# FYS3150 – Project 1

Kristian Gregorius Hustad (`krihus`)
Jonas Gahr Sturtzel Lunde (`jonassl`)

September 17, 2016

With Python, life is much simpler.

*Morten Hjorth-Jensen*

**Abstract**

In this report, we show how the problem of finding a numerical solution to an ODE can be expressed in terms of a matrix equation, we derive and discuss algorithms for solving a (sparse) tridiagonal matrix and compare those algorithms to an more general LU factorization algorithm for dense matrices.
Our findings show that the specialized algorithm yields better performance while using less memory than the general algorithm.

GitHub repository at `https://github.com/KGHustad/FYS3150`

# 1 Introduction

In this project, we aim to find a numerical solution to the following ODE with Dirichlet boundary conditions

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

where the source term, $f(x)$, is a known function.

We derive a numerical scheme, which we solve as a matrix equation, by viewing the scheme as a set of linear equation, which can be mapped onto a tridiagonal matrix. We devise a general solution by gaussian elimination, which we then specialize, and lastly we solve the matrix equation by LU-factorization.

In section 2, we derive and analyze the algorithms, and in section 3, we discuss the implementation and analyze the performance of the algoritms. Finally, a conclusion is given in section 4.

# 2 Discussion of methods

## 2.1 An analytical solution

With the source term

$$f(x) = 100e^{-10x} \tag{2}$$

there exists an analytical solution

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{3}$$

We should verify that (3) with the source term (2) is in fact a solution to (1).

$$\begin{aligned}
\frac{d^2}{dx^2}u(x) &= \frac{d^2}{dx^2}\left(1 - (1 - e^{-10})x - e^{-10x}\right) \\
&= \frac{d}{dx}\left((1 - e^{-10}) + 10e^{-10x}\right) \\
&= -100e^{-10x} \\
&= -f(x) \\
u(0) &= 1 - (1 - e^{-10}) \cdot 0 - e^{-10 \cdot 0} \\
&= 1 - 0 - 1 \\
&= 0 \\
u(1) &= 1 - (1 - e^{-10}) - e^{-10} \\
&= 1 - 1 + e^{-10} - e^{-10} \\
&= 0
\end{aligned}$$

We see that (3) and (2) satifies (1), hence we do indeed have an analytical solution. This analytical solution will be used in the rest of this project to

assert the correctness and measure the rate of convergence of our numerical approximations. Therefore, it is instructive to plot the exact solution here, before we proceed into the inexact realm of numerics.

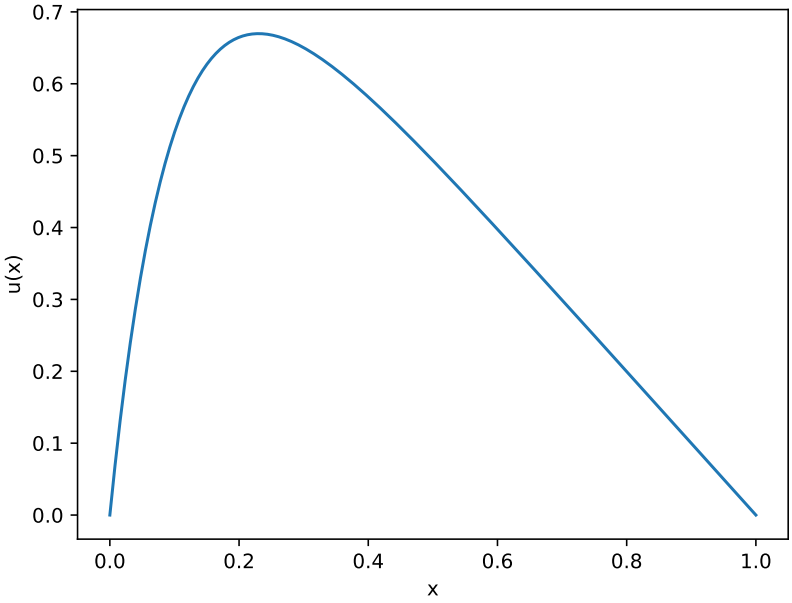


Figure 1: Plotting the analytical solution, (3)

## 2.2 Deriving a numerical scheme

Deriving a numerical scheme for solving (1) by the finite difference method is rather trivial.

