FYS4150 - Computational physics
Project 1

Henrik Schou Røising

September 10, 2015

Abstract

In this project we study the numerical solution to a one dimensional Poisson equation with Dirichlet boundary conditions. The second derivative is approximated with a three point formula and the problem is solved as a set of linear equations for the discretized variable. The equations are handled in two ways which in the end are compared. First by a a simplified form of Gaussian elimination and then by use of LU decomposition. The numerical algorithms are compared in terms of FLOPS and computation time.

- Github repository link with all source files: https://github.com/henrisro/Project1

1 Introduction

We start out with the one dimensional Poisson equation for a special source function. This equation is to be solved numerically by a discretization of the interval $x \in [0,1]$ where the solution is defined. In the theory section we show that finding the discretized solution is equivalent to solving a set of linear equations, i.e. solving for $\boldsymbol{v}$ in a matrix equation on the form $A\boldsymbol{v} = \tilde{\boldsymbol{b}}$. It will turn out that the matrix A is tridiagonal, which in turn makes Gaussian elimination particularly simple for this system. First this method is implemented in a C++ program for the three different number of grid points $n = 10$, $n = 100$ and $n = 1000$. The numerical results are compared to the analytical solution of the equation (which is found for the given source function). Secondly we compute the maximal relative error (maximal over $x \in (0,1)$) of the numerical solution when varying the number of grid points and study the result as a log-log table (and plot) of error vs. grid step size. Finally we implement another method for solving the linear set of equations, namely by use of LU decomposition of the matrix A. For this we use pre written library functions provided on the web page of the course (given in files called `lib.h` and `lib.cpp`). The number of FLOPS and the overall calculation time is compared for the two different methods and presented in a table.

2 Theory

The three dimensional Poisson equation

$$\nabla^2\Phi = -4\pi\rho(\boldsymbol{r}) \tag{1}$$

can be reduced, by assumption of spherical symmetry and the substitution $\Phi(r) = \phi(r)/r$, to

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

If one further let $\phi \to u$, $r \to x$ and define $f(x) \equiv 4\pi x\rho(x)$, the equation becomes simply

$$\frac{d^2u}{dx^2} \equiv u''(x) = -f(x) \tag{3}$$

This equation will in the following be studied on the open interval $x \in (0, 1)$ with Dirichlet boundary conditions, $u(0) = u(1) = 0$ and source function $f(x) = 100e^{-10x}$. For this source function, a quick integration reveals that the analytical solution of the problem is given by the function $u(x) = 1-(1-e^{-10})x-e^{-10x}$. We discretize and label the numerical solution (at grid point x_i) v_i where $x_i = ih$ and $h = \frac{1}{n+1}$ is the step size. With this convention $v_0 = v_{n+1} = 0$ and the interior solution $v_i\ \forall i \in \{1, \dots, n\}$ is to be found. We approximate the second derivative in equation (3) with a three point formula (with error of the order $\mathcal{O}(h^2)$):

$$-\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = f_i \tag{4}$$

Now define $\tilde{b}_i \equiv h^2 f_i$ such that equation (4) can be written as $-v_{i+1} + 2v_i - v_{i-1} = \tilde{b}_i$. If we omit the uninteresting end points, $i = 0$ and $i = n + 1$, this equation corresponds to the i'th component of the matrix equation

$$\begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\ & \vdots & & & \ddots & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \vdots \\ \tilde{b}_{n-1} \\ \tilde{b}_n \end{bmatrix} \tag{5}$$

From now on label this tridiagonal matrix A, the vector $(v_1, \dots, v_n)^T \equiv \boldsymbol{v}$ and $(\tilde{b}_1, \dots, \tilde{b}_n)^T \equiv \tilde{\boldsymbol{b}}$.

3 Method / Algorithm

Two methods are implemented and compared. The first one is derived from the results in the theory section and the second one is mostly carried out by the use of pre written library functions.

3.1 Method 1: Gaussian elimination of tridiagonal matrix

In order to find an algorithm for solving the matrix equation (5), let us be slightly more general and write the i'th component as $a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i$ with of course $a_i = -1$ (and $a_1 = 0$), $b_i = 2$ and $c_i = -1$ (and $c_n = 0$) in our case. From here on we will actually keep it the equations this general such that the code can more easily be applied to other systems at a later stage. We now start a row reduction of the system (Gaussian elimination) and deduce an algorithm with $n = 4$ (simply to save some space):

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & \tilde{b}_1 \\ a_2 & b_2 & c_2 & 0 & \tilde{b}_2 \\ 0 & a_3 & b_3 & c_3 & \tilde{b}_3 \\ 0 & 0 & a_4 & b_4 & \tilde{b}_4 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & c_1/b_1 & 0 & 0 & \tilde{b}_1/b_1 \\ 0 & b_2 - \frac{c_1}{b_1}a_2 & c_2 & 0 & \tilde{b}_2 - \frac{\tilde{b}_1}{b_1}a_2 \\ 0 & a_3 & b_3 & c_3 & \tilde{b}_3 \\ 0 & 0 & a_4 & b_4 & \tilde{b}_4 \end{array}\right] \tag{6}$$

Now let $\tilde{d}_1 = b_1$ and $\tilde{d}_2 = b_2 - \frac{c_1}{b_1}a_2 = b_2 - \frac{c_1}{\tilde{d}_1}a_2$. Relabel also the elements appearing on the right of the vertical bar $\tilde{v}_i$ as they are updated, so that for example $\tilde{v}_1 = \tilde{b}_1/\tilde{d}_1$. Let us normalize the second row and then do one more iteration to see the pattern:

