FYS4150 - Computational Physics

PROJECT 1

Solving a differential equation using matrices

Jon Vegard Sparre
jonvsp@uio.no

Dato: September 11, 2015

Abstract

In this project we're solving a differential equation with two different methods, with a algorithm for a tridiagonal matrix equation and with Armadillo's `solve`-function. We look at the precision of the two methods and their speed.

Note: I have collaborated with Henrik S. Limseth and Anne-Marthe Hovda in this project, but we have written our own programs and therefore deliver different reports.

Link to my GitHub domain: `https://github.com/jonvegards/FYS4150`

Exercise a)

We're going to solve the one dimensional Poisson equation given on the form

$$-u''(x) = f(x), \tag{1}$$

with Dirichlet boundary conditions.

Rewriting of differential equation We'll show that we can write our differential equation (1) as the matrix equation $Av = \tilde{b}$. First we discretize our equation such that we have $-u_i'' = f_i$, then we use the three point formula for a second derivative to get

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i, \quad \text{for } i = 0, \dots, n,$$

where we multiply both sides by h^2 such that we get $\tilde{b}_i = h^2 f_i$ where $h = 1/(n+1)$. Then we simply write the matrix A given in the exercise text and multiply it by v to see what we get

$$Av = \begin{pmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ \vdots \\ \vdots \\ v_n \end{pmatrix}$$

$$= \begin{pmatrix} 2v_1 & -v_2 & 0 & 0 & \dots & 0 \\ -v_1 & 2v_2 & -v_3 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & -v_n \\ 0 & \dots & \dots & 0 & -v_{n-1} & 2v_n \end{pmatrix},$$

where we recognize each row as the three point formula for the i-th component of v, *e.g.* for $i = 2$ we have

$$-v_1 + 2v_2 - v_3 = \tilde{b}_2.$$

Thus we've shown that we can write the given differential equation as a system of linear equations.

Checking the given solution We're given the source term $f(x) = 100e^{-10x}$, and we'll check if the solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$ is correct, by inserting it into to Poisson's equation.

$$
\begin{aligned}
u(x) &= 1 - (1 - e^{-10})x - e^{-10x} \\
u'(x) &= (1 - e^{-10}) + 10e^{-10x} \\
u''(x) &= -100e^{-10x} = -f(x).
\end{aligned}
$$

It is indeed correct and we can move on to next exercise!

Exercise b)

The matrix A is rewritten as three column vectors in order to optimize our code, we can do this because the matrix A is zero everywhere except at the three diagonals in the middle. We then get n equations,

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i.$$

To solve this we do a forward and then a backward substitution. The algorithm is described below

```
Declaring variables
- Vectors a, b, c which is the subdiagonal, diagonal and superdiagonal of the matrix A.
- Vector v which is the one we're solving for.
- Vector f which is the source term in our diff. eq. discretized.

Setting initial conditions
...

We declare then a temporary varible which we'll use in the row reduction of A.
btemp = b_1;
The first element of v is then
v_1 = f_1/b_1

Make for loops that calculate v_i
for( i up to n):
    To do the substitution we need a temporary variable such that we can perform the row reduction of A, this we have to save this variable in order to do the backward substitution correct
    temp_i = c_{i-1} / btemp

    btemp = b_i - a_i*temp_i
    v_i = ( f_i - a_i * v_(i-1) ) / btemp

Then we do the backward substitution, starting from the second last element in v and moving backwards to i=1.
for(i=n-1 to  i>= 1):
    v_i = v_i - temp_(i+1) * v_(i+1)
```

For our calculation we have $8n$ FLOPS, $6n$ in the first for-loop and then $2n$ more in the second for-loop, a LU-decomposition requires $(2/3)n^3$ FLOPS so it's much more efficient to do it the way it's been done in this project.

I made a function to calculate the array v which i named `num_solve.cpp`, the main program is calling on this function when solving the problem using the tridiagonal solver. The main program is giving the parameters and variables a, b, c, f and no. of grid points n, and also the object v which is being filled by the function. All variables are called by reference, in order to avoid moving on large objects in the memory, except the integer number n.

```cpp
#include <iostream>
#include <time.h>
#include "num_solve.h"

using namespace std;

/////////////////////////////////////////////////
// Solving the problem with a tridiagonal
// solving algorithm
/////////////////////////////////////////////////

void num_solve(int n, double *a, double *b, double *c, double *v, double *f){
    int i;

    // Copied from lecture notes p. 186
    // First: forward substitution
    clock_t start , finish ; // declare start and final time start = clock () ;
    double temp[n+1];

    start = clock(); // Starting the timer

    double btemp = b[1];
    v[1] = f[1]/btemp;

    for(i=2; i <= n ; i++) {
        temp[i] = c[i-1]/btemp;
        btemp = b[i]-c[i]*temp[i];
        v[i] = (f[i] - a[i]*v[i-1])/btemp;
    }

    // Secondly: backsubstitution
    for(i=n-1 ; i >= 1 ; i--) {
        v[i] -= temp[i+1]*v[i+1];
    }
    finish = clock(); // Stopping the timer

    printf ("Elapsed time numerical: %5.9f seconds.\n", ( (float)( finish - start ) / CLOCKS_PER_SEC ));
}
```

The plots resulting from this method is shown in the end of this report together with the results from the method using Armadillo's `solve`-function.

