Project 1

Abstract

In this project we solved a differential equation by manipulating it into a tridiagonal matrix and we developed our own tridiagonal solver that proved much more efficient for this problem than Armadillo's LUdecompostion. Lastly we discovered it was more effective to multiply matrices row-wise than column-wise.

Introduction

The main task for this project is to solve the differential equation

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

where $f(x) = 100e^{-10x}$. We are going to solve this numerically and the derivative we are going to approximate by

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

Our first task is to transform this into a matrix, and then come up with an algorithm that will solve this problem. We also have to compare the error we get for different step lengths h, e.g. the number of points we use. All of this we have to implement into c++, a language I have never used before. We are then going to implement the LUdecomposition method to compare it to our own algorithm. The way we are going to compare it is by measuring the time taken for the programs to run.

Methods

The first thing we have to show is that $-\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i$ for $i = 1, \ldots, n$ can be written on the form

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

where

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{pmatrix}, \quad \tilde{b}_i = h^2 f_i. \tag{1}$$

The first thing we realise is that $-\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i$ can be written as a set of linear equations.

$$-v_1 + 2v_0 = \tilde{b_0}$$

$$-v_2 + 2v_1 - v_0 = \tilde{b_1}$$
$$-v_2 + 2v_1 - v_0 = \tilde{b_2}$$
$$\cdots = .$$
$$\cdots = .$$
$$-v_{n+1} + 2v_n - v_{n-1} = \tilde{b_n}$$

This we can again write as

$$v_1 \begin{pmatrix} 2 \\ -1 \\ \dots \\ \dots \\ \dots \\ 0 \end{pmatrix} + v_2 \begin{pmatrix} -1 \\ 2 \\ -1 \\ \dots \\ \dots \\ 0 \end{pmatrix} + v_3 \begin{pmatrix} 0 \\ -1 \\ 2 \\ -1 \\ \dots \\ 0 \end{pmatrix} + \cdots + v_{n-1} \begin{pmatrix} 0 \\ 0 \\ \dots \\ -1 \\ 2 \\ -1 \end{pmatrix} + v_n \begin{pmatrix} 0 \\ 0 \\ \dots \\ \dots \\ -1 \\ 2 \end{pmatrix} = \begin{pmatrix} \tilde{b_1} \\ \tilde{b_2} \\ \dots \\ \dots \\ \dots \\ \tilde{b_n} \end{pmatrix} \quad (2)$$

Which can be written as

$$\begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b_1} \\ \tilde{b_2} \\ \dots \\ \dots \\ \dots \\ \tilde{b_n} \end{pmatrix}$$

Our next task is to come up with an algorithm that could solve this problem. We already heard in the lectures that all that is needed for this problem is a subtraction and a forward substitution and then a backwards substitution. The way I did this was to remove the first non-zero element on each line by subtracting the line with the line above it. With that approach we get an upper triangular matrix where we can easily find the solution. We have the last element of the solution in the last row, so we can just work our way backwards.

I then implemented this into c++ and got this program:

```
#include <iostream>      //including modules I might need
#include <cmath>
#include <fstream>

using namespace std;     // preventing me from writing std:: all the
    time

int main(){
        // Declaring all numbers and variables
        int n;
        int i;
```

```
double h;
n = 100;
double b[n];
double a[n];
double c[n];
double v[n];
double f[n];
double x[n];
double u[n];
//double eps[n];
double btemp;
double temp[n];
double y;
double z;

for(i=0; i < n ; i++){
        // The different diagonals in the matrix
        b[i] = 2.0;
        a[i] = -1.0;
        c[i] = -1.0;
}

h = 1./(n+1);               // The step-length
btemp = b[0];               // The temporary b's in the solver
        // The initial values
x[0] = 0*h;
f[0] = pow(h, 2.0)*100*exp(-10*x[0]);
v[0] = f[0]/btemp;

for(i=1; i < n; i++){
        x[i] = i*h;       // Different points where we calculate
            the derivatives
        f[i] = pow(h,2.0)*100*exp(-10*x[i]);      // The
            function we consider
        temp[i] = c[i-1]/btemp;                   // The forward
            substitution
        btemp = b[i] -a[i]*temp[i];               //
        v[i] = (f[i] - a[i]*v[i-1])/btemp;        // The
            numerical solution
        u[i] = 1-(1-exp(-10))*x[i]-exp(-10*x[i]);         // The
            exact solution
        //eps[i] = log10(abs((v[i]-u[i])/u[i]));// The
            relative error
}
for(i=n-1; i>=0 ; i--){
```

```
                v[i] -= temp[i+1]*v[i+1];          // The backward
                    substitution
        }
        /*ofstream myfile;        // Writing the results to file
        myfile.open ("project.txt", ios::out | ios::app);
        y = 0;
        for(i=0; i<n; i++){
                // The largest relative error value
                z = abs(eps[i]);
                if (z > y){
                        y = z;
                }
                //cout << v[i] << endl;
                //myfile << v[i] << endl;
        }
        myfile << y << endl;
        myfile.close(); */
}
```

