PHY 480 - Computational Physics
Project 1: Linear Algebra Methods

Thomas Bolden

February 12, 2016

Github Repository at https://github.com/ThomasBolden/PHY-480-Spring-2016

Abstract

In this project, a solution was found to the one-dimensional Poissson equation with Dirichlet boundary conditions. This was done by rewriting the Poisson equation as a set of linear equations and implimenting a tridiagonal matrix solver. It was confirmed that the tridiagonal solver approximation became more accurate as the step size decreased. Then, two other methods (Gaussian elimination, and LU decomposition) were implemented and compared. The error in the results of the LU defactorization method were found to be smaller than the error in the Gaussian elimination method.However, the calculation time became impractical after $n =$ steps.

Contents

Introduction

An important skill in physics is being able to efficiently solve differential equations. There are many situations in which differential equations can be solved as a system of linear equations. Such equations are called linear second-order differential equations, of the form

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + k^2(x)y = f(x) \;, \tag{1}$$

where $f(x)$ is the inhomogenous term, and k^2 is a real function.

An example of this being useful is in electromagnetism. Poisson's equation describes the electrostatic potential energy field Φ caused by a given charge density distribution ρ. The equation in three dimensions is

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

where the electrostatic potential and charge density are spherically symmetric. This allows one to simplify the equation to one dimension

$$\frac{1}{r^2}\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2\frac{\mathrm{d}\Phi}{\mathrm{d}r}\right) = -4\pi\rho(r) \tag{3}$$

which can be rewritten using the substitution $\Phi(r) = \phi(r)/r$

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

Now, like equation (1), this is linear second order in r. This becomes clear when we let $k^2(r) = 0$, $f(r) = -4\pi r\rho(r)$. If we let $\phi \to u$ and $r \to x$, we get the general Poisson equation

$$-u'' = f(x). \tag{5}$$

If we apply certain boundary conditions, we can rewritten as a set of linear equations. In this project, I explored several methods of solving systems of linear equations, including Gaussian elimination, LU decomposition, and analytically. In the section that follows, I outline the methods and algorithms used to write the C++ code, along with some examples of the output one should expect when running the code themself. The next section contains the useful results. In it, I compare the run times and efficiency of each method used, along with the magnitude of the associated error. Finally, the source code is presented for reference.

Methods

Given a differential equation of the form

$$-\frac{\mathrm{d}^2}{\mathrm{d}x^2}u(x) = f(x) \tag{6}$$

where $f(x)$ is continuous on the domain $x \in (0,1)$. We also assume the Dirichlet boundary conditions $u(0) = u(1) = 0$. We can define a discretized approximation second derivative of u as v_i with grid points $x_i = ih$ in the interval $x_0 = 0$ to $x_{n+1} = 1$, and with step lengths $h = 1/(n+1)$. The boundary conditions become $v_0 = 0 = v_{n+1}$. If the source term is $f(x) = 100e^{-10x}$, the exact closed-form solution is $u(x) = 1-(1-e^{-10})x-e^{-10x}$. The second derivative of u can be approximated as

$$-u'' = -\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i \quad \text{for } \ i = 1,\dots,n \tag{7}$$

This equation can be written as a set of linear equations of the form

$$\mathbf{Av} = \tilde{\mathbf{b}} \tag{8}$$

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & -1 & 2 & -1 & \ddots & & \vdots \\ \vdots & 0 & -1 & 2 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & & \ddots & \ddots & \ddots & -1 \\ 0 & 0 & \cdots & \cdots & 0 & -1 & 2 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix}, \quad \tilde{b}_i = h^2 f_i$$

One way to solve tridiagonal matrices (like the one above) is Gaussian elimination. Basically, Gaussian elimination uses the first equation to eliminate the first unknown x_1 from the remaining equations, and continues this trend for the remaining equations until an upper triangular matrix is formed. An upper triangular matrix can be easily solved using back substitution. This process requires a total of $\mathcal{O}(8n)$ floating point operations. The general algorithm is below.