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & (\tilde{b}_2 - \tilde{v}_1 a_2)/\tilde{d}_2 \\ 0 & a_3 & b_3 & c_3 & \tilde{b}_3 \\ 0 & 0 & a_4 & b_4 & \tilde{b}_4 \end{array}\right] = \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & a_3 & b_3 & c_3 & \tilde{b}_3 \\ 0 & 0 & a_4 & b_4 & \tilde{b}_4 \end{array}\right] \sim \tag{7}$$

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & 0 & b_3 - \frac{c_2}{\tilde{d}_2}a_3 & c_3 & \tilde{b}_3 - \tilde{v}_2 a_3 \\ 0 & 0 & a_4 & b_4 & \tilde{b}_4 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & 0 & 1 & c_3/\tilde{d}_3 & (\tilde{b}_3 - \tilde{v}_2 a_3)/\tilde{d}_3 \\ 0 & 0 & a_4 & b_4 & \tilde{b}_4 \end{array}\right] \tag{8}$$

$$\sim \cdots \sim \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & 0 & 1 & c_3/\tilde{d}_3 & \tilde{v}_3 \\ 0 & 0 & 0 & 1 & \tilde{v}_4 \end{array}\right] \tag{9}$$

The pattern seen here can be summarized in the following recursion relations:

$$\tilde{d}_i = b_i - \frac{c_{i-1}}{\tilde{d}_{i-1}}a_i \qquad \text{and} \qquad \tilde{v}_i = (\tilde{b}_i - \tilde{v}_{i-1}a_i)/\tilde{d}_i \tag{10}$$

for $i \in \{2, 3, \ldots, n\}$ with $\tilde{d}_1 = b_1$ and $\tilde{v}_1 = \tilde{b}_1/\tilde{d}_1$ (note that b_i is not to be confused with $\tilde{b}_i = h^2 f_i$. This is a somewhat unfortunate notation). In C++ these recursion relations could be implemented in the following way (where we also store the values $c_{i-1}/\tilde{d}_{i-1}$ for use in the next algorithm):

```cpp
double b_temp = b[1]; // Temporary variable corresponding to \tilde{d
  }[1]
v[1] = b_twidd[1]/b_temp; // First updated element in v-array
for (int i=2; i<=n; i++) {
  // Updating temporary diagonal element:
  diag_temp[i] = c[i-1]/b_temp;
  // Updating diagonal element:
  b_temp = b[i] - a[i]*diag_temp[i];
  // Updating solution:
  v[i] = (b_twidd[i]-v[i-1]*a[i])/b_temp;
}
```

Listing 1: Algorithm 1: forward substitution in Gaussian elimination.

In order to complete the Gaussian elimination we also have to iterate backwards to get pivot elements in all columns of the row reduced matrix in equation (9). But that is simple:

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & 0 & 1 & c_3/\tilde{d}_3 & \tilde{v}_3 \\ 0 & 0 & 0 & 1 & \tilde{v}_4 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d}_1 & 0 & 0 & \tilde{v}_1 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 & \tilde{v}_2 \\ 0 & 0 & 1 & 0 & \tilde{v}_3 - \frac{c_3}{\tilde{d}_3}\tilde{v}_4 \\ 0 & 0 & 0 & 1 & \tilde{v}_4 \end{array}\right] \tag{11}$$

so let us update $\tilde{\tilde{v}}_3 = \tilde{v}_3 - \frac{c_3}{\tilde{d}_3}\tilde{v}_4$. It is quickly seen that we can just continue this iteration all the way to the first element until the left block matrix has been reduced to the identity:

$$\tilde{\tilde{v}}_i = \tilde{v}_i - \frac{c_i}{\tilde{d}_i}\tilde{v}_{i+1} \tag{12}$$

for $i \in \{n-1, n-2, \dots, 1\}$. We also note that the factor arising in this update, $c_i/\tilde{d}_i$, is stored in the variable `diag_temp[i+1]` in the above code piece. Translated to C++ we can implement this new update by:

```
for (int i=n-1; i>=1; i--) {
    v[i] -= diag_temp[i+1]*v[i+1];
}
```

Listing 2: Algorithm 2: backward substitution in Gaussian elimination.

The two algorithms shown in Listing 1 and Listing 2 together solves the problem, returning the discretized solution v_i. By a quick inspection of the number of FLOPS in Algorithm 1 (we count the number of operations `+`,`-`,`*` and `/`) we see that there are 6 FLOPS in each loop iteration and thus $6(n-1)$ required FLOPS for the whole loop. Adding the $2(n-1)$ FLOPS required for Algorithm 2 we end up with a total of

$$N_{\text{tridiagonal}} = 8(n-1) = \mathcal{O}(8n) \tag{13}$$

This linear relation in n can also be approximately assumed (if each FLOP is assumed to take the same amount of time) to be the case for the calculation time of this method. Now it is worth mentioning that the implementation can be optimized for our specific case. Since $a_i = c_i = -1$, the algorithms presented in equation (10) and (12) can be simplified to

$$\tilde{d}_i = b_i + 1/\tilde{d}_{i-1} \qquad \text{and} \qquad \tilde{v}_i = (\tilde{b}_i + \tilde{v}_{i-1})/\tilde{d}_i \tag{14}$$

for $i \in \{2, 3, \dots, n\}$. Similarly for the backward substitution:

$$\tilde{\tilde{v}}_i = \tilde{v}_i + \tilde{v}_{i+1}/\tilde{d}_i \tag{15}$$

for $i \in \{n-1, n-2, \ldots, 1\}$, giving a total cost of $6(n-1)$ FLOPS. This optimization will not be implemented in what follows in an attempt to keep the code structure more general. The number $N_{\text{tridiagonal}}$ should anyhow be compared to the FLOPS of standard Gaussian elimination, which is $N_{\text{Gaussian}} = \frac{2}{3}n^3 + \mathcal{O}(n^2)$ [1], such that the algorithm discussed here is a major speed up in comparison. In fact the major simplification lies in the forward substitution where the standard algorithm gives raise to the cubic term.

