Project 1

Kristen Parzuchowski, Parker Brue, Sam Edwards[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. This report is specialized to Sam's results for his Python version of the algorithms.

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:

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

Using Dirichlet boundary conditions, $u_0 = u_{\mathrm{n}+1} = 0$, 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):

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_{i-n} \\ 0 & 0 & e_{i-n} & d_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_n \end{bmatrix}$$

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:

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_{n-1} \\ 0 & 0 & 0 & \tilde{d}_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_n \end{bmatrix}$$

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

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

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

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

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.

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

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

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:

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

Equation (7) can be reduced to:

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

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.

In LU decomposition, the lower triangular matrix takes the form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix}$$

The upper triangular matrix takes the form:

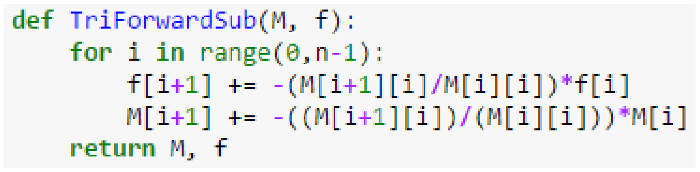
$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

In first using the lower triangular matrix, equation (10) can be solved. The solution found for x can then be used to solve equation (11).

METHODS

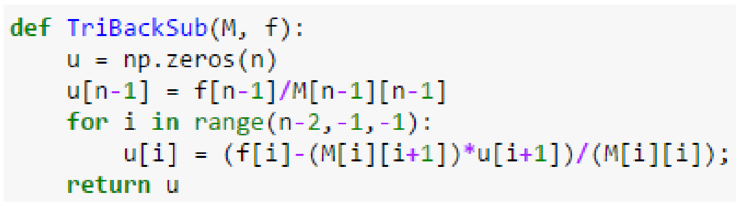
We will briefly explain the methods implemented in this project. For a complete understanding, visit our Github page: (https://github.com/parzuch9/KESP/Project1).

Figure 1 and Figure 2 show the algorithms to solve a general tridiagonal matrix. The fact that they take into account a lack of symmetry gives them a few more operations per loop iteration than the specialized algorithms, although both can be used to solve the special tridiagonal matrix. The notation M in the code corresponds to any 10^n x 10^n tridiagonal matrix, and f is our discretized function.



```
def TriForwardSub(M, f):
    for i in range(0,n-1):
        f[i+1] += -(M[i+1][i]/M[i][i])*f[i]
        M[i+1] += -((M[i+1][i])/(M[i][i]))*M[i]
    return M, f
```

FIG. 1. The forward substitution algorithm as implemented in Python.

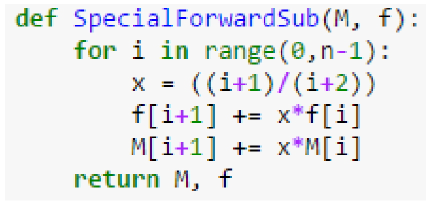


```
def TriBackSub(M, f):
    u = np.zeros(n)
    u[n-1] = f[n-1]/M[n-1][n-1]
    for i in range(n-2,-1,-1):
        u[i] = (f[i]-(M[i][i+1])*u[i+1])/(M[i][i]);
    return u
```

FIG. 2. The back substitution algorithm as implemented in Python.

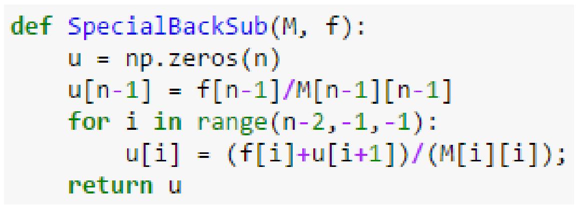
Implementing a Specialized Algorithm

This algorithm solves the specialized tridiagonal matrix from the Algorithms section found when solving the poisson equation. The Python functions that describe these special cases of forward and backward substitution can be seen in Figures 3 and 4. Notice the much simpler form due to the repeated 2's and -1's that replace elements M[i][i+1], M[i+1]M[i], and M[i][i].



```
def SpecialForwardSub(M, f):
    for i in range(0,n-1):
        x = ((i+1)/(i+2))
        f[i+1] += x*f[i]
        M[i+1] += x*M[i]
    return M, f
```

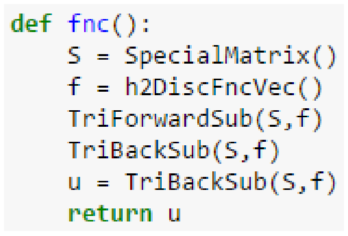
FIG. 3. The specialized forward substitution algorithm as implemented in Python.



```
def SpecialBackSub(M, f):
    u = np.zeros(n)
    u[n-1] = f[n-1]/M[n-1][n-1]
    for i in range(n-2,-1,-1):
        u[i] = (f[i]+u[i+1])/(M[i][i]);
    return u
```

FIG. 4. The specialized back substitution algorithm as implemented in Python.