Exercise c)

In this part of the project we did an analysis of the relative error between our own numerical solution and the exact solution. We calculated the error as described in the exercise text,

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

In my program it looked like this.

```
void error( double *err, double *u, double *v, int n){
    int i;
    double max;
    // Setting first element to 0.
    err[0] = 0;
    for (i=1; i<n; i++){
        // Calc. error using the given formula
        err[i] = log10( fabs( (v[i] - u[i]) / u[i] ) );
        if (fabs(err[i]) > fabs(err[i-1])){     // Picking out max error
            max = err[i];
        }
    }
    cout << "max␣error:␣" << max << endl;       // Printing max value
}
```

Below we see table 1 with our maximum relative errors. We see that it decreases when n is increasing, remember that the error is written as the logarithm of the relative error. In fig. 1, 2, 3, 4 and 5 we see the errors plotted as a function of x. When $n = 10$ the error is constant equal -1.1797, which is what I should get for every n if the algorithm is correct. This was discovered when Henrik and Anne-Marthe plotted the relative error between their solution from the LU-decomposition and the exact solution, since the LU-method is certainly correct the relative error between a numerical and exact solution should always, in this case, be a constant. As we can see from my own error this only happens for $n = 10$, otherwise it increases with x. The bigger n is the more the relative error increases, this may be due to some index-errors in my program which I haven't found.

n	ϵ_{max}
10	-1.1797
10^2	-3.0880
10^3	-5.0800
10^4	-7.0753
10^5	-8.3667

Table 1: Maximum error for different n when using LU-decomposition with forward and backward substitution. At $n = 10^6$ our program crashes. Also note that the error is decreasing with increasing n.

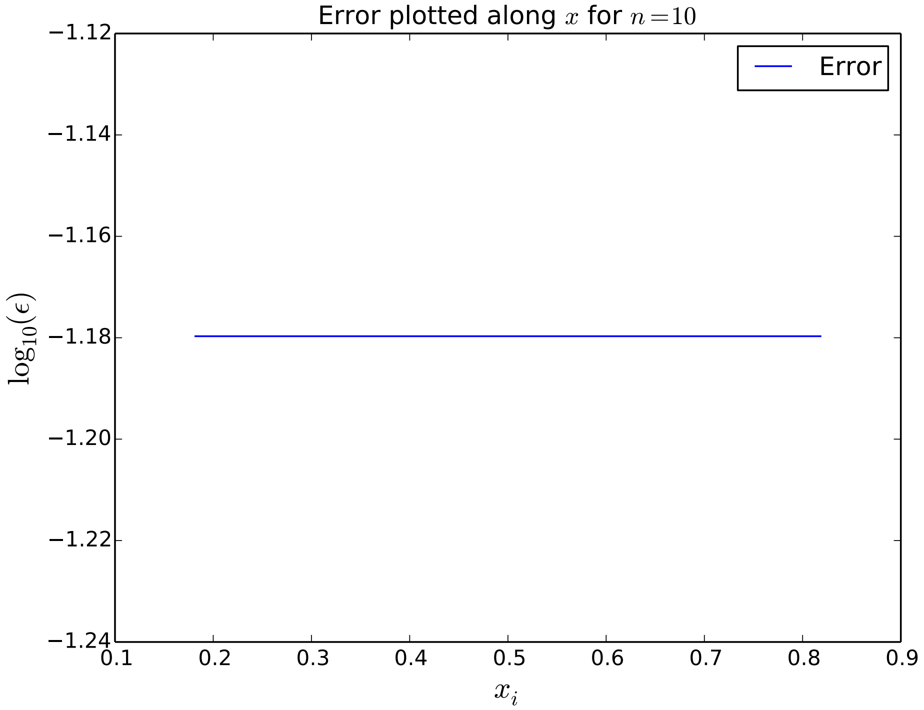

Figure 1: When $n = 10$ the error is constant, this may be a coincidence. The error is quite big anyway, which we also see clearly in fig. 6.