3.2 Method 2: LU decomposition of matrix A

Let us return to the matrix equation (5) written as $A\boldsymbol{v} = \tilde{\boldsymbol{b}}$. Assuming A is invertible (which definitely is the case here) one can decompose A into a product of a lower diagonal and an upper diagonal matrix: $A = LU$:

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ & \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \tag{16}$$

With the matrix equation now written as $L(U\boldsymbol{v}) = \tilde{\boldsymbol{b}}$, we see that the task of determining $\boldsymbol{v}$ is reduced to two sets of linear equations:

$$L\boldsymbol{y} = \tilde{\boldsymbol{b}} \qquad \text{and} \qquad U\boldsymbol{v} = \boldsymbol{y}. \tag{17}$$

The first of these matrix equations is used to find $\boldsymbol{y}$ and then the second is solved for $\boldsymbol{v}$. This is quickly done because of the constructions of L and U. To see this explicitly we can write out the components of the equations in (17):

$$y_1 = \tilde{b}_1 \tag{18}$$
$$l_{21}y_1 + y_2 = \tilde{b}_2 \tag{19}$$
$$\vdots \tag{20}$$
$$l_{n1}y_1 + l_{n2}y_2 + \cdots + y_n = \tilde{b}_n \tag{21}$$

$$u_{11}v_1 + u_{12}v_2 + \cdots + u_{1n}v_n = y_1 \tag{22}$$
$$\vdots \tag{23}$$
$$u_{(n-1)(n-1)}v_{n-1} + u_{(n-1)n}v_n = y_{n-1} \tag{24}$$
$$u_{nn}v_n = y_n \tag{25}$$

y_i is determined iteratively through (18) etc. and then v_i iteratively through (25) etc. This decomposition and solution to the set of equations is found numerically by calling two functions in the pre written library (from the course web page), `lib.h` and `lib.cpp`. The functions are called `ludcmp` (finds the LU decomposition) and `lubksb` (finds the vector $\boldsymbol{v}$), and their documentation is found in `lib.cpp`. A code piece which shows an implementation of them and the corresponding time used for calculation is shown in Listing 3.

```cpp
int *indx;
double d;
indx = new int[n];

// Time of calculation starts here:
clock_t start, finish;
start = clock();

ludcmp(A, n, indx, &d); // LU decompose A[][]
lubksb(A, n, indx, b_twidd); // Solve linear equations

// The solution v is now stored in b_twidd[]!
finish = clock();
calculation_time = (finish-start)/(double)CLOCKS_PER_SEC;
```

Listing 3: LU decomposition: use of pre written library functions.

Finally we state that the number of FLOPS required by this algorithm is (n^2 iterations for both forward and backward substitution and $2/3n^3$ for the LU decomposition):

$$N_{\text{LU}} = \frac{2}{3}n^3 + 2n^2 = \mathcal{O}(2/3n^3) \tag{26}$$

We see that this method is in fact a *quadratic slowdown* in n compared to method 1.

4 Results

Method 1 presented in the last section was implemented (code is attached at the end of this document), the results written to a text file and a small python script was written to plot the resulting solution v_i together with the analytical solution u_i as function of the grid point x_i. The results can be seen in figure 1, 2 and 3 for the number of grid points $n = 10$, $n = 100$ and $n = 1000$ respectively. The relative error of the numerical solution is defined as

$$\epsilon_i \equiv \log_{10}\left(\frac{|v_i - u_i|}{|u_i|}\right) \tag{27}$$

The maximum of this function,

$$\epsilon_{\max} = \max\{\epsilon_i \mid i \in \{1, 2, \ldots, n\}\}, \tag{28}$$

is taken as a measure of the quality of the numerical solution. Since an increasing n correspond to a decreasing step length, one of course expect $\epsilon_{\max}$ to be reduced as n increases (until round of errors take over). In particular by plotting $\log_{10}(\epsilon_{\max})$ against $\log_{10}(h)$, we expect a slope of 2 (not for very small h) as the truncation error we do by the approximation in equation (4) goes as $\mathcal{O}(h^2)$. In table 1 the maximum of the relative error, in particular $\log_{10}(\epsilon_{\max})$, is shown as function of n in powers of 10. A corresponding plot of $\log_{10}(\epsilon_{\max})$ against $\log_{10}(h)$ can be seen in figure 4 (there with n increasing in powers of 2 to make visualization better).

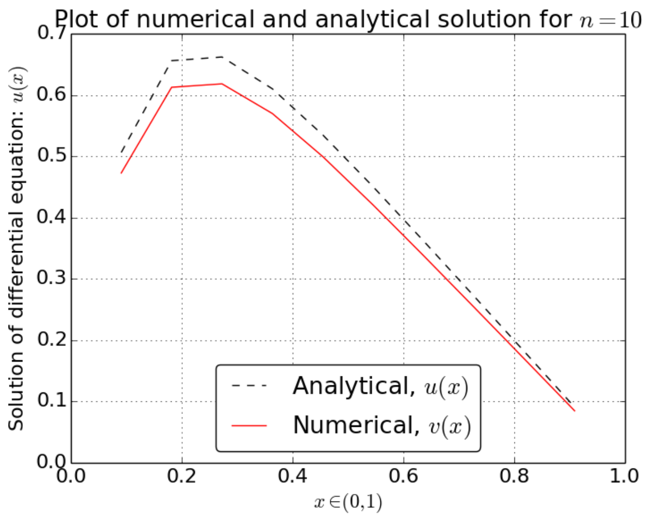

Figure 1: Analytical and numerical solution for $n = 10$ on the interior of the interval, $x \in (0, 1)$.

