Project 1
FYS3150

Ann-Silje Kirkvik

github.com/annsilje/fys3150

September 16, 2016

Abstract

This project investigates different numerical methods for solving linear second order differential equations of a certain form. The second order derivative is approximated numerically and yields a linear equation system with a sparse matrix. Three methods are tested for this specific problem. One is a full scale LU-decomposition and backward substitution that also works on dense matrices. The other two methods utilize the fact that the matrix is tridiagonal and has constant values for the diagonals. As expected the fastest algorithm is the one specifically tailored to the second order derivative approximation.

1 Introduction

This project will explore different numerical methods for solving linear second order differential equations of the form

$$-u''(x) = f(x), x \in (0,1), u(0) = u(1) = 0. \tag{1}$$

This problem can be rewritten into a set of linear equation $\boldsymbol{Ax} = \boldsymbol{b}$ with the nice property that the matrix $\boldsymbol{A}$ is tridiagonal. This particular matrix even has an analytical solution, which enables the creation of a tailored algorithm that is fast and memory efficient compared to a brute force linear equation solver. Section 2 explains the theoretical background for the different numerical algorithms tested in this project. Section 3 contains the results for the tests performed and section 4 has some concluding remarks.

2 Description

Starting with a Taylor expansion of $u(x_i + h) = u_{i+1}$ and $u(x_i - h) = u_{i-1}$ and adding these together yields the three point formula, from chapter 3.1 in [1], for the second order derivative:

$$u''(x) = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + \mathcal{O}(h^2).$$

The numerical approximation $v_i''(x_i)$ to $u''(x)$ is then

$$v_i''(x_i) = \frac{v_{i+1} + v_{i-1} - 2v_i}{h^2}.$$

Inserting this approximation into the original differential equation in 1 yields the following formula:

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

for a given set of grid points $(x_i, f_i(x_i))$ for $i = 1, ..., n$, where $h = 1/(n+1)$ is the step length between each x_i. With the boundary conditions $v_0 = v_{n+1} = 0$ this becomes n linear equations:

$$\begin{aligned} 2v_1 - v_2 &= h^2 f_1 \\ -v_1 + 2v_2 - v_3 &= h^2 f_2 \\ -v_2 + 2v_3 - v_4 &= h^2 f_3 \\ &\vdots \\ -v_{n-1} + 2v_n - v_{n+1} &= h^2 f_n \end{aligned}$$

In matrix form this becomes $\boldsymbol{Av} = h^2 \boldsymbol{f}$ or

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & \cdots & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

This linear equation system can be solved numerically in at least three different ways:

- By LU-decomposition and backward substitution.
- With a tridiagonal solver and backward substitution.
- With a analytical custom made solver and backward substitution.

2.1 LU-decomposition

LU-decomposition consists of finding a factorization $\boldsymbol{A} = \boldsymbol{LU}$ such that $\boldsymbol{L}$ is a lower triangular matrix and $\boldsymbol{U}$ is an upper triangular matrix. For instance if $\boldsymbol{A}$ is a 4×4 matrix one such factorization is:

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \quad \text{and} \quad \boldsymbol{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

Once this factorization is found the equation system $\boldsymbol{Av} = \boldsymbol{LUv} = h^2 \boldsymbol{f}$ can be solved by setting $\boldsymbol{Ly} = h^2 \boldsymbol{f}$ and $\boldsymbol{Uv} = \boldsymbol{y}$ and applying backward substitution twice. This execution time of this algorithm scales as $\mathcal{O}(n^3)$ as described in appendix B.