For plotting the results for various step lengths I wrote it to file and used python to plot.

```
# Python-program for plotting the results
from scitools.std import * # INF1100-package that include everything
    we need

n = 100          # The n from previously
infile = open('project.txt', 'r')       # Opens the file we wrote to
lines = infile.readlines()                      # Reading in the lines
infile.close()                                          # Closing the
    file
v = zeros(n)                                            # Empty vector
     to store the values

for i in range(n):
        v[i] = lines[i]                                 # Putting the
            values into the vector

x = linspace(0, 1, n)                           # The x-values used
u = 1-(1-exp(-10))*x-exp(-10*x)          # The exact solution

# The plot commands, plotting the numerical and exact solution
    together
plot(x, v)
legend('numerical')
hold('on')
```

```
plot(x, u)
legend('exact')

hardcopy('plot.png')
raw_input('enter')
```

I calculated the maximum error for different values of n, as instructed in the exercise text. How I did it is included in the main program.

Then it is on to the LU-decomposition, we shall solve the exact same equation as before just now it is another method involved. To solve this we use the c++ package armadillo and its built in function LU that splits a matrix into an upper triangular and a lower triangular matrix. With this we are quipped to solve the entire equation and we only need to use the built-in armadillo function solve(). We see that the LU-decomposition gives the exact same results as the tridiagonal solver from earlier.

```
#include <iostream>      // Including modules
#include <armadillo>

using namespace std;
using namespace arma;

int main(){
        // Defining the variables
        int n;
        int i;
        int j;
        n = 100;          // The dimension of the matrices/vectors
        vec x(n);
        double h = 1./(n+1);      // The step length
        mat A = 2*eye<mat>(n,n);          // The matrix we use
        for(i=0; i < n;i++){x[i] = i*h;}          // Filling the x
            vector
        for(i=0; i < n; i++){             // Filling the matrix
                for(j=0; j < n; j++){
                        if(j == i-1){
                                A(i,j) = -1;}
                        else if(j == i+1){
                                A(i,j) = -1;}
                        }
                }

        mat L, U;         // Initilazing the matrices we use to solve
            the problem
        lu(L, U, A);      // The LU-decomposition
        //mat B = L*U;    // Checking if it worked
        vec b(n);         // The vector with the function
```

```
        for(i=0; i < n ;i++){b(i) = pow(h, 2)*100*exp(-10*x(i));}

        vec y = solve(L, b);      // The solution for
        vec X = solve(U, y);      // the LU-decomposition

        //cout << X << endl;

}
```

Our task however is to measure the difference in time and this I did by the built-in Unix command "time" before running the program. The last task asked us to multiply two matrices, row-wise and column-wise, to see which is the most effective. I again use armadillo to create two matrices of equal size with random elements, and then use the code presented in the text to multiply them. I used the built-in "time" function in Unix to measure the time the programs use. The program for multiplying row-wise:

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

main(){
        int i;
        int j;
        int k;
        int n = 100;
        mat A = zeros<mat>(n,n);
        mat B = randu<mat>(n,n);
        mat C = randu<mat>(n,n);
        for(i=0; i<n; i++){
                for(j=0; j<n; j++){
                        for(k=0; k<n; k++){
                                A(i,j) += B(i,k)*C(k,j);
                                }
                        }
                }
        }
```

and column-wise:

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;
```

```
main(){
        int i;
        int j;
        int k;
        int n = 1000;
        mat A = zeros<mat>(n,n);
        mat B = randu<mat>(n,n);
        mat C = randu<mat>(n,n);
        for(j=0; j<n; j++){
                for(i=0; i<n; i++){
                        for(k=0; k<n; k++){
                                A(i,j) += B(i,k)*C(k,j);
                                }
                        }
                }
        }
```

Results

The first thing we are asked to say something about is the number of flops in our algorithm. In my algorithm I get $8n$ flops, if I use the fact some of the elements can only have the values -1 and 2 I could probably get it down some more, but that will not solve a general tridiagonal matrix equation. It is either way much better than the LU-decomposition that uses $\frac{2}{3}n^3$ flops. Standard Gaussian elimination takes $\frac{2}{3}n^3$ flops, so that would be even more ineffective.

From our tridiagonal solver and the LU decomposition we got some plots, unfortunately I could not get plots for n = 10000 for LU decomposition. We see from the plots that both methods give about the same result, and that for small n they are not that accurate. We have to use almost 1000 points before we get acceptable results. The results for the tridiagonal solver for n = 10000 and 100000 are almost perfect and they do not take that long time to produce, while for LU you use a lot of time for the same results. Figure 1 shows us n = 10, figure 2 n = 100, figure 3 n = 1000 and figure 4 shows us the tridiagonal solution for n = 10000 and 100000.

I have also computed the maximum relative error for each n for the tridiagonal solver and plotted them with respect to $log_{10}(h)$ and $log_{10}(n)$. The relavtive error in a table:

$log_{10}(n)$	ϵ
1	1.76259
2	0.348336
3	0.30541
4	0.301465
5	0.301073

Figure 5shows the relative error as a plot for different n values and h values. We say they are almost the exact inverse of each other. The error falls steeply