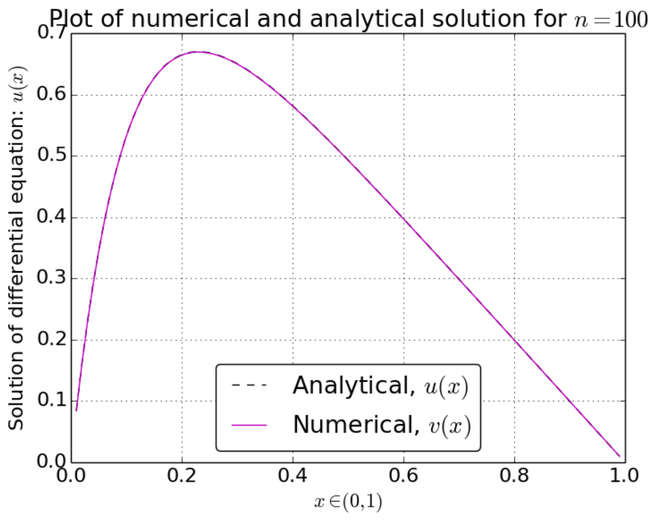

Figure 2: Analytical and numerical solution for $n = 100$ on the interior of the interval, $x \in (0, 1)$.

For comparing the time usage of method 1 (tridiagonal Gaussian elimination, t_{tridiag}) and method 2 (LU decomposition, t_{LU}), a C++ program was written to print out the calculation time of the basic algorithms shown in Listing 1 and Listing 2 vs. the algorithm in Listing 3 (where the implementation of the calculation time is shown explicitly). If each FLOP is assumed to take the same amount of time, one expects to see (at least for large n) that t_{tridiag} has a linear dependence in n while t_{LU} has a cubic dependence. The results of a program run is shown in Table 2. We chose to stop the use of method 2 when n passes about 3000 since t_{LU} blows up to the order of minutes around that value. In this calculation we increased n with a factor of 4 in each iteration to get a reasonable amount of values (the full program is attached in the end of this document).

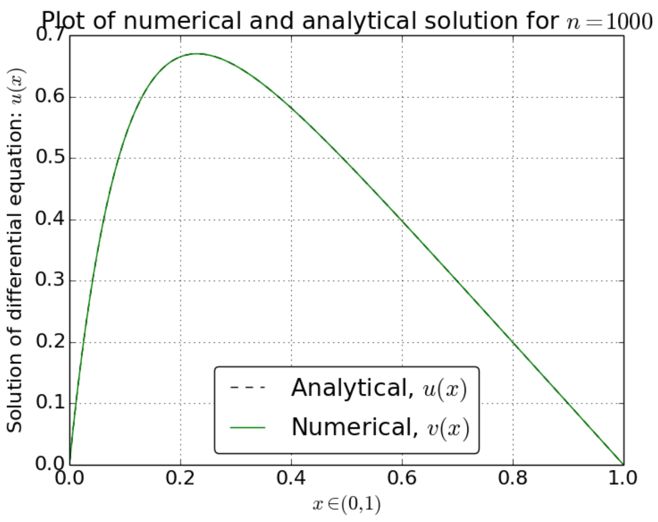

Figure 3: Analytical and numerical solution for $n = 1000$ on the interior of the interval, $x \in (0, 1)$.

Number of grid points, n	Log of step size, $\log_{10}(h)$	Log of maximal relative error, $\log_{10}(\max(\{\epsilon_i\}))$
10^1	-1.0413927	-1.3589135
10^2	-2.0043214	-3.2621439
10^3	-3.0004341	-5.2541384
10^4	-4.0000434	-7.2533984
10^5	-5.0000043	-9.0436161
10^6	-6.0000004	-6.3508347

Table 1: Table of maximal relative error as function of number of grid points, n.

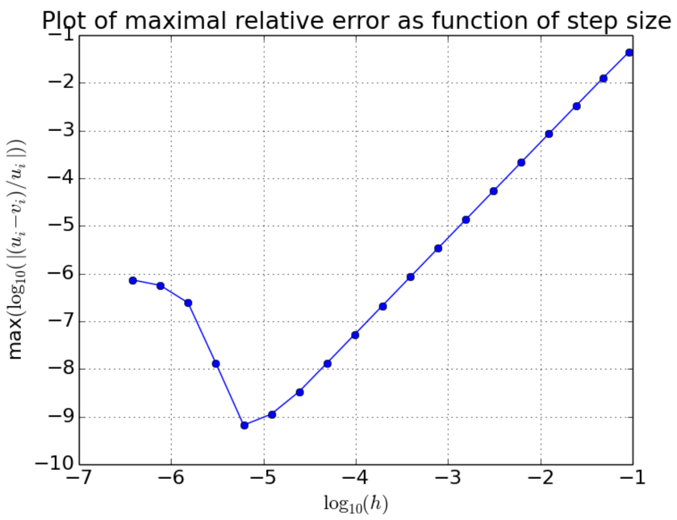

Figure 4: Logarithmic plot of relative error as function of step size.

Number of grid pints, n	Calculation time for tridiagonal method, t_{tridiag} [s]	Calculation time for LU method, t_{LU} [s]
10	0.0	$1.9 \cdot 10^{-5}$
40	$3 \cdot 10^{-6}$	0.000207
160	$7 \cdot 10^{-6}$	0.010694
640	$1.7 \cdot 10^{-5}$	0.534527
2560	$7.6 \cdot 10^{-5}$	77.847840
10240	0.000430	- -
40960	0.001594	- -
163840	0.005500	- -
655360	0.021694	- -
2621440	0.085793	- -

Table 2: Table of calculation time comparison for the two different methods as function of number of grid points, n. The number of points is increased by a factor of 4 in each row. The calculation time computed in this way had precision in μs, which indicates that the calculation with the tridiagonal method for $n = 10$ took less than 1 μs.

