Diffusion Equation

Vegard Falmår and Sigurd Sørlie Rustad

University of Oslo
Norway

December 18, 2020

CONTENTS

The diffusion equation is used to predict the evolution of many phenomena. In this report we are working with a constiant collective diffusion coefficient, giving us a linear differential equation identical to the heat equation. We study it in both one an two dimensions. In one dimension we use explicit forward Euler, implicit backward Euler and implicit Crank-Nicolson, and in two dimensions only forward Euler.

Crank-Nicolson outperformed both algorithms in one dimension. both when looking at mean absolute percentage error (MAPE) and when looking at the pure difference between the closed form and computed result. Forward and backward Euler performed similarly when stability conditions were complied with. When they weren't, forward Euler became unstable and produced values in the order of magnitude 10^8 (expected values in the range 0-10).

For our two dimensional problem we used forward Euler as our solver. Here we also tested above and below the stability conditions. As we saw in the one dimensional case, when below stability conditions we got good results (MAPE of around 10^{-3}%). When above the condition however, the solver produced results in the order of 10^{35-116}, and MAPE in the same order.

In this report we explain ways to implement and optimize code for solving the algorithms. We did not arrive at the best algorithm, however we have discussed this extensively in the discussion-section.

I. INTRODUCTION

The diffusion equation is used in many areas. Everything from how heat is transferred through a stick, to how an oil spill will spread. In this report we will try to solve the diffusion equation for one and two dimensions, and using different algorithms. We are going to study the diffusion equation with a constant collective diffusion coefficient, giving us a linear differential equation identical to the heat equation. All central theory is covered in the theory section, and we recommend reading it even before the introduction.

For our one dimensional problem we are using dirichlet boundary conditions. We use three different algorithms, explicit Forward Euler, implicit Backward Euler and implicit Crank-Nicolson. The situation we are looking at is $x \in [0, 1]$, zero in one end and one in the other as boundary conditions. We implement the aforementioned algorithms for $N_x = 10, 100$ steps in x-direction. The Δt is decided by the stability condition of forward Euler. We try a Δt just above and below the stability condition, looking at how the algorithms differ. Since forward Euler is conditionally stable (the condition mentioned above) we expect large errors. However since backward Euler and Crank-Nicolson is unconditionally stable, they should still give sensible results. We also look at how the errors vary between the methods, plotting the closed solution above the computed values. To further investigate the performance we also plot the difference between closed and computed results, calculating mean absolute percentage error while we are at it.

In our two dimensional problem we use forward Euler as a solver, with homogeneous dirichlet boundary conditions (zero along all edges). Similarly to the one dimensional problem we also use $x, y \in [0, 1]$ and the same number of grid points in x- and y-direction ($N_x = N_y = 50$). The Δt is again decided by the stability conditions, running for Δt just above and below the condition. We then look at how the results compare with the closed solution. Like we did in the one dimensional we plot the difference as a plane, and calculate the mean percentage error. Again we expect worse results when the stability condition is broken, however it is interesting to see how much.

For those interested we have included our methods of implementation, as well as discussion. For our studies we have used c++ for heavy computation, python for visualization and automation. All the code along with instructions on how to run it, can be cloned from our GitHub repository[1]. In our code we have done extensive unit-tests in order to make sure our code is running correctly. The tests performed are covered in the method section under Unit-tests. We also parallelized our two dimensional solver. How this is done is also covered in the method section (under Parallelization), as well as extensive discussion on optimization of our parallelization in the discussion-section.

II. THEORY

The diffusion equation

The full diffusion equation reads

$$\frac{\partial u(\mathbf{r}, t)}{\partial t} = \nabla \cdot [D(u, \mathbf{r}) \nabla u(\mathbf{r}, t)],$$

where $\mathbf{r}$ is a positional vector and $D(u, r)$ the collective diffusion coefficient. If $D(u, \mathbf{r}) = 1$ the equation simplifies to a linear differential equation

$$\frac{\partial u}{\partial t} = \nabla^2 u(\mathbf{r}, t),$$

or

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) u(x, y, z, t) = \frac{\partial u(x, y, z, t)}{\partial t} \tag{1}$$

in cartesian coordinates. In this report we are mainly going to work with the diffusion equation in one and two

[1] github.com/sigurdru/FYS3150/tree/master/Project5

dimensions, i.e.

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}$$
$$\wedge$$
$$\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} = \frac{\partial u(x,y,t)}{\partial t}.$$

Mean absolute percentage error

The mean absolute percentage error (MAPE), is used to predict the accuracy of a prediction. We will use it to measure how good our numerical methods are, compared to theoretical values. Equation (2) shows how to calculate MAPE in percentage,

$$M = \left(\frac{1}{n} \sum_{n=1}^{n} \left| \frac{A_\mathrm{t} - A_\mathrm{c}}{A_\mathrm{t}} \right| \right) \cdot 100\%. \tag{2}$$

Here A_t and A_c are the theoretical and computed values, n number of data points and M MAPE in percent.

Discretization

Equation (1) in one dimension reads

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \quad \text{or} \quad u_{xx} = u_t. \tag{3}$$

With $x \in [0, L]$ and boundary conditions

$$u(0,t) = a(t), \quad t \geq 0 \quad \wedge \quad u(L,t) = b(t), \quad t \geq 0,$$

we can approximate the solution by discretization. First introducing $\Delta x = L/(n+1)$ and Δt as small steps in x-direction and time. Then we can define the value domain of t and x,

$$t_j = j\Delta t, \quad j \in \mathbb{N}_0 \quad \wedge \quad x_i = i\Delta x, \quad \{i \in \mathbb{N}_0 | i \leq n+1\}.$$

Explicit and implicit schemes

It is common to divide numerical algorithms into explicit and implicit schemes. When performing numerical integration, we iterate over a discrete set of grid points at which we evaluate the function in question. In explicit schemes, the value at the next grid point is determined entirely by known or previously calculated values. In implicit schemes, the value is determined by solving a coupled set of equations, often involving matrix or iterative techniques.