2.2 Tridiagonal solver

A tridiagonal solver exploits the fact that only the diagonal elements and the elements directly above and below the diagonal are different from zero. The matrix $\boldsymbol{A}$ then has the form:

$$\boldsymbol{A} = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_n & b_n \end{bmatrix}$$

By performing Gauss elimination on the equation system it can be shown that the elements directly above the diagonal remains unchanged and the diagonal elements d_i become

$$d_i = b_i - a_i c_{i-1}/d_{i-1} \qquad \text{with } d_1 = b_1. \tag{2}$$

The elements directly below the diagonal becomes zero, as is the goal of the Gauss elimination. The column vector elements w_i becomes

$$w_i = f_i - a_i w_{i-1}/d_{i-1} \qquad \text{with } w_1 = f_1. \tag{3}$$

The detailed steps of the Gauss elimination are listed in appendix A. The equation system can then finally be solved with backward substitution where

$$v_i = w_{i-1} - c_{i-1} v_i/d_{i-1} \qquad \text{with } v_n = h^2 w_n/d_n. \tag{4}$$

Equation 2 consists of three floating point operations which are then performed n times. (Ignoring the fact that the first element is easier to find). The same applies for equation 3. Finding the upper triangular equation system then requires $2 * 3n = 6n$ floating point operations, which scales as $\mathcal{O}(n)$. The backward substitution also requires three floating point operations and therefore the algorithm as a whole scales as $\mathcal{O}(n)$. The factor h^2 is not a part of the generic tridiagonal solver, but rather a constant specific to the problem of approximating the second order derivative and is therefore not considered when counting floating point operations.

2.3 Custom solver

The matrix $\boldsymbol{A}$ is not only tridiagonal but all the elements $a_i = -1$, $b_i = 2$ and $c_i = -1$. Inserting this into equations 2, 3 and 4 respectively yields

$$
\begin{aligned}
d_i &= 2 - 1/d_{i-1} \qquad \text{with } d_1 = 2 \\
w_i &= f_i + w_{i-1}/d_{i-1} \qquad \text{with } w_1 = f_1 \\
v_i &= w_{i-1} + v_i/d_{i-1} \qquad \text{with } v_n = h^2 w_n/d_n
\end{aligned}
$$

Calculating the first few instances of d_i reveals a pattern:

$$
\begin{aligned}
d_1 &= 2 \\
d_2 &= 2 - 1/2 = 3/2 \\
d_3 &= 2 - 1/(3/2) = 4/3 \\
d_4 &= 2 - 1/(4/3) = 5/4 \\
&\vdots \\
d_i &= (i+1)/i.
\end{aligned}
$$

Having precalculated some of the floating point operations, this algorithm is even faster than the generic tridiagonal solver. Finding the diagonal requires one floating point operation per element, a total of n floating point operations. Finding the new column vector is reduced to two floating point operations per element and the same with the backward substitution. This gives a total of $(2+1)n = 3n$ floating point operations for finding the upper triangular equation system and $2n$ floating point operations for the backward substitution. As with the generic tridiagonal solver the algorithm scales as $\mathcal{O}(n)$.

3 Results

If $f(x) = 100e^{-10x}$ the analytical solution to the differential equation in 1 is $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. Because the analytical solution is known the function $f(x)$ is useful for testing the numerical algorithms. The three solvers are tested for different values of n and the source code is found on the project's github address https://github.com/annsilje/fys3150/.

3.1 Precision

Figure 1 shows the difference between the analytical solution and the numerical solution using the generic tridiagonal solver for $n = 10, 10^2, ..10^7$. The endpoints are excluded in the plot. With shorter step lengths h (higher n) the difference gets smaller and there does not seem to be any problems with loss of precision due to round of errors with the values that were tested. Similar plots for the custom made solver and the LU-decomposition solver is shown in figure 3 and 4 in appendix C.

