Fys4150: Computational Physics

Project 1

Due date **19.th of September, 2016 - 23:59**

Øyvind B. Svendsen, Magnus Christopher Bareid
un: oyvinbsv, magnucb

September 19, 2016

Abstract

The aim of this project is for the students to get familiar with various vector and matrix operations, from dynamic memory allocation to the usage of programs in the library package of the course.

The students were invited to use either brute force-algorithms to calculate linear algebra, or to use a set of recommended linear algebra packages, i.e. Armadillo, that simplify the syntax of linear algebra. Additionally, dynamic memory handling is expected.

The students will showcase necessary algebra to perform the tasks given to them, and explain the way said algebra is implemented into algorithms. In essence, we're asked to simplify a linear second-order differential equation from the form of the Poisson equation, seen as

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

into a one-dimensional form bounded by Dirichlet boundary conditions.

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

so that discretized linear algebra may be committed unto the equation, yielding a number of numerical methods for aquiring the underivated function $u(x)$.

Contents

1 Computational Physics, first project **3**
1.1 The fundamental math . . . 3
1.1.1 Introduction . . . 3
1.1.2 Problem . . . 3

2 Method **3**

3 Explanation of programs **7**
3.1 main.cpp 7
3.1.1 Step by step, in short terms 7
3.2 make_data.py 8
3.3 plot_stuff.py 9

4 Results **9**
4.1 General tridiagonal solver 9
4.2 CPU-time 10
4.3 Relative error 10

5 Conclusion and discussion **11**

6 Appendix - Github **12**

1 Computational Physics, first project

1.1 The fundamental math

1.1.1 Introduction

The production of this document will inevitably familiarize its authors with the programming language `C++`, and to this end mathematical groundwork must first be elaborated to translate a Poisson equation from continuous calculus form, into a discretized numerical form.

The Poisson equation is rewritten to a simplified form, for which a real solution is given, with which we will compare our numerical approximation to the real solution.

The real motivation for this project, though, is to find a viable numerical method for double integration and the implementation of such.

1.1.2 Problem

2 Method

Reviewing the Poisson equation:

$$\begin{aligned}
&\nabla^2\Phi = -4\pi\rho(\mathbf{r}), \text{ which is simplified one-dimensionally by } \Phi(r) = \phi(r)/r \\
\Rightarrow &\frac{d^2\phi}{dr^2} = -4\pi r\rho(r), \text{ which is further simplified by these substitutions:} \\
&r \to x, \\
&\phi \to u, \\
&4\pi r\rho(r) \to f, \qquad \text{which produces the simplified form}
\end{aligned}$$

$$\begin{aligned}
&-u''(x) = f(x), \qquad \text{for which we assume that} \qquad f(x) = 100e^{-10x}, \qquad (1) \\
\Rightarrow &u(x) = 1 - (1 - e^{-10})x - e^{-10x}, \text{ with bounds: } x \in [0,1],\ u(0) = u(1) = 0
\end{aligned}$$

From here on and out, the methods for finding the doubly integrated function $u(x)$ numerically will be deduced.

To more easily comprehend the syntax from a programming viewpoint, one may refer to the each discretized representation of x and u; we know the span of x, and therefore we may divide it up into appropriate chunks. Each of these x_i will yield a corresponding u_i.

We may calculate each discrete x_i by the form $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$ as it is linearly increasing, meaning we use $n+1$ points in our approximation, yielding the step length $h = 1/(n+1)$. Of course, this also yields the discretized representation of $u(x_i) = u_i$.

Through Euler's teachings on discretized numerical derivation methods, a second derivative may be constructed through the form of

$$\begin{aligned}
&\left(\frac{\partial}{\partial x}\right)_{fw} u(x) = \frac{u_{i+1} - u_i}{h}\ , \qquad \left(\frac{\partial}{\partial x}\right)_{bw} u(x) = \frac{u_i - u_{i-1}}{h}\ , \\
&\left(\frac{\partial}{\partial x}\right)^2 u_i = \left(\frac{\partial}{\partial x}\right)_{bw}\left(\frac{\partial}{\partial x}\right)_{fw} u_i = \left(\frac{\partial}{\partial x}\right)_{bw}\left(\frac{u_{i+1} - u_i}{h}\right) = \frac{\left(\frac{\partial}{\partial x}\right)_{bw} u_{i+1} - \left(\frac{\partial}{\partial x}\right)_{bw} u_i}{h} \\
&\left(\frac{\partial}{\partial x}\right)^2 u_i = \frac{u_{+1} - 2u_i + u_{i-1}}{h^2}\ , \qquad \text{which we then use for the problem in question, (1).} \\
\Rightarrow &-\left(\frac{\partial}{\partial x}\right)^2 u(x) = -\frac{u_{+1} - 2u_i + u_{i-1}}{h^2} = \frac{2u_i - u_{i+1} - u_{i-1}}{h^2} = f_i, \qquad \text{for } i = 1, ..., n \qquad (2)
\end{aligned}$$