Using an implicit method instead of an explicit method usually requires more computation in every step, and they are often harder to implement. It can, in turn, save computation by allowing larger step sizes. Explicit methods are always conditionally stable, however the implicit methods we will use in this report are unconditionally stable. Of course, in order to achieve a desired *accuracy*, the step sizes can not be arbitrarily large for either the explicit or implicit schemes. It is generally the case, though, that implicit schemes allow for larger step sizes than explicit schemes.

Explicit Forward Euler

The algorithm for explicit forward Euler in one dimension (from [1] chapter 10.2.1) reads

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j} \tag{4}$$

where

$$\alpha = \frac{\Delta t}{\Delta x^2},$$

and a local approximate error of $O(\Delta t)$ and $O(\Delta x^2)$. The discretization is explained in the appropriate section. Note that the expression on the right hand side, used to calculate the value at a time $t_j + \Delta t$, only contains the state of the system at time t_j. This can be written as a matrix equation (see [1] chapter 10.2.1)

$$\mathbf{u}_{j+1} = (\mathbb{1} - \alpha \mathbf{B})\mathbf{u}_j$$

where $\mathbb{1}$ is the identity matrix and

$$\begin{bmatrix} 2 & -1 & 0 & 0\dots \\ -1 & 2 & -1 & 0\dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & -1 & 2 \end{bmatrix} \quad \wedge \quad \mathbf{u}_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \dots \\ u_{n,j} \end{bmatrix}$$

The stability requirement for this algorithm (also from [1] chapter 10.2.1) is

$$\rho(\mathbf{B}) \leq 1 \implies \frac{\Delta t}{(\Delta x)^2} \leq 1/2 \tag{5}$$

Where $\rho(\mathbf{B})$ is the spectral radius of $\mathbf{B}$:

$$\rho(\mathbf{B}) = \max\{|\lambda| : \det(\mathbf{B} - \lambda \mathbb{1}) = 0\}.$$

In two dimensions ($u(x,y)$) the method for solving is similar. Assuming the same number of integration points in x- and y-direction ($\Delta x = \Delta y = \Delta l$), the algorithm (from [1] chapter 10.2.5) reads

$$u_{i,j}^{l+1} = u_{i,j}^{l} + \alpha \left[u_{i+1,j}^{l} + u_{i-1,j}^{l} + u_{i,j+1}^{l} + u_{i,j-1}^{l} - 4u_{i,j}^{l} \right] \tag{6}$$

where $u_{i,j}^{l} = u(i\Delta l, j\Delta l, l\Delta t)$ and $\alpha = \Delta t/\Delta l^2$. With this algorithm we will get slightly different stability conditions. The Von Neumann stability analysis gives

$$\frac{\Delta t}{\Delta l^2} \leq \frac{1}{4} \tag{7}$$

see [3] for derivation.

Implicit Backward Euler

The Backward Euler algorithm uses the same centered difference in space as Forward Euler to approximate the second derivative

$$u_{xx} \approx \frac{u(x_i+\Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j))}{\Delta x^2}, \tag{8}$$

but a backward formula for the time derivative:

$$u_t \approx \frac{u(x_i, t_j) - u(x_i, t_j - \Delta t)}{\Delta t}, \tag{9}$$

which also has a local truncation error of $O(\Delta t)$ and $O(\Delta x^2)$. Again, defining

$$\alpha = \frac{\Delta t}{(\Delta x)^2}$$

we obtain by inserting (8) and (9) into our differential equation the equation describing Backward Euler:

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1+2\alpha)u_{i,j} - \alpha u_{i+1,j} \tag{10}$$

Written out for all i, equation 10 becomes

$$\begin{aligned} -\alpha u_{0,j} + (1+2\alpha)u_{1,j} - \alpha u_{2,j} &= u_{1,j-1} \\ -\alpha u_{1,j} + (1+2\alpha)u_{2,j} - \alpha u_{3,j} &= u_{2,j-1} \\ &\cdots \\ -\alpha u_{n-3,j} + (1+2\alpha)u_{n-2,j} - \alpha u_{n-1,j} &= u_{n-2,j-1} \\ -\alpha u_{n-2,j} + (1+2\alpha)u_{n-1,j} - \alpha u_{n,j} &= u_{n-1,j-1} \end{aligned}$$

In general, this can be rearranged slightly so that

$$\begin{aligned} (1+2\alpha)u_{1,j} - \alpha u_{2,j} &= u_{1,j-1} + \alpha u_{0,j} \\ -\alpha u_{1,j} + (1+2\alpha)u_{2,j} - \alpha u_{3,j} &= u_{2,j-1} \\ &\cdots \\ -\alpha u_{n-3,j} + (1+2\alpha)u_{n-2,j} - \alpha u_{n-1,j} &= u_{n-2,j-1} \\ -\alpha u_{n-2,j} + (1+2\alpha)u_{n-1,j} &= u_{n-1,j-1} + \alpha u_{n,j} \end{aligned}$$