### 2.2.1 Taylor expansion

A differentiable function $g(x)$ can be written as

$$g(x) = \sum_{n=0} \frac{g^{(n)}(a)}{n!}(x-a)^n \tag{4}$$

If we want our approximation to be accurate, $|x-a|$ should be close to zero. To ease further notation, we introduce $d = (x-a)$, we get

$$
\begin{aligned}
g(a+d) &= \sum_{n=0} \frac{g^{(n)}(a)}{n!} d^n \\
&= g(a) + g'(a)d + \frac{g''(a)}{2!}d^2 + \frac{g'''(a)}{3!}d^3 + \frac{g''''(a)}{4!}d^4 + \dots \qquad (5) \\
g(a-h) &= \sum_{n=0} \frac{f^{(n)}(a)}{n!} (-d)^n \\
&= g(a) - g'(a)d + \frac{g''(a)}{2!}d^2 - \frac{g'''(a)}{3!}d^3 + \frac{g''''(a)}{4!}d^4 + \dots \qquad (6)
\end{aligned}
$$

The traditional application for such an expansion is to approximate the value of $g$ in a point $x$ close to another point $a$, where $g$ and its derivatives are easy to evaluate. However, we shall take a different view. Recall from section 1 that we know $u''(x)$ – what we seek is $u(x)$. We see that the sum of (5) and (6) becomes

$$
\begin{aligned}
g(a+d) + g(a-d) &= 2\left(g(a) + \frac{g''(a)}{2!}d^2 + \frac{g''''(a)}{4!}d^4 + \dots\right) \qquad (7) \\
&= 2g(a) + g''(a)d^2 + \mathcal{O}(d^4) \qquad (8)
\end{aligned}
$$

Since we only know $g''(a)$ [1], we rewrite as

$$
g(a-d) - 2g(a) + g(a+d) = g''(a)d^2 + \mathcal{O}(d^4) \qquad (9)
$$

### 2.2.2 Discretization

We discretize $u(x), \quad x \in [0,1]$ as $n+2$ ($n$ excluding the boundaries) equally spaced points $v_0, \dots, v_{n+1}$ so that $\Delta x = \frac{1-0}{n+1}$ and $v_i \approxeq u(i\Delta x)$. To denote the set of indices for the points, we employ notation from [1] and introduce $\mathcal{I} = \{i \mid i \in \mathbb{Z}, 0 \leq i \leq n+1\}$ and for the inner points (excluding the boundaries) $\mathcal{I}_i = \{i \mid i \in \mathbb{Z}, 0 < i < n+1\}$. The boundary conditions imply

$$
v_0 = v_{n+1} = 0 \qquad (10)
$$

The source term, $f(x), \quad x \in [0,1]$ is discretized similarly. We can discretize (1) as [2]

[1] In this case, we happen to have an analytical expression for $g''(x)$, which would let us find $g''''(x)$ and find an even more accurate formula, but that approach is not always feasible, so we will not explore it further here.

[2] Here we take a slightly different approach, which is used extensively in [1], by first estimating the first derivate in two points and then the second derivative in the point in between.

$$
\begin{aligned}
-[D_x D_x u]_i &= f_i \\
\frac{-1}{\Delta x}\left([D_x u]_{i+\frac{1}{2}} - [D_x u]_{i-\frac{1}{2}}\right) &= f_i \\
\frac{-1}{\Delta x}\left(\frac{v_{i+1}-v_i}{\Delta x} - \frac{v_i - v_{i-1}}{\Delta x}\right) &= f_i \\
\frac{-1}{\Delta x^2}\left(v_{i+1} - 2v_i + v_{i-1}\right) &= f_i \\
-v_{i+1} + 2v_i - v_{i-1} &= \Delta x^2 f_i
\end{aligned}
$$

where $i \in \mathcal{I}_i$. Setting $h = \Delta x$, we obtain the following equation

$$
-v_{i+1} + 2v_i - v_{i-1} = h^2 f_i, \quad i \in \mathcal{I}_i \tag{11}
$$

This corresponds to a system of linear equations where the unknowns are $v_i, \quad i \in \mathcal{I}_i$. To save space, we introduce $s_i = h^2 f_i, i \in \mathcal{I}_i$.

$$
\begin{aligned}
a_1 v_0 + b_1 v1 + c_1 v_2 &= s_1 \\
a_2 v_1 + b_2 v2 + c_2 v_3 &= s_2 \\
&\vdots \\
a_n v_{n-1} + b_n vn + c_n v_{n+1} &= s_n
\end{aligned} \tag{12}
$$

In our scheme,

$$
a_i = -1, b_i = 2, c_i = -1 \quad \forall \quad i \in \mathcal{I}_i \tag{13}
$$

but we will extend the theory to a more general case.