5 Discussion

The implementation of method 1 is seen to work in accordance to what is expected (figure 1, 2 and 3) as the relative error decreases - truncation error of order $\mathcal{O}(h^2)$ - with step size. In figure 4 one can read of a slope very close to 2 as n is increased from 10, but when the step size is smaller than about $h \approx 10^{-5}$, loss of numerical precision takes over and the relative error increases again. This is due to the fact that numerical derivation is based on subtracting numbers which become close to equal as h is getting small, and one eventually subtract numbers with an amount of common decimals close to the number of decimals used in the representation of the numbers themselves. Thus increasing n further than about 10^5 gives no further precision in the numerical solution, v_i.

One thing one might be worried about is the fact that the numerical solution is strictly smaller than the analytical one as seen for $n = 10$ (figure 4). A possible explanation of this lies on the other hand in the explicit form of the truncation error of the second derivative. By Taylor expansion the exact second derivative of u is given by:

$$u_0^{''} = \underbrace{\frac{u_h + u_{-h} - 2u_0}{h^2}}_{\equiv v_0^{''}} - 2\sum_{j=1}^{\infty} \frac{u_0^{(2j+2)}}{(2j+2)} h^{2j} = v_0^{''} - \frac{u_0^{(4)}}{12} h^2 + \mathcal{O}(h^4) \tag{29}$$

so the lowest order order truncation error depends on the fourth derivative of u at x_0. By use of the closed form of the analytical solution, $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$, one finds that this reduces to:

$$u_0^{''} = v_0^{''} + \frac{10^4}{12} e^{-10x} h^2 + \mathcal{O}(h^4) \tag{30}$$

and since the lowest order neglected term here is strictly positive on the interval $x \in (0,1)$ (it is still positive when integrated two times), this shows that u is strictly larger than v on this interval.

When the method of tridiagonal Gaussian elimination is compared in terms of time usage to the LU decomposition method, we see the superior advantage of the first one: a quadratic speed up when n gets large. In fact, if one were to plot the time usage as function of n using the data in Listing 3, the resulting graphs are in good agreement with what was mentioned in the last section: a linear dependency on n for method 1 and a cubic dependency for method 2. One of the problems one encounter with method 2 as n is increased is the storage of all the $n \times n$ values of matrix A in the computer memory. Assume for instance that $n = 10^5$. If one were to store all the 10^{10} components of A as double precision floating points, it would take some 80 GB of memory, which is way out of reach for a simple laptop. Further: by a quick cubic extrapolation of the data in Listing 3 the computation time needed for such a calculation accumulates to over 60 days.

6 Conclusion

In this project we have seen that solving a one dimensional Poisson equation numerically will depend heavily, in terms of number of FLOPS and computation time, on the method of implementation. First we used a simplified form of Gaussian elimination which could be used for tridiagonal matrices. This allowed us to skip the storage of the matrix itself and gave a total number of flops $N_{\text{tridiag}} = \mathcal{O}(8n)$. The precision of the numerical solution was seen to behave in accordance to the expected truncation error of the method. Then we applied a LU decomposing method, which is standard routine for solving linear equations and which works for any invertible matrix. The number of FLOPS for this method goes as $N_{\text{LU}} = \mathcal{O}(2/3n^3)$. An implementation of this method showed that the method was practically useless for systems with $n \gtrsim 10^3$ as the time consumption becomes large. The numerical solution reached its maximal precision around $n = 10^5$ where the tridiagonal method required a computation time in orders of ms.

7 References

[1] M. Hjorth-Jensen. Computational Physics, Lecture Notes Fall 2015. Department of Physics, University of Oslo, 2015.

8 Code attachment

The code is attached in chronological order beneath and the caption instructs how the programs were ran to give the results in this report.

```
/*
**      Project1: a) and b)
**      The algorithm for solving the tridiagonal matrix
**      equation is implemented (requiering O(8n) FLOPS).
**      Results are written to textfile and read by a python
**      script (project1_b_plot.py) to make plots.
*/
# include <iostream>
# include <fstream>
# include <iomanip>
# include <cmath>

using namespace std;
ofstream ofile;

// Declaring two functions that will be used:
double Solution(double x) {return 1.0-(1-exp(-10))*x-exp(-10*x);}

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

// Main program reads filename and n from command line:
int main(int argc, char* argv[]) {

    // Declaration of initial variables:
    char *outfilename;
    int n;

    // Read in output file and n,
    // abort if there are too few command-line arguments:
    if( argc <= 2 ){
      cout << "Bad Usage: " << argv[0] <<
          " read also output file and n (int) on same line" << endl;
      exit(1);
    }
    else{
      outfilename = argv[1]; // first command line argument.
      n = atoi(argv[2]);  // second command line argument.
    }

    // Constants of the problem:
    double h = 1.0/(n+1.0);
    double *x = new double[n+2];
    double *b_twidd = new double[n+1]; // construction with n+1 points to make
                                       // indexing close to mathematics.
    b_twidd[0] = 0;

    // The constituents of the tridiagonal matrix A:
    // Zeroth element not needed, but included to make indexing easy:
    int *a = new int[n+1];
    int *b = new int[n+1];
    int *c = new int[n+1];

    // Temporal variabel in Gaussian elimination:
    double *diag_temp = new double[n+1];

    // Real solution and approximated one:
    double *u = new double[n+2]; // Analytical solution
    double *v = new double[n+2]; // Numerical solution
    // Including extra points to make the indexing easy:
    u[0] = 0;
    v[0] = 0;

    // Filling up x-array, making x[0] = 0 and x[n+1] = 1:
    for (int i=0; i<=n+1; i++) {
        x[i] = i*h;
        // Could print results to check:
        //cout << "x = " << x[i] << " and " << "h^2*f(x) = " << h*h*f(x[i]) << endl;
    }

    // Filling up b_twiddle array, i.e. right hand side of equation:
    for (int i=1; i<=n; i++) {
        b_twidd[i] = h*h*f(x[i]);
        // Could print here to check:
```

```
        //cout << "b_twidd = " << b_twidd[i] << "for x = " << x[i] << endl;
        u[i] = Solution(x[i]);
        //cout << "u = " << u[i] << " for x = " << x[i] <<  endl;
        b[i] = 2;
        a[i] = -1;
        c[i] = -1;
    }
    c[n] = 0;
    a[1] = 0;

    // Algorithm for finding v:
    // a(i)*v(i-1) + b(i)*v(i) + c(i)*v(i+1) = b_twidd(i)
    // Row reduction; forward substitution:
    double b_temp = b[1];
    v[1] = b_twidd[1]/b_temp;
    for (int i=2;i<=n;i++) {
       // Temporary value needed also in next loop:
       diag_temp[i] = c[i-1]/b_temp;
       // Temporary diagonal element:
       b_temp = b[i] - a[i]*diag_temp[i];
       // Updating right hand side of matrix equation:
       v[i] = (b_twidd[i]-v[i-1]*a[i])/b_temp;
    }

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

    // Open file and write results to file:
    ofile.open(outfilename);
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << "       x:             u(x):          v(x):  " << endl;
    for (int i=1;i<=n;i++) {
       ofile << setw(15) << setprecision(8) << x[i];
       ofile << setw(15) << setprecision(8) << u[i];
       ofile << setw(15) << setprecision(8) << v[i] << endl;
    }
    ofile.close();

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

    return 0;
}
```