The discretized prolem can now be solved as a linear algebra problem. Looking closer at the discretized problem:

$$\begin{aligned} -u''(x_i) = \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} &= f_i \ , \qquad \text{for } i = 1, \ldots, n. \\ \Rightarrow \qquad -u_{i+1} + 2u_i - u_{i-1} &= h^2 f_i \ , \qquad \text{substitute } h^2 f_i = y_i, \text{ and test for some values:} \end{aligned}$$

$$\begin{aligned} i = 1: \qquad & -u_2 + 2u_1 - u_0 = y_1 \\ i = 2: \qquad & -u_3 + 2u_2 - u_1 = y_2 \\ i = 3: \qquad & -u_4 + 2u_3 - u_2 = y_3 \\ & \vdots \\ i = n: \qquad & -u_{n+1} + 2u_n - u_{n-1} = y_n \end{aligned}$$

By now it should be obvious to recognize that the coefficients corresponding to each of these terms and their corresponding values of $u(x)$ looks very similar to a tridiagonal matrix multiplication problem which could be represented such as this:

$$-\left(\frac{\partial}{\partial x}\right)^2 u(x) = f(x) \Rightarrow \hat{\mathbf{A}}\hat{\mathbf{u}} = \hat{\mathbf{y}} \Rightarrow \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & -1 \\ 0 & \ldots & \ldots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_{n+1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_{n+1} \end{pmatrix} \tag{3}$$

This matrix equation will not be valid for the first, and last, values of $\hat{\mathbf{y}}$ because they would require elements of $\hat{\mathbf{u}}$ that are not defined; u_{-1} and u_{n+1}, respectively. Given this constraint we see that the matrix-equation gives the same set of equations that we require.

$$\begin{aligned} i = 1: \quad & -u_2 + 2u_1 - u_0 = y_1 \\ i = 2: \quad & -u_3 + 2u_2 - u_1 = y_2 \\ i = 3: \quad & -u_4 + 2u_3 - u_2 = y_3 \\ & \vdots \\ i = n: \quad & -u_{n+1} + 2u_n - u_{n-1} = y_n \end{aligned}$$

The original problem at hand (the simplified Poisson equation) has now been reduced to a numerical linear algebra problem. Solving a tridiagonal matrix-problem like this is done by Gaussian elimination of the tridiagonal matrix $\hat{\mathbf{A}}$, and thereby solving $\hat{\mathbf{u}}$ for the resulting diagonal-matrix, as presumably $\hat{\mathbf{A}}$ and $\hat{\mathbf{y}}$ are the knowns in this set.

Firstly the tridiagonal matrix $\hat{\mathbf{A}}$ is rewritten to a series of three vectors $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, and $\hat{\mathbf{c}}$ that will represent a general tridiagonal matrix. This will make it easier to include other problems of a general form later.

The tridiagonal matrix $\hat{\mathbf{A}}$ with the vector $\hat{\mathbf{y}}$ included now looks like:

$$\begin{pmatrix} b_0 & c_0 & 0 & \dots & \dots & 0 & y_0 \\ a_1 & b_1 & c_1 & \ddots & & \vdots & \vdots \\ 0 & a_2 & b_2 & c_2 & \ddots & \vdots & \vdots \\ \vdots & \ddots & a_3 & b_3 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_n & \vdots \\ 0 & \dots & \dots & 0 & a_{n+1} & b_{n+1} & y_{n+1} \end{pmatrix}$$

, but we only work with rows from $i = 1$ to $i = n$ because of the Dirichlet conditions, as explained above. The matrix which we row-reduce thus looks like this

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 & y_1 \\ a_2 & b_2 & c_2 & \ddots & & \vdots & \vdots \\ 0 & a_3 & b_3 & c_3 & \ddots & \vdots & \vdots \\ \vdots & \ddots & a_4 & b_4 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_n & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n & y_n \end{pmatrix}$$

The Gaussian elimination can be split into two parts; a forward substitution were the matrix-elements a_i are set to zero, and a backward substituion were the vector-elements u_i are calculated from known values.