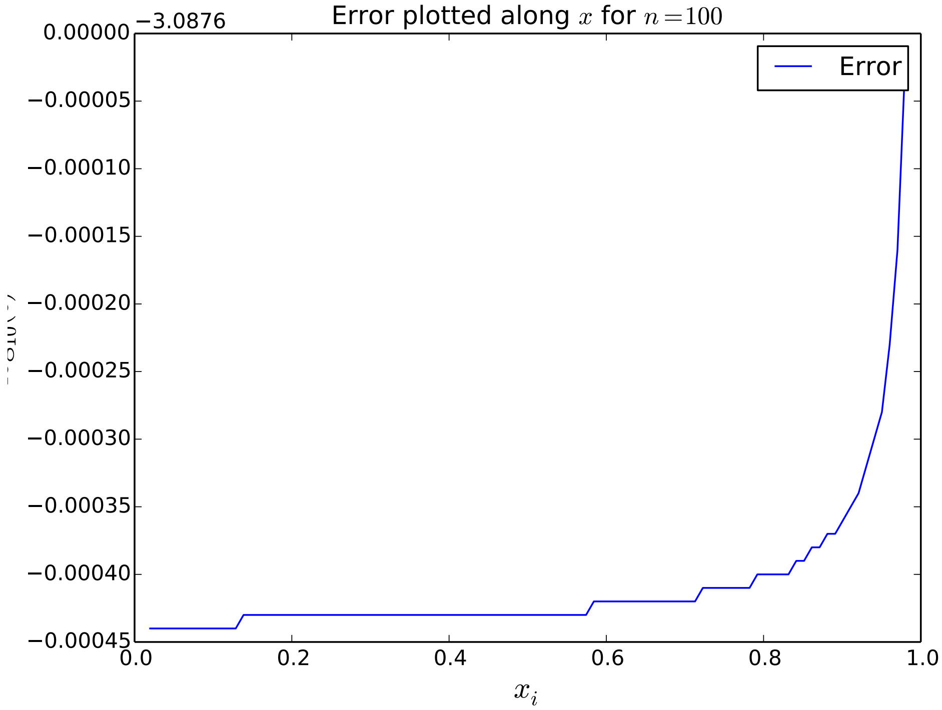

Figure 2: The relative error is for $n = 100$ around $\log_{10}(-3.08)$, which is an expected improvement from $n = 10$.