Listing 4: Program `project1_b.cpp`. This program was compiled and ran for $n = 10$, $n = 100$ and $n = 1000$ producing the .txt files `data_n10.txt`, `data_n100.txt` and `data_n1000.txt` respectively.

```
#
#     Project1: b)
#     The algorithm for solving the tridiagonal matrix
#     equation is implemented in project1_b.cpp.
#     Results are read from textfile and plotted here.
#
from math import *
import numpy as np
import matplotlib.pyplot as plt

def read_x_u_v(filename):
    infile = open(filename, 'r')
    # Elements to be read in file:
```

```
    x = []; u = []; v = [];
    # Read lines except for the first one:
    lines = infile.readlines()[1:]
    for line in lines:
      words = line.split()
        x.append(float(words[0]))
        u.append(float(words[1]))
        v.append(float(words[2]))
    infile.close()
    return x, u, v

# Fetching data by a call on read_x_u_v for three different n:
x1, u1, v1 = read_x_u_v('data_n10.txt')
x2, u2, v2 = read_x_u_v('data_n100.txt')
x3, u3, v3 = read_x_u_v('data_n1000.txt')


# Plotting commands:
plt.rcParams.update({'font.size': 16})
fig, ax = plt.subplots(1)
ax.plot(x1,u1,'k--',label='Analytical, $u(x)$')
ax.plot(x1,v1,'r-',label='Numerical, $v(x)$')
ax.set_xlabel('$x\in (0,1)$')
ax.set_ylabel('Solution of differential equation: $u(x)$')
ax.legend(loc='lower center',fancybox='True')
ax.set_title('Plot of numerical and analytical solution for $n = 10$')
ax.grid()
plt.show()

fig, ax = plt.subplots(1)
ax.plot(x2,u2,'k--',label='Analytical, $u(x)$')
ax.plot(x2,v2,'m-',label='Numerical, $v(x)$')
ax.set_xlabel('$x\in (0,1)$')
ax.set_ylabel('Solution of differential equation: $u(x)$')
ax.legend(loc='lower center',fancybox='True')
ax.set_title('Plot of numerical and analytical solution for $n = 100$')
ax.grid()
plt.show()

fig, ax = plt.subplots(1)
ax.plot(x3,u3,'k--',label='Analytical, $u(x)$')
ax.plot(x3,v3,'g-',label='Numerical, $v(x)$')
ax.set_xlabel('$x\in (0,1)$')
ax.set_ylabel('Solution of differential equation: $u(x)$')
ax.legend(loc='lower center',fancybox='True')
ax.set_title('Plot of numerical and analytical solution for $n = 1000$')
ax.grid()
plt.show()
```

Listing 5: Program `project1_b_plot.py`. This program was ran after producing the .txt files `data_n10.txt`, `data_n100.txt` and `data_n1000.txt` to make the plots shown in figure 1, 2 and 3.

```
/*
**      Project1: c)
**      The algorithm for solving the tridiagonal matrix
**      equation is implemented (requiering O(8n) FLOPS).
**      Log of the absolute error is written to file for
**      several different values of n. initial n, number of steps
**      and step_increase is read from screen. Name of output file
**      is read from command line. Results read and plotted in python
**      script project1_c_plot.py
*/
# include  <iostream>
# include <fstream>
# include <iomanip>
# include <cmath>