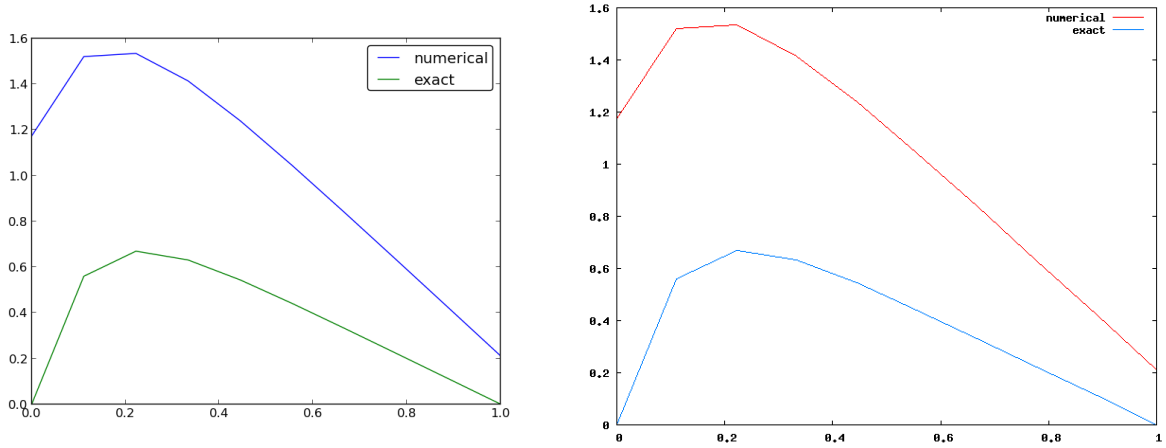

Figure 1: Tridiagonal solver to the left, LU decompostion to the right for n=10

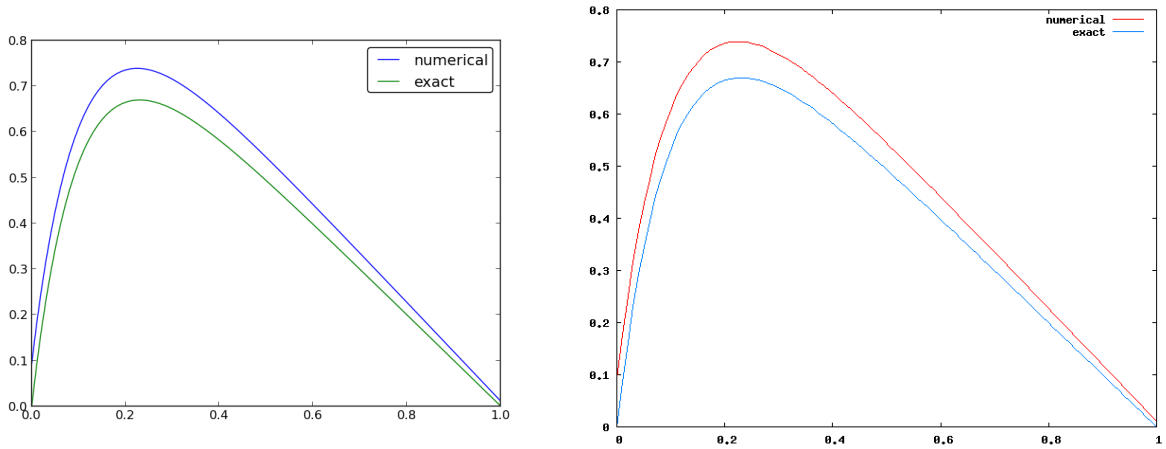

Figure 2: Tridiagonal solver to the left, LU decompostion to the right for n=100

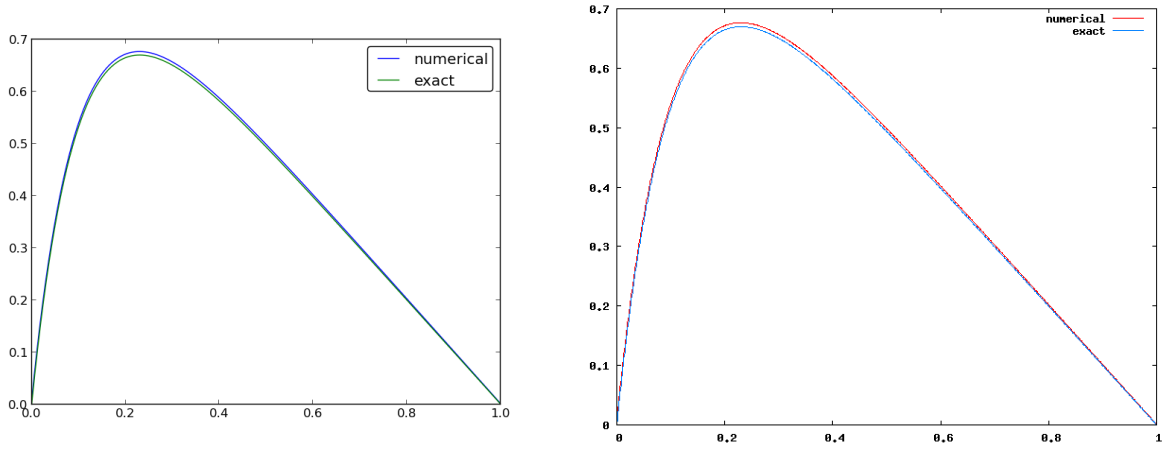

Figure 3: Tridiagonal solver to the left, LU decompostion to the right for n=1000

