FYS3150 - Project 1 Report

Nils Johannes Mikkelsen
(Dated: September 10, 2018)

A linear second order ordinary differential equation is discretised using a finite difference approximation leading to a tridiagonal matrix equation with constant diagonal entries. The problem is solved using four different algorithms: an LU decomposition-based approach, the Thomas algorithm for tridiagonal matrix equations, a simplification of the Thomas algorithm using constant diagonal entries and a final simplification based on knowing the specific entries in the tridiagonal matrix. The LU decomposition is shown to be very ineffective compared to the other algorithms, while the Thomas algorithm and the first simplified algorithm both significantly improve on the efficiency of the overall computation. The final algorithm is shown to yield no further improvement over the already-simplified algorithm. Adequate numerical solutions with relative errors on the scale $8.3 \cdot 10^{-6}$ are achieved using a discretisation with $n = 1000$ grid points.

ABOUT PROJECT 1

This is a report for Project 1 in FYS3150 Computational Physics at UiO, due September 10th, 2018. [3] The project description was accessed August 27th, 2018 with the following web address:

http://compphysics.github.io/ComputationalPhysics/doc/Projects/2018/Project1/pdf/Project1.pdf

All material written for this report can be found in this GitHub repository:

https://github.com/njmikkelsen/comphys2018/tree/master/Project1

I. INTRODUCTION

Many problems in physics boil down to a second order ordinary differential equation. It is the object of this report to investigate four different numerical algorithms for solving a such equation derived from electrostatics, namely Poisson's equation. By discretising the second order differential using a central finite difference approximation, the differential equation is effectively split into a system of linear equations. This system may be combined into a more compact matrix notation such that the problem becomes a linear algebra problem. This specific problem happens to include a tridiagonal matrix, whose solution may be found using various approaches.

It is the object of this report to investigate four different numerical algorithms for solving the tridigonal matrix equation. The report is most concerned with the accuracy of the approximation, including errors related to floating point arithmetic, and the algorithms' efficiency. The four algorithms include an LU decomposition algorithm and three tridiagonal matrix equation algorithms.

The LU decomposition is performed using the `C++` library `armadillo`. The first tridiagonal matrix algorithm is the standard Thomas algorithm for solving tridiagonal matrix equations, the next algorithm is a simplification of the Thomas algorithm based on constant diagonal entries. Finally, the last algorithm is based on knowing the exact entries in the specific tridiagonal matrix that is going to be solved.

II. THEORY

A. Linear Second Order Ordinary Differential Equations without First Order Terms

1. The general equation

A linear second order ordinary differential equation (ODE) without first order terms can be written as

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} y(x) + k(x)^2 y(x) = f(x) \tag{1}$$

where x and $y = y(x)$ are the independent and dependent variables, $k(x)$ is a variable coefficient and $f(x)$ is the so-called *source* term. The most famous example of equation (1) in Physics is Newton's Second Law of Motion. Section II B introduces another common example, namely Poisson's equation from electrostatics.

2. A specific case

Let $k(x) = 0$ and $f(x) = -100e^{-10x}$ such that equation 1 may be simplified to

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} y(x) = -100e^{-10x} \tag{2}$$

The solution to equation (2), namely $y(x)$, is easily found by integrating twice:

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}x} y(x) &= -10^2 \int \mathrm{d}x\, e^{-10x} \\ &= 10e^{-10x} + C_1 \end{aligned}$$

$$\begin{aligned} y(x) &= 10 \int \mathrm{d}x\, e^{-10x} + C_1 x \\ &= -e^{-10x} + C_1 x + C_2 \end{aligned} \tag{3}$$

Furthermore, introducing the boundary conditions:

$$y(x = 0) = y(x = 1) = 0$$

imposes restraints on C_1 and C_2 such that

$$y(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{4}$$

B. Poisson's Equation from Electrostatics

1. Derivation

Using Gaussian units, Gauss's law of electricity and the Maxwell-Faraday equation may be written as

$$\boldsymbol{\nabla} \cdot \mathbf{E} = 4\pi\rho \tag{5}$$

$$\boldsymbol{\nabla} \times \mathbf{E} = -c^{-1}\frac{\partial}{\partial t}\mathbf{B} \tag{6}$$

where $\mathbf{E}$ and $\mathbf{B}$ are the electric field and the magnetic flux density, ρ is the scalar electric charge distribution and c is the speed of light.

Assuming electrostatics implies that $\partial\mathbf{B}/\partial t = 0$, which further implies that $\boldsymbol{\nabla} \times \mathbf{E} = 0$. As the curl of $\mathbf{E}$ is zero, $\mathbf{E}$ is defined by scalar electric potential φ:

$$\mathbf{E} = -\boldsymbol{\nabla}\varphi \tag{7}$$

Finally, combining equations (5) and (7) yields Poisson's equation:

$$\nabla^2\varphi = -4\pi\rho \tag{8}$$

2. Further development using a spherical distribution

Assuming a spherically symmetric charge distribution $\rho = \rho(r)$ allows for further simplification of equation (8) (the angular terms in $\nabla^2\varphi$ are ignored due to symmetry):

$$\nabla^2\varphi(r) = r^{-2}\frac{\mathrm{d}}{\mathrm{d}r}\left[r^2\frac{\mathrm{d}}{\mathrm{d}r}\varphi(r)\right] = r^{-2}\left[2r\varphi(r) + r^2\frac{\mathrm{d}^2}{\mathrm{d}r^2}\varphi(r)\right]$$
$$= r^{-1}\left[\varphi(r) + r\frac{\mathrm{d}}{\mathrm{d}r}\varphi(r)\right] = r^{-1}\frac{\mathrm{d}^2}{\mathrm{d}r^2}\left[r\varphi(r)\right]$$