Let $\mathbf{v}_j$ be a vector containing the values of u at $n-1$ points in space at a time t_j

$$\mathbf{v}_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \vdots \\ u_{n-2,j} \\ u_{n-1,j} \end{bmatrix} \tag{11}$$

and the vector $\mathbf{b}_j$ be defined as follows:

$$\mathbf{b}_j = \begin{bmatrix} u_{1,j-1} + \alpha u_{0,j} \\ u_{2,j-1} \\ \vdots \\ u_{n-2,j-1} \\ u_{n-1,j-1} + \alpha u_{n,j} \end{bmatrix} \tag{12}$$

As the boundary conditions $u_{0,j}$ and $u_{n,j}$ are specified, we already know every component of $\mathbf{b}_j$. We can then rewrite equation (10) as the matrix equation

$$\mathbf{A}\mathbf{v}_j = \mathbf{b}_j \tag{13}$$

where $\mathbf{A}$ is defined as

$$A = \begin{bmatrix} (1+2\alpha) & -\alpha & 0 & 0 & \dots & 0 \\ -\alpha & (1+2\alpha) & -\alpha & 0 & \dots & 0 \\ \vdots & & \ddots & & & \vdots \\ 0 & \dots & 0 & -\alpha & (1+2\alpha) & -\alpha \\ 0 & \dots & 0 & 0 & -\alpha & (1+2\alpha) \end{bmatrix} \tag{14}$$

This is a tridiagonal matrix with $(1+2\alpha)$ on the diagonal and $-\alpha$ directly above and below the diagonal.

Implicit Crank-Nicolson

The Crank-Nicolson algorithm uses a time-centered scheme centered around $t+\Delta t/2$, with a truncation error of $O(\Delta t^2)$ and $O(\Delta x^2)$. The time derivative is given by

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t} \tag{15}$$

$$\begin{aligned} u_{xx} \approx \frac{1}{2\Delta x^2}\Big(& u(x_i+\Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j) \\ & + u(x_i + \Delta x, t_j + \Delta t) - 2u(x_i, t_j + \Delta t) \\ & + u(x_i - \Delta x, t_j + \Delta t)\Big) \end{aligned}$$

Inserting these two equations into our differential equation, we obtain (see Appendix, section VII A for derivation) that the equation describing the Crank-Nicolson algorithm can be written as a matrix equation

$$\mathbf{A}\mathbf{v}_j = \mathbf{b}_j \tag{16}$$

The vector $\mathbf{v}_j$ is the same as in the case of Backward Euler, defined in (11). The matrix $\mathbf{A}$ is a tridiagonal matrix with $2(1+\alpha)$ on the diagonal and $-\alpha$ directly above and below the diagonal. The vector $\mathbf{b}_j$ is

$$\mathbf{b}_j = \begin{bmatrix} \alpha u_{0,j-1} + 2(1-\alpha)u_{1,j-1} + \alpha u_{2,j-1} + \alpha u_{0,j} \\ \alpha u_{1,j-1} + 2(1-\alpha)u_{2,j-1} + \alpha u_{3,j-1} \\ \vdots \\ \alpha u_{n-3,j-1} + 2(1-\alpha)u_{n-2,j-1} + \alpha u_{n-1,j-1} \\ \alpha u_{n-2,j-1} + 2(1-\alpha)u_{n-1,j-1} + \alpha u_{n,j-1} + \alpha u_{n,j} \end{bmatrix} \tag{17}$$

III. METHODS

A. One dimension

We will start by solving the one dimensional diffusion equation

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, \quad x \in [0,1]$$

with initial conditions

$$u(x,0) = 0, \quad 0 < x < 1 \tag{18}$$

and boundary conditions

$$\begin{aligned} u(0,t) &= 0, \quad t \geq 0 \quad \text{and} \\ u(L,t) &= 1, \quad t \geq 0. \end{aligned} \tag{19}$$

As mentioned we use three methods, explicit forward Euler, implicit backward Euler and Crank-Nicolson. When using Explicit forward Euler, we only need to solve the difference-equation described by equation (4). When using Implicit backward Euler er Crank-Nicolson we need to solve the matrix-equations (13) and (16). There are several methods for solving such tridiagonal matrix equations, however we will use the method covered in a previous project. see project[2] page 2 (section A. Solve tridiagonal matrix equation). We will use $\Delta x = 1/10$, $\Delta x = 1/100$ and Δt decided by equation (5). We then study compare our results with analytical ones (analytical results are derived below), at two points in time. First t_1 where our solution is smooth but curved, then t_2 where we have a linear (stable) solution. Hopefully we will be able to decide on what method is best. We also check what happens when the difference in equation (5) is not satisfied. For all our results we also plot the difference between the results, as well as calculate the MAPE in percent (see equation (2)).

Analytic solution to the 1D diffusion equation

With the initial and boundary conditions described by equations (18) and (19) we can find an analytical solution. We expect the heat distribution to converge to a stable state u_E as $t \to \infty$:

$$\lim_{t\to\infty} u(x,t) = u_E(x)$$

This final state should still satisfy our differential equation such that

$$\frac{\partial^2 u_E(x)}{\partial x^2} = \frac{\partial u_E(x)}{\partial t} = 0$$

[2] https://github.com/sigurdru/FYS3150/tree/master/Project1

The solution to this equation is

$$u_E(x) = Ax + B,$$

and the boundary conditions give

$$\begin{aligned} u_E(0) &= B = 0 \\ u_E(L) &= AL = 1 \quad \Rightarrow \quad u_E(x) = \frac{x}{L} \end{aligned}$$

