PHYS 905 - Project 1

Terri Poxon-Pearson

February 7, 2016

Contents

1 Introduction

In this report we study different algorithms which can be used to solve linear, second order, differential equations. As as example, we look at the example of Poisson's equation with Dirichlet boundary conditions. For each method used, we cast Poisson's equation as a triadiagonal matrix equations. First we employ a general algorithm for solving tridiagonal matricies. Next, we specialize this algorithm to the specific features of this problem and study the effect on the number of Floating Point Operations per Second (FLOPS) which manifests in the CPU time. Finally, we use a an algorithm for LU decomposition, a much more general method which does not require a tridiagonal form of the matrix. We study how this powerful algorithm scales with matrix size.

In the following report, we introduce the matrix equation we solve, outine the general algorithm for solving tridiagonal matricies, and discuss how this algorithm

can be tailored for our specific case. These discussions will incude discussions of the FLOPS required for each calculation. We will end the Methods portion of this report with a brief introduction to the LU decomposition algorithm. Then we wil discuss the implementation of these algorithms and the codes we used for our calculations. We will then present our results which include a study of convergence to the annalytical solutions, a comparision of computational speeds between different algorithms, and a detailed discussion of errors in the calculation. Finally, we will present some conclusions and perspectives.

2 Methods

2.1 Poisson's Equation in Differential Form

In this project we will be using Poisson's equation as an example of a linear, second order, differential equation. This equation describes the electrostatic potential generated by a charge distrubution and can be written as

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

If we assume spherical symmetry this equation can be reduced to a one dimmensional equation in r which can be generalized to

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

In this project we described Dirichlet boundary conditions, meaning that

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

In our case, we will assime that sources term is $f(x) = 100e^{-10x}$, giving the closed form solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. We will use this exact solution to study the results of our algorithm.

2.2 Poisson's Equation in Matrix Form

In order to solve the differential equations, the functions must be discritized. In this work, u will be approximated byt v_i where $x_i = ih$. The discretized second derivative can be approximated using the three point formula

$$\frac{d^2u}{dx^2} \approx \frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} + O(h^2).$$

A full derivation of this expression can be found in [1]. Plugging this into our generalized Poisson's equation, we are left with

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

This now gives us a set of equations corespondeing to each f_i. This set of equations can be cast as a matrix equation

$$\mathbf{Av} = \tilde{\mathbf{b}}.$$

Because there are only up to 3 v_i terms in each equation corresponding to each f_i, this matrix equation will be tridiagonal and have the form

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

and $\tilde{b}_i = h^2 f_i$.

2.3 General Tridiagonal Matrix Solver

We can re-write the matrix expression in terms vectors a, b, c of length n. These vectors will not inculde the endpoints, as those values are already fixed by the boundary conditions. The equation becomes

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & \dots & \dots & \dots \\ e_1 & d_2 & e_2 & \dots & \dots & \dots \\ & e_2 & d_3 & e_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & e_{n-2} & d_{n-1} & e_{n-1} \\ & & & & e_{n-1} & d_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{bmatrix}.$$

This equation can be solved using a simplified form of Gaussian elimination which involves two steps: a decomposition and forward substitution, follwed by a backward substitution. In the case of our matrix, we want to first eliminate the e_1 from the second row. To do this, we multiply the first equation by e_1/e_1 and subtract it from the second equation. This leaves us with

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & \dots & \dots & \dots \\ 0 & d_2 - \frac{e_1^2}{d_2} & e_2 & \dots & \dots & \dots \\ & e_2 & d_3 & e_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & e_{n-2} & d_{n-1} & e_{n-1} \\ & & & & e_{n-1} & d_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 - \tilde{b}_1 \frac{e_1}{d_1} \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{bmatrix}.$$

These new diagonal elements are will be renamed $\tilde{d}_i$ and the new sorce terms will be renamed $\tilde{b}^*$. This process is then repeated for each row of the matrix, finally leaving us with

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & \dots & \dots & \dots \\ 0 & \tilde{d}_2 & \dots & \dots & \dots & \\ & 0 & \tilde{d}_3 & e_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & \tilde{d}_{n-1} & e_{n-1} \\ & & & & 0 & \tilde{d}_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}^* \\ \dots \\ \dots \\ \dots \\ \tilde{b}^*_n \end{bmatrix}.$$

Where the general expressions for the matrix elements are:

$$\tilde{d}_i = d_i - \frac{e^2_{i-1}}{\tilde{d}_{i-1}} \qquad \text{and} \qquad \tilde{b}^*_i = \tilde{b}_i - \tilde{b}^*_{i-1}\frac{e_{i-1}}{\tilde{d}_{i-1}} \qquad \text{for} \qquad i = 2, 3, ...n$$

Next, we can do a backward substitution begining with the final row of the matrix equation. Solving the the unknown function, we get that $v_n = \tilde{b}^*_n/\tilde{d}_n$. Now that v_n is knows, we substitute this value into the equation from the second to last row and solve for v_{n-1}. We can continue this backward substitution until all values v_i are know. The general formula for these elements are

$$v_i = \frac{\tilde{b}^*_i - e_i u_{i+1}}{\tilde{d}_i} \qquad \text{for} \qquad i = n-2, n-1, ...1.$$

In the next section we will explore the efficiency of an algorithm which uses this method.

2.4 FLOPS in

3 Code and Implementation

4 Results and Discussion

5 Conclusions

6 Appendices

References

[1] Hjorth-Jensen, Mortehn. Computational Physics, Lecture Notes Fall 2015. August 2015.