Finally, define $\phi = \varphi(r)/r$ such that Poisson's equation can be written as

$$\frac{\mathrm{d}^2}{\mathrm{d}r^2}\phi(r) = -4\pi r\rho(r) \tag{9}$$

Equation (9) is recognised as a special case of equation (1) with

$$y(x) = \phi(x), \quad k(x) = 0 \quad \text{and} \quad f(x) = -4\pi x\rho(x)$$

To arrive on the specific case introduced in section II A 2, the charge distribution must be given by:

$$\rho(r) = \frac{25}{\pi}r^{-1}e^{-10r}$$

C. Finite Difference Approximation Methods for Second Order Derivatives

The Taylor expansion of any single-variable function $f(x)$, centered about $x = a$, can be written as

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{6}(x-a)^3 + \frac{f^{(4)}(a)}{24}(x-a)^4 + \cdots \tag{10}$$

Let $a = x_0$ and $x = x_0 \pm \Delta x$ such that:

$$f(x_0 \pm \Delta x) = f(x_0) \pm f'(x_0)\Delta x + \frac{f''(x_0)}{2}\Delta x^2 \pm \frac{f'''(x_0)}{6}\Delta x^3 + \frac{f^{(4)}(x_0)}{24}\Delta x^4 + \cdots \tag{11}$$

This implies that:

$$f(x_0 + \Delta x) + f(x_0 - \Delta x) = 2f(x_0) + f''(x_0)\Delta x^2 + \frac{f^{(4)}(x_0)}{12}\Delta x^4 + \cdots$$

Solving for $f''(x_0)$ yields the central second order finite difference approximation for second order derivatives:

$$f''(x_0) = \frac{f(x_0 - \Delta x) - 2f(x_0) + f(x_0 + \Delta x)}{\Delta x^2} - \frac{f^{(4)}(x_0)}{12}\Delta x^2 - \cdots \tag{12}$$

III. METHOD

The main equation that is solved in this report is equation (2) with Dirichlet boundary conditions $y(0) = y(1) = 0$, such that the analytic solution is given by equation (4). To limit the scope of the numerical solution, x values outside the range $[0, 1]$ are ignored.

A. Discretisation of The Differential Equation

To begin the process, the x-range $[0, 1]$ is divided into $n + 1$ equal slices such that the continuous x axis is replaced by an x grid with $n + 2$ points. Each grid point is denoted by x_i, where $i \in \{0, 1, \ldots, n + 1\}$, and is seperated by a grid spacing of $h = (x_{n+1} - x_0)/(n + 1)$. As expected, the grid boundaries are $x_0 = 0$ and $x_{n+1} = 1$, which implies that $h = 1/(n + 1)$.

Furthermore, the continuous functions $y(x)$ and $f(x)$ are replaced by the discrete sets $\{y_i\}$ and $\{f_i\}$ respectively. The set elements are defined by $y_i = y(x_i)$ and $f_i = f(x_i)$. The Dirichlet boundary conditions demand that $y_0 = y_{n+1} = 0$.

Using the discretised variables, equation (2) may be discretised using the finite difference approximation of second order derivatives from section II C:

$$y''(x_i) = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} + \mathcal{O}\big(h^2\big) \tag{13}$$

where $\mathcal{O}\big(h^2\big)$ is local truncation error. With n approximations, the global truncation error is $n\mathcal{O}\big(h^2\big) \approx \mathcal{O}(h)$ for small h.

The differential equation may thus be written as a coupled system of linear equations:

$$\begin{aligned} y_0 - 2y_1 + y_2 &= h^2 f_1 \\ y_1 - 2y_2 + y_3 &= h^2 f_2 \\ &\vdots \\ y_{n-1} - 2y_n + y_{n+1} &= h^2 f_n \end{aligned}$$

As $y_0 = y_{n+1} = 0$, this system may be written as a matrix equation:

$$\begin{bmatrix} -2 & 1 & 0 & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \vdots \\ 0 & 1 & -2 & 1 & \cdots & \vdots \\ \vdots & 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & -2 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

By introducing the new variable $\tilde{y}_i = y_i/h^2$, the matrix equation may be written as

$$D\tilde{\mathbf{y}} = \mathbf{f} \tag{14}$$

where D is the left hand side tridiagonal matrix, and $\tilde{\mathbf{y}}$ and $\mathbf{f}$ are column vectors.

B. Matrix Equation Algorithms

The goal now is to develope a series of algorithms that are capable of solving equation (14). The algorithms have varying "degrees of knowledge" of the equation in that they are based on different assumptions. However, all algorithms assume the matrix is a square matrix. Limited by their assumptions, the scope of the algorithms' areas of application differ (although they obviously share the ability to solve equation (14)) and thus solve the same matrix equations with different efficiencies.

Each algorithm is only focused on equation (14), thus the final solution $\mathbf{y}$ requires an additional calculation. It involves calculating h^2 and computing each element $y_i = (h^2)\tilde{y}_i$, which in total amounts to $n + 1$ additional FLOPS for each algorithm.

To avoid having to reintroduce notation, the following definitions will be used throughout. Any linear matrix equation may be written as

$$M\mathbf{v} = \mathbf{u} \tag{15}$$

where M is an $n \times n$ matrix with entries m_{ij}, and $\mathbf{v}$ and $\mathbf{u}$ are n-dimensional vectors with entries v_i and u_i respectively. Note that $i, j \in \{1, \ldots, n\}$.

1. *General matrix equation algorithm: LU decomposition*

The first algorithm to be presented in this report is the well-known LU decomposition solution. The process is divided into two components: the LU factorisation and solving the resulting matrix equations. As it happens, Project 1 does not require the report to include an in-depth description of LU decompositioning. The details surrounding the algorithm are therefore ignored. Nevertheless, the general principle behind the solution is explained below. When later performing the LU decomposition numerically, a standard algorithm supplied by the `armadillo` library is used.