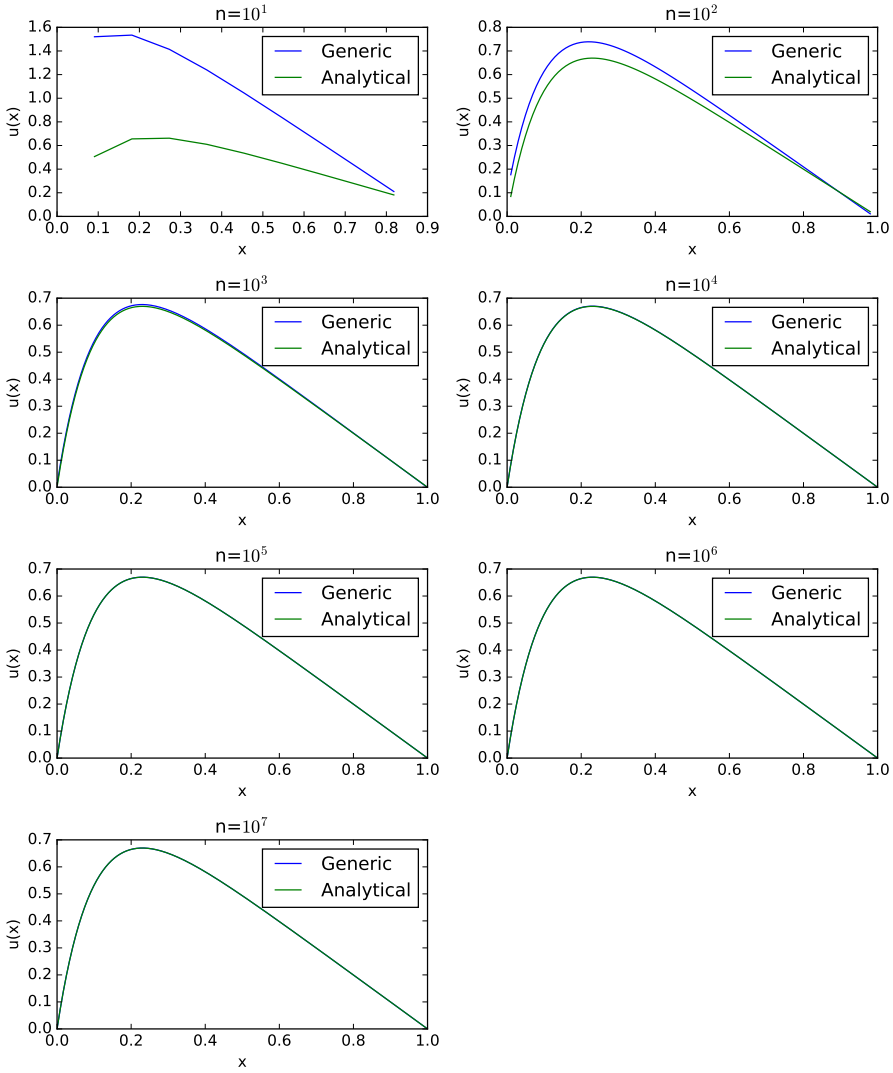

Figure 1: Numerical solution using the generic tridiagonal solver compared to the analytical solution.

The relative error of each step size is computed as

$$\epsilon_i = \left| \frac{v_i - u_i}{u_i} \right| \tag{5}$$

and is plotted in figure 2. The relative error seems to be close to zero for the middle region of $x \in (0, 1)$, but as x approaches the boundary conditions the error increases and reaches its maximum value at x_1. Since u_i is approaching

zero as $x \to 0$ and $x \to 1$, this seems reasonable.