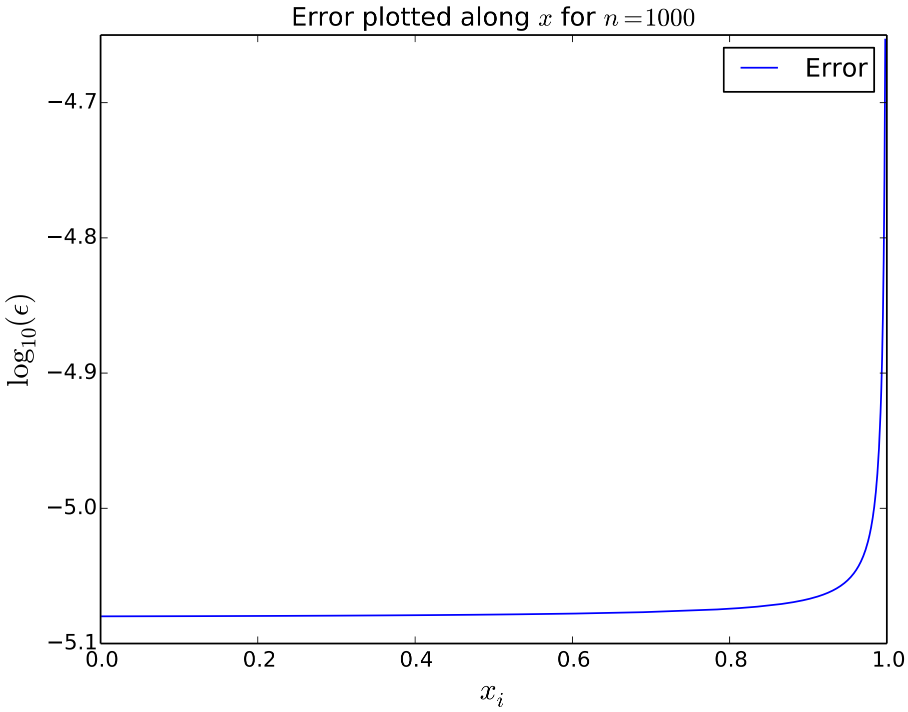

Figure 3: Error plot for $n = 10^3$

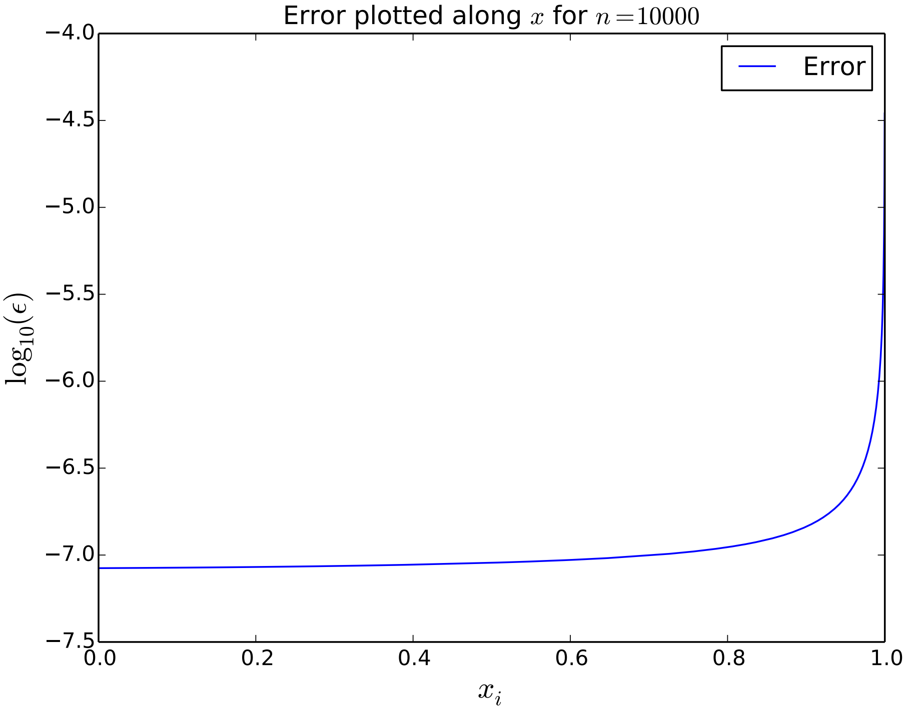

Figure 4: Error plot for $n = 10^4$

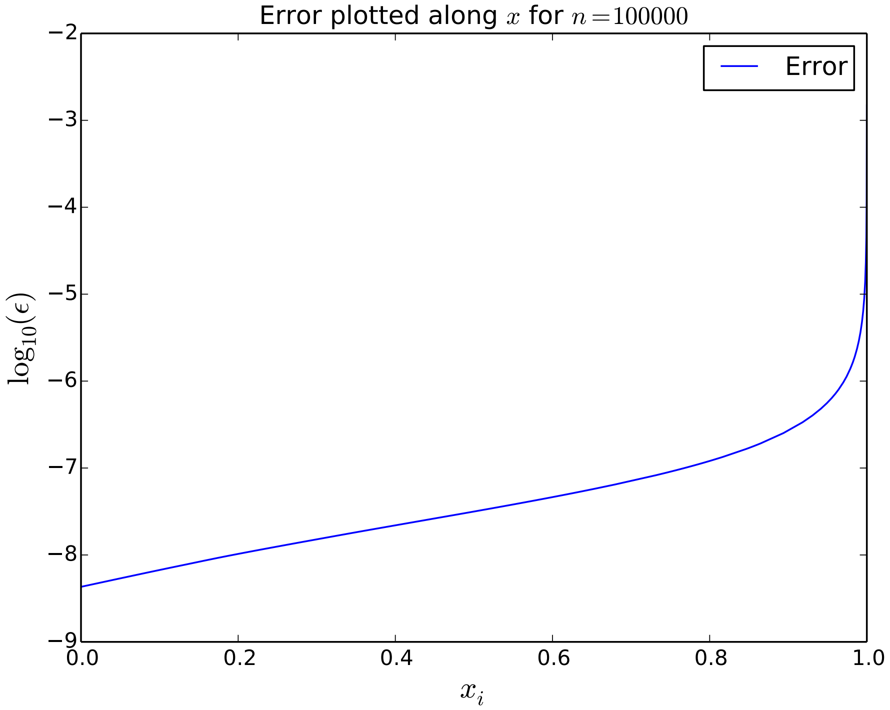

Figure 5: Error plot for $n = 10^5$. It seems like the numerical precision is decreasing when n is getting too large.

When we compare all our relative error-plots we see that the relative error is indeed decreasing for higher n, we also se that the relative error is always increasing a lot when $x \to 1$. This may be due to the accumulation of numerical error in the calculation.

Exercise d)

In this last part we're asked to compare our numerical results with the ones we get by using packages like Armadillo. First we can take a look at the function I made for solving the equation with Armadillo. The program is more or less self contained with comments that explain how it works. As we can see this program is much simpler than the one using, once we have initialized necessary objects we call on Armadillo's `solve`-function and we're (almost) done, we have to shift the `x2`-array in order to implement the boundary conditions correct.

```
#include <iostream>
#include <time.h>
#include "armadillo"
#include "arma_solve.h"

using namespace std;
using namespace arma;

/////////////////////////////////////////////////
// Solving the same problem with Armadillo
/////////////////////////////////////////////////

mat arma_solve(int n, double h, mat x_mat, mat x2){
    int i;

    clock_t start , finish ; // declare start and final time start = clock () ;

    // Declaring matrices
    mat A = zeros<mat>(n+1,n+1);
    mat f2 = zeros<mat>(n+1,1);

    // Initializing source term-vector
    for (i=1; i<=n; i++){
        f2(i) = h*h*100*exp(-10*x_mat(i));
    }

    // Filling matrix A
    A.diag() += 2.;
    A.diag(1) += -1.;
    A.diag(-1) += -1.;

    // Starting clock
    start = clock();

    // Calling on Armadillo's solve-function
    // to solve A * x2 = f2
    x2 = solve(A, f2);

    // Stopping clock
    finish = clock();

    printf ("Elapsed time Armadillo: %5.9f seconds.\n", ( (float( finish - start )) / CLOCKS_PER_SEC ));

    // Shifting the array one element in order to get the
    // boundary conditions correct
    x2.reshape(n+3,1);
    for (i=n; i>=0; i--){
        x2(i+1) = x2(i);
    }

    x2(0) = 0.;

    return x2;
}
```