Suppose the matrix M may be written as the product of two matrices L and U: $M = LU$, where L is a unit (diagonal entries equals 1) *lower* triangular matrix and U is an *upper* triangular matrix. Equation (15) may thus be written as $L(U\mathbf{v}) = \mathbf{u}$, which can be further separated as two matrix equations:

$$U\mathbf{v} = \mathbf{y} \quad \text{and} \quad L\mathbf{y} = \mathbf{u} \tag{16}$$

There are several algorithms for finding L and U. Although not necessarily the most efficient, the one presented here is a simple algorithm:

1. M is row-reduced to echelon form U by a sequence of row-replacement operations (if possible).

2. Then, the same sequence of row-replacement operations reduces L to I (the identity matrix).

Having found L and U, the next steps are as follows:

1. $\mathbf{y}$ may be found by solving $L\mathbf{y} = \mathbf{u}$, which is very simplified by the fact that L is unit lower triangular: Starting from the upper row, each element below the current diagonal entry is directly removed by simple row-operations.

2. Finally, the solution $\mathbf{v}$ is found by solving $U\mathbf{v} = \mathbf{y}$. The entries in $\mathbf{v}$ may computed sequentially by a backward substitution, starting from the lowest row.

The benefits of LU decompositioning really arises when solving several matrix equations on the form $M\mathbf{v} = \mathbf{d}$, where $\mathbf{d}$ may differ with each equation. Nontheless, the process is fairly quick in on itself: as opposed to Gaussian elimination's $\mathcal{O}(n^3)$ FLOPS, LU decompositioning requires $\mathcal{O}(2n^2)$ FLOPS (this includes both the LU fatorisation and the solving of equations (16)).

2. A generalised tridiagonal matrix equation algorithm

The next matrix algorithm solves a tridiagonal matrix equation with arbitrary diagonal entries. The algorithm presented here is often known as the Thomas algorithm and is in principle just Gaussian elimination, however specialised for tridiagonal matrices.

As M is tridiagonal, Equation (14) may be written as:

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_1 & b_2 & c_2 & 0 & \cdots & \vdots \\ 0 & a_2 & b_3 & c_3 & \cdots & \vdots \\ \vdots & 0 & a_3 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix}$$

such that the linear equations may be written as

$$\begin{aligned} b_1 v_1 + c_1 v_2 &= u_1 \\ a_{i-1} v_{i-1} + b_i v_i + c_i v_{i+1} &= u_i, \quad i = 2, \ldots, n-1 \\ a_{n-1} v_{n-1} + b_n v_n &= u_n \end{aligned}$$

The first step is to perform a so-called "forward sweep" in which the lower diagonal is removed and the central diagonal is normalised:

1. The first equation is divided by b_1 so that $m_{11} = 1$.
2. For each equation from $i = 2$ to $i = n$:
   (a) The current equation is normalised so that $m_{i,i-1} = 1$.
   (b) The previous equation is subtracted from the current equation so that $m_{i,i-1} = 0$.
   (c) The current equation is renormalised so that $m_{ii} = 1$.

Having completed the forward sweep, the matrix equation now reads

$$\begin{bmatrix} 1 & \tilde{c}_1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \tilde{c}_2 & 0 & \cdots & \vdots \\ 0 & 0 & 1 & \tilde{c}_3 & \cdots & \vdots \\ \vdots & 0 & 0 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 & \tilde{c}_{n-1} \\ 0 & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{u}_1 \\ \tilde{u}_2 \\ \vdots \\ \tilde{u}_{n-1} \\ \tilde{u}_n \end{bmatrix}$$

where

$$\tilde{c}_i = \frac{c_i}{b_i - a_i \tilde{c}_{i-1}} \quad \text{with} \quad \tilde{c}_1 = \frac{c_1}{b_1} \tag{17}$$

$$\tilde{u}_i = \frac{u_i - a_i \tilde{u}_{i-1}}{b_i - a_i \tilde{c}_{i-1}} \quad \text{with} \quad \tilde{u}_1 = \frac{u_1}{b_1} \tag{18}$$

A forward sweep thus consists of computing:

× $\tilde{c}_1$ and $\tilde{u}_1$. 2 FLOPS

× For $i = 2, \ldots, n-1$: ($n-2$ iterations)

→ $\alpha_i = b_i - a_i \tilde{c}_{i-1}$. 2 FLOPS

→ $\tilde{c}_i = c_i / \alpha_i$. 1 FLOPS

→ $\tilde{u}_i = (u_i - a \tilde{u}_{i-1}) / \alpha_i$. 3 FLOPS

× $\tilde{u}_n = (u_n - a_n \tilde{u}_{n-1}) / (b_n - a_n \tilde{c}_{n-1})$. 5 FLOPS

The total number of FLOPS during a forward sweep is therefore

$$\begin{aligned} \text{FLOPS}_{\text{sweep}} &= 2 + (n-2)(2+1+3) + 5 \\ &= 6n - 5 \end{aligned} \tag{19}$$

Following the forward sweep, the solution $\mathbf{v}$ is obtained via a backward substitution:

1. The final element is found directly: $v_n = \tilde{u}_n$.
2. For each equation from $j = n-1$ to $j = 1$:
   (a) The previous equation is multiplied with $\tilde{c}_j$ and subtracted from the current equation so that $m_{j-1,j} = 0$.