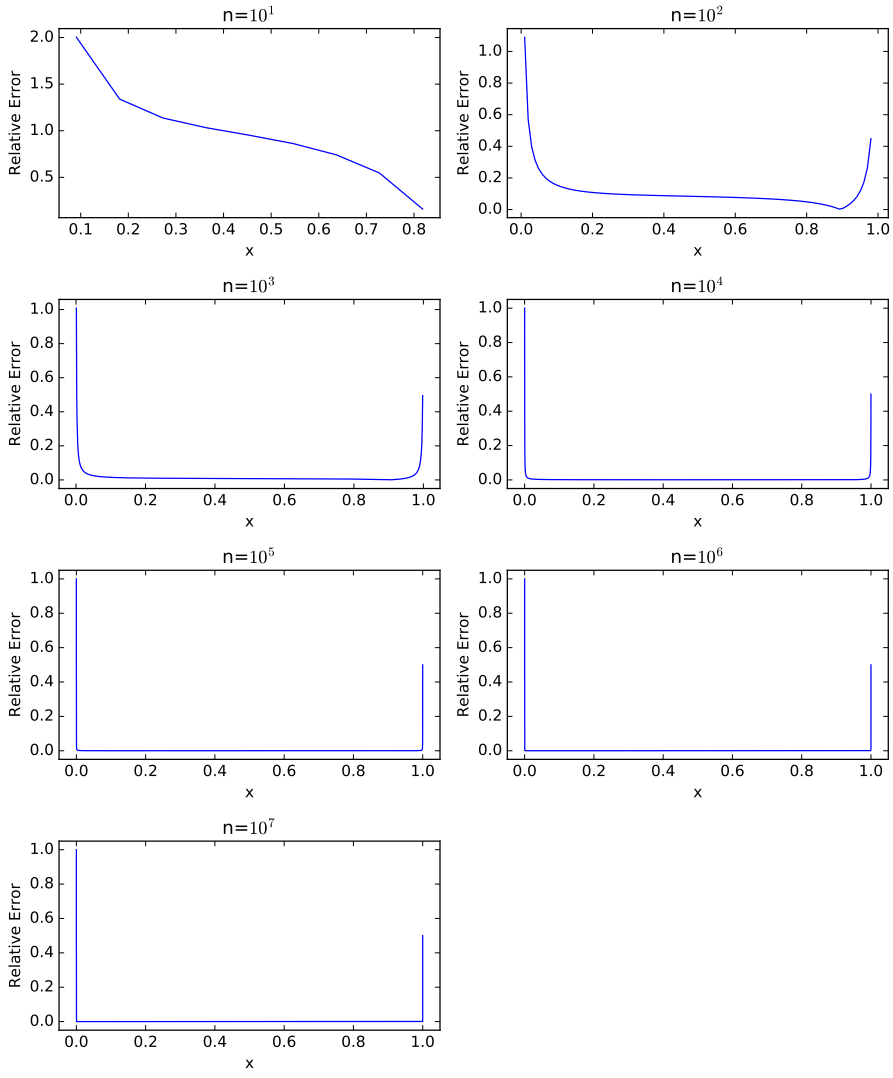

Figure 2: Relative error of the numerical solution obtained with the generic tridiagonal solver.

The maximum value of the relative error for each step length h is displayed in table 1 and it seems to approach 1, with the logarithm approaching zero. Obviously, both the nominator and denominator in equation 5 approach zero as x_i approaches x_0, but these results may indicate that they approach zero at different speeds.

Step size (h)	Max relative error(ϵ)	Max $log(\epsilon)$
9×10^{-2}	2.00329251	6.95×10^{-1}
1×10^{-2}	1.09015546	8.63×10^{-2}
1×10^{-3}	1.00890166	8.86×10^{-3}
1×10^{-4}	1.00088902	8.89×10^{-4}
1×10^{-5}	1.00008889	8.89×10^{-5}
1×10^{-6}	1.00000923	9.23×10^{-6}
1×10^{-7}	1.00000307	3.07×10^{-6}

Table 1: Maximum relative error for each step length using the generic tridiagonal solver.

3.2 Execution time and memory consumption

The execution times of the different algorithms measured using the C++ function `std::clock()` are listed in table 2. As expected the fastest algorithm is the custom made solver, followed by the generic tridiagonal solver. By the number of floating point operations the custom made algorithm was expected to be twice as fast as the generic tridiagonal solver, but in general that is not the case. The method used for timing is probably not suited for very fast algorithms and running the program multiple times yields different results, but in general the custom made algorithm is the fastest. Also, the implementations can consist of operations other than floating point operations, even though these often are very fast. The LU-decomposition, however, is visibly slower even for small n, which is expected. The numbers indicate that the execution time of the LU-decomposition scale by a factor of n^2 faster than the other algorithms. Due to the rapidly increasing execution time and memory consumption the LU-decomposition solver is not tested for values of $n > 1000$.

n	Generic	Custom	G/C	LUD	LUD/C
10^1	1.40×10^{-5}s	3.00×10^{-6}s	4.67	1.90×10^{-5}s	6.33
10^2	1.40×10^{-5}s	1.20×10^{-5}s	1.17	7.39×10^{-3}s	6.16×10^2
10^3	5.00×10^{-5}s	4.10×10^{-5}s	1.22	1.18s	2.88×10^4
10^4	2.94×10^{-4}s	2.86×10^{-4}s	1.03s		
10^5	3.02×10^{-3}s	2.55×10^{-3}s	1.18s		
10^6	2.82×10^{-2}s	2.62×10^{-2}s	1.08s		
10^7	2.80×10^{-1}s	2.55×10^{-1}s	1.10s		

Table 2: Execution times for the different algorithms and their ratio.