```

```
using namespace std;
ofstream ofile;

// Declaration of functions:
void initialize (int *, int *, int *);
double f(double);
double Solution(double);
double eps(double, double);
double calculate_max_eps(int);

// Functions:
double Solution(double x) {return 1.0-(1-exp(-10))*x-exp(-10*x);}

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

double eps(double u, double v) {return log10(fabs(v-u));}

void initialize (int *initial_n, int *number_of_steps, int *step_increase)
{
  printf("Read in from screen initial_n, number_of_steps and step_increase \n");
  scanf("%d %d %d",initial_n, number_of_steps , step_increase);
  return;
}

double calculate_max_eps(int n) {
    // Constants of the problem:
    double max_eps;
    double h = 1.0/(n+1.0);
    double *x = new double[n+2];
    double *b_twidd = new double[n+1];
    b_twidd[0] = 0;

    // The constituents of the tridiagonal matrix A:
    // Zeroth element not needed, but included to make indexing easy:
    int *a = new int[n+1];
    int *b = new int[n+1];
    int *c = new int[n+1];

    // Temporal variabel in Gaussian elimination:
    double *diag_temp = new double[n+1];

    // Real solution and approximated one:
    double *u = new double[n+2]; // Analytical solution
    double *v = new double[n+2]; // Numerical solution
    // Including extra points to make the indexing easy:
    u[0] = 0;
    v[0] = 0;

    // Filling up x-array, making x[0] = 0 and x[n+1] = 1:
    for (int i=0; i<=n+1; i++) {x[i] = i*h;}

    // Filling up b_twiddle array, i.e. right hand side of equation:
    for (int i=1; i<=n; i++) {
        b_twidd[i] = h*h*f(x[i]);
        u[i] = Solution(x[i]);
        b[i] = 2;
        a[i] = -1;
        c[i] = -1;
    }
    c[n] = 0;
    a[1] = 0;

    // Algorithm for finding v:
    // a(i)*v(i-1) + b(i)*v(i) + c(i)*v(i+1) = b_twidd(i)
    // Row reduction; forward substitution:
    double b_temp = b[1];
    v[1] = b_twidd[1]/b_temp;
    for (int i=2;i<=n;i++) {
       // Temporary value needed also in next loop:
        diag_temp[i] = c[i-1]/b_temp;
        // Temporary diagonal element:
        b_temp = b[i] - a[i]*diag_temp[i];
        // Updating right hand side of matrix equation:
        v[i] = (b_twidd[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];
    }

    // Finding largest relative error and returning it:
    max_eps = fabs(eps(u[1],v[1]));
    for (int i=1; i<=n; i++) {
      // Recall that eps() calculates log_10 of error. Want this to
      // be as small as possible in aboslute value, meaning
      // the largest possible error:
      if (fabs(eps(u[i],v[i])) < fabs(max_eps)) {
         // found new largest value:
         max_eps = eps(u[i],v[i]);
      }
    }

    // Delete arrays, free memory:
    delete [] diag_temp;
    delete [] x;
    delete [] b_twidd;
    delete [] a;
    delete [] b;
    delete [] c;
    delete [] u;
    delete [] v;

  return max_eps;
}

int main(int argc, char* argv[]) {
  // Declaration of initial variables:
  int initial_n, number_of_steps, step_increase;
  double max_error;
  char *outfilename;

  if( argc <= 1 ){
      cout << "Bad Usage: " << argv[0] <<
          " read also output file on same line" << endl;
      exit(1);
    }
    else{
      outfilename = argv[1];
    }

  initialize (&initial_n, &number_of_steps, &step_increase);
  // More variables to be declared:
  double *h_log10 = new double[number_of_steps];
  double *eps_max_log10 = new double[number_of_steps];

  // Write results to txt file:
  ofile.open(outfilename);
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << " n:             log10(h):      log10(max(eps)): " << endl;
  int n = initial_n;
  // Loop over all n producing log_10(h) and log_10(max_eps):
  for (int i=0; i<number_of_steps; i++) {
      h_log10[i] = log10(1.0/(n+1.0));
      eps_max_log10[i] = calculate_max_eps(n);
      ofile << setw(10) << setprecision(5) << n;
      ofile << setw(15) << setprecision(8) << h_log10[i];
      ofile << setw(15) << setprecision(8) << eps_max_log10[i] << endl;
      n *= step_increase;
  }
  ofile.close();
  return 0;
}
```

Listing 6: Program `project1_c.cpp`. This program was ran to produce the .txt files `log_error.txt` (used by the next python script to produce figure 4) and `log_error_powersof10.txt` which is shown in table 1.

```
#
#     Project1: c)
#     The algorithm for solving the tridiagonal matrix
#     equation is implemented in project1_c.cpp.
#     Log of the absolute error is read from file and plotted:
#
from math import *
import numpy as np
import matplotlib.pyplot as plt

def read_logh_logeps(filename):
    infile = open(filename, 'r')
    # Elements to be read in file:
    log_h = []; log_eps = [];
    # Read lines except for the first one:
    lines = infile.readlines()[1:]
    for line in lines:
      words = line.split()
        log_h.append(float(words[1]))
        log_eps.append(float(words[2]))
    infile.close()
    return log_h, log_eps

# Fetching data and plotting them:
log_h, log_eps = read_logh_logeps('log_error.txt')

# Plotting commands:
plt.rcParams.update({'font.size': 16})
fig, ax = plt.subplots(1)
ax.plot(log_h,log_eps,'bo-')
ax.set_xlabel('$\log_{10}(h)$')
ax.set_ylabel('max$(\log_{10}(\mid (u_i-v_i)/u_i \mid ))$')
ax.set_title('Plot of maximal relative error as function of step size')
ax.grid()
plt.show()
```