Algorithm 1 Gaussian Elimination

function GaussElim(A)
  for $k = 1$ to $n-1$ **do**
    for $i = k+1$ to n **do**
      $a_{ik} = \frac{a_{ik}}{a_{kk}}$
      for $j = k+1$ to $n+1$ **do**
        $a_{ij} = a_{ij} - a_{ik} \times a_{kj}$
      end for
    end for
  end for
end function

We can compare this algorithm to the standard lower-upper (LU) decomposition. Equation (8) can also be written as $\mathbf{A} = LU$, assumin $\mathbf{A}$ is invertible.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & l_{n,n-1} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & \cdots & u_{1n} \\ 0 & u_{22} & \ddots & & \vdots \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & u_{n-1,n} \\ 0 & 0 & \cdots & 0 & u_{nn} \end{bmatrix} \tag{9}$$

This requires approximately $\frac{2}{3}n^3$ floating point operations on an n by n matrix.

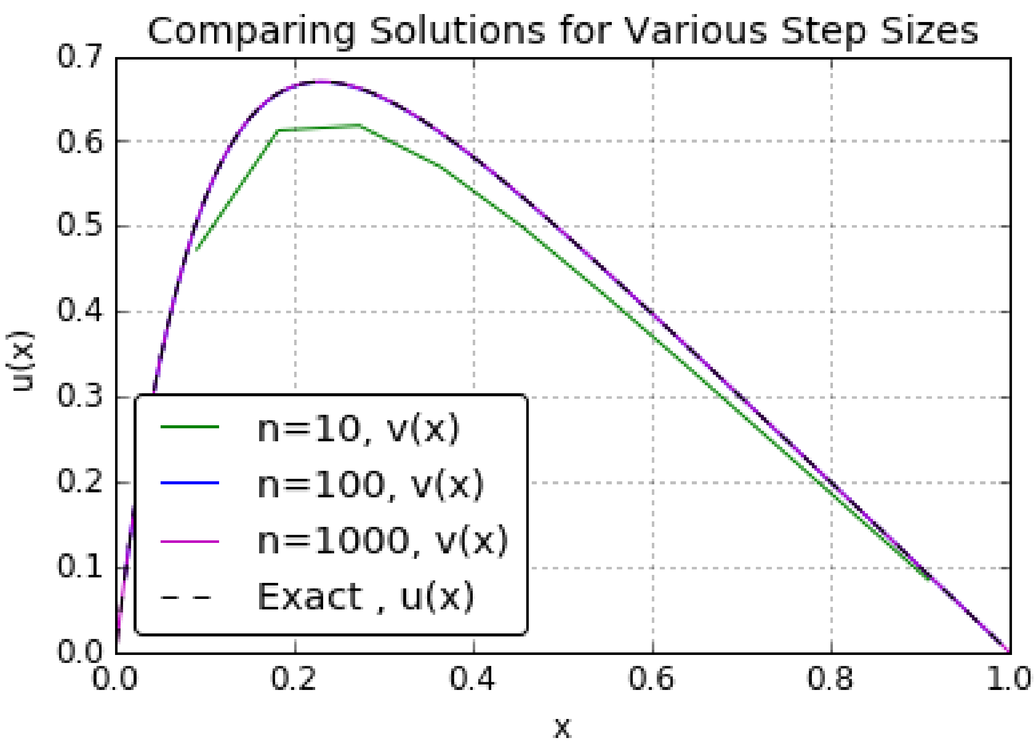
Algorithm 2 LU Decomposition

```
function LUdecomp(A)
    for i = 1 to n do
        for j = i to n do
            L_ik U kj = a_ij
        end for
        for j = i + 1 to n do
            L_jk U_ki = a_ji
        end for
    end for
end function
```

The final task is to time each of the algorithms above. This is done most easily using the "time.h" header in C++.

Results

The graph below shows how the accuracy of the traditional tridiagonal matrix solver increases as the step length increases.



Conclusions

I have show how the accuracy of the solution to a linear system depends on the size of the system. For smaller systems, the number of floating point operations did not impede the ability of the user to impliment more accurate methods.

