Project 1

Nat Hawkins, Victor Ramirez, Mike Roosa, Pranjal Tiwari

26 Jan, 2017

Abstract

The aim of this project is two-fold: firstly to familiarize ourselves with some computational practices and the use of C++ coding methods (i.e. dynamic memory allocation, compiling of cpp files, etc.) as well as testing the limits of machine accuracy, and second to perform computations using vector and matrix operations in order to solve mathematical problems. We will be comparing the efficiency of use of vectors versus matrices in our solving of these problems.

1 Introduction

Before we begin, although C++ will be used for some parts of this project in order to learn the language, the majority of our coding will be done in Python. The three of us have experience in using this language and we decided that it was best to play to our strengths in using the coding language we are most comfortable with. We will do some work with C++ and a discussion of this will be provided in our report.

1.1 Mathematical Motivation

The motivation behind this project is simple: Many important differential equations in Science can be written as linear second-order differential equations

$$\frac{d^2y}{dx^2} + k^2(x)y = f(x),$$

where f is normally called the inhomogeneous term and k^2 is a real function.

A classical equation from electromagnetism is Poisson's equation. The electrostatic potential Φ is generated by a localized charge distribution $\rho(\mathbf{r})$. In three dimensions it reads

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r}).$$

With a spherically symmetric Φ and $\rho(\mathbf{r})$ the equations simplifies to a one-dimensional equation in r, namely

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

which can be rewritten via a substitution $\Phi(r) = \phi(r)/r$ as

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

The inhomogeneous term f or source term is given by the charge distribution ρ multiplied by r and the constant -4π.

We will rewrite this equation by letting $\phi \rightarrow u$ and $r \rightarrow x$. The general one-dimensional Poisson equation reads then

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

2 Solution

2.1 Problem Set-Up

In this project we solved the one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations.

In doing so, our mathematics looked something like the following:

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

In our case we will assume that the source term is $f(x) = 100e^{-10x}$, and keep the same interval and boundary conditions. Then the above differential equation has a closed-form solution given by $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. We will compare our numerical solution with this result in the future exercise.

We fed the program an input value i, which was then used to determine x and the step size h. We did this by defining the variables as follows

$$i = [input], \qquad n = 10^i, \qquad h = \frac{1.0}{n}$$

We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n,$$

where $f_i = f(x_i)$.

We began with a tridiagonal matrix, A, and transformed the problem into the following:

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}},$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{bmatrix},$$

and $\tilde{b}_i = h^2 f_i$. A can be derived by looking at the formula for the approximation of the second derivative. Note how the v_{i+1} and the v_{i-1} terms have a coefficient of -1. This is where the off diagonal elements come from in our matrix A. Since the two off diagonal entries are equivalent, we can approximate them into one vector, e. However, since all of the entries are -1, we decided to not allocate memory to this. Instead, we simplified our algorithm to account for these values. We will discuss our problem solving algorithms in the following section.

The reason for the conversion from a matrix to vectors comes about in our number of floating point operations (FLOPS). In using the matrix approximation, we were looking at (reference) $\frac{2}{3}n^3$ FLOPS in calculating our solutions. However, in being able to reduce this down to a vector problem, we managed to reduced our calculations to $9n$, and then further to $4n$, again to be discussed in the next section. The short of this discussion is that we were able to reduce our floating point operations by two orders of magnitude. In the sense of computational efficiency, this makes for much smarter problem solving and much more effective computation.

2.2 Algorithms

The algorithms that we implemented in the solving of this problem are as follows. We denote the dioagonal elements as $d[i]$, the solution values as $u[i]$, and the values of the function as $f[i]$. The problem starts when look at a general case for a $4x4$ matrix:

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{bmatrix},$$

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix},$$

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix},$$

And now we can look at solving $\mathbf{Au} = \tilde{\mathbf{f}}$, through the process of Gaussian Elimination. For this process, reference (reference). In essence, we begin by multiplying the second row of A by the ratio of the terms in the first column of rows one and two. Explicitly, this looks like $\frac{e_1}{d_1}$ on both the right hand side and left hand side. This transforms the equation into the following:

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 \\ 0 & d_2 - \frac{e_1^2}{d_1} & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{bmatrix} * \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 - f_1 \frac{e_1}{d_1} \\ f_3 \\ f_4 \end{bmatrix}$$

We now can redefine $\tilde{d_2} = d_2 - \frac{e_1^2}{d_1}$ and $\tilde{f_2} = f_2 - f_1 \frac{e_1}{d_1}$. This changes the matrix to a new form that we can use to repeat the process further.

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d_2} & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{bmatrix} * \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ \tilde{f_2} \\ f_3 \\ f_4 \end{bmatrix}$$

Now, we can perform the same operation and multiply the second row by $\frac{e_2}{\tilde{d_2}}$. As we now with Gaussian Elimination, the final result will be an upper triangular matrix of the form:

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d_2} & e_2 & 0 \\ 0 & 0 & \tilde{d_3} & e_3 \\ 0 & 0 & 0 & \tilde{d_4} \end{bmatrix} * \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ \tilde{f_4} \end{bmatrix}$$

At this point, we can begin the process of forward substitution and equate $\tilde{d_4} * u_4 = \tilde{f_4}$. Using this value, we can solve for u_4 and then solve $\tilde{d_3} * u_3 + \tilde{d_4} * u_4 = \tilde{f_3}$ for u_3, and so on and so forth. With pre-populated arrays for u and f, we now have our solution. This process yields the algorithms used in our program.

