Project 1

Eric Aboud[1]

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

We present our algorithm for solving linear equations. Using Dirichlet boundary conditions, we were able to write the one-dimensional Poisson equation using a set of linear equations. Our best algorithm runs as $4n$ FLOPS with n the dimensionality of the matrix. We were then able to rewrite our linear set of equations as a tridiagonal matrix, solving the equations. Comparions between the two methods provides insight on the optimal way of solving the set of linear equations in C++.

INTRODUCTION

In order to become more familiar with vector and matrix programming, this project was set up to write matrix operations and the linear operations behind them. Using Dirichlet boundary conditions, we were able to write a set of linear equations to solve the one dimensional Poisson equation. We were also able write a linear set of equations to form a tridiagonal matrix.

THEORY, ALGORITHMS AND METHODS

Theoretical solution of the one dimensional Poisson equation

We can solve the one dimensional Poisson equation by setting a function equal to the negative of the second derivative of another function:

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

Via descritization, 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, we are able to solve the set of linear equations using forward and backward substitutions. We can then rewrite this as a set of linear equations in the form of a tridiagonal matrix:

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

Computational solution of the one dimensional Poisson equation

It is possible to solve the poisson equation using a set of linear equations and matrix mathematics with C++. By assume a function

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

and a closed form solution:

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

We can write a simple program that uses forward (6) and backward (7) subsitutions 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 (Figure 1).

```
void MatrixInverse(double **A, int n)
{
  // allocate space in memory
  int *indx;
  double *column;
  indx = new int[n];
  column  = new double[n];
  double **Y    = AllocateMatrix(n,n);
  // Perform the LU decomposition
  LUDecomposition(A, n, indx);   // LU decompose  a[][]
  cout << "LU decomposed matrix  A:" << endl;
  WriteMatrix(A,n);
  // find inverse of a[][] by columns
  for(int j = 0; j < n; j++) {
    // initialize right-side of linear equations
    for(int i = 0; i < n; i++) column[i] = 0.0;
    column[j] = 1.0;
    LUBackwardSubstitution(A, n, indx, column);
    // save result in y[][]
    for(int i = 0; i < n; i++) Y[i][j] = column[i];
  }
  // return the inverse matrix in A[][]
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) A[i][j] = Y[i][j];
  }
  DeallocateMatrix(Y, n, n);     // release local memory
  delete [] column;
  delete []indx;

}
```

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

We are able to calculate the relative error of the calculations using:

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

RESULTS AND DISCUSSION

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 2). 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.

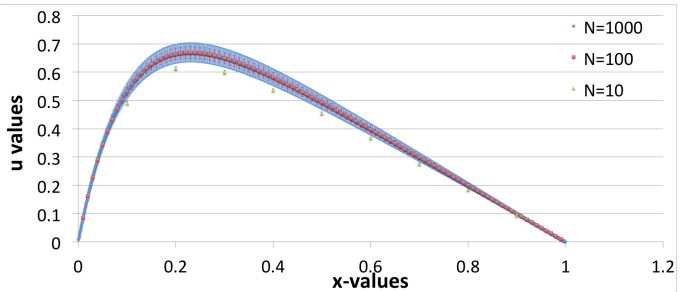

FIG. 2. Comparison between different step sizes. The x axis are the x values in the range [0,1] and the y axis are the resulting u values.

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)	Average Relative Error ($\|\epsilon\|$)
1	1.101
2	3.079
3	5.079
4	7.079
5	9.079
6	11.50
7	12.27

By including equation 8 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 Elements	Time$_{LU}$ (s)	Approximate Time$_{ClosedForm}(s)$
10	0	0
100	0	0
1000	30	0
10000	N/A	0
1000000	N/A	7
10000000	N/A	74

It appears that the LU decomposition process includes 5 FLOPS, however to use of the LU decomposition, such as finding the inverse of the matrix, brings the total FLOPS up. It may be possible to run a 10^5 column elements, but it would take a very long time. At 10^3, the LU decomposition calculations took too long and too much ram for my computer to calculate the solution in under an hour. As we can see from the above table, it takes a while, compared to the closed form method, to run 10^2 column elements implying that it would take an unreasonable time to run the 10^5 column elements, therefore the LU decomposition method was not used to calculate the solution for matrices with more than 10000 column elements.

We can also see from this table that it is much quicker to solve the set of linear equations via the closed form method. While it took 30 seconds to calculate it with LU decomposition, closed form method solves the set in less than one second. Using the closed form method we don't see an appreciable time delay until the 10^7 column elements.

CONCLUSION

By evaluating a set of linear equations with two methods we can compare and find the best way to solve the set in a program. Using the special case (diagonal elements equal to 2 and off diagonal equal to -1) we are able to significantly decrease the time required to compute the solution of the set of linear equations. We found that there is a saddle point in the number of matrix elements where we get and accurate description of the solutions without overwhelming error bars. We also found that there is also a maximum number of step sizes before our program breaks down.

We also found that the closed form method of solving the set of linear equations was much more efficient

than calculating it with LU composition. The LU decomposition method took much more time to calculate the solution of the set.

This analysis has showed that there are multiple ways of solving a set of linear equations as well as ways of optimizing the calculations to provide a more efficient computation.