PHY480 Project 1

David Butts, Daniel Coulter, and Liam Clink
Michigan State University
(Dated: February 6, 2018)

Abstract: In physics, many differential equations can be approximated using linear systems of equations, making efficient algorithms for solving the associated matrix equations necessary. We compare five algorithms for solving the one-dimensional Poisson equation this way, three of which were our implementations. The reference algorithms we used were NumPy's Gaussian Elimination method and SciPy's LU decomposition method, which are $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ respectively. We compared these to algorithms which took advantage of the sparse symmetric character of the matrix which are $\mathcal{O}(n)$. The best performing algorithm was written in C++, which ran an order of magnitude faster than our best Python algorithm, and with better accuracy.

I. INTRODUCTION

The goal of this project is to numerically solve the Poisson equation with Dirichlet boundary conditions. The general Poisson equation in three dimensions is given by:

$$\nabla^2 u(x,y,z) = f(x,y,z) \tag{1}$$

We aim only to solve the one-dimensional case, so Equation 1 can be written as:

$$-\frac{\mathrm{d}^2 u(x)}{\mathrm{d}x^2} = f(x) \tag{2}$$

where we are using the following boundary conditions:

$$x \in (0,1), \quad u(0) = u(1) = 0 \tag{3}$$

The differential equation can be approximated as a system of linear equations

$$\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = -f(x_i) \tag{4}$$

where the approximate solution is v. We can then apply computational matrix solution methods to approach this problem.

II. ALGORITHMS

We wrote three separate algorithms for this project. The first is a Python algorithm for general tridiagonal matrice [2], the second is a Python algorithm for the specialized case where the values along the diagonals are the same, and the third is a C++ algorithm for the general case [1]. We compared our solving algorithms to the LU solver from SciPy and the linalg solver from NumPy.

A. Python

We implemented our algorithm by rewriting the system of equations as the following matrix (with $a_i = c_i = -1, b_i = 2$) which we can treat as a tridiagonal matrix problem, which allows for $\mathcal{O}(n)$ algorithms to be used, as opposed to Gaussian elimination, which is $\mathcal{O}(n^3)$.

$$\begin{bmatrix} b_0 & c_1 & 0 & \dots & 0 \\ a_1 & b_1 & c_2 & 0 & \vdots \\ 0 & a_2 & \ddots & \ddots & 0 \\ \vdots & 0 & \ddots & \ddots & c_{n-1} \\ 0 & \dots & 0 & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ \vdots \\ \vdots \\ f_n \end{bmatrix} \tag{5}$$

We let the vector f be defined as

$$f = 100e^{-10x} \tag{6}$$

In the Python symmetric positive definite tridiagonal matrix implementation we used a forward substitution to find new diagonal elements b_i with (7) and the new right hand side of the equation with (8). Then we use a backward substitution (9) to find the solution.

$$\tilde{b}_i = b_i - \frac{a_i c_{i-1}}{\tilde{b}_{i-1}} \tag{7}$$

$$\tilde{f}_i = f_i - \frac{a_i \tilde{f}_{i-1}}{\tilde{b}_{i-1}} \tag{8}$$

$$u_{i-1} = \frac{\tilde{f}_{i-1} - c_{i-1} u_i}{\tilde{b}_{i-1}} \tag{9}$$

B. C++

We also implemented a similar algorithm for symmetric positive definite tridiagonal matrices in C++. This implementation uses 2 vectors of N values and 1 vector of $N-1$ values, which is two fewer than the Python implementation. This takes advantage of how the superdiagonal is the same as the infradiagonal, and also overwrites the values in the initial vector with the values of the old vector as they aren't needed anymore. When solving matrix equations of dimension in excess of 10^8, this can mean a savings of many GB in memory usage.