My main program calls on the different solving methods and saves the results as `.txt`-files, see below. These files is read by a Python-script and plotted. The Python script is included at the end of the report and is self contained with comments.

```
// Project 1 main program
#include <iostream>
#include <cmath>
#include <typeinfo>
#include <time.h>
#include <stdio.h>
#include "armadillo"    // How do I compile with this included?
#include "arma_solve.h"
#include "num_solve.h"
#include "lu_decomposition.h"

using namespace std;
using namespace arma;

void exact( double *, double *, int );
void error( double *, double *, double *, int );
void save_results( double *, double *, double *, double*, int );
void save_results_arma(mat, mat, int);
void error_lu( double *, double *, mat , int);

int main()
{
    int n = 100; // Program runs w/ n=10,100,1000,10000.
    double x[n+1], h, a[n+1], b[n+1], c[n+1], v[n+2], f[n+1], u_exact[n+1], err[n+1];
    int i;

    // /////////////////////////////////////////////////////////////////////////
    // Initializing arrays etc.
    // /////////////////////////////////////////////////////////////////////////

    h = 1 / (float(n)+1);
    // Imposing boundary conditions
    v[0] = 0; v[n] = 0; x[0] = 0; x[n+1] = 1;

    u_exact[n] = 0; err[n+1] = 0;
    a[0] = c[n+1] = 0;

    for (i=1; i <= n; i++){
        x[i] = i*h;
        f[i] = h*h*100*exp(-10*x[i]);
    }
    for (i=1; i <= n; i++){
        c[i] = -1.;
        a[i] = -1.;
        b[i] = 2.;
    }

    //////////////////////////////////////
    // Initializing matrices
    //////////////////////////////////////

    mat x_mat = zeros<mat>(n+1,1);
    mat x_mat2 = zeros<mat>(n+1,1);
    mat x2 = zeros<mat>(n+1,1);
    mat x3 = zeros<mat>(n+1,1);

    //////////////////////////////////////
    // Solving with our tridiagonal solver

    num_solve(n, a, b, c, v, f);

    //////////////////////////////////////
    // Solving with Armadillo

    for (i=1; i<=n; i++){
        x_mat(i) = i*h;
    }
    x_mat.reshape(n+2,1);
    x_mat(n+1) = 1.;

    x2 = arma_solve(n, h, x_mat, x2);

    //////////////////////////////////////
    // Solving by LU-decomposition

    x3 = lu_decomposition(n, h, x_mat, x3);

    //////////////////////////////////////

    // Finding the exact result
    exact( x, u_exact, n);

    // Computing the error
    error(err, u_exact, v, n);

    // Writing results to file
    //save_results( x, v, u_exact, err, n);
    //save_results_arma(x_mat, x2, n);
    return 0;
}

void exact( double *x, double *u_exact, int n){
    int i;
    // f(x) = 100*exp(-10*x)
    for (i=1; i<=n; i++){
        u_exact[i] = 1 - ( 1 - exp(-10) )*x[i] - exp(-10*x[i]);
    }
}

void error( double *err, double *u, double *v, int n){
    int i;
    double max;
    // Setting first element to 0.
    err[0] = 0;
    for (i=1; i<n; i++){
        // Calc. error using the given formula
        err[i] = log10( fabs( (v[i] - u[i]) / u[i] ) );
        if (fabs(err[i]) > fabs(err[i-1])){    // Picking out max error
            max = err[i];
        }
    }
    cout << "max error: " << max << endl;        // Printing max value
}

void save_results( double *x, double *v, double *u, double *err, int n){
    FILE *output_file;
    output_file = fopen("oppgave_b_n_100000.txt" , "w") ;  // Is there a way to produce several output files with different names?
    fprintf(output_file, "   %s    %s    %s    %s \n", "x", "v_numerical", "u", "error");
    int i;
    for (i=0; i<=n+1; i++){
        fprintf(output_file, "%12.5f %12.5f %12.5f %12.5f \n",
                      x[i], v[i], u[i], err[i] );
    }
    fclose (output_file);
}

void save_results_arma(mat err2, mat x2, int n){
    int i;
    ofstream myfile;
    myfile.open ("arm_solve_10000.txt");
    myfile << "x_mat" << "     " << "x2" << endl;
    for (i=0; i<n+1; i++){
        myfile << err2(i) << "    " << x2(i) << endl;
    }
```

```
        myfile.close();
}
```

We'll now take a look at the results from our to methods. We have plotted three graphs in one plot: the exact solution, the solution from our tridiagonal solver and the solution from Armadillo's solver. We immediately see that Armadillo is useless when $n = 10$. We get the same shape on the solution, but it is far too high, see fig. 6. This happens again when $n = 100$, but it's getting closer, see fig. 7. We have to set $n = 1000$ before we can say that Armadillo is giving any precision in it's calculations, with the tridiagonal solver the exact and numerical solution are so close that it's hard to distinguish between the two graphs.