By the end of the substitution, the matrix M will have been row-reduced to the identity matrix and the coefficients of $\mathbf{v}$ are determined using the recursive relation:

$$v_n = \tilde{u}_n, \quad v_j = \tilde{u}_j - \tilde{c}_j v_{j+1} \tag{20}$$

Calculating v_n requires 0 FLOPS, but calculating v_j requires 2 FLOPS. Thus in total:

$$\text{FLOPS}_{\text{sub}} = 2(n-1) = 2n - 2 \tag{21}$$

which implies that the complete algorithm for solving any tridiagonal matrix equation requires

$$\text{FLOPS}_{\text{general}} = 8n - 7 \tag{22}$$

3. A specialised tridiagonal matrix equation algorithm

Having presented a generalised algorithm for solving tridiagonal matrix equations, this subsection will focus on a specialised algorithm that assumes each individual diagonal contains identical entries. Mathematically, this implies that

$$a_i = a, \quad b_i = b \quad \text{and} \quad c_i = c,$$

which simplifies the linear equations to

$$\begin{aligned} b v_1 + c v_2 &= u_1 \\ a v_{i-1} + b v_i + c v_{i+1} &= u_i, \quad i = 2, \ldots, n-1 \\ a v_{n-1} + b v_n &= u_n \end{aligned}$$

The specialised algorithm is based on a simplification of $\tilde{c}_i$ and $\tilde{u}_i$ from the generalised algorithm:

$$\tilde{c}_i = \frac{c_i}{b_i - a_i\tilde{c}_{i-1}} = \frac{\frac{c}{a}}{\frac{b}{a} - \tilde{c}_{i-1}}$$
$$\tilde{u}_i = \frac{u_i - a_i\tilde{u}_{i-1}}{b_i - a_i\tilde{c}_{i-1}} = \frac{\frac{u_i}{a} - \tilde{u}_{i-1}}{\frac{b}{a} - \tilde{c}_{i-1}}$$

The first step is to divide the first equation by b and all other equations by a such that the linear equations read

$$\begin{aligned} v_1 + \rho_1 v_2 &= u_1' \\ v_{i-1} + \rho_2 v_i + \rho_3 v_{i+1} &= u_i', \quad i = 2, \ldots, n-1 \\ v_{n-1} + \rho_2 v_n &= u_n' \end{aligned}$$

where

$$\rho_1 = \frac{c}{b}, \quad \rho_2 = \frac{b}{a}, \quad \rho_3 = \frac{c}{a},$$
$$u_1' = \frac{u_1}{b} \quad \text{and} \quad u_i' = \frac{u_i}{a} \quad \text{for} \quad i = 2, \ldots, n.$$

This initial operation requires 3 FLOPS for ρ_1, ρ_2 and ρ_3, in addition to n FLOPS for $\mathbf{u}'$. Hence,

$$\text{FLOPS}_{\text{setup}} = n + 3 \tag{23}$$

Before even having started on the forward sweep, the tridiagonal matrix equation thus reads

$$\begin{bmatrix} 1 & \rho_1 & 0 & \cdots & \cdots & 0 \\ 1 & \rho_2 & \rho_3 & 0 & \cdots & \vdots \\ 0 & 1 & \rho_2 & \rho_3 & \cdots & \vdots \\ \vdots & 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \rho_2 & \rho_3 \\ 0 & \cdots & \cdots & 0 & 1 & \rho_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} u_1' \\ u_2' \\ \vdots \\ u_{n-1}' \\ u_n' \end{bmatrix}$$

The next step is to perform a simpler, although conceptually identical, forward sweep:

For each equation from $i = 2$ to $i = n$:

(a) The previous equation is subtracted from the current equation so that $m_{i,i-1} = 0$.

(b) The current equation is renormalised so that $m_{ii} = 1$.

Having completed the forward sweep, the matrix equation now reads

$$\begin{bmatrix} 1 & \tilde{c}_1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \tilde{c}_2 & 0 & \cdots & \vdots \\ 0 & 0 & 1 & \tilde{c}_3 & \cdots & \vdots \\ \vdots & 0 & 0 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 & \tilde{c}_{n-1} \\ 0 & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{u}_1 \\ \tilde{u}_2 \\ \vdots \\ \tilde{u}_{n-1} \\ \tilde{u}_n \end{bmatrix}$$

where the new coefficients are

$$\tilde{c}_i = \frac{\rho_3}{\rho_2 - \tilde{c}_{i-1}} \quad \text{with} \quad \tilde{c}_1 = \rho_1 \tag{24}$$
$$\tilde{u}_i = \frac{u_i' - \tilde{u}_{i-1}}{\rho_2 - \tilde{c}_{i-1}} \quad \text{with} \quad \tilde{u}_1 = u_1' \tag{25}$$

The specialised forward sweep consists of computing:

× For $i = 2, \ldots, n-1$: ($n-2$ iterations)

→ $\alpha_i = \rho_2 - \tilde{c}_{i-1}$. 1 FLOPS

→ $\tilde{c}_i = \rho_3/\alpha_i$. 1 FLOPS

→ $\tilde{u}_i = (u_i' - \tilde{u}_{i-1})/\alpha_i$. 2 FLOPS

× $\tilde{u}_n = (u_n' - \tilde{u}_{n-1})/(\rho_2 - \tilde{c}_{n-1})$. 3 FLOPS

The total number of FLOPS during a specialised forward sweep is therefore

$$\begin{aligned} \text{FLOPS}_{\text{special sweep}} &= (n-2)(1+1+2) + 3 \\ &= 4n - 5 \end{aligned} \tag{26}$$

The backward substitution in the specialised algorithm is identical to the backward substitution in the generalised algorithm and will thus not be discussed here. The complete specialised algorithm therefore requires

$$\text{FLOPS}_{\text{special}} = 7n - 4 \tag{27}$$

This suggests that the specialised algorithm is about $8/7 \approx 1.14$ times faster than the generalised algorithm.