Code

../Code/Project1.cpp

```
/*

Project 1 a,b - Vector and Matrix Operations
Solving a tridiagonal matrix

*/

#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>
#include <string>
#include <armadillo>
#include "time.h"

using namespace std;
using namespace arma;

ofstream myfile;

// -~- Functions -~- \\

double f(double x){
    return 100*exp(-10*x);
}

double analyze(double x){
    return 1.0-(1-exp(-10))*x-exp(-10*x);
}

// -~- Main -~- \\

int main(){

    // -~- Declaration of Variables -~- \\

    int n;
    string outfilename;

    cout << "Dimensions of the nxn matrix: ";
    while(!(cin >> n)){
        cout << "Not a valid number! Try again: ";
        cin.clear();
```

```
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    }
    cout << "Enter a name for the output file: ";
    cin >> outfilename;

    // -~- Body of the program -~- \\

    clock_t start , finish ;
    start = clock();

    double h = (1.0) / (n + 1.0);
    double *x = new double[n+2];
    double *tildeb = new double[n+1];
    tildeb[0] = 0;

    int *a = new int[n+1];
    int *b = new int[n+1];
    int *c = new int[n+1];

    double *diag_temp = new double[n+1];

    double *u = new double[n+2]; // Analytical solution
    double *v = new double[n+2]; // Numerical solution

    u[0] = 0;
    v[0] = 0;

    for (int i=0; i<=n+1; i++) {
        x[i] = i*h;
    }

    for (int i=1; i<=n; i++) {
        tildeb[i] = h*h*f(x[i]);
        u[i] = analyze(x[i]);
        a[i] = -1;
        b[i] = 2;
        c[i] = -1;
    }

    c[n] = 0;
    a[1] = 0;

    // Algorithm for finding v:
    double b_temp = b[1];
    v[1] = tildeb[1]/b_temp;
    for (int i=2;i<=n;i++) {
       diag_temp[i] = c[i-1]/b_temp;
       b_temp = b[i] - a[i]*diag_temp[i];
       v[i] = (tildeb[i]-v[i-1]*a[i])/b_temp;
    }

    // Row reduction; backward substition:
    for (int i=n-1;i>=1;i--) {
        v[i] -= diag_temp[i+1]*v[i+1];
```

```
    }

    finish = clock() - start;

    double processortime = ((double)finish)/CLOCKS_PER_SEC;

    // -~- writing results to file, to be read and graphed in python -~- \\

    myfile.open(outfilename);
    //myfile << setiosflags(ios::showpoint | ios::uppercase); //sci notation
    myfile << "Solution to tridiagonal matrix of size n=" << n << endl;
    myfile << "Time elapsed = " << processortime << " seconds" << endl ;
    myfile << "       x:             u(x):          v(x):  " << endl;
    for (int i=1;i<=n;i++) {
       myfile << setw(15) << setprecision(8) << x[i];
       myfile << setw(15) << setprecision(8) << u[i];
       myfile << setw(15) << setprecision(8) << v[i] << endl;
    }

    myfile.close();

    delete [] x;
    delete [] tildeb;
    delete [] a;
    delete [] b;
    delete [] c;
    delete [] u;
    delete [] v;

    return 0;

}
```

../Code/plots.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 14 00:10:43 2016

@author: Thomas
"""

# testing testing 123

import math
import matplotlib.pyplot as plt

def convert(rawdata):

    x = []
    u = []
    v = []

    file = open(rawdata , 'r')
```

```
    valid_data = file.readlines()[3:]

    for line in valid_data:

        xuv = line.split()
        x.append(float(xuv[0]))
        u.append(float(xuv[1]))
        v.append(float(xuv[2]))

    file.close()

    return x, u, v

x10 , u10 , v10 = convert('n=10')
x100 , u100 , v100 = convert('n=100')
x1000 , u1000 , v1000 = convert('n=1000')

exact = []

for value in x1000:

    f = 1.0-(1-math.exp(-10))*value-math.exp(-10*value)

    exact.append(f)

hfont = {'fontname':'Courier'}

fig , ax = plt.subplots(1)

ax.plot(x10,v10,'g-',label='n=10, v(x)')
ax.plot(x100,v100,'b-',label='n=100, v(x)')
ax.plot(x1000,v1000,'m-',label='n=1000, v(x)')
ax.plot(x1000,exact,'k--',label='Exact , u(x)')
ax.set_xlabel('x',**hfont)
ax.set_ylabel('u(x)',**hfont)
ax.legend(loc='lower left',fancybox='True')
ax.set_title('Comparing Solutions for Various Step Sizes',**hfont)
ax.grid()
plt.show()
```

References

[1] M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2013).

[2] W. McLean, *Poisson Solvers*, Northwestern University (2004).