The big difference in the precision of these two methods may be due to the number of floating point operations. The more FLOPS we have, the more numerical error we get in the final result because the error will accumulate.

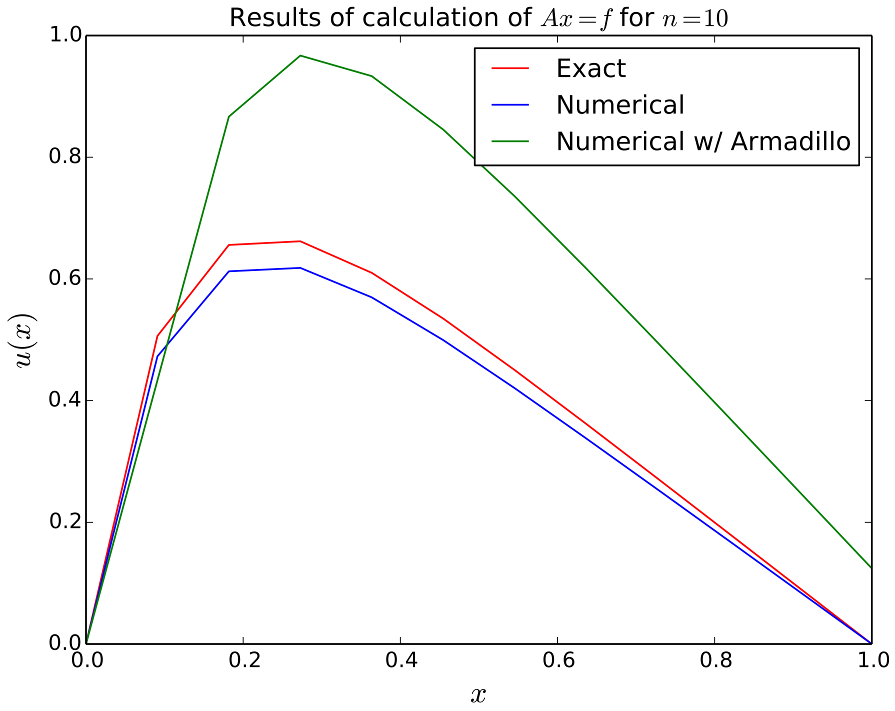

Figure 6: Numerical solution for $n = 10$. Armadillo's function 'solve' is pretty far off from the exact and the first numerical solution.

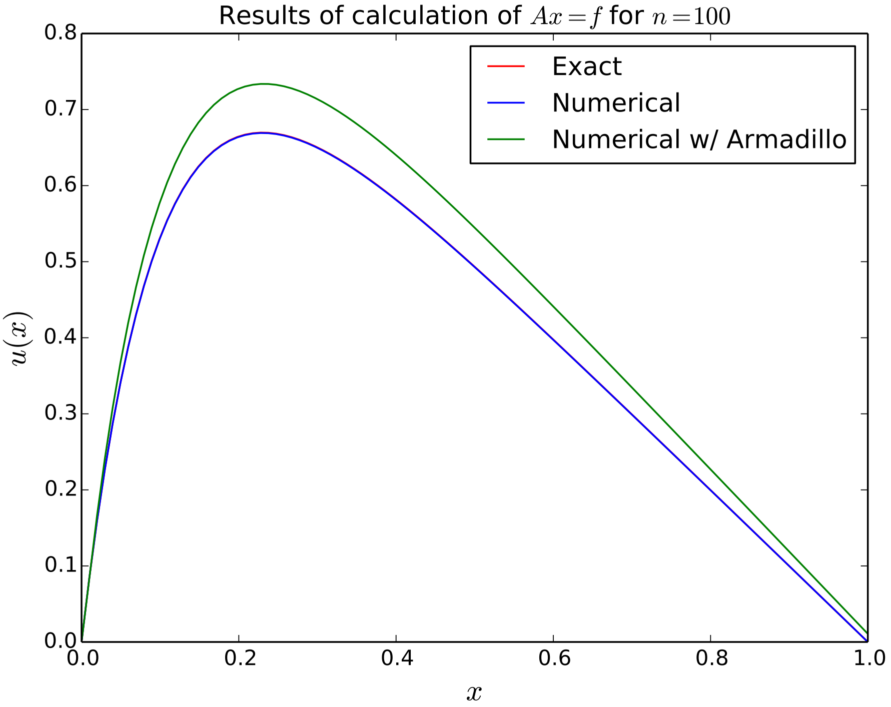

Figure 7: Numerical solution for $n = 10^2$.