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 $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$ (convince yourself that this is correct by inserting the solution in the Poisson equation). We will compare our numerical solution with this result in the next exercise.

Project 1 b): We can rewrite our matrix **A** in terms of one-dimensional vectors a, b, c of length $1 : n$. Our linear equation reads

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{bmatrix}.$$

Note well that we do not include the endpoints since the boundary conditions are used resulting in a fixed value for v_i. A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. Develop a general algorithm first which does not assume that we have a matrix with the same elements along the diagonal and the non-diagonal elements. The algorithm for solving this set of equations is rather simple and requires two steps only, a decomposition and forward substitution and finally a backward substitution.

Your first task is to set up the general algorithm (assuming different values for the matrix elements) for solving this set of linear equations. Find also the precise number of floating point operations needed to solve the above equations.

Then you should code the above algorithm and solve the problem for matrices of the size 10×10, 100×100 and 1000×1000. That means that you select $n = 10$, $n = 100$ and $n = 1000$ grid points.

Compare your results (make plots) with the closed-form solution for the different number of grid points in the interval $x \in (0, 1)$. The different number of grid points corresponds to different step lengths h.

Project 1 c): Use thereafter the fact that the matrix has identical matrix elements along the diagonal and identical (but different) values for the non-diagonal elements. Specialize your algorithm to the special case and find the number of floating point operations for this specific tri-diagonal matrix. Compare the CPU time with the general algorithm from the previous point for matrices up to $n = 10^6$ grid points.

Project 1 d): Compute the relative error in the data set $i = 1, \dots, n$,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 extract the max value of the relative error. Try to increase n to $n = 10^7$ or higher. Make a table of the results and comment your results.

Project 1 e): Compare your results with those from the LU decomposition codes for the matrix of sizes 10×10, 100×100 and 1000×1000. Here you should use the library functions provided on the webpage of the course. Alternatively, if you use armadillo as a library, you can use the similar function for LU decomposition. The armadillo function for the LU decomposition is called *LU* while the function for solving linear sets of equations is called *solve*. Use for example the unix function *time* when you run your codes and compare the time usage between LU decomposition and your tridiagonal solver. Alternatively, you can use the functions in C++, Fortran or Python that measure the time used.

Make a table of the results and comment the differences in execution time How many floating point operations does the LU decomposition use to solve the set of linear equations? Can you run the standard LU decomposition for a matrix of the size $10^5 \times 10^5$? Comment your results.

To compute the elapsed time in c++ you can use the following statements

```
...
#include "time.h"   //  you have to include the time.h header
int main()
{
    // declarations of variables
    ...
    clock_t start, finish;  //  declare start and final time
    start = clock();
    // your code is here, do something and then get final time
    finish = clock();
    ( (finish - start)/CLOCKS_PER_SEC );
...
```

Similarly, in Fortran, this simple example shows how to compute the elapsed time.

```
PROGRAM time
 REAL :: etime          ! Declare the type of etime()
 REAL :: elapsed(2)     ! For receiving user and system time
 REAL :: total          ! For receiving total time
 INTEGER :: i, j

 WRITE(*,*) 'Start'

 DO i = 1, 5000000
      j = j + 1
 ENDDO

 total = ETIME(elapsed)
 WRITE(*,*) 'End: total=', total, ' user=', elapsed(1), &
              ' system=', elapsed(2)

END PROGRAM time
```

Introduction to numerical projects

Here follows a brief recipe and recommendation on how to write a report for each project.

- Give a short description of the nature of the problem and the eventual numerical methods you have used.
- Describe the algorithm you have used and/or developed. Here you may find it convenient to use pseudocoding. In many cases you can describe the algorithm in the program itself.
- Include the source code of your program. Comment your program properly.
- If possible, try to find analytic solutions, or known limits in order to test your program when developing the code.
- Include your results either in figure form or in a table. Remember to label your results. All tables and figures should have relevant captions and labels on the axes.
- Try to evaluate the reliabilty and numerical stability/precision of your results. If possible, include a qualitative and/or quantitative discussion of the numerical stability, eventual loss of precision etc.
- Try to give an interpretation of you results in your answers to the problems.
- Critique: if possible include your comments and reflections about the exercise, whether you felt you learnt something, ideas for improvements and other thoughts you've made when solving the exercise. We wish to keep this course at the interactive level and your comments can help us improve it.
- Try to establish a practice where you log your work at the computerlab. You may find such a logbook very handy at later stages in your work, especially when you don't properly remember what a previous test version of your program did. Here you could also record the time spent on solving the exercise, various algorithms you may have tested or other topics which you feel worthy of mentioning.

Format for electronic delivery of report and programs

The preferred format for the report is a PDF file. You can also use DOC or postscript formats or as an ipython notebook file. As programming language we prefer that you choose between C/C++, Fortran2008 or Python. The following prescription should be followed when preparing the report:

- Use your github repository to upload your report. Indicate where the report is by creating for example a **Report** folder. Please send us as soon as possible your github username.

- Place your programs in a folder called for example **Programs** or **src**, in order to indicate where your programs are. You can use a README file to tell us how your github folders are organized.
- In your git repository, please include a folder which contains selected results. These can be in the form of output from your code for a selected set of runs and input parameters.
- In this and all later projects, you should include tests (for example unit tests) of your code(s).
- Comments from us on your projects, with score and detailed feedback will be emailed to you.

Finally, we encourage you to work two and two together. Optimal working groups consist of 2-3 students. You can then hand in a common report.