Obviously, storing only the nonzero matrix elements in diagonal vectors makes the memory requirement $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$. Memory usage puts a practical limitation on speed, because if more memory is required than is available, the usage of virtual memory bottlenecks the process heavily. The algorithm is described below, where α is the diagonal, β is the superdiagonal, and f is the initial vector.

$$\tilde{\beta}_k = \beta_k/\alpha_k,\ \tilde{\alpha}_{k+1} = \alpha_{k+1} - \beta_k/\tilde{\beta}_k \tag{10}$$

$$\tilde{b}_{k+1} = b_{k+1} - \tilde{\beta}_k \cdot b_k \tag{11}$$

$$\tilde{b}_{n-1} = b_{n-1}/a_{n-1} \tag{12}$$

$$\tilde{b}'_{k-1} = \tilde{b}_{k-1}/\tilde{\alpha}_{k-1} - \tilde{\beta}_{k-1} \cdot \tilde{b}_k \tag{13}$$

Equations 10 and 11 represent loops through $0 \leq k < n-1$, k increasing, and equation 13 represents a loop through $n-1 \geq k > 0$ decreasing. A comparison of the runtimes for these algorithms can be seen in figure 5.

III. RESULTS

A. Accuracy

We compared our general tridiagonal solver to the closed form solution of Equation 2:

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{14}$$

The results of this comparison for a 10 x 10, 100 x 100, and 1000 x 1000 matrix can be seen in figures 1, 2, and 3. By 1000 x 1000, the solution from our algorithm very nearly matches that of the analytic solution.

We also performed error analysis to compare the error in Python and C++. Error was calculated using

$$\varepsilon_i = \log_{10}\left(\frac{v_i - u_i}{u_i}\right) \tag{15}$$

where v_i is the calculated value and u_i is the analytic value. The results can be found in Figure 4.

B. Timing Analysis

Here we compared the timing for the four algorithms we used in this study, the LU solver from SciPy, `linalg.solve()` from Numpy, our general tridiagonal solver, and our specialized tridiagonal solver. The results can be seen in Table 1 and Figure 5.

First we implemented our SciPy solver by constructing the full n x n matrix and calling the `linalg.solve()` method on the matrix with the vector f. We also used the LU solver in SciPy by first calling the `scipy.linalg.lu_factor()` method on our matrix, and then called the `scipy.linalg.lu_solve` on our factored matrix and the vector f.

We compared the general tridiagonal solver to our special case algorithm. Figure 6 Shows the results of this comparison. Our Special case algorithm is slightly faster than the general algorithm.

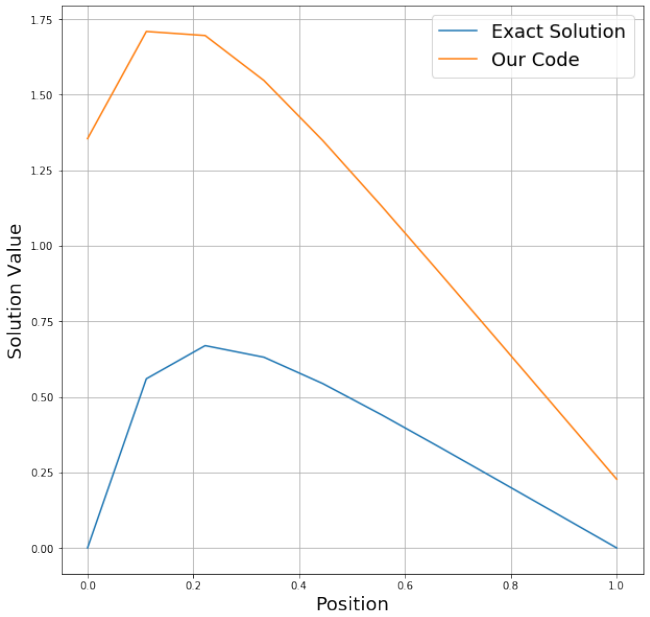

FIG. 1. A comparison of the exact solution of Equation 2 and our tridiagonal algorithm for 10 grid points