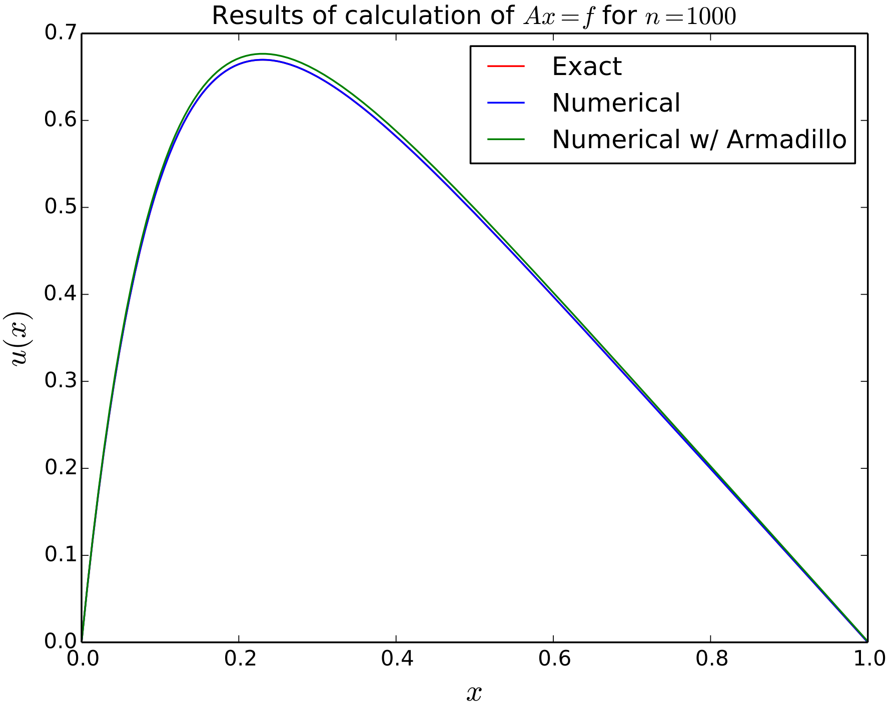

Figure 8: Numerical solution for $n = 10^3$.

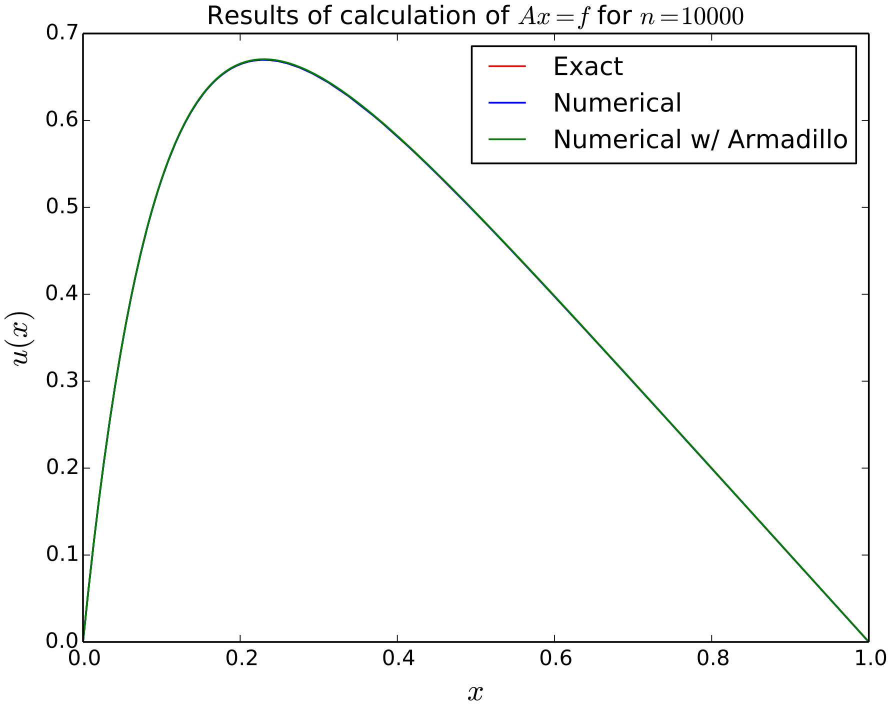

Figure 9: Numerical solution for $n = 10^4$.

Now we have seen that the precision of the two algorithms we've used are pretty different, now it's time to look at the time usage of the methods. Just for comparison I also added a function using LU-decomposition (see further down for code) for solving the equation. As mentioned earlier the LU-decomposition use $(2/3)n^3$ FLOPS while our own method use $8n$ FLOPS, it's pretty easy to guess which method is the fastest one.

In table 2 we see the results for different n. As expected the tridiagonal solver is the fastest one, and the LU-decomposition method is the slowest. When $n = 10^4$ something strange happens to the time function in the library `time.h`, it suddenly gives me a time far longer than it actually takes to run the program.