Let us now define the function

$$\begin{aligned} v(x,t) &= u(x,t) - u_E(x) \quad \text{which gives} \\ u(x,t) &= v(x,t) + u_E(x) \end{aligned}$$

We then have

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial v}{\partial t} + \frac{\partial u_E}{\partial t} = \frac{\partial v}{\partial t} \\ \frac{\partial^2 u}{\partial x^2} &= \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 u_E}{\partial x^2} = \frac{\partial^2 v}{\partial x^2} \end{aligned}$$

Thus, $u(x,t)$ and $v(x,t)$ should both satisfy the same differential equation (3). The intial and boundary conditions for $v(x,t)$ are

$$\begin{aligned} v(x,0) &= u(x,0) - u_E(x) = -u_E(x) \\ v(0,t) &= u(0,t) - u_E(0) = 0 \\ v(0,t) &= u(L,t) - u_E(L) = 0 \end{aligned}$$

In other words, v obeys the same equation as u, but with boundary conditions 0.

It is rather straightforward to see that (however see [4] chapter 3 for a more thorough derivation)

$$f_n(x,t) = e^{-C_n^2 t} \left(A \sin(C_n x) + B \cos(C_n x)\right) \tag{20}$$

is a particular solution to the one dimensional diffusion equation:

$$\frac{\partial f_n(x,t)}{\partial t} = \frac{\partial^2 f_n(x,t)}{\partial x^2} = -C_n^2 f_n(x,t)$$

The equation is linear and from the prinsiple of superposition we have that any sum of functions $f_n(x,t)$ with different values of C_n is also a solution. The boundary conditions $v(0,t) = 0$ and $v(L,t) = 0$ give

$$\begin{aligned} f_n(0,t) &= B_n e^{-C_n^2 t} = 0 \quad \Rightarrow \quad B_n = 0 \\ f_n(L,t) &= A_n e^{-C_n^2 t} \sin(C_n L) = 0 \quad \Rightarrow \quad C_n = \frac{n\pi}{L}, n \in \mathbb{Z} \end{aligned}$$

The solution $v(x,t)$ can then be written as sum

$$v(x,t) = \sum_n A_n e^{-\left(\frac{n\pi}{L}\right)^2 t} \sin\left(\frac{n\pi}{L} x\right)$$

The coefficients A_n are given by the initial condition $v(x,0) = -u_E(x)$ and can be determined from (see [1] chapter 10.2.4)

$$\begin{aligned} A_n &= \frac{2}{L} \int_0^L -u_E(x) \, \sin\left(\frac{n\pi}{L} x\right) \, \mathrm{d}x \\ &= -\frac{2}{L^2} \int_0^L x \, \sin\left(\frac{n\pi}{L} x\right) \, \mathrm{d}x \end{aligned}$$

The solution to this is

$$A_n = \frac{2\left(\pi n\cos(\pi n) - \sin(\pi n)\right)}{\pi^2 n^2} \tag{21}$$

The complete solution to our differential equation is thus

$$u(x,t) = \frac{x}{L} + \sum_{n=1}^{\infty} A_n e^{-\left(\frac{n\pi}{L}\right)^2 t} \sin\Big(\frac{n\pi}{L}x\Big), \tag{22}$$

where the coefficients A_n are given by equation (21).

B. Two dimensions

Moving on to two dimensions we use explicit forward Euler. This means we need to solve the difference equation (6), for $x, y \in [0,1]$. We will use homogeneous Dirichlet boundary conditions

$$\begin{aligned} &u(0,y,t) = u(1,y,t) = 0, \quad 0 \le y \le 1, \quad t \ge 0 \text{ and} \\ &u(x,0,t) = u(x,1,t) = 0, \quad 0 \le x \le 1, \quad t \ge . \end{aligned} \tag{23}$$

and initial conditions given by

$$u(x,y,0) = \begin{cases} 1 & \text{if } 0 < y \le \frac{1}{2} \text{ and } 0 < x < 1 \\ 0 & \text{if } \frac{1}{2} < y < 1 \text{ and } 0 < x < 1 \end{cases} \tag{24}$$

As for the one we will test for different Δt's, and compare it to the closed form solution covered bellow. We do one case where the difference in equation (7) is satisfied and one where it's not. We then plot the result and difference between them, also calculating MAPE in percent (equation (2)). For all cases we used $\Delta x = \Delta y = 1/50$.

Parallelization

When solving the difference equation (6) we parallelize the program using OpenMp. While calculating $u^l_{i,j}$ we only need to worry about the surrounding points for the previous step, namely $u^{l-1}_{i,j}$ and $u^{l-1}_{i\pm1,j\pm1}$. In a grid pattern this would look something like:

$$\begin{array}{ccc} & u^{l-1}_{i,j+1} & \\ u^{l-1}_{i-1,j} & u^{l-1}_{i,j} & u^{l-1}_{i+1,j} \\ & u^{l-1}_{i,j-1} & \end{array}$$

This means that whenever we are done calculating $u^l_{i,j}$ and $u^l_{i\pm1,j\pm1}$ we can start calculating the next time step $u^{l+1}_{i,j}$. We found that if we run through $u^l_{i,j}$ diagonally, we can start calculating the next time step ($u^{l+1}_{i,j}$) one diagonal behind. A visual representation of this can be seen in figure 1. The red and black dots represent values of $u^l_{i,j}$ we have and haven't calculated respectively. We can then make sure one thread is calculating $u^l_{i,j}$ for one value of l diagonally, another core the next time step one diagonal behind, and so forth. We will be timing our code for a selected few runs to determine the effect of parallelization.