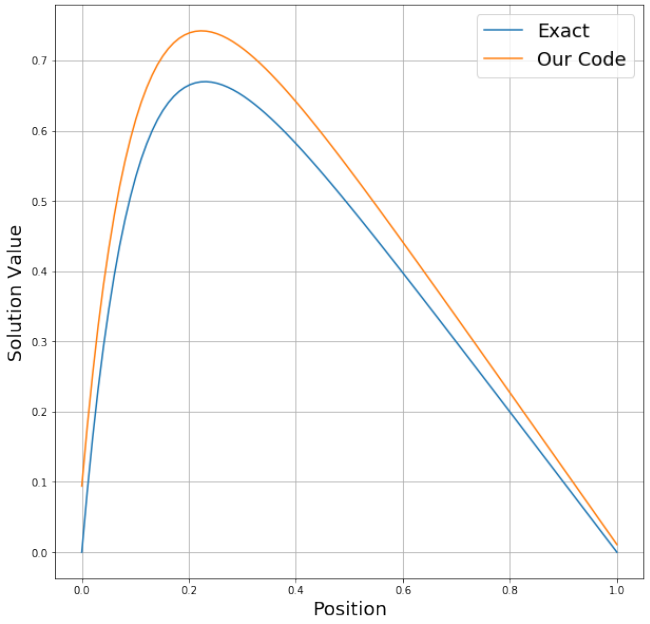

FIG. 2. A comparison of the exact solution of Equation 2 and our tridiagonal algorithm for 100 grid points

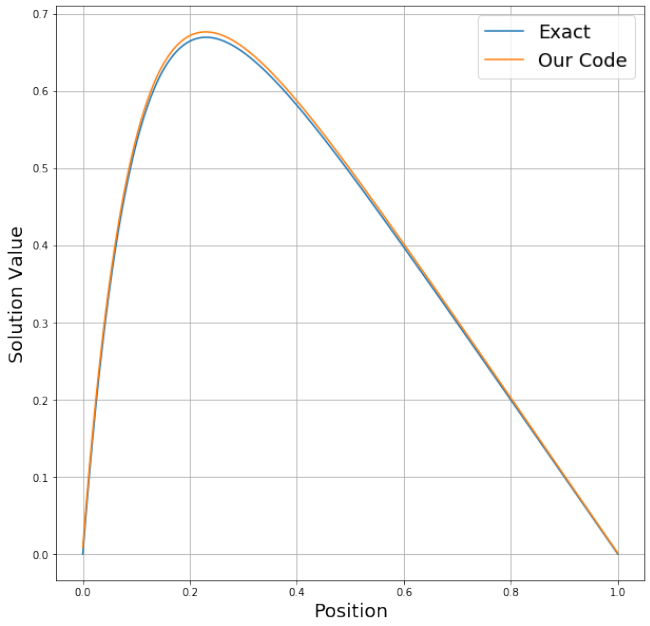

FIG. 3. A comparison of the exact solution of Equation 2 and our tridiagonal algorithm for 1000 grid points

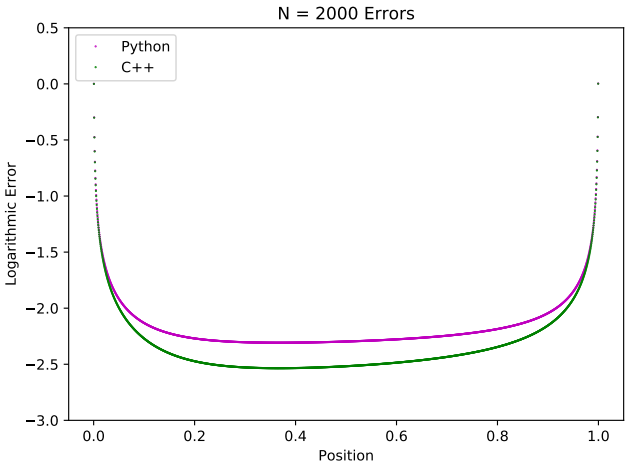

FIG. 4. An error comparison using $\epsilon_i = \log_{10}\left(\frac{v_i - u_i}{u_i}\right)$, where v_i is the calculated value and u_i is the analytic value. Interestingly, all of the algorithms in Python had the same error, but C++ has a higher accuracy.