It could have been interesting to look how the time usage evolved with higher n, but the program crashes when I'm setting $n = 10^5$. This may be due to memory handling and how I compile the program.

n	$t_{\text{Numerical}}$	$t_{\text{Armadillo}}$	$t_{\text{LU decomp.}}$
10	0.000002 s	0.000075 s	0.000123 s
10^2	0.000004 s	0.000701 s	0.002931 s
10^3	0.000027 s	0.068193 s	0.459989 s
10^4	0.000312 s	40.02037 s	297.6299 s

Table 2: Time spent by the different methods used to solve our problem. Something weird happens with the time-function when we set $n = 10^4$ cause it didn't take five minutes to run the program on my own computer.

Below the program for calculating the differential equation with LU-decomposition is listed. The program is using the LU-decomposition-functions from Armadillo. Mathematically it is done like this,

$$
\begin{aligned}
&Ax = b \Rightarrow LUx = b\\
&Ux = y \Rightarrow Ly = b\\
&Ux = L^{-1}b = y \Rightarrow x = U^{-1}y = U^{-1}L^{-1}b.
\end{aligned}
$$

```cpp
#include <iostream>
#include <time.h>
#include "armadillo"
#include "lu_decomposition.h"

using namespace std;
using namespace arma;

///////////////////////////////////////////////
// Solving the same problem by LU-decomposition
///////////////////////////////////////////////

mat lu_decomposition(int n, double h, mat x_mat, mat x3){
    int i;
    mat w, y, L_inv, U_inv;

    clock_t start , finish ; // declare start and final time start = clock () ;

    // Initializing matrices for the calculations
    mat A = zeros<mat>(n+1,n+1);

    // Filling matrix A
    A.diag() += 2.;
    A.diag(1) += -1.;
    A.diag(-1) += -1.;

    mat f2 = zeros<mat>(n+1,1);

    for (i=1; i<=n; i++){
        f2(i) = h*h*100*exp(-10*x_mat(i));
    }

    w = f2;

    mat L,U,P;

    // Doing the LU-decomposition and finding x
    // as described in the lecture notes.

    start = clock();

    lu(L,U,P,A);

    y = solve(L, f2);
    x3 = solve(U, y);

    finish = clock();

    // Shifting the array one element
    x3.reshape(n+3,1);
    for (i=n; i>=0; i--){
        x3(i+1) = x3(i);
    }

    x3(0) = 0.;

    printf ("Elapsed time LU-decomposition: %5.9f seconds.\n"
        , ( (float)( finish - start ) / CLOCKS_PER_SEC ));

    cout << "LU_decomp. success" << endl;
    return x3;
}
```

Conclusion

The far best method, of the two we tested properly in this project, for solving the differential equation $-u''(x) = f(x)$ is the tridiagonal solver. This gave us the most precise result – if we should judge by looking at the plots – and was also the fastest one. My project have some weaknesses, the algorithm could have been improved in order to reduce the number of FLOPS and the whole program could have been written with vectors instead of doubles, then it would have been easier to find index errors for example.

Python code for plotting

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
from numpy import log

# Script for plotting results from main.cpp
# Remember to change n everywhere where necessary
# Declaring lists
x = []
v = []
u = []
err = []

# Fetching data from first calculation
with open('oppgave_b_n_100.txt') as oppgave_b:
        next(oppgave_b)
        for line in oppgave_b:
                x.append(float(line.split()[0]))
                v.append(float(line.split()[1]))        # numerical
                u.append(float(line.split()[2]))        # exact
                err.append((float(line.split()[3])))

# Fetching data from Armadillo calculation
x_arm = []
v_arm = []

with open('arm_solve_100.txt') as oppgave_b:
        next(oppgave_b)
        for line in oppgave_b:
                x_arm.append(float(line.split()[0]))
                v_arm.append(float(line.split()[1]))    # numerical

plt.figure(1)
log, = plt.plot(x[2:-2], err[2:-2])
plt.legend([log], ['Error'])
plt.title(r'Error plotted along $x$ for $n=100$')
plt.xlabel('$x_i$',size=17)
plt.ylabel('$\log_{10}(\epsilon)$',size=17)
plt.savefig("error_n_100.pdf")



plt.figure(2)
ex,  = plt.plot(x,u, 'r', label='exact')
num, = plt.plot(x,v, 'b', label='numerical')
num_arm, = plt.plot(x_arm, v_arm, 'g', label='armadillo')
plt.legend([ex, num, num_arm], ['Exact', 'Numerical', 'Numerical w/ Armadillo'])
plt.xlabel('$x$',size=17)
plt.ylabel('$u(x)$',size=17)
plt.title(r'Results␣of␣calculation␣of␣$Ax␣=␣f$␣for␣$n=100$')
plt.savefig("d_n_100.pdf")

plt.show()
```