Being a system of linear equations, (12) can be rewritten as a matrix equation $A\mathbf{v} = \mathbf{s}$, where $\mathbf{v} = (v_1, \ldots, v_n)$, $\mathbf{s} = h^2\mathbf{f} = (h^2 s_1, \ldots, h^2 s_n)$ and $A$ is a $n \times n$ matrix defined as

$$
A = \begin{pmatrix}
b_1 & c_1 & 0 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
a_2 & b_2 & c_2 & 0 & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & a_3 & b_3 & c_3 & 0 & \ldots & \ldots & \ldots & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\ldots & \ldots & \ldots & \ldots & \ldots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & 0 & a_n & b_n
\end{pmatrix} \tag{14}
$$

Notice that $a_1$ and $c_n$ do not appear in the matrix although they are present in the system of equations. As a consequence of the boundary conditions, we have (10), which in turn implies $a_1 v_0 = a_1 \cdot 0 = 0$ and $c_n v_{n+1} = c_n \cdot 0 = 0$, hence we can omit $a_1$ and $c_n$ from the matrix.

## 2.3 A general algorithm for solving an equation with a tridiagonal matrix

We can solve $A\mathbf{v} = \mathbf{s}$ for any tridiagonal matrix $A$ with standard Gaussian elimination. Since most of the matrix elements are zero, we choose to represent the matrix as three arrays, `a`, `b`, and `c`. Additionally, $v_i$ and $s_i$ are stored in the arrays `v` and `s`, respectively.

**Algorithm 1** Gaussian elimination for a tridiagonal matrix

1: **for** $i \leftarrow 2, \ldots, n$ **do** ▷ Forward substitution eliminating $a_i$
2: $\quad b_i \leftarrow b_i - c_{i-1} \cdot \frac{a_i}{b_{i-1}}$ ▷ Update $b_i$
3: $\quad s_i \leftarrow s_i - s_{i-1} \cdot \frac{a_i}{b_{i-1}}$ ▷ Update $s_i$
4: $\quad a_i \leftarrow a_i - b_{i-1} \cdot \frac{a_i}{b_{i-1}} = 0$ ▷ Set $a_i$ to 0 (can be skipped)
5: **end for**
▷ Backward substitution obtaining $v_i$
6: $v_n \leftarrow \frac{s_n}{b_n}$
7: **for** $i \leftarrow n-1, \ldots, 1$ **do**
8: $\quad v_i \leftarrow \frac{s_i - c_i v_{i+1}}{b_i}$
9: **end for**

We see that algorithm 1 requires a number of floating point operations on the order of $8n$ [3], provided that we avoid computing the ratio $\frac{a_i}{b_{i-1}}$ twice and skip line 4. We also note that the space requirements are on the order of $5n$.

## 2.4 A more specialized algorithm

In 2.3, we devised a *general* algorithm, meaning we did not exploit the fact that $a_i = k_a, b_i = k_b, c_i = k_c \quad \forall \quad i \in \mathcal{I}_i$ where the constants $k_a$, $k_b$ and $k_c$ have values as in (13).

[3] To be precise, there are $(n-1)5 + 1 + (n-1)3 = 8n - 7$ flops, but there is no point in fine counting the exact number, as the runtime will be dominated by the leading term, $8n$.

In order to see how we can exploit this fact, we will study the case of $n = 4$.

$$
\begin{aligned}
A_4 &= \begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} \\
&\sim \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & 2-(-1)\cdot\frac{-1}{2} & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} \\
&\sim \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & 2-(-1)\cdot\frac{-1}{\frac{3}{2}} & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} \\
&\sim \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & 0 & 2-(-1)\cdot\frac{-1}{\frac{4}{3}} \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & 0 & \frac{5}{4} \end{pmatrix}
\end{aligned}
$$

At this point we should be recognising a pattern along the diagonal. Namely, that $b_i = \frac{i+1}{i}$. This allows us to precalculate $b_i$. Furthermore, the formulae for $s_i$ and $v_i$ can be simplified by exploiting the fact that $a_i = -1 \quad \forall \quad i \in \mathcal{I}_i$. These observations give rise to a new algorithm.

**Algorithm 2** Gaussian elimination for a special tridiagonal matrix with $a_i = c_i = -1$ and $b_i = 2 \quad \forall \quad i \in \mathcal{I}_i$

**Require:** $b_i = \frac{i+1}{i} \quad \forall \quad i \in \mathcal{I}_i$

1: **for** $i \leftarrow 2, \ldots, n$ **do** ▷ Forward substitution eliminating $a_i$
2: $\quad s_i \leftarrow s_i + \frac{s_{i-1}}{b_{i-1}}$ ▷ Update $s_i$
3: **end for**

▷ Backward substitution obtaining $v_i$

4: $v_n \leftarrow \frac{s_n}{b_n}$
5: **for** $i \leftarrow n-1, \ldots, 1$ **do**
6: $\quad v_i \leftarrow \frac{s_i + v_{i+1}}{b_i}$
7: **end for**

We see that algorithm 2 requires a number of floating point operations on the order of $4n$ [4], which is half of what algorithm 1 requires!