Starting the Gaussian elimination with the second row, row II, a row operation is performed to maintain the validity of the system. The goal is to remove element a_2 from the row. This is done by subtracting row II, multiplied with some constant k from row I. For every next row operation, there will then be a new k_i calculated.

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 & y_1 \\ \tilde{a}_2 & \tilde{b}_2 & \tilde{c}_2 & \ddots & & \vdots & \tilde{y}_2 \\ 0 & a_3 & b_3 & c_3 & \ddots & \vdots & \vdots \\ \vdots & \ddots & a_4 & b_4 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_{n-1} & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n & y_n \end{pmatrix}$$

$$\tilde{\text{II}} = \text{II} - k_{\text{I}} \times \text{I}$$

where k_{I} is determined by

$$\tilde{a}_2 = 0 = a_2 - k_{\text{I}} b_1 \quad \Rightarrow k_{\text{I}} = \frac{a_2}{b_1}$$

$$\tilde{b}_2 = b_2 - \frac{a_2}{b_1} c_1$$

$$\tilde{c}_2 = c_2 - \frac{a_2}{b_1} \times 0 = c_2$$

$$\tilde{y}_2 = y_2 - \frac{a_2}{b_1} y_1$$

Moving on to row 3, and performing a similar operation:

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 & y_1 \\ 0 & \tilde{b}_2 & c_2 & \ddots & & \vdots & \tilde{y}_2 \\ 0 & \tilde{a}_3 & \tilde{b}_3 & \tilde{c}_3 & \ddots & \vdots & \tilde{y}_3 \\ \vdots & \ddots & a_4 & b_4 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_{n-1} & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n & y_n \end{pmatrix}$$

$$\tilde{\tilde{\text{II}}} = \text{II} - k_{\text{II}} \times \text{II}$$

where k_{II} is determined by

$$\tilde{a}_3 = 0 = a_3 - k_{\text{II}} \tilde{b}_2 \quad \Rightarrow k_{\text{II}} = \frac{a_3}{\tilde{b}_2}$$

$$\tilde{b}_3 = b_3 - \frac{a_3}{\tilde{b}_2} c_2$$

$$\tilde{c}_3 = c_3 - \frac{a_3}{\tilde{b}_2} \times 0 = c_3$$

$$\tilde{y}_3 = y_3 - \frac{a_3}{\tilde{b}_2} \tilde{y}_2$$

Having repeated this procedure, a pattern emerges and an algorithm can be formulated:

$$\tilde{b}_{i+1} = b_{i+1} - \frac{a_{i+1}}{\tilde{b}_i} c_i \ , \qquad \text{for } i = 1, 2, \ldots, n-1$$
$$\tilde{y}_{i+1} = y_{i+1} - \frac{a_{i+1}}{\tilde{b}_i} \tilde{y}_i \ , \qquad \text{for } i = 1, 2, \ldots, n-1$$

After this procedure, the tridiagonal matrix $\hat{\mathbf{A}}$ is transformed into an uppertriangular matrix, $\hat{\tilde{\mathbf{A}}}$. This sort of equation set can be solved iteratively for $\hat{\mathbf{u}}$, since the last equation has one unknown and the other equations has only two unknowns, and thus begins a backward substitution, starting at $i = n$. The matrix multiplication now looks like this:

$$\begin{pmatrix} b_1 & c_1 & 0 & \ldots & \ldots & 0 \\ 0 & \tilde{b}_2 & c_2 & \ddots & & \vdots \\ \vdots & 0 & \tilde{b}_3 & c_3 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & \ddots & \tilde{b}_{n-1} & c_{n-1} \\ 0 & \ldots & \ldots & \ldots & 0 & \tilde{b}_n \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ \vdots \\ \vdots \\ \tilde{y}_{n-1} \\ \tilde{y}_n \end{pmatrix}$$

