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

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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 $\phi \rightarrow u$ and $r \rightarrow x$, 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 $v_0 = 0 = v_{n+1}$. 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

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

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

If the source term is $f(x) = 100e^{-10x}$

Results

.

$$d\tilde{i} = i/i - 1$$

relative error should give a flat line

Conclusions

.

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>
```

```cpp
#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;

    float processortime = ((float)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(10) << x[i];
       myfile << setw(15) << setprecision(10) << u[i];
       myfile << setw(15) << setprecision(10) << v[i] << endl;
    }

    myfile.close();

    delete [] x;
    delete [] tildeb;
    delete [] a;
```

```
    delete [] b;
    delete [] c;
    delete [] u;
    delete [] v;

    return 0;

}
```

../Code/plots.py

```
# From matplotlib examples

# obvi not real useful yet

import numpy as np
import matplotlib.pyplot as plt

plt.subplots_adjust(hspace=0.4)
t = np.arange(0.01, 20.0, 0.01)

# log y axis
plt.subplot(221)
plt.semilogy(t, np.exp(-t/5.0))
plt.title('semilogy')
plt.grid(True)

# log x axis
plt.subplot(222)
plt.semilogx(t, np.sin(2*np.pi*t))
plt.title('semilogx')
plt.grid(True)

# log x and y axis
plt.subplot(223)
plt.loglog(t, 20*np.exp(-t/10.0), basex=2)
plt.grid(True)
plt.title('loglog base 4 on x')

# with errorbars: clip non-positive values
ax = plt.subplot(224)
ax.set_xscale("log", nonposx='clip')
ax.set_yscale("log", nonposy='clip')

x = 10.0**np.linspace(0.0, 2.0, 20)
y = x**2.0
plt.errorbar(x, y, xerr=0.1*x, yerr=5.0 + 0.75*y)
ax.set_ylim(ymin=0.1)
ax.set_title('Errorbars go negative')


plt.show()
```

References

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

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