## 2.5 LU decomposition

If we want to solve a series of matrix equations $A\mathbf{v} = \mathbf{s}$ with the same matrix $A$, we could invest some time in performing an LU decomposition of $A$ in $\Theta(n^3)$

[4] The computation of $b_i$ is not counted in as that operation can be vectorized, and in addition $b_i$ is never modified, so we would be able to reuse it for many sets of $v_i$.

time, while we could solve $U\mathbf{v} = \mathbf{w}, L\mathbf{w} = \mathbf{s}$ in $\Theta(n^2)$ time. However, we would need to store a full $n \times n$ matrix, so our memory requirements are on the order of $\Theta(n^2)$.

# 3 Implementation and results

We chose Python as our implementation language because it allows for rapid development and has a syntax which is very close to the algorithmic language we are translating from. For future projects, we will consider using C++, but its main advantage lies is its efficiency, which was not a big problem with our programs.

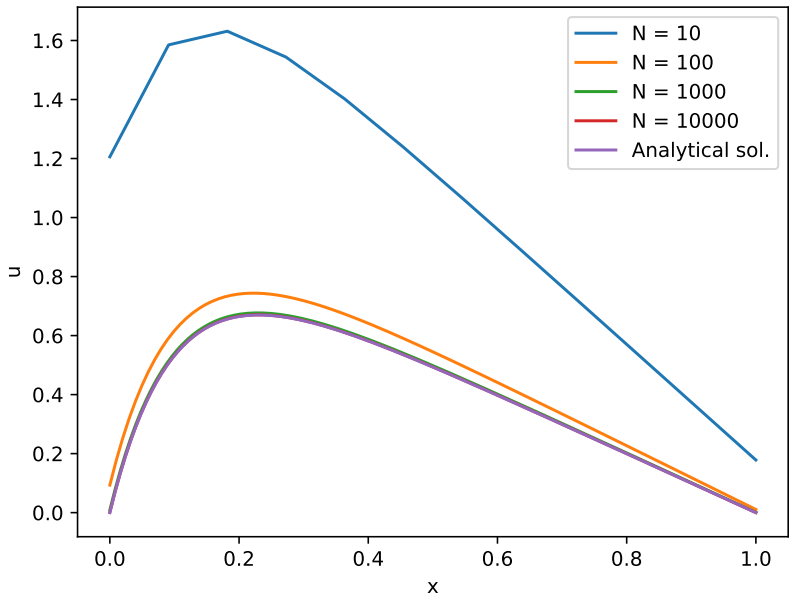


Figure 2: Approximation to $u$ by algorithm 1

## 3.1 Studying the error

We see that

$$\epsilon(10N) = \epsilon(\frac{\Delta x}{10}) \approx \epsilon(N) - \log_{10} 100 = \epsilon(N) - 2 \tag{15}$$

That is, the error is $\Theta(h^2)$, which is in agreement with the derivations we made in section 2.2.1.

| $N$ | $\epsilon$ |
|---|---|
| 1.0e+01 | -1.18 |
| 1.0e+02 | -3.09 |
| 1.0e+03 | -5.08 |
| 1.0e+04 | -7.08 |
| 1.0e+05 | -9.08 |
| 1.0e+06 | -10.16 |

Table 1: Errors for various $N$ with the specialized solver

### 3.2 Efficiency

We find that algorithm 1 takes about twice as long as algorithm 2 for a sufficiently large $n$.

LU factorization, on the other hand, runs much slower.

| N | Gen. | Spec. | LU |
|---|---|---|---|
| 10 | 5.2e−05 | 3.5e−05 | 7.0e−04 |
| 100 | 1.6e−04 | 9.8e−05 | 4.2e−04 |
| 1000 | 1.4e−03 | 8.0e−04 | 2.5e−01 |
| 10000 | 1.3e−02 | 7.7e−03 | 2.4e+02 |

This should come as no surprise, since we only solve a single equation, and the LU factorization runs in $\Theta(n^3)$ time, as mentioned in section 2.5.

## 4 Conclusion

We have shown that it can be very benificial to develop a specialized algorithm to solve the matrix equation $A\mathbf{v} = \mathbf{s}$, where $A$ is a tridiagonal matrix.

By specializing our algorithms, we are able to both lower the memory requirements and the number of floating point operations required.

## References

[1] H. P. Langtangen and S. Linge, *Finite Difference Computing with Partial Differential Equations*. Springer, 2016. [Online]. Available: http://hplgit.github.io/fdm-book/doc/web/index.html