4. A taylored matrix equation algorithm

The fourth and final algorithm presented here is an algorithm taylored to the actual problem at hand, namely equation (14). The notation used here will match the original notation used in section III A, note in particular:

$$M = D, \quad \mathbf{v} = \mathbf{y} \quad \text{and} \quad \mathbf{u} = h^2\mathbf{f}$$

Furthermore, the diagonal entries are as follows:

$$a = c = 1 \quad \text{and} \quad b = -2$$

In order to avoid having to multiply by h^2, the equation is written in terms of $\tilde{y}_i = y_i/h^2$. Moreover, when the first equation is divided by -2 and the rest are kept the same, the linear system of equations read

$$\begin{aligned} \tilde{y}_1 - \frac{1}{2}\tilde{y}_2 &= -\frac{1}{2}f_1 \\ \tilde{y}_{i-1} - 2y_i + \tilde{y}_{i+1} &= f_i, \quad i = 2, \ldots, n-1 \\ \tilde{y}_{n-1} - 2\tilde{y}_n &= f_n \end{aligned}$$

which in matrix form is:

$$\begin{bmatrix} 1 & -\frac{1}{2} & 0 & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \vdots \\ 0 & 1 & -2 & 1 & \cdots & \vdots \\ \vdots & 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & -2 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_{n-1} \\ \tilde{y}_n \end{bmatrix} = \begin{bmatrix} -\frac{1}{2}f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}$$

By performing a forward sweep in the same manner as presented in the specialised matrix equation algorithm, the resulting matrix equation is

$$\begin{bmatrix} 1 & -\frac{1}{2} & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \tilde{c}_2 & 0 & \cdots & \vdots \\ 0 & 0 & 1 & \tilde{c}_2 & \cdots & \vdots \\ \vdots & 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 & \tilde{c}_{n-1} \\ 0 & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_{n-1} \\ \tilde{y}_n \end{bmatrix} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \vdots \\ \tilde{f}_{n-1} \\ \tilde{f}_n \end{bmatrix}$$

where

$$\tilde{c}_i = \frac{1}{-2 - \tilde{c}_{i-1}} \quad \text{with} \quad \tilde{c}_1 = -\frac{1}{2} \tag{28}$$

$$\tilde{f}_i = \frac{f_i - \tilde{f}_{i-1}}{-2 - \tilde{c}_{i-1}} \quad \text{with} \quad \tilde{f}_1 = -\frac{1}{2}f_1 \tag{29}$$

The relationship between $\tilde{c}_i$ and $\tilde{f}_i$ is immediately noticeable:

$$\tilde{f}_i = (f_i - \tilde{f}_{i-1})\tilde{c}_i = (\tilde{f}_{i-1} - f_i)(-\tilde{c}_i) \tag{30}$$

(The reasoning behind the second equality will become apparent in due time.)

The next step is to explore $\tilde{c}_i$ a little more in detail. For starters, it may be written as

$$\tilde{c}_i = \frac{-1}{2 + \tilde{c}_{i-1}} \quad \text{with} \quad \tilde{c}_1 = -\frac{1}{2} \tag{31}$$

The goal is to rewrite the expression on a closed form (without chained fractions). By writing out the first iterations, a clear pattern emerges:

$$\tilde{c}_2 = \frac{-1}{2 - \frac{1}{2}} = -\frac{2}{3}$$
$$\tilde{c}_3 = \frac{-1}{2 - \frac{2}{3}} = -\frac{3}{4}$$
$$\tilde{c}_4 = \frac{-1}{2 - \frac{3}{4}} = -\frac{4}{5}$$

It is apparent that $\tilde{c}_i$ behaves according to $-i/(i+1)$. However, in order for $\tilde{c}_i = -i/(i+1)$, equation (31) must be satisfied:

$$\frac{-1}{2 + \tilde{c}_{i-1}} = \frac{-1}{2 + \left(\frac{i-1}{i}\right)} = \frac{-i}{2i - i + 1} = -\frac{i}{i+1} = \tilde{c}_i$$

Hence, $\tilde{c}_i = -i/(i+1)$, which implies that to calculate (30) requires only 2 FLOPS for $(-\tilde{c}_i)$ and 2 additional FLOPS for $\tilde{f}_i$. Both $(-\tilde{c}_i)$ and $\tilde{f}_i$ require n iterations such that the taylored forward sweep requires

$$\begin{aligned} \text{FLOPS}_{\text{taylored sweep}} &= n(2+2) + 1 \\ &= 4n + 1 \end{aligned} \tag{32}$$

(The +1 stems from the computation of $\tilde{f}_1$.)

Having calculated $(-\tilde{c}_i)$ and $\tilde{f}_i$, the backward substitution is completed using the following recursion relation (the concept has already been explained in previous algorithms):

$$\tilde{y}_n = \tilde{f}_n \quad \text{and} \quad \tilde{y}_j = \tilde{f}_j + (-\tilde{c}_j)\tilde{y}_{j+1} \tag{33}$$

The number of FLOPS required for the taylored substitution is the same as in previous algorithms. In total, the number of FLOPS for completing the taylored algorithm is

$$\text{FLOPS}_{\text{taylored}} = 6n - 1 \tag{34}$$

This suggests that the taylored algorithm is about $8/6 \approx 1.34$ times faster than the generalised algorithm, and about $7/6 \approx 1.17$ times faster than the specialised algorithm. It is likely that the actual efficiency of the taylored algorithm is even greater than this due to its ability to vectorise some of its calculations (in particular $(-\tilde{c}_i)$). Among the algorithms presented in this report, the ability to vectorise some computations is unique to the taylored approach, and often may improve the accuracy significantly.

C. Comparing The Numerical Solutions

1. *Accuracy*

Seeing that the differential equation (eq.(2)) was solved analytically in section II A 2, this provides an excellent opportunity to compare the numerical solutions to the exact solution.

The relative error between the numerial and exact solution in x_i is given by

$$\epsilon_i = \left| \frac{y(x_i) - y_i}{y(x_i)} \right| \tag{35}$$

The relative error is highly dependent on the value of n, often changing by orders of magnitude. Thus instead of the regular relative error, this report will study the magnitude of the relative error with respect to the magnitude of h (i.e., $\log(h)$):

$$\varepsilon_i = \log(|\epsilon_i|) = \log\left[\left| \frac{y(x_i) - y_i}{y_i} \right| \right] \tag{36}$$

(Here, it is implied that the logarithm is with respect to base 10.)

The greater number of FLOPS an algorithm requires, the more prone the algorithm is to accumulated errors from floating point arithmetic. This error is also contained in ϵ_i, thus the various algorithms may achieve different relative errors depending on the number of grid points.

