PHY 480/905 Computational Physics

Project I

Joshua Milem
February 7, 2016

Abstract

The purpose of this project is to solve the three-dimensional Poisson equation with Dirichlet boundary conditions. This will be accomplished by solving a set of linear equations for a discretized variable. The first part of this project will be solved using a given source term, while the second part will be a more general case. The latter part of this report will include error in the data set. Two methods will be used to solve this equation: LU decomposition and Gaussian elimination. These are then compared in terms of the number of floating point operations (FLOPS) and, consequently, computation time.

Introduction

This project will solve Poisson's equation for charge distribution by transforming it into a simplified second-order differential equation first, and then finally into a set of linear equations of the form **Av=b** which can be solved easily using code. We will be using Dirichlet boundary conditions in these calculations, under the domain x(0,1)with u(0)=u(1)=0. This tridiagonal matrix will be solved using two different methods, namely Gaussian elimination and LU decomposition. Both of these functions are using pre-written library functions as suggested in the project documentation. Error calculations are included in the latter part of the report with a graphical representation comparing the two methods. Both methods are also compared in terms of the number floating point operations (FLOPS) each uses, and it follows that computation time will also be compared. Graphical representations of all data will be created using Python.

Theory

The problem we will solve begins with a 3-dimensional differential equation known as the Poisson equation. This equation is used widely in electromagnetism to describe electrostatic potential from charge distribution and thus is useful to solve. This equation will be solved using a certain set of assumptions. This equation is stated as such:

$$\nabla^2\varphi = -4\pi r\rho(r)$$

Where φ is the electrostatic potential, ∇ is the gradient, and $\rho(r)$ is the local charge distribution. Dirichlet boundary conditions will be used to eliminate the endpoint values, or:

$$u(0) = u(1) = 0$$

This equation can be simplified down to a 1-dimensional form by assuming that both the localized charge distribution and electrostatic potential are both spherically symmetric. This will be reduced to:

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\varphi}{dr}\right) = -4\pi\rho(r)$$

In order to reduce this equation down to a further generalized form (which is ideal for computations) we use a substitution as given by the lecture notes, where a reduced form is given as:

$$\frac{d^2\emptyset}{dr^2} = -4\pi r\rho(r)$$

Where $\emptyset$ is $r\varphi(r)$. Finally, if we simplify the right side of the equation to a constant multiplied by the charge distribution and rename this simply f(x), our equation becomes an elegant one-dimensional second order differential equation of the form:

$$-u''(x) = f(x)$$

From this we can solve using the numerical definition of the second derivative,

$$f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$$

Given the discretized approximation $u \approx v_i$ as suggested in the lecture notes, we will define our step spacing as $h = \frac{1}{(n+1)}$ where *n* is the index of the matrix. We will also use the interval $x_0 = 0\ to\ x_{n+1} = 1$. However, given this interval, both endpoints are equal to zero as given by the Dirichlet boundary conditions. This second derivative of approximation can be solved in the form **Av = b** if we simply multiply both sides of our equation by the step spacing squared to yield a linear equation.

$$-v(i+1) - 2v(i) + v(i-1) = f(i)h^2$$

This matrix set '**Av = b**' is a tridiagonal matrix of the form:

$$\begin{vmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{vmatrix} \begin{vmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{vmatrix} = \begin{vmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{vmatrix}$$

This concludes the initial theory behind the project. The next section will discuss how exactly to solve this problem.

Methods

The first method will be using is Gaussian elimination. Our system is tridiagonal with the form stated above. G.E. is split into two parts for this algorithm, a forwards substitution step and a backwards substitution step. First, let us state the algorithm in a general sense since using our integer values directly would not yield a very modular code. Of the tridiagonal form,

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

Our equation has the values of $a = 2$, $b = -1$, and $c = -1$. So, our starting matrix with a, b, and c is:

$$\begin{bmatrix} a_1 & c_1 & 0 & 0 \\ b_1 & a_2 & c_2 & 0 \\ 0 & b_2 & a_3 & c_3 \\ 0 & 0 & b_3 & a_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix}$$

The first step of G.E. is to perform the forward substitution by reducing the rows down to a single value along the diagonal. The first iteration of this would be:

$$\begin{bmatrix} a_1 & c_1 & 0 & 0 \\ 0 & a_2 - \frac{c_1}{a_1} b_2 & c_2 & 0 \\ 0 & b_3 & a_3 & c_3 \\ 0 & 0 & b_4 & a_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

This iteration is carried all the way through the matrix, with each successive iteration calling on the iteration before it i.e. a_3 will use the formula in a_2 for its calculation. This is why this elimination is great for computers because it becomes more and more tedious as the calculation goes on, and for large matrices this would take more time than a person would care to spend on it. The final result of this process will be a matrix that reduces down to a diagonal composed of 1's, where the adjacent diagonal becomes a ratio of c_1, c_{n-1} divided by A_1, A_{n-1} where A is the substituted form of the original diagonal.

The second step of G.E. is to perform a backwards substitution where the augmented part of the matrix (v values) are substituted such that whole matrix can be reduced to a single identity of:

$$w_i = x_i - \frac{c_i}{d_i} x_{i+1}$$

This was coded in python using the diag function which creates an array based solely on the diagonal. However, the final compiled code did not run for this function and I suspect the reason is an indexing error with the algorithm.

The second method used to solve the equation is LU decomposition. This refers to the factorization of a matrix A into the form A = LU where L and U are the lower and upper triangular matrices respectively. The general form of a 4x4 LU decomposition looks as such:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ l_{31} & l_{32} & l_{33} & \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ & u_{22} & u_{23} & u_{24} \\ & & u_{33} & u_{34} \\ & & & u_{44} \end{bmatrix}$$

These two systems represent equations that can be solved for the decomposition. Expanding the matrix multiplication will yield a set of algebraic equations which can be solved using code, since it requires small manipulation of essentially the same equation. This was coded using the linalg library included in python, using the same set of diagonally defined matrices as used in the G.E. function. Unfortunately, the final code for the LU function did not produce a working result because of a matrix size error.

Results and Comparison

The number of floating point operations involved in LU decomposition can be expressed as 2/3m^3 FLOPS because the system of equations will be larger than that of G.E. This is a very high value, especially since a given data set taken from an instrument will produce an array of order 10^4 or larger. Using the LU function, an extremely powerful computer would be required to manipulate larger matrices, and even with smaller matrices the computation time will be great, and when one is paying for time on a supercomputer, this is not ideal. However, using G.E. method, the manipulations are much more simple since there are only two substitutions that need to

be done using floating point operations. This varies as linear dependence as opposed to the cubic dependence of LU decomposition. LU decomposition is certainly more straightforward in terms of the raw mathematics, however when it comes to computing G.E. clearly the better method to use because of its lightweight algorithm.

Afterword

This project was very discouraging for me. I received a large amount of help from a friend that is a mathematics major. I have not taken linear algebra yet and I am just now learning a deeper level of coding, so trying to learn both aspects of the project at the same time proved to be difficult. I used python to produce my code for this since it is what I am probably the best at, albeit still not very good. In the future I would like to use C++, but it proved to be too overwhelming to try and code in C++ and understand the mathematics behind the project at the same time. As a result of these factors my code does not compile from a matrix size error and my report appears dry. This being said, I greatly enjoy the course so far despite this an am looking forward to the next project where I actually know the theory behind it.