The generic and tridiagonal solver only uses arrays and the memory consumption therefore scales as $\mathcal{O}(n)$ with each number occupying 64bits, or 8bytes. The LU-decomposition however uses matrices and the memory consumption scales as $\mathcal{O}(n^2)$ and will become and issue for large values of n. For instance, with $n = 10^5$ the matrix will require around 80GB of memory, while one array only requires about 0.8MB, where the implementation requires 4-5 such arrays

for the generic tridiagonal solver. The custom solver requires even less.

4 Conclusions

The execution time of the LU-decomposition method is obviously a lot longer than the execution time of the tridiagonal solvers. When the matrix $\boldsymbol{A}$ is sparse a lot of the floating point operations in LU-decomposition is multiplication by zero, and the execution time can be reduced significantly by exploiting this knowledg,e as is done with the tridiagonal solvers.

The approximation to the second order derivative is reasonable for smaller values of h, for instance with $n = 10^4$. But the relative error becomes large near the boundary conditions because the analytical solution is approaching zero.

A Gauss elimination

Gauss elimination of the equation system $\boldsymbol{Av} = h^2\boldsymbol{f}$, where a_{ij} is the matrix element of matrix $\boldsymbol{A}$ at row i and column j and R_i is row number i.

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

Set $b_1 = d_1$, $f_1 = w_1$ and apply $R_2 = R_2 - R_1 * a_{j1}/a_{11}$.

$$\begin{bmatrix} d_1 & c_1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & b_2 - \frac{a_2 c_1}{d_1} & c_2 - \frac{a_2 * 0}{b_1} & 0 & \cdots & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} w_1 \\ f_2 - \frac{a_2 w_1}{d_1} \\ f_3 \\ \vdots \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

Set $d_2 = b_2 - a_2 c_1/d_1$, $w_2 = f_2 - a_2 w_1/d_1$ and apply $R_3 = R_3 - R_2 * a_{j2}/a_{22}$.

$$\begin{bmatrix} d_1 & c_1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & d_2 & c_2 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & b_3 - \frac{a_3 c_2}{d_2} & c_3 - \frac{a_3 * 0}{d_2} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} w_1 \\ w_2 \\ f_3 - \frac{a_3 w_2}{d_1} \\ \vdots \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

Continuing this process for every row finally yields

$$\begin{bmatrix} d_1 & c_1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & d_2 & c_2 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & d_3 & c_3 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \cdots & \cdots & \cdots & 0 & d_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & d_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ \vdots \\ w_{n-1} \\ w_n \end{bmatrix}$$

With $d_i = b_i - a_i c_{i-1}/d_{i-1}$ and $w_i = f_i - a_i w_{i-1}/d_{i-1}$.

B LU-decomposition and backward substitution flops

Finding the LU-decomposition involves finding the value for $n \times n$ matrix elements (assuming a square matrix). These elements are found with the following equations, as described in [1]:

$$
\begin{aligned}
u_{1j} &= a_{1j} \\
u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \quad \text{for } i = 2, ..., j-1 \\
u_{jj} &= a_{jj} - \sum_{k=1}^{j-1} l_{jk} u_{kj} \quad \text{for } j = 1, ..., n \\
l_{ij} &= \frac{1}{u_{jj}} \big(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \big) \quad \text{for } i > j
\end{aligned}
$$

This shows that determining a matrix element requires at the maximum n floating point operations. This means that the runtime of the LU-decomposition scales as $\mathcal{O}(n^3)$. The backward substitution is performed twice, but each application only scales as $\mathcal{O}(n^2)$ as with the tridiagonal solvers. The dominating term is therefore $\mathcal{O}(n^3)$.