$$\tilde{d_i} = d_i - \frac{e_{i-1}^2}{d_{i-1}} \tag{1}$$

$$\tilde{f_i} = f_i - f_{i-1} \frac{e_{i-1}}{d_{i-1}} \tag{2}$$

$$u_i = \frac{\tilde{f_i} - e_i u_{i+1}}{d_3} \tag{3}$$

Now we can use the fact that for our specific case all of the e_i values are equal to -1 to simplify our algorithms. This also causes a drastic simplification for the d_i values.

$$\tilde{d}_i = \frac{i+1}{i} \tag{4}$$

$$\tilde{f}_i = f_i + f_{i-1}\frac{1}{d_{i-1}} \tag{5}$$

$$u_i = \frac{\tilde{f}_i + u_{i+1}}{d_3} \tag{6}$$

Thus, we see a reduced floating point operation count down to $\approx 4n$. This is where the efficiency of this approach really becomes evident.

2.3 Problem 1c

2.4 Relative Error

We calculated the relative error in the data set $i = 1, \ldots, n$,by setting up

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

as function of $log_{10}(h)$ for the function values u_i and v_i. For each step length, we saw that the error was pretty much a constant value. That meaning that for each value of x, the relative error was approximately the exact same. Since the error was the exact same for all x values, we elected to choose only one points error from each trial. However, we elected to choose the first value in each trial for our error (i.e. $i = 1, x = 0.1 \quad i = 2, x = 0.01, etc.$). We were consistent with our choice, so whatever error that this may have imposed, we were at least consistent with this. We then conducted multiple trials utilizing values of i ranging from $i = 0$ to $i = 8$. The results we got are below:

2.5 Problem 1e

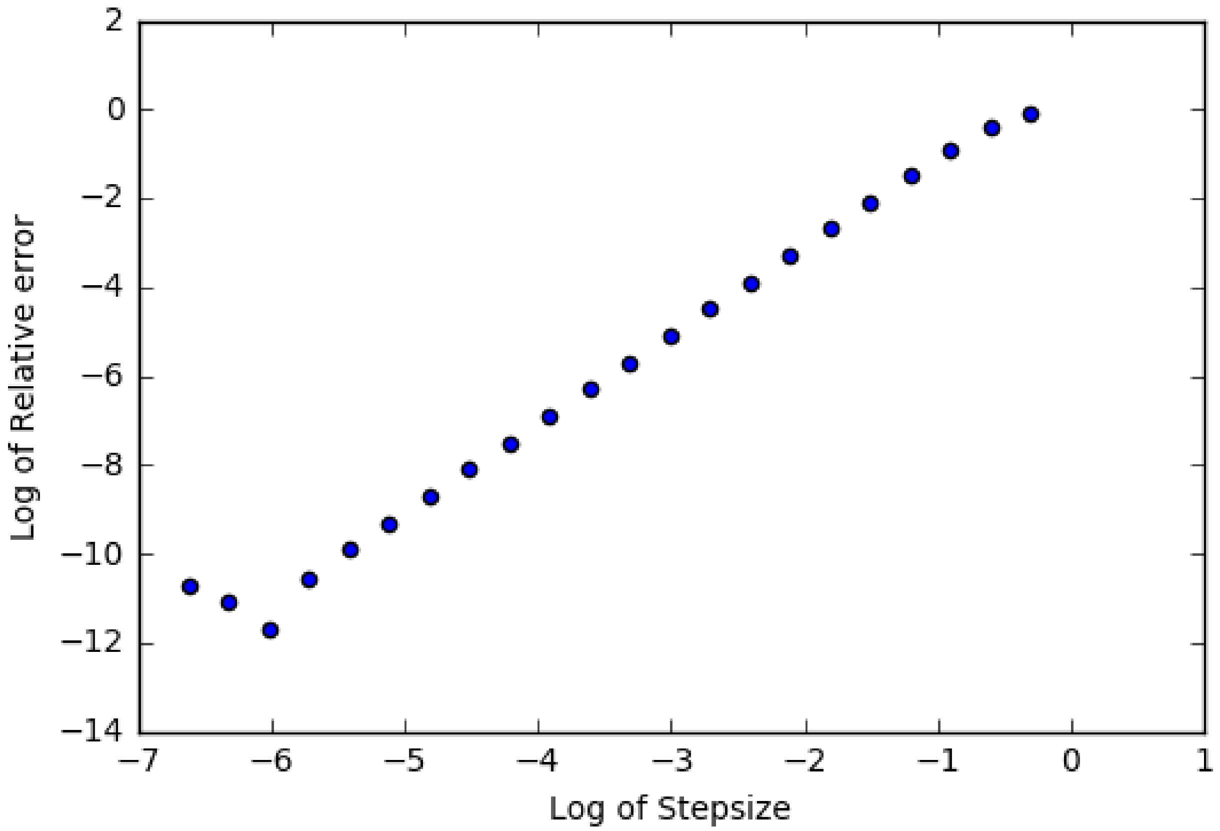

Figure 1: Plot of Log of Relative Error. On the x-axis, we see the log of the step-size. This is equivalent to looking at which i value we chose to use in order to approximate our solution. The y-axis shows the log of the relative error given in the equation above. What we see is that the log of error has a slope of approximately -2, as expected in our initial mathematic discussion. The error begins to increase again after $i = 6$ or $n = 10^6, h \approx 10^{-6}$. This is due to the truncation error that is getting overwritten by the number of bits. The increase in error can be accounted to machine error.