Code/Size	10	100	1000
LU Solver	.00031	.00111	.03169
Numpy linalg.solve	2.1875e-5	.00038	.03376
Specialized Tridiagonal	2.3085e-5	.00011	.00196
Tridiagonal	3.5319e-5	.00033	.00291

When we attempted to measure the timing for the LU decomposition we found that it could not handle a matrix at the size of 10^5 x 10^5. This is because it must store the entire matrix which will take up about 80GB. When we attempted to create a matrix of this size, the code ran until it used all available memory and crashed.

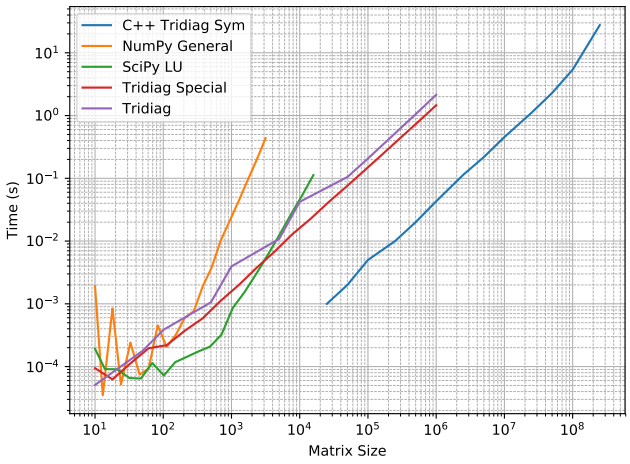

FIG. 5. A comparison of `scipy.lu_solve()`, `np.linalg.solve()`, our general tridiagonal solver, our special case tridiagonal solver for the Poisson equation, and a symmetric tridiagonal solver written in C++

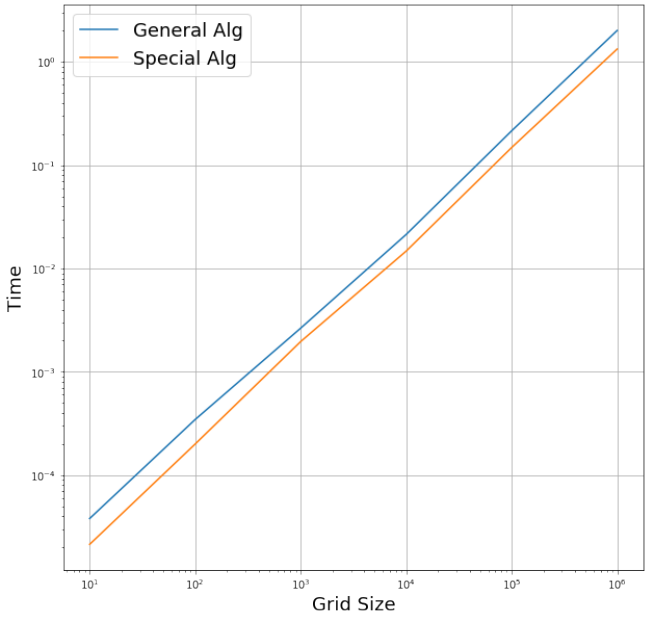

FIG. 6. A comparison of the time taken for our general tridiagonal solver compared to our specialized algorithm.

IV. CONCLUSION

Through this study we found that for our algorithms it was better to use C++ over Python due to memory allocation and its native speed. When comparing algorithms without respect to language choice it is important to think

about how much memory is required to use an algorithm and how an algorithm scales. The naive approaches of using Gaussian elimination or LU decomposition to solve a linear algebra problem requires an entire matrix to be stored which will drastically limit the dimension of the matrix that can be stored. These algorithms scale like $\mathcal{O}(n^3)$. By noticing symmetries in our problem we can write specialized algorithms. For the problem we solved we had a tridiagonal matrix where the off diagonals were all the same value, and the main diagonal was a single value. This allowed us to only store two vectors instead of the entire matrix. We used the algorithm described in Equations 7 - 9 which scaled as $\mathcal{O}(n)$.

[1] Gene Golub and Charles Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2012.

[2] Mark Newman. *Computational Physics*. CreateSpace Independent Publishing Platform, 2012.