In the code you will see on GitHub, functions named fnc() (and SpecialFnc())seen in Figure 5 execute nearly the entirety of the problem. The function SpecialMatrix() simply produces our matrix of 2's and -1's, and h2DiscFncVec() produces the discretized function vector. Returning the one dimensional array u gives points that begin to converge to the actual function u, as the dimension of the matrices increase.



```
def fnc():
    S = SpecialMatrix()
    f = h2DiscFncVec()
    TriForwardSub(S,f)
    TriBackSub(S,f)
    u = TriBackSub(S,f)
    return u
```

FIG. 5. The specialized back substitution algorithm as implemented in Python.

Relative error

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

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

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 (Figure 6). You can see that the approximated numerical solution gets closer to the analytical solution as n increases by multiples of ten. n = 10^3 is actually so close to the analytical solution that it is hard to tell them apart. I did not include n = 10^5 because it was indistinguishable from the analytical plot.

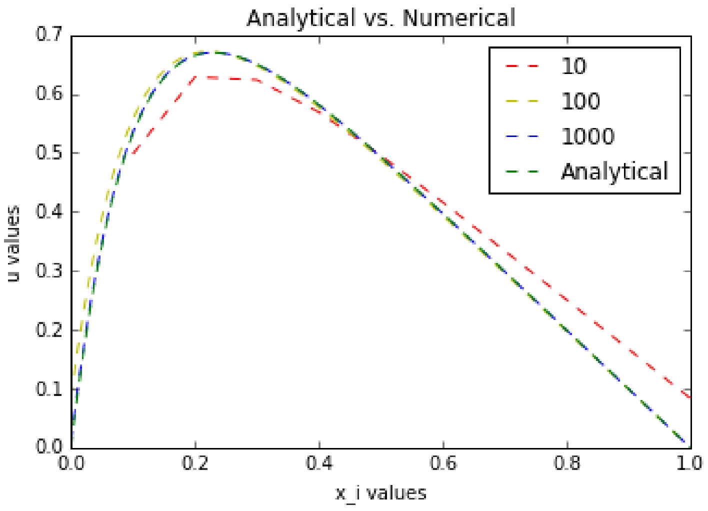


FIG. 6. Comparison between different step sizes. The x axis are the x values in the range [0,1] and the y axis are the resulting u values.

Step Size (n)	Average Relative Error ($\|\epsilon\|$)
1	1.200
2	2.019
3	2.928
4	3.921

By including equation 14 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=1,2,3,4 (Table above). It should be noted that these are the absolute values and averages (of ten trials) of the errors. It should also be noted that it breaks down for n=10^5; there, my computer had a memory error. It appears that n=4 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 using both the special algorithm and the LU decomposition algorithm. The timer comes is imported into the iPython notebook via "timeit". By adding this timing mechanism, we calculated the time required to solve square matrices with various column lengths via both the specialized algorithm and LU decomposition algorithms. Again, a memory error occured at n = 10^5 for the specialized algorithm, and at n = 10^4 for the LU decomposition.

Column Size	Specialized Time (s)	LU Time (s)
10^1	0.000102	0.000403
10^2	0.000865	0.00627
10^3	0.00961	0.0913
10^4	0.27745	

CONCLUSIONS

Results

We investigated a few ways of solving a tridiagonal matrix problem. As expected, specialization outperforms generalized "brute force" methods. Specialization is faster, and in this case, can be taken out into greater orders of magnitude. Although there is improvement in this realm, the error did steadily increase as the power of ten increased.

Future Prospects

There may be a method to optimize the LU decomposition program, to allow for us to solve problems greater than 10^4, it might be useful to look into this. Similar reduction in FLOPs using knowledge of 2's and -1's could increase speeds.

Acknowledgements

We thank Professor Morten Hjorth-Jensen for our fruitful discussions and for providing us with much of the C++ code to get started on this project.