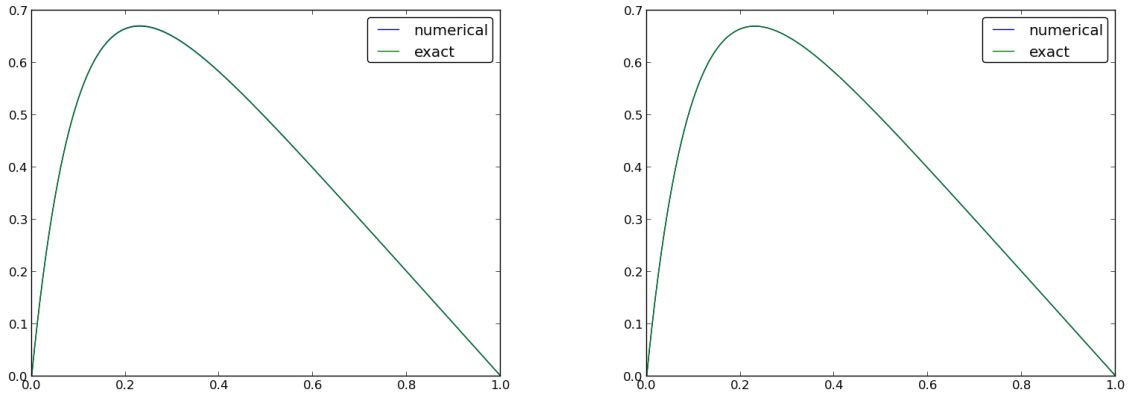

Figure 4: To the left tridiagonal solver for n = 10000, right for n = 100000

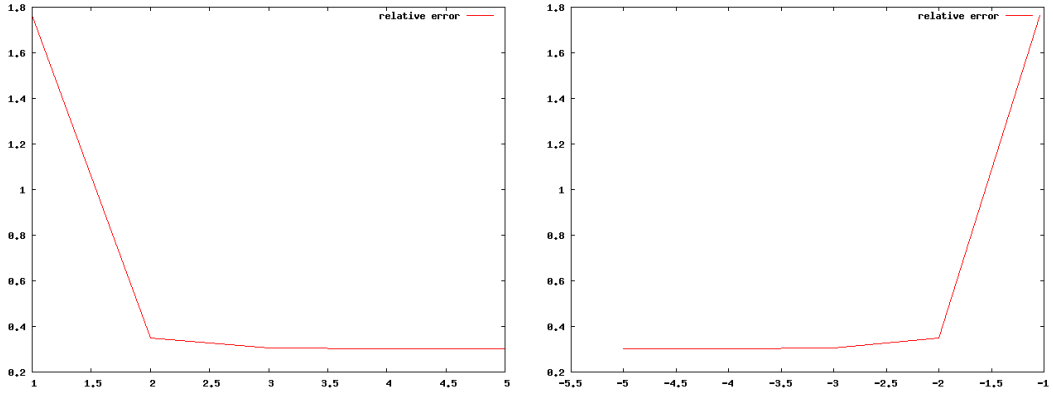

Figure 5: Relative error for different n values for the left plot, the right plot is for different h values

from n = 10 to 100, but after that it seeks towards 0.3.

Time usage of the different solvers I have said much about and here comes some numbers on the time usage for different n for the two numerical methods. We see what I have said earlier that the tridiagonal solver is extremely efficient for this problem, while the otherwise efficient LU decomposition provides the same results after a much longer time.

$log_{10}(n)$	time tridia	time LU dec
1	0.015s	0.018s
2	0.013s	0.019s
3	0.014s	0.365s
4	0.215s	4min30.182s
5	0.254s	armadillo hates me

We also have that last exercise to consider where we shall see what is the most efficient of row and column multiplication of matrices. We are asked to consider the case $10^4 \times 10^4$, that would take many hours for

each of the multiplications, and I soon gave that up. The results I got in the end was:

$log_{10}(n)$	row	column
1	0.002s	0.002s
2	0.027s	0.029s
3	27.467s	30.902s

We see that row multiplication goes slightly faster than column multiplication. For 10^4 the time difference will probably grow to a few hours , which is significant.

Conclusion

We have seen how easily one can solve differential equations numerically if you know the right methods, and the correct step length for the precision you want. It was not very much difference in time even if you used a step length $\frac{1}{10}$ of what you need if you use the correct method.

In this project we have discovered that a simple problem can be solved considerably faster by just using a different algorithm, and that seemingly equal things like row multiplication and column multiplication, take a different amount of time. We also have a basis for future problems where we need to equations with a tridiagonal matrix. It was also a nice first experience with c++ and latex, the exercise gave me great experience for future work with those.