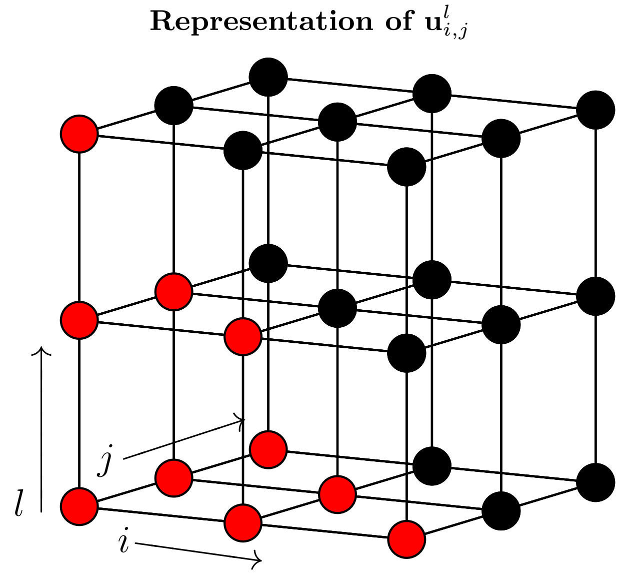

Figure 1. Visualization of how the parallelization works. i, j and l represent the indexes of $u^l_{i,j}$ and the red and black dots represent the values of $u^l_{i,j}$ we have and haven't calculated.

Analytical solution to the 2D diffusion equation

with boundary and initial conditions described by equations (23) and (24) we can find analytical solutions to the diffusion equation. Lets say $x \in [0,a]$ and $y \in [0,b]$, with homogeneous Dirichlet boundary and initial conditions defined by some function $f(x,y)$. Then, from [2], the general solution is given by equation (25).

$$u(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} A_{mn} \sin(\mu_m x)\sin(\nu_n y) e^{-\lambda^2_{mn} t}, \tag{25}$$

where

$$\mu_m = \frac{m\pi}{a} \quad \wedge \quad \nu_n = \frac{n\pi}{b} \quad \wedge \quad \lambda_{mn} = \sqrt{\mu_m^2 + \nu_n^2},$$

for $m, n, \in \mathbb{N}$ and

$$A_{mn} = \frac{4}{ab}\int_0^a \int_0^b dydx f(x,y)\sin\Big(\frac{m\pi}{a}x\Big)\sin\Big(\frac{n\pi}{b}\Big).$$

Inserting our initial conditions (24) and $a = b = 1$ into the equation above, we get the coefficients

$$\begin{aligned} A_{mn} &= 4\int_0^1\int_0^1 dydxu(x,y,0)\sin(m\pi x)\sin(n\pi y) \\ &= 4\int_0^1 \sin(m\pi x)dx \int_0^{1/2}\sin(n\pi y)dy \\ &= 4\left[-\frac{\cos(m\pi x)}{m\pi}\right]_0^1\left[-\frac{\cos(n\pi y)}{n\pi}\right]_0^{1/2} \\ &= 4\left(-\frac{(-1)^m-1}{m\pi}\right)\left(-\frac{\cos\left(\frac{n\pi}{2}\right)-1}{n\pi}\right) \\ &= \frac{4}{\pi^2}\left(\frac{\left((-1)^m-1\right)\left(\cos\left(\frac{n\pi}{2}\right)-1\right)}{mn}\right) \end{aligned}$$

Inserting this into equation (25)

$$\begin{aligned} u(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty}\frac{4}{\pi^2}\left(\frac{\left((-1)^m-1\right)\left(\cos\left(\frac{n\pi}{2}\right)-1\right)}{mn}\right)\cdot \\ \sin(m\pi x)\sin(n\pi y)\exp\left(-((m\pi)^2+(n\pi)^2)t\right) \end{aligned} \tag{26}$$

C. Unit testing

There are many moving parts in the code we use to produce the results. In order to make sure that our implementation is correct and to help track down potential errors, we have developed specific tests for the different parts of the code. To facilitate testing we have structured our code in distinct units with separate classes for the tridiagonal matrix solver, each of the three different one-dimensional solvers, and the two-dimensional solver. We use Catch2[3] to assert that the computed results match the expected and Valgrind[4] to verify that our implementation does not suffer from memory leakage.

Tridiagonal matrix solver

This class solves a matrix equation $\mathbf{Au} = \mathbf{b}$ for $\mathbf{u}$ when $\mathbf{A}$ is a tridiagonal matrix. The test case we have used is

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 6 \end{bmatrix}$$

which has the analytic solution

$$\mathbf{u} = [6,\ 11,\ 14,\ 14,\ 10]^{\mathrm{T}}$$

[3] https://github.com/catchorg/Catch2
[4] https://valgrind.org/

Explicit Forward Euler

Equation (4) describes how the vector $\mathbf{u}$ changes from one time step to the next with Forward Euler. The test case we have used for this class is that with $\alpha = 2$

$$\mathbf{u}_j = [0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 3,\ 2,\ 1,\ 4,\ 6]^{\mathrm{T}}$$

should give

$$\mathbf{u}_{j+1} = [1,\ 1,\ 2,\ 3,\ 4,\ -1,\ 5,\ 2,\ 9,\ 2,\ 2]^{\mathrm{T}}$$

in the next time step.

Implicit Backward Euler

Equation (10) describes how the vector $\mathbf{u}$ changes from one time step to the next with Backward Euler. Here, the test verifies that with $\alpha = 2$

$$\mathbf{u}_j = [1,\ 14,\ -10,\ 18,\ 4,\ 2]^{\mathrm{T}}$$

gives

$$\mathbf{u}_{j+1} = [1,\ 4,\ 2,\ 6,\ 4,\ 2]^{\mathrm{T}}$$

in the next time step

Implicit Crank-Nicolson

Using equation (27), which describes the Crank-Nicolson algorithm, we have developed the test case that with $\alpha = 2$, the initial state

$$\mathbf{u}_j = [1,\ 6,\ 14,\ 4,\ 2,\ 2]^{\mathrm{T}}$$

should give

$$\mathbf{u}_{j+1} = [1,\ 4,\ 2,\ 6,\ 4,\ 2]^{\mathrm{T}}$$

in the next time step.

Two dimensions

In the two-dimensional case, knowing the state $\mathbf{u}^j$ in one time step, equation (6) directly gives the state $\mathbf{u}^{j+1}$ in the next time step. Starting from a state

$$\mathbf{u}^j = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

we have solved the equation for two time steps with $\alpha = 2$, giving us the following:

$$\mathbf{u}^{j+1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 4 & -5 & 0 \\ 0 & -2 & 5 & -8 & 0 \\ 0 & -25 & -14 & -35 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{u}^{j+2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & -31 & -18 & 27 & 0 \\ 0 & -16 & -75 & -14 & 0 \\ 0 & 143 & -12 & 201 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We run the 2D-solver using both one and two cores and verify that the computed results match the expected.

IV. RESULTS

For all results in this section, we have used 200 addends in the Fourier sum in equation (22) for the analytic solution.

Figure 2 shows $u(x,t)$ as a function of x calculated using the Forward Euler algorithm for a few select time steps. The dashed lines show the analytic solution. The top two figures show the solution with $N_x = 10$, that is, eleven grid points with a distance $\Delta x = 1/10$ between. The bottom figure shows $N_x = 100$, giving $\Delta x = 1/100$. The first and third figure are produced with the stability condition satisfied ($\Delta t/\Delta x^2 = 0.4$), whereas the second has $\Delta t/\Delta x^2 = 0.6$.

Figure 3 shows the error for different number of integration points and values of Δt. Since the error for $\Delta x = 1/10$ and $\Delta t/\Delta x^2 = 0.6$ is obviously very large (see the second plot in figure 2), we have chosen to include here the errors for $\Delta t/\Delta x^2 = 0.4$ (the first and third plot) and $\Delta t/\Delta x^2 = 0.5$ (the second plot). The first two plots have $N_x = 10$, i.e. eleven integration points, and the third plot has $N_x = 100$. Different colors correspond to different times. The mean absolute percentage error (MAPE) can be read from the title for the different times.

Figure 4 shows the difference between the computed values $u_c(x,t)$ and the theoretical values $u_t(x,t)$ as a function of x calculated using the Backward Euler algorithm. The first plot is made with $N_x = 10$ and thus $\Delta x = 1/10$. For the second and third plot, we have $N_x = 100$ and $\Delta x = 1/100$. For the first two plots we have used a value of Δt such that $\Delta t/\Delta x^2 = 0.4$, i.e. the stability conditions for the explicit scheme were satisfied. In the third plot, we used a value of Δt such that $\Delta t/\Delta x^2 = 0.6$ and the stability condition was not satisfied. Different colors correspond to different times. The mean absolute percentage error (MAPE) can be read from the title for the different times. Figure 5 shows the same for the Crank-Nicolson scheme.

Plots showing the actual values of $u(x,t)$ as a function of x are not included in the report as they are visually indistinguishable from the last plot in figure 2. The can be viewed in our GitHub repository previously linked to, and will be automatically produced by the code.

In figure 6 we have plotted the analytical and computed results for stable and unstable conditions (equation (7) satisfied and not). The top four plots are the stable solutions, at four different times. The bottom four are the unstable results at the same times. Above all of them we have plotted the analytical solution with a black mesh plot. To produce the analytical results we have used equation (26) with $n = m = 50$ (larger values did not yield better results). Note that the unstable result has very large oscillations.

Figure 7 shows the difference between the analytical and stable computed result. We also made a table I that shows MAPE (calculated using equation (2)) for the stable and unstable solution, at the different time steps displayed in figure 6. There was no point plotting the difference between unstable and theoretical solution, because it look identical to the unstable results in figure 6.

Unstable		Stable	
MAPE [%]	Time	MAPE [%]	Time
0.039	0.000	0.039	0.000
3.95e+34	0.033	0.0049	0.033
1.27e+74	0.067	0.0026	0.067
9.38e+115	0.100	0.0013	0.100

Table I. In this table we have calculated the MAPE (in percent) for the stable and unstable computed values, and for different time steps. The first two columns are the stable values, and the second two unstable ones.

V. DISCUSSION

From figure 2 we see very clearly the importance of satisfying the stability condition for the explicit Forward Euler scheme. The first and third plots show the function changing rapidly in the initial stage and then converging more and more slowly to a stationary state (which is a linear function of x), exactly as we expected. The second plot shows disastrous oscillations that leave the computed solution entirely useless. The function values oscillate between $\pm 6 \cdot 10^8$, when we know for certain that the correct solution should always be in the range $[0, 1]$. This was caused by simply increasing the value of Δt by 50% such that $\Delta t/\Delta x^2$ changes from 0.4 to 0.6. It is therefore very important to be aware of and consider the stability condition when using the Forward Euler algorithm.

We see from figure 3 that the error is reduced by approximately a factor of 100 when increasing the number of integration steps from 10 to 100, while also adjusting Δt such that $\Delta t/\Delta x^2 = 0.4$ in both cases. This also seems to hold true for the mean absolute percentage error. The bottom plot, where $\Delta t/\Delta x^2 = 0.5$ just satisfies the stability condition, we already see the start of the oscillations so clearly visible in figure 2. The oscillations cause mean absolute percentage error to be over twice as large for $t = 0.083$ (0.0350% vs. 0.0155%). We have verified that the error is not caused by too few addends in the Fourier sum by increasing the number from 200 to 400. To the degree of precision with which we have presented our results, the values for the errors were identical. From the stability condition, one might presume that $\Delta t/\Delta x^2 = 0.5$ would be a safe choice. Indeed, the results are nowhere near as bad as with $\Delta t/\Delta x^2 = 0.6$,

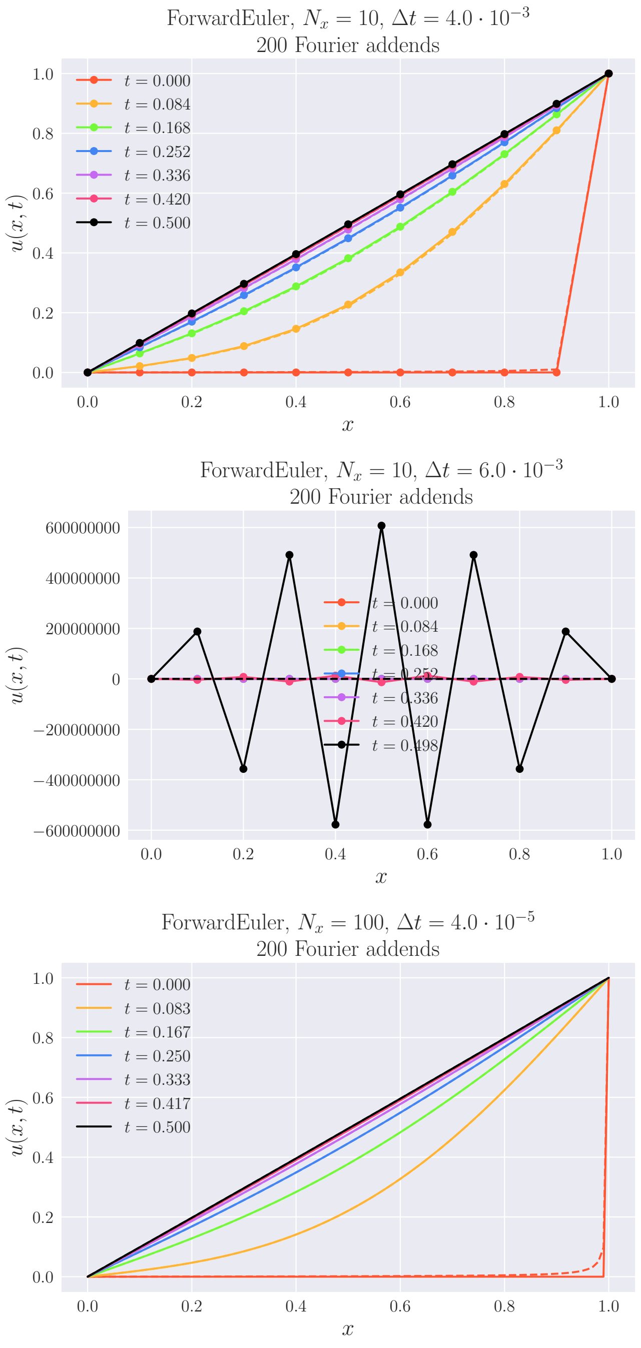

Figure 2. $u(x,t)$ as a function of x calculated using the Forward Euler algorithm. The dashed lines show the analytical results, calculated with 200 addends in the Fourier sum in equation (22). The different colors denote the solution at different times. The top two figures show the solution with $N_x = 10$, that is, eleven grid points with a distance $\Delta x = 1/10$ between. The bottom figure shows $N_x = 100$, giving $\Delta x = 1/100$. The first and third figure are produced with the stability condition satisfied ($\Delta t/\Delta x^2 = 0.4$), whereas the second has $\Delta t/\Delta x^2 = 0.6$.

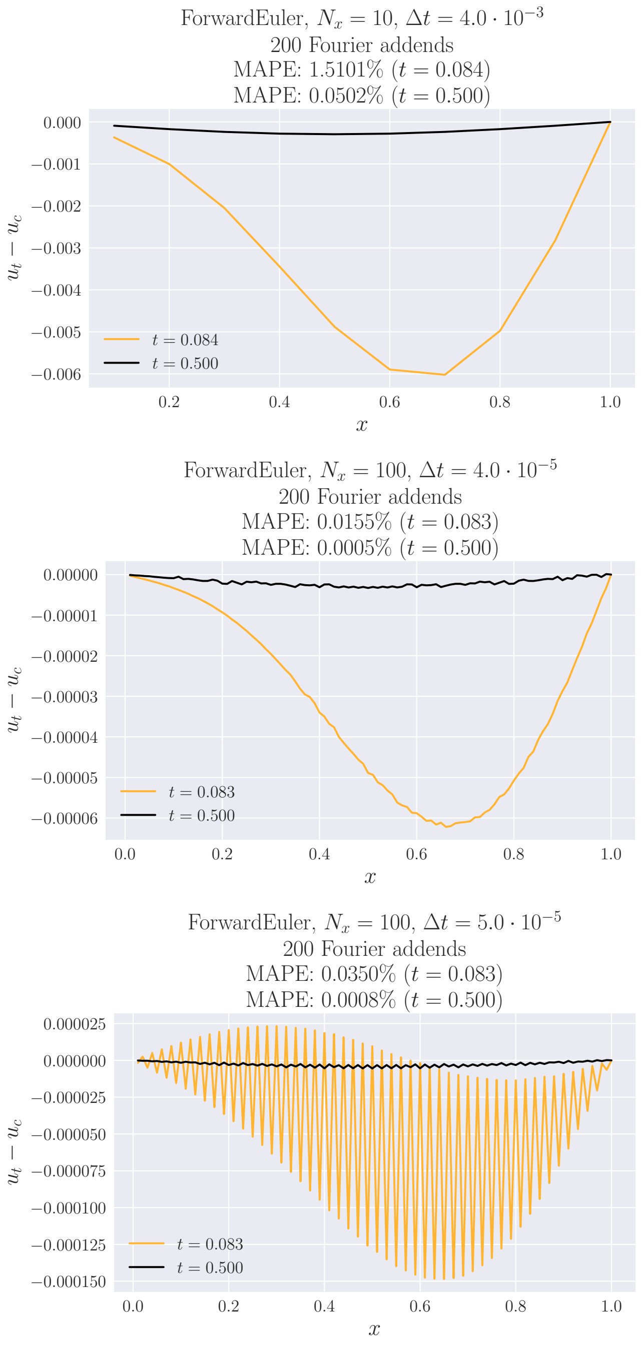

Figure 3. The error for different number of integration points and values of Δt for the Forward Euler algorithm. Different colors correspond to different times. The mean absolute percentage error (MAPE) can be read from the title for the different times. The first and third plot have $\Delta t/\Delta x^2 = 0.4$ and the second has $\Delta t/\Delta x^2 = 0.5$. The first two plots have $N_x = 10$, i.e. eleven integration points, and the third plot has $N_x = 100$.

but we do see the a tendency of the same problem with oscillations. It seems therefore that the wise approach is to make sure that you strictly satisfy the stability condition ($\Delta t/\Delta x^2 < 0.5$) in order to be confident in your results.

The results produced by the Backward Euler and

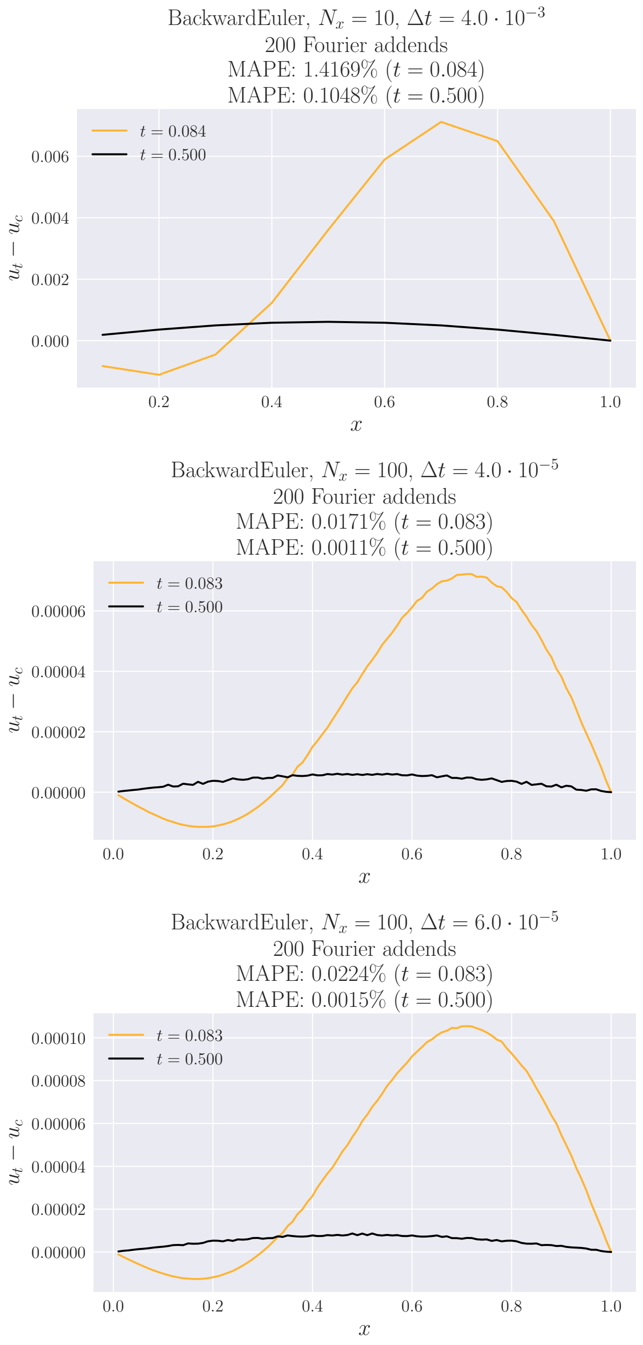

Figure 4. The difference between the computed values $u_c(x,t)$ and the theoretical values $u_t(x,t)$ as a function of x calculated using the Backward Euler algorithm. The first plot is made with $N_x = 10$ and thus $\Delta x = 1/10$. For the second and third plot, we have $N_x = 100$ and $\Delta x = 1/100$. For the first two plots we have used a value of Δt such that $\Delta t/\Delta x^2 = 0.4$. In the third plot, we used a value of Δt such that $\Delta t/\Delta x^2 = 0.6$. Different colors correspond to different times. The mean absolute percentage error (MAPE) can be read from the title for the different times.