Solving from the end and back, we get

$$\tilde{y}_n = \tilde{b}_n u_n \quad \Rightarrow u_n = \frac{\tilde{y}_n}{\tilde{a}_n}$$

which will be used in the equation for $n-1$;

$$\tilde{y}_{n-1} = \tilde{b}_{n-1} u_{n-1} + c_{n-1} u_n$$
$$\Rightarrow u_{n-1} = \frac{\tilde{y}_{n-1} - c_{n-1} u_n}{\tilde{b}_{n-1}}$$
$$\Rightarrow u_{n-1} = \frac{\tilde{y}_{n-1} - \frac{c_{n-1}\tilde{y}_n}{\tilde{b}_n}}{\tilde{b}_{n-1}}$$

Which in turn leads directly to the algorithm that solves u_i:

$$u_i = \frac{\tilde{y}_i - \frac{c_i \tilde{y}_{i+1}}{\tilde{b}_{i+1}}}{\tilde{b}_i} \ , \qquad \text{for } i = n-1, n-2, \ldots, 1$$

This method now yields all u_i, except for u_0 and u_{n+1} which were determined by the Dirichlet conditions in the first place. Thusly, $u(x)$ may be aquired through numerical means, when vectors $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, $\hat{\mathbf{c}}$ are provided through a script, providing all their components for the necessary iterations.

This also seems like quite a generalised method to deal with triangular matrix equations, considering the vectors $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, and $\hat{\mathbf{c}}$'s components' vagueness, suggesting that this method has a broader use than our mere double derivation representation.

However, the matrix $\hat{\mathbf{A}}$ in question that we are dealing with, is a very specific matrix where all the components of $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, $\hat{\mathbf{c}}$ are repeating ad nauseam. As such, the row reduced variant of $\hat{\mathbf{A}}$, namely $\hat{\tilde{\mathbf{A}}}$, should be easy to aquire via a specific numerical algorithm, rather than the algebraic algorithm deduced above.

For this purpose, we may simply insert the values that we know each of the vectors $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, $\hat{\mathbf{c}}$ hold into the completedly row reduced form - into $\hat{\tilde{\mathbf{A}}}$ - and consider the shown values for our specific matrix.

$$\begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 & \tilde{y}_1 \\ 0 & \tilde{b}_2 & -1 & \ddots & & \vdots & \tilde{y}_2 \\ \vdots & 0 & \tilde{b}_3 & -1 & \ddots & \vdots & \tilde{y}_3 \\ \vdots & & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & & & \ddots & \tilde{b}_{n-1} & -1 & \tilde{y}_{n-1} \\ 0 & \ldots & \ldots & \ldots & 0 & \tilde{b}_n & \tilde{y}_n \end{pmatrix}$$

with all a_i = -1, all $b_i = 2$, and all c_i = -1:

$$\tilde{b}_2 = b_2 - \frac{a_2}{b_1} c_1 = 2 - \frac{-1}{2} - 1$$
$$\tilde{y}_2 = y_2 - \frac{-1}{b_1} y_1$$

3 Explanation of programs

3.1 main.cpp

3.1.1 Step by step, in short terms

- The main-program is a C++-program designed to take a cmd-line argument that decides the size of the array u, and a boolean argument (0 or 1 that decides wether or not the armadillo-function "solve" should be used.
- An x-array between 0 and 1 is calculated and the appropriate a-, b-, c-, y-arrays are calculated as well. For this program the tridiagonal elements are always -1, 2, and -1, while y(x) follows the function described in the introduction.
- If the boolean cmd-line argument is false, then the program will calculate $\hat{\mathbf{u}}$ using the general tridiagonal method(explanation below) and store the CPU-time it takes to calculate $\hat{\mathbf{u}}$ using this algorithm.
- Next, $\hat{\mathbf{u}}$ will be calculated using the specialized tridiagonal method(explanation below) and store the CPU-time it takes to calculate $\hat{\mathbf{u}}$ using this algorithm.
- Afterwards, since C++ is terrible at plotting data, all the data (CPU-time of methods, x-arrays, and u-arrays for both methods) are stored in separate files in the data-folder with the csv-format.
- If the boolean cmd-line argument is true, then the program will calculate $\hat{\mathbf{u}}$ using LU-decomposition method in the armadillo-library, measure the CPU-time it takes and store this time in the time-datafile.
- The function 'write2file' does exactly what the name suggest, it writes a string to a file. The file-name is a argument to the function and the string is also a argument to the function.

  It is worth noting that this function will only append to the end of a file so that it does not accidentally destroy lots of data.
- The function 'general_tridiag' solves the equation $\hat{\mathbf{A}}\hat{\mathbf{u}} = \hat{\mathbf{y}}$ for $\hat{\mathbf{x}}$ when $\hat{\mathbf{A}}$ is a tri-diagonal matrix. The tridiagonal elements are three arrays of length n that must be given to the function as arguments along with the vector y and the size of the arrays. In figure figure 1 the syntax for solving x_i is presented, calculating $\tilde{b_i}$ and $\tilde{y_i}$ in the forward substitution, and u_i in backward substitution, according to the algorithm in section 2.
- The comments in figure figure 1 counts the number of floting point operations in each line. This ammounts to

$$5flops \times (n-3)iterations + 4flops \times (n-2)iterations = (9n-23)flops$$

  This strange number comes from the fact that the program requires n to be at least 3, since the Dirichlet Boundary Conditions are included in the for-loops. (i.e. excluding calculation of the endpoints, leaving them to be zero according to DBC).
- The function 'specific_tridiag' is an attempt at optimizing the code in 'general_tridiag' and lowering the number of floating point operations per for-loop. This is primarly done by inserting -1.0 for every tridiagonal element a_i and c_i, and precalculating the diagonal elements d_i using the formula $d_i = \frac{i+1}{i}$.

- In figure 2 the algorithm for an optimized, special-case of the tri-diagonal solver is included. The diagonal elements are already calculated, meaning the forward substitution only applies to $\hat{\mathbf{y}}$. The comments counts the number of floating point operations in each iteration.

$$2flops \times (n-3)iterations + 2flops \times (n-2)iterations = (4n-10)flops$$

  The same restraint applies to this calculation, meaning since the algorithm includes the DBC, n must be three or higher to actually compute any values for $\hat{\mathbf{u}}$.

- The function 'LU-decomp' simply solves the set of equtions by making the matrix $\hat{\mathbf{A}}$ with armadillo arrays and using the solve function in armadillo's library. The function returns the CPU-time it takes to calculate $\hat{\mathbf{u}}$.

```
t0 = clock();
//forward substitution
for (int i=1; i<=arg_n-3; i++){
    k = arg_a(i+1)/( (double) arg_b(i) ); //1 flop
    arg_b(i+1) -= k*arg_c(i); //2 flops
    arg_y(i+1) -= k*arg_y(i); //2 flops
}
//backward substitution
for (int i=arg_n-2; i>=1; i--){
    arg_u(i) = (arg_y(i) - arg_u(i+1)*arg_c(i))/( (double) arg_b(i) ); //4 flops
}
t1 = clock();
return (t1 - t0)/((double) CLOCKS_PER_SEC); //measure time of forward and backward substitution
```

Figure 1: The syntax used in the function 'general_tridiag' where all variables beginning with arg are function arguments.

```
t0 = clock();
//forward substitution
for (int i=2; i<=arg_n-2; i++){
  arg_y(i) += arg_y(i-1)/d(i-1); //2 flops
}
//backward substition
for (int i=arg_n-2; i>=1; i--){
    arg_u(i) = (arg_y(i) + arg_u(i+1))/d(i); //2 flops
}
t1 = clock();
return (t1 - t0)/((double) CLOCKS_PER_SEC); //measure time of forward and backward substitution
```

Figure 2: The syntax used in the function 'specific_tridiag' where all variables beginning with arg are function arguments.

3.2 make_data.py

The program performs the experiment (running 'main.cpp') for several values of n.

Firstly it calculates both tridiagonal solvers and LU-decomposition for n = 10,100,1000 in order to compare the CPU-time for the three methods.

Then it chooses a logarithmic scale of n-values between n=10 and n=10'000 in order to produce data-sets for plotting $u(x)$ and ϵ.

3.3 plot_stuff.py

This program fetches relevant data produced by 'main.cpp'/'make_data.py' and stores these data as arrays in a nested dictionary.

Three plotting functions are defined for making appropriate plots:

• 'compare_methods' plots u(x) against x for both tridiagonal solvers and the exact solution for a specific n.

• 'compare_approx_n' plots u(x) against x for several values of n for one of the tri-diagonal solvers.

• 'epsilon_plots' calculates the maximum relative error and makes a log-log-plot against step-length. This done by calculating the maximum difference between u_{exact} and $u_{general}$ for a series of n-values, using this difference to find the relative error $10^{\epsilon} = \left|\frac{v-u}{u}\right|$ for every n-value.

These functions are called to make the plots in section 4.

4 Results

The data can be found in the data-folder on the github-repository[1] and all the figures presented in this section can found in the img-folder in the same repository.

4.1 General tridiagonal solver

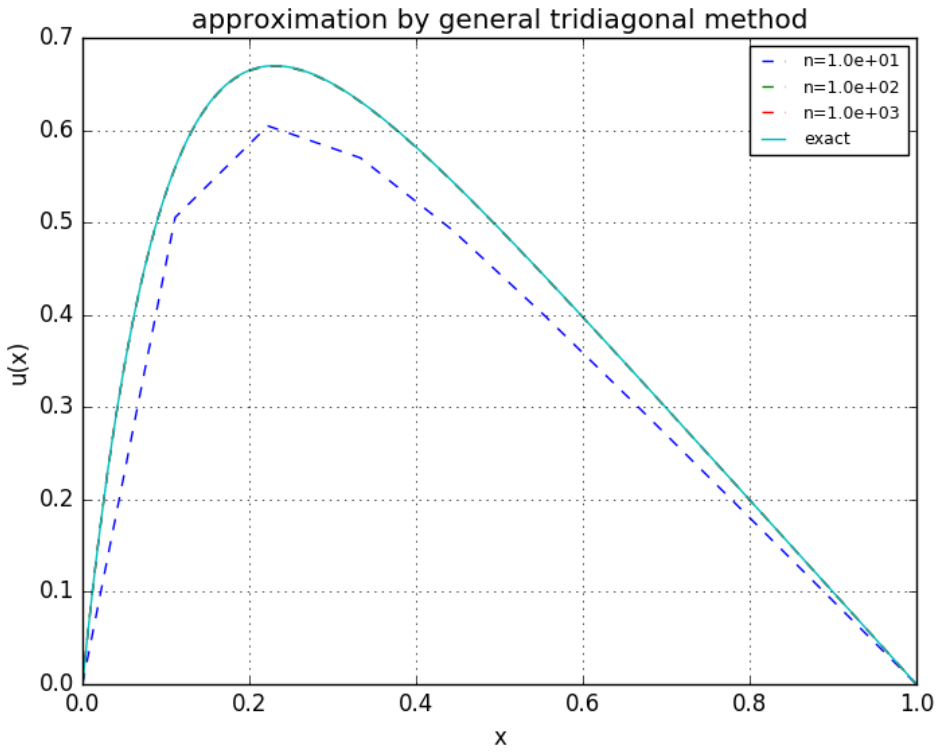

Figure 3: Compare calculated u(x) with exact solution of u(x) for three values of n, using the general tridiagonal solver.

[1] see section 6

4.2 CPU-time

$\log_{10}(n)$	general tridiagonal[s]	specific tridiagonal[s]	LU decomposition [s]
1	0.000010	0.000006	0.008585
2	0.000047	0.000007	0.015610
3	0.000159	0.000123	0.699342
4	0.001421	0.001749	
5	0.020054	0.024508	
6	0.056499	0.064189	

Table 1: CPU-time each methods use for calculating u(x) in seconds.

$\log_{10}(n)$	general tridiagonal[s]	specific tridiagonal[s]	LU decomposition [s]
1	1e+6	1.67e+6	1.16e+3
2	2.13e+6	1.43e+7	6.40e+3
3	6.29e+6	8.13e+6	1.43e+3
4	7.03e+6	5.72e+6	
5	4.99e+6	4.08e+6	
6	1.77e+7	1.56e+7	

Table 2: Matrix size per computation time. This data was found by dividing n (in table table 1 by computational time in the same table).