2. *Efficiency*

As already mentioned, the different algorithms require different number of FLOPS, thus performing with different efficiencies. Measuring the efficiency of an algorithm is easily done using the computer's CPU time. However, the uncertainty in this calculation depends on the CPU time granularity, and the precision of the time ticks. The granularity is easily found, but the precision of the ticks is difficult to handle. Thus, the uncertainty should be considered as an estimate of the standard deviation as opposed to a strict uncertainty.

IV. RESULTS

All programs may be found in the GitHub repository.

The numerical solutions have been computed using `solve_equation.x`, which was compiled using the following commands on a laptop running the Ubuntu 16.04.5 LST distribution:

```
[terminal]$ g++ -std=c++14 -o solve_equation.x solve_equation.cpp
tridiagmateq.cpp -l armadillo -O3
```

Plots generated using `plot_errors.py` (which is written using Python 3 syntax).

A. An overview

1. *The generalised algorithm*

Figures 1, 2 and 3 show plots of the numerical solutions provided by the generalised algorithm with $n = 10$, 100 and 1000 grid points respectively. The exact solution have been added to the plots for ease of comparing the accuracy of the computations. It is immediately obvious that this algorithm quickly approaches the exact solution rather quickly: the difference between the exact and numerial solution is minimal with only $n = 100$ grid points.

Furthermore, table I shows the average time spent during calculation. The impact from increasing n from 10 to 100 is actually much less (in proportion) to increasing n from 10 to 100. That is, there seems to be a non-linear relationship between n and time spent.

2. *The specialised and taylored algorithms in comparison to the generalised algorithm*

Figures 4 and 5 show plots of the numerical solutions provided by the specialised and taylored algorithms respectively. The solutions both used $n = 1000$ grid points in order to be compared to figure 3. Again, the exact solution has been added to the plots for ease of comparing the accuracy of the computations. As expected, seeing that these algorithms are based on the generalised algorithm, these figures are, for all intents and purposes, completely identical to figure 3.

The difference between the two algorithms lies in the time spent computing the solutions. Table II shows the average time spent computing the numerical solutions using the specialised and taylored algorithms: Surprisingly, the taylored algorithm spent more time compared to the specialised algorithm. This is unexpected as the taylored algorithm requires less flops than the specialised algorithm. Nonetheless, both the specialised and taylored algorithms spent less time than the generalised algorithm.

3. *Testing the specialised and taylored algorithms for greater n*

Table III shows how the time spent during a computation of the numerical solution evolves with increasing n for both the specialised and the taylored algorithm. There is a clear trend in the data: while the taylored algorithm is initially slower than the specialised algorithm, the rate at which the algorithms get progressively slower with increasing n is greater for the specialised algorithm than for the taylored algorithm. This is likely due to the vectorisation capabilities of the taylored algorithm. Thus with respect to computation time, choosing the most advantageous algorithm is dependent on your chosen n.

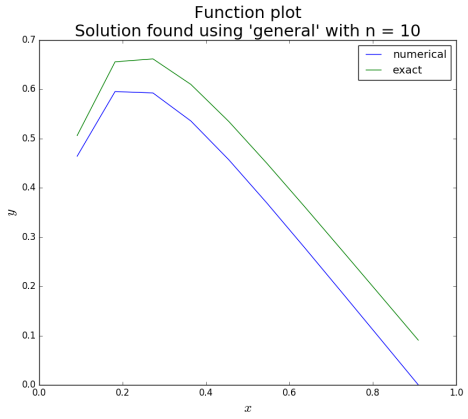

Figure 1. A plot of the numerical solution on top of the exact solution. The numerical solution was computed using the generalised algorithm with $n = 10$.

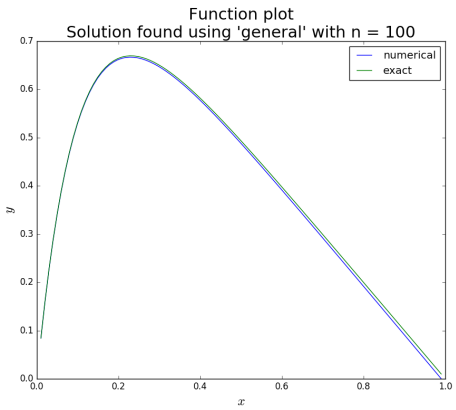

Figure 2. A plot of the numerical solution on top of the exact solution. The numerical solution was computed using the generalised algorithm with $n = 100$.

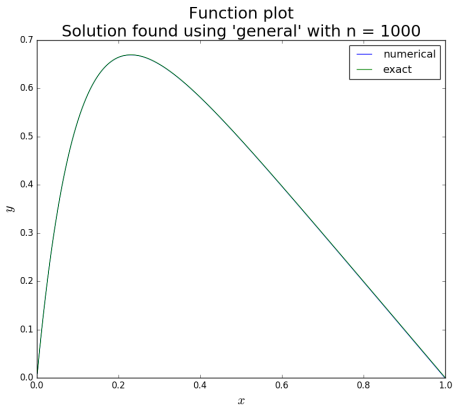

Figure 3. A plot of the numerical solution on top of the exact solution. The numerical solution was computed using the generalised algorithm with $n = 1000$.

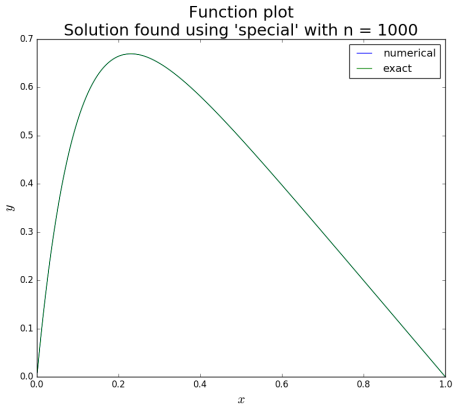

Figure 4. A plot of the numerical solution on top of the exact solution. The numerical solution was computed using the specialised algorithm with $n = 1000$.