Listing 7: Program `project1_c_plot.py`. This program was ran after producing the .txt file `log_error.txt` to plot the data shown in figure 4.

```
/*
**     Project1: d)
**     The algorithm for solving the tridiagonal matrix
**     equation is implemented (requiering O(8n) FLOPS) and
**     compared to the method of LU decomposition. The time
**     usage results are written to a txt file as a table.
*/
# include <iostream>
# include <fstream>
# include <iomanip>
# include <time.h>
# include <new>
# include <string>
# include <cstdio>
# include <cstdlib>
# include <cmath>
# include <cstring>
# include "lib.h"

using namespace std;
ofstream ofile;

// Declaration of functions:
void initialize (int *, int *, int *);
double f(double);
double Solution(double);
double eps(double, double);
double time_of_tridiagonal_method(int);
double time_of_LU_method(int);

// Functions:
double Solution(double x) {return 1.0-(1-exp(-10))*x-exp(-10*x);}

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

double eps(double u, double v) {return log10(fabs(v-u));}

void initialize (int *initial_n, int *number_of_steps, int *step_increase)
{
  printf("Read in from screen initial_n, number_of_steps and step_increase \n");
  scanf("%d %d %d",initial_n, number_of_steps, step_increase);
  return;
}

double time_of_LU_method(int n) {
  double *b_twidd = new double[n];
  double *x = new double[n+2];
  double **A;
  double calculation_time;
  double h = 1.0/(n+1.0);

  // Constructing x-array:
  for (int i=0; i<=n+1; i++) {x[i] = i*h;}

  // Constructing right hand side of equation:
  for (int i=0; i<n; i++) {b_twidd[i] = h*h*f(x[i+1]);}

  // Constructing matrix A the brute force way
  // without use of Armadillo here.
  A = new double*[n];
  for (int i=0; i<n; i++) {
    A[i] = new double[n];
  }
  int flag;
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
       flag = 0;
       if (i == j) {
        A[i][j] = 2;
        flag = 1;
       }
       if (i == j+1 || j == i+1 ) {
        A[i][j] = -1;
        flag = 1;
       }
       if (flag == 0) {
        A[i][j] = 0;
       }
    // Could print results to check that A comes out right:
    //cout << " " << A[i][j] << " ";
    }
  //cout << endl;
  }

  // LU decomposition of A:
  // allocate space in memory:
```

```
  int *indx;
  double d;
  indx = new int[n];

  // Time of calculation starts here:
  clock_t start, finish;
  start = clock();

  ludcmp(A, n, indx, &d);   // LU decompose  A[][]
  lubksb(A, n, indx, b_twidd); // Solve linear equation

  // The solution v is now in b_twidd!
  finish = clock();
  calculation_time = (finish-start)/(double)CLOCKS_PER_SEC;

  // Print to compare results:
  //for (int i = 0; i<n; i++) {
  //  cout << " LU-method: v =  " << b_twidd[i] << endl;
  //}

  delete [] x;
  delete [] b_twidd;
  for (int i=0; i<n; i++) {
    delete [] A[i];
  }
  delete [] A;

  return calculation_time;
}

double time_of_tridiagonal_method(int n) {
    // Constants of the problem:
    double calculation_time;
    double h = 1.0/(n+1.0);
    double *x = new double[n+2];
    double *b_twidd = new double[n+1];
    b_twidd[0] = 0;

    // The constituents of the tridiagonal matrix A:
    // Zeroth element not needed, but included to make indexing easy:
    int *a = new int[n+1];
    int *b = new int[n+1];
    int *c = new int[n+1];

    // Temporal variabel in Gaussian elimination:
    double *diag_temp = new double[n+1];

    // Real solution and approximated one:
    double *u = new double[n+2]; // Analytical solution
    double *v = new double[n+2]; // Numerical solution
    // Including extra points to make the indexing easy:
    u[0] = 0;
    v[0] = 0;

    // Filling up x-array:
    for (int i=0; i<=n+1; i++) {
        // Making x[0] = 0 and x[n+1] = 1:
        x[i] = i*h;
    }

    // Filling up b_twiddle array, i.e. right hand side of equation:
    for (int i=1; i<=n; i++) {
        b_twidd[i] = h*h*f(x[i]);
        u[i] = Solution(x[i]);
        b[i] = 2;
        a[i] = -1;
        c[i] = -1;
    }
    c[n] = 0;
    a[1] = 0;

    // Algorithm for finding v:
    // a(i)*v(i-1) + b(i)*v(i) + c(i)*v(i+1) = b_twidd(i)
    // Row reduction; forward substitution:
    // Time of calculation starts here:
    clock_t start, finish;
    start = clock();

    double b_temp = b[1];
    v[1] = b_twidd[1]/b_temp;
    for (int i=2;i<=n;i++) {
       // Temporary value needed also in next loop:
       diag_temp[i] = c[i-1]/b_temp;
       // Temporary diagonal element:
       b_temp = b[i] - a[i]*diag_temp[i];
       // Updating right hand side of matrix equation:
       v[i] = (b_twidd[i]-v[i-1]*a[i])/b_temp;
    }

    // Row reduction; backward substition:
    for (int i=n-1;i>=1;i--) {
        // Can use diag_temp here since a[i] = c[i] = -1:
        v[i] -= diag_temp[i+1]*v[i+1];
    }

    finish = clock();
    calculation_time = (finish-start)/(double)CLOCKS_PER_SEC;

    // Could print to compare results:
    //for (int i=1;i<n+1;i++) {
    //  cout << " Tridiagonal method: v =  " << v[i] << endl;
    //}

    // Delete arrays, free memory:
    delete [] diag_temp;
    delete [] x;
    delete [] b_twidd;
    delete [] a;
    delete [] b;
    delete [] c;
    delete [] u;
    delete [] v;

  return calculation_time;
}

int main(int argc, char* argv[]) {
  // Declaration of initial variables:
  int initial_n, number_of_steps, step_increase;
  double max_error, calculation_time_used_trid, calculation_time_used_LU;

  // Want to store table of result in a .txt file:
  string outfilename;
  outfilename = "Time_comparison.txt";

  initialize (&initial_n, &number_of_steps, &step_increase);

  ofile.open(outfilename);
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << "        n:        time (LU):      time (tridiag): " << endl;
  int n = initial_n;
  // Loop over all n producing table of time used:
  for (int i=0; i<number_of_steps; i++) {
      ofile << setw(10) << setprecision(5) << n;

      // Calculate only with LU decompostition if n is not "too big":
      if ( n <= 3000) {
        ofile << setw(15) << setprecision(8) << time_of_LU_method(n);
        }
      if (n > 3000) {
        ofile << setw(15) << setprecision(8) << " - - ";
      }
      ofile << setw(15) << setprecision(8) << time_of_tridiagonal_method(n) << endl;

      n *= step_increase;
  }
  ofile.close();

  return 0;
}
```

Listing 8: Program `project1_d.cpp`. This program was ran to produce the .txt file `Time_comparison.txt` which is presented in table 2.