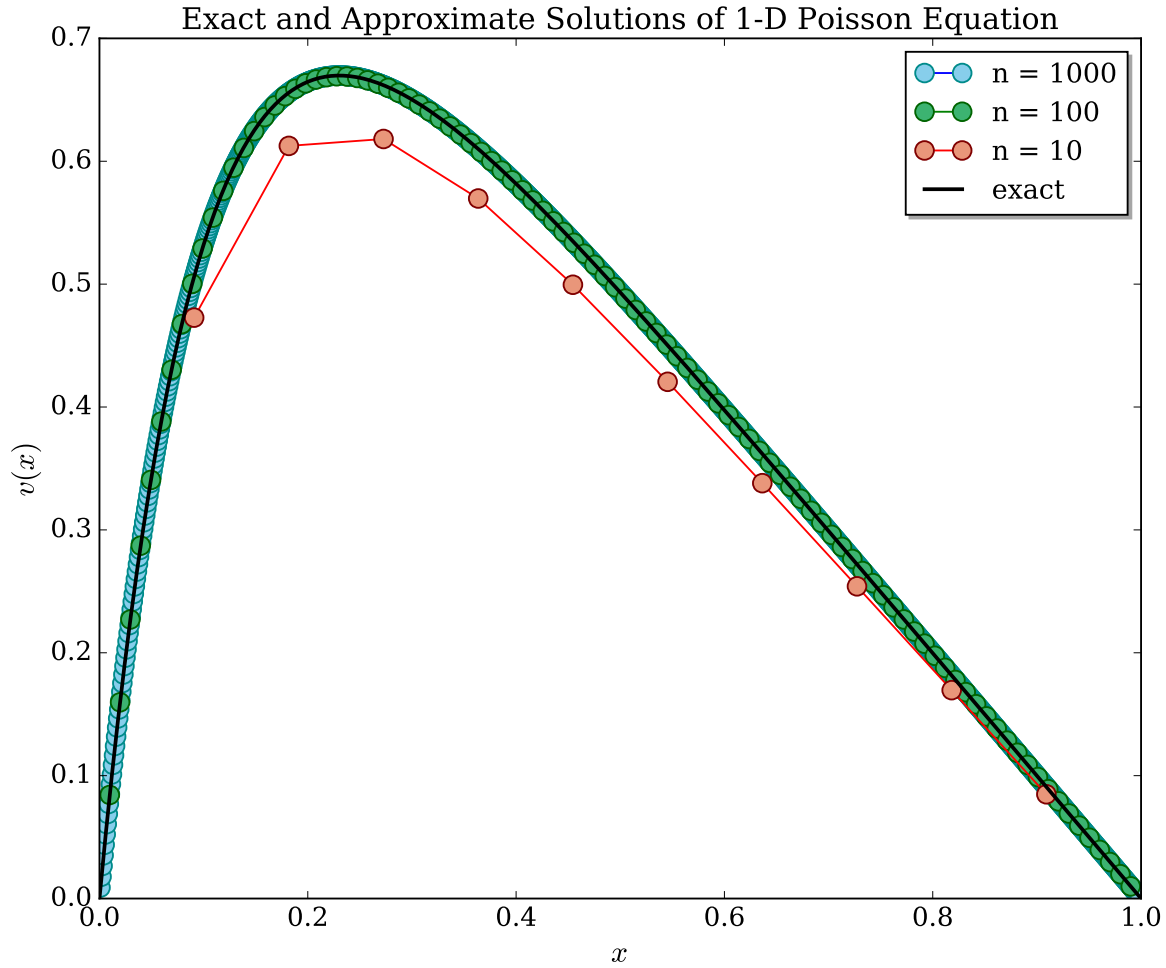
### IV. RESULTS AND ANALYSIS

$n =$	10	$10^2$	$10^3$	$10^4$	$10^5$	$10^6$
A	$2.1E-4$	$1.4E-4$	0.288	N/A	N/A	N/A
B	$2.0E-6$	$6.0E-6$	$4.4E-5$	$4.1E-4$	$2.4E-3$	0.023
C	$2.0E-6$	$4.0E-6$	$3.4E-5$	$3.6E-4$	$3.1E-3$	0.030
D	$1.0E-6$	$3.0E-6$	$3.1E-5$	$3.2E-4$	$2.5E-3$	0.024

Table 1. Time used for computation (in seconds) for methods A, B, C, and D and for  $n = 10 - 10^6$ . Benchmark calculations can be found in 'benchmark' folder under 'report'.

Figure 1. The discretized solution  $v_i$  for various  $n$  are plotted alongside the closed-form solution  $u(x)$ . For  $n > 100$ , the approximate and exact solutions are almost indistinguishable.

discuss main findings, link them up with existing lit-



erature, discuss eventual errors, effectiveness of your algorithm, stability of the calculations. 20 pts

blah biddy blah

The computation times for all four programs are shown in Table 1.

## V. CONCLUSION

summary of discussions and critical comments. what was learned about the methods i used? and results obtained? what about weird things when i was writing the code? future improvements and possible directions? comparison with iterative methods? 10pts

clarity of figures/overall presentation: 10 pts. references: 5 pts