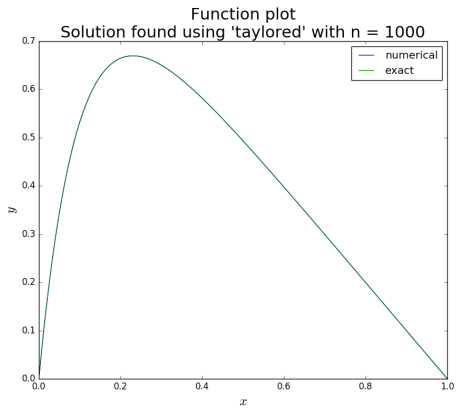

Figure 5. A plot of the numerical solution on top of the exact solution. The numerical solution was computed using the taylored algorithm with $n = 1000$.

Table I. Average time spent by the generalised algorithm during aomputation of the numerical solution.

n	time spent
10	0.010 ms
100	0.019 ms
1000	0.126 ms

Table II. Average time spent by the specialised and taylored algorithms during a computation of the numerical solution using $n = 1000$ grid points.

algorithm	time spent
specialised	0.097 ms
taylored	0.158 ms

Table III. Average time spent by the specialised and taylored algorithms during a computation of the numerical solution using $n = 10^4$, 10^5, 10^6 and 10^7 grid points.

algorithm	n	time spent
specialised	10 000	1.012 ms
specialised	100 000	8.677 ms
specialised	1 000 000	122.440 ms
specialised	10 000 000	278.338 ms
taylored	10 000	1.139 ms
taylored	100 000	14.516 ms
taylored	1 000 000	97.157 ms
taylored	10 000 000	201.197 ms

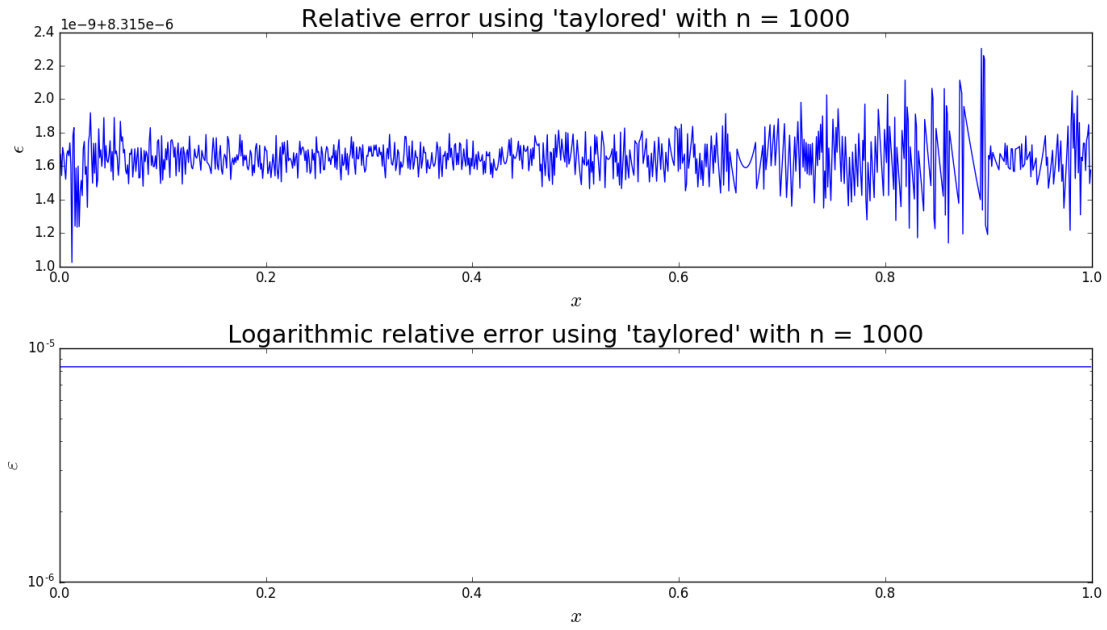

Figure 6. A plot of the relative error and logarithmic relative error in the numerical solution as function of x_i. The numerical solution was computed using the taylored algorithm with $n = 1000$.

B. Relative error

Figure 6 below shows the relative error in the numerical solution ϵ_i as a function of x_i when it is computed using the taylored algorithm with $n = 1000$ grid points. There is no clear relationship between the ϵ_i and x_i, although the error seems to be somewhat-normally distributed about $\langle\epsilon\rangle \approx 8.3166 \cdot 10^{-6}$. Plots of the relative errors from the other numerical solutions are more or less identical and have therefore been left out.

Below in figures 7 and 8, the maximum logarithmic relative errors $\varepsilon = \log(\max(\epsilon_i))$ in the specialised and taylored numerical solutions have been plotted with respect to $\log(h)$. Both algorithms show a linear relationship between ε and $\log(h)$ for $\log(h)$ values greater than -5. However, while the specialised algorithm is destabilised as $\log(h)$ decreases, the taylored algorithm remains stabilised. The behaviour of ε for the specialised algorithm becomes unpredictable for $\log(h) < -5$ whereas the behaviour of ε for the taylored algorithm is somewhat predictable.

C. LU decomposition using `armadillo`

The LU decomposition process provided excellent results, however spent much more time than the other algorithms. What's more, whereas the other algorithms only needed to store $1 \times n$ dimensional arrays, LU decompositioning needs to store $n \times n$ matrices. This process thus demands considerably more memory than the other algorithms. So much so in fact that the computer performing the computations ran out of memory when trying to decompose an $10^5 \times 10^5$ matrix.

Table IV shows the average time spent by the LU decomposition algorithm when computing the numerical solution. Overall the algorithm is very slow compared to the other algorithms, in particular, the $n = 10000$ computation is a ridiculous $\approx 2 \cdot 10^5$ times slower than the specialised and taylored algorithms. Hence, not only is the LU decomposition algorithm memory-inefficient, it also performs with extraordinarily terrible efficiency when compared to the other algorithms.