C Numerical vs Analytical solutions

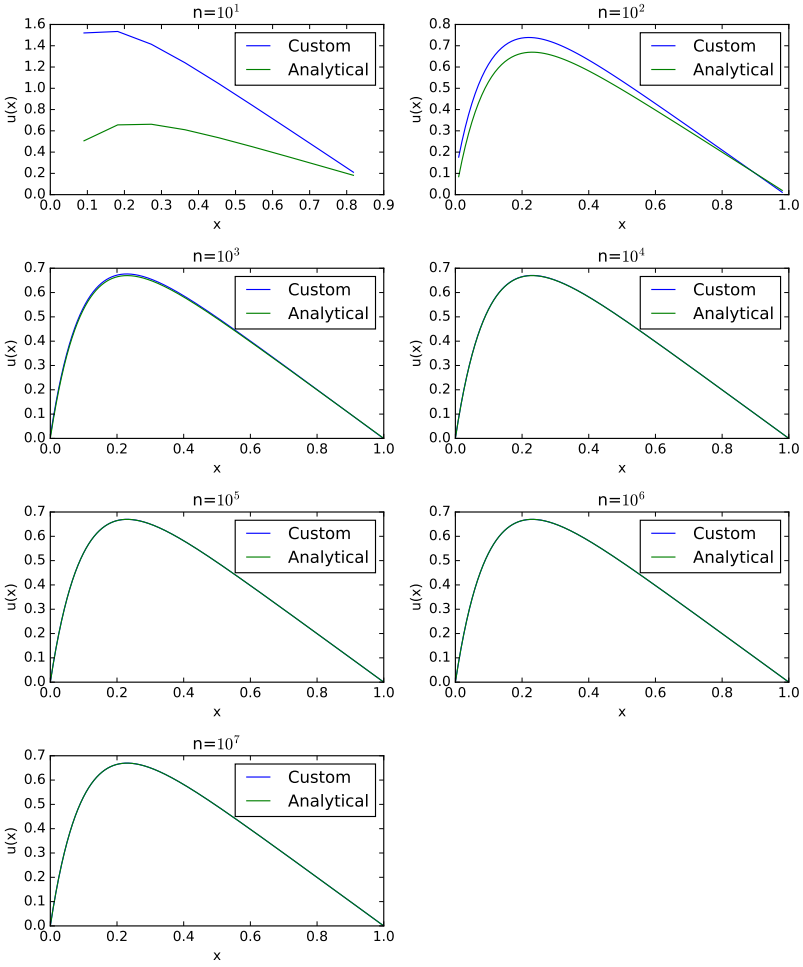

Figure 3: Numerical solution using the tridiagonal solver tailored to this specific matrix compared to the analytical solution.

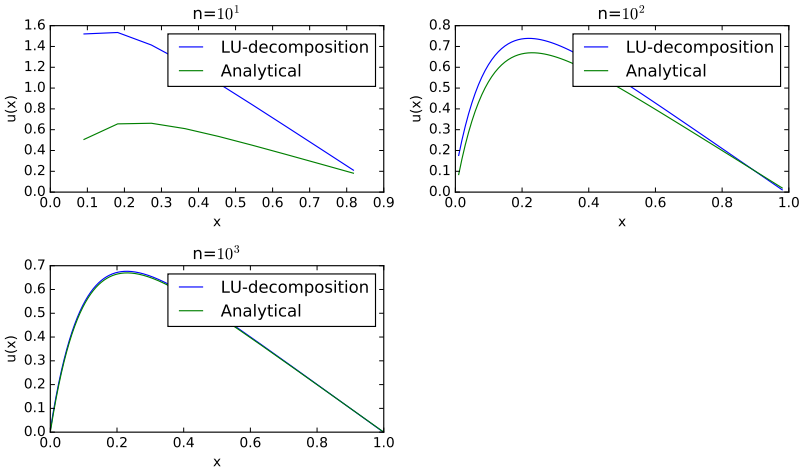

Figure 4: Numerical solution using the LU-decomposition and backward substitution compared to the analytical solution.

References

[1] Hjort-Jensen, M., 2015. Computational physics. Available at https://github.com/CompPhysics/ComputationalPhysics/