$\log_{10}(n)$	general tridiagonal[s]	specific tridiagonal[s]	LU decomposition [s]
1	9e+6	6.68e+6	x $\times$ 1.16e+3
2	1.92e+7	5.72e+7	x $\times$ 6.4e+3
3	5.66e+7	3.25e+7	x $\times$ 1.43e+3
4	6.33e+7	2.29e+7	
5	4.49e+7	1.63e+7	
6	1.59e+8	6.24e+7	

Table 3: Flops per second where found by multiplying table 1 by the number of flops per iteration (9 for general and 4 for specific)

4.3 Relative error

n	$\log_{10}(h)$	ϵ
10	-1.0414	-1.0140
100	-2.0043	-3.0709
1000	-3.0004	-5.0381
10000	-4.0000	-3.9641

Table 4: table of epsilon-values of various step-lengths

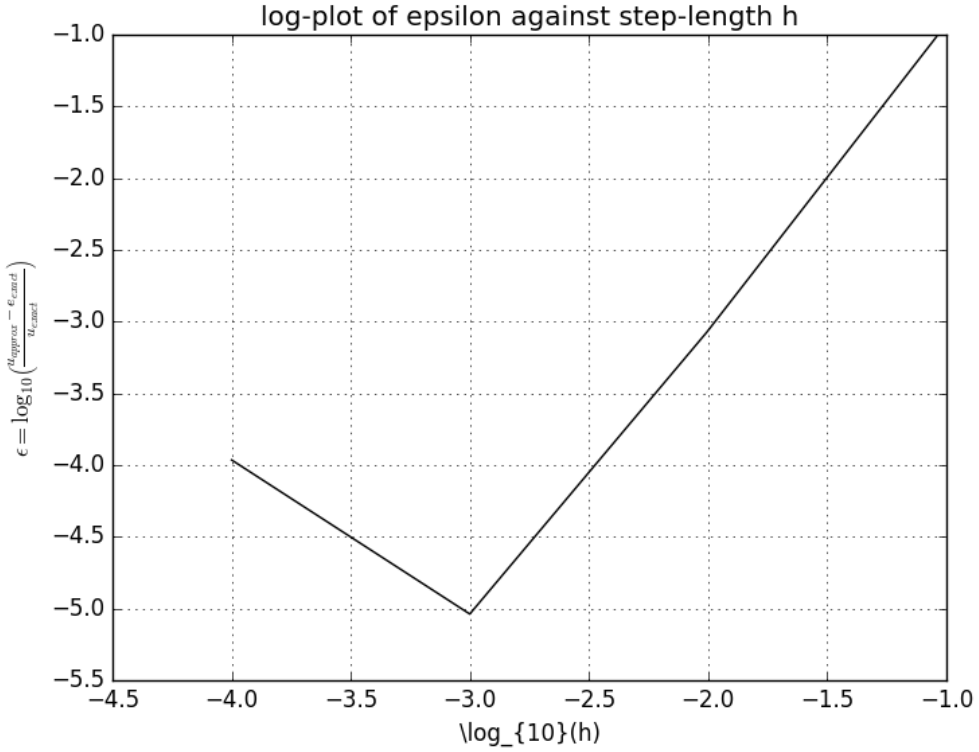

Figure 4: log-log-plot of table ??

5 Conclusion and discussion

From section 4 it becomes clear that the numerical solvers are quite accurate for n=100 and above(h=1/99 and below). This becomes clear when one examines figure figure 3 where the approximation by solver lies on top of the exact solution for n=10 and n=100.

When it comes to computing time, the LU-decomposition is much less efficient(to no surprise for anyone), however there seems to be a slight bug in the timing of specific tridiagonal solver. This method becomes *slower* than the general solver for high values of n, as opposed to the previous expectation (recall that the specific algorithm calculates by $\simeq$ 4n flops, while the general, algebraic algorithm calculates by $\simeq$ 9n flops).

By looking at table 2 the number of total floating point operations per second should the same for all three methods since they are all run on the same computer, however there are discrepencies when comparing the specific and general solvers, as mentioned earlier. the number of floating point operations per iteration for the LU-decomposition can now be determined by solving for x. This factor x is 7758, 3000 and 39580 for the three recorded instances of LU-decomposition. The results are way away of from eachother (hinting towards errors in the program CPU-timing or calculation), but they are all very much larger than the factors 9 and 4 in the general and specific methods.

The program was unsuccesful at storing the data for values of n larger then 1e+4, so the graph of epsilon streches over a limited span of values, however a trend still appears. As the step-length decreases the relative error also decreases (with a slope of $\epsilon = h^2$), until it hits a minimum value at $h = 10^{-3}$ and starts increasing with a slope of $\epsilon = -h$ as h decreases further.

6 Appendix - Github