It should be noted that the numbers presented here reflect the algorithm's performance with respect to a single matrix equation. It is well-established that one of the strongest benefits of LU decomposition is its ability to study several right-hand-sides of a matrix equation in succession. This was not done in this numerical experiment.

Table IV. Average time spent by the LU decomposition algorithms during a computation of the numerical solution.

n	time spent
10	0.722 ms
100	10.750 ms
1000	575.917 ms
10000	217058.608 ms

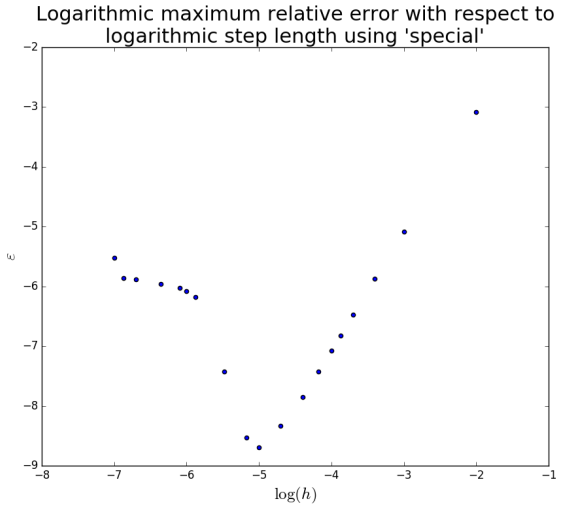

Figure 7. A plot of the maximim logarithmic relative error in the numerical solution as function of $\log(h)$. The numerical solution was computed using the special algorithm.

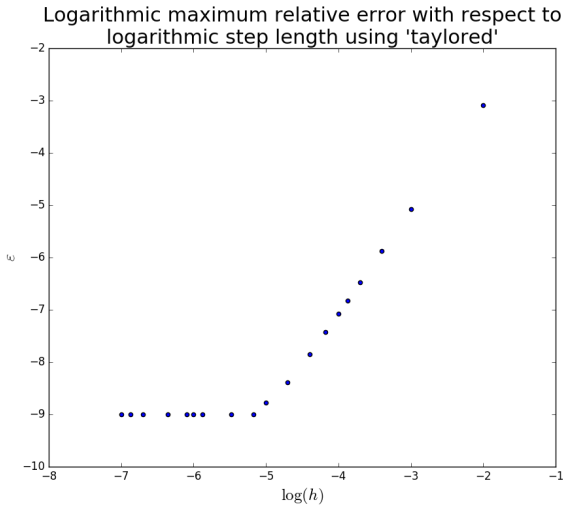

Figure 8. A plot of the maximim logarithmic relative error in the numerical solution as function of $\log(h)$. The numerical solution was computed using the taylored algorithm.

V. DISCUSSION

Not surprisingly, the algorithms designed to make fewer computations generally performed better than the more general algorithms.

The most comprehensive algorithm, the LU decomposition, is (ideally) capable of handling any matrix equation and therefore suffers the consequences of memory-heavy and inefficiency. The generalised algorithm is massively more efficient than the LU decomposition, it is however only capable of handling tridiagonal matrix equations. While seemingly a limitation, tridiagonal matrix equations are very common in applied linear algebra. Hence, an algorithm that is much more efficient than the LU decompositino is a nice addition to the numerical toolbox of any physicist.

The specialised algorithm is further simplification that improves on the efficiency of the generalised algorithm. As it is not uncommon for tridiagonal matrices to have constant diagonal entries, this algorithm is also a welcome addition to the numerical toolbox.

The final taylored algorithm actually did not improve on the efficiency of the specialised algorithm to any large extent. The specialised algorithm did grow slower than the taylored algorithm for large n (the specialised algorithm is about ≈ 1.38 times slower when $n = 10^7$), but employing the taylored algorithm really does not have any meaningful effect other than in computations with $n > 10^7$ or successive computations with $n \gtrsim 10^5$.

Furthermore, figures 7 and 8 show that the specialised algorithm performed unpredictably for very large n, while the taylored algorithm performed with unchanging accuracy. It may therefore seem benefitial to select the taylored algorithm over the specialised algorithm for very large n, however increasing n essenetially only leads to less efficiency with no added accuracy (see region $\log(h) < -5$ in figure 8). Seeing that table III suggests that the specialised algorithm performed faster than the taylored algorithm when $n = 10^5$ (i.e., when $\log(h) \cong -5$), there is no substantial benefit in choosing the taylored algorithm over the specialised algorithm.

VI. CONCLUSION

In conclusion, the algorithms all provided accurate results given a descent number of grid points n. The generality of the LU decomposition makes it memory-inefficient and slow when dealing with large matrices. The preferred choice when dealing with tridiagonal matrix equations is therefore either the generalised or the specialised algorithm depending on the matrix in question. Improving the specialised algorithm to a taylored algorithm when solving the specific matrix equation introduced in section III A is inadvisable as it does not lead to any meaningful benefit.

The specialised algorithm performed optimally with a step length of about 10^{-5} (i.e., approximately 10^5 grid points), decreasing the step length further leads to unpredictable behaviour that may be difficult to handle.

To solve the specific problem at hand (equation (14)) is satisfactorily done with a maximum relative error of about $8.3 \cdot 10^{-6}$ using $n = 10^3$ grid points and any of the four algorithms.

[1] Morten Hjorth-Jensen. Computational physics lectures: Linear algebra methods, 2018. http://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/pdf/linalg-print.pdf.

[2] Morten Hjorth-Jensen. Computational physics lectures: Ordinary differential equations, 2018. http://compphysics.github.io/ComputationalPhysics/doc/pub/ode/pdf/ode-print.pdf.

[3] Morten Hjorth-Jensen. Fys3150 computational physics - project 1, 2018. http://compphysics.github.io/ComputationalPhysics/doc/Projects/2018/Project1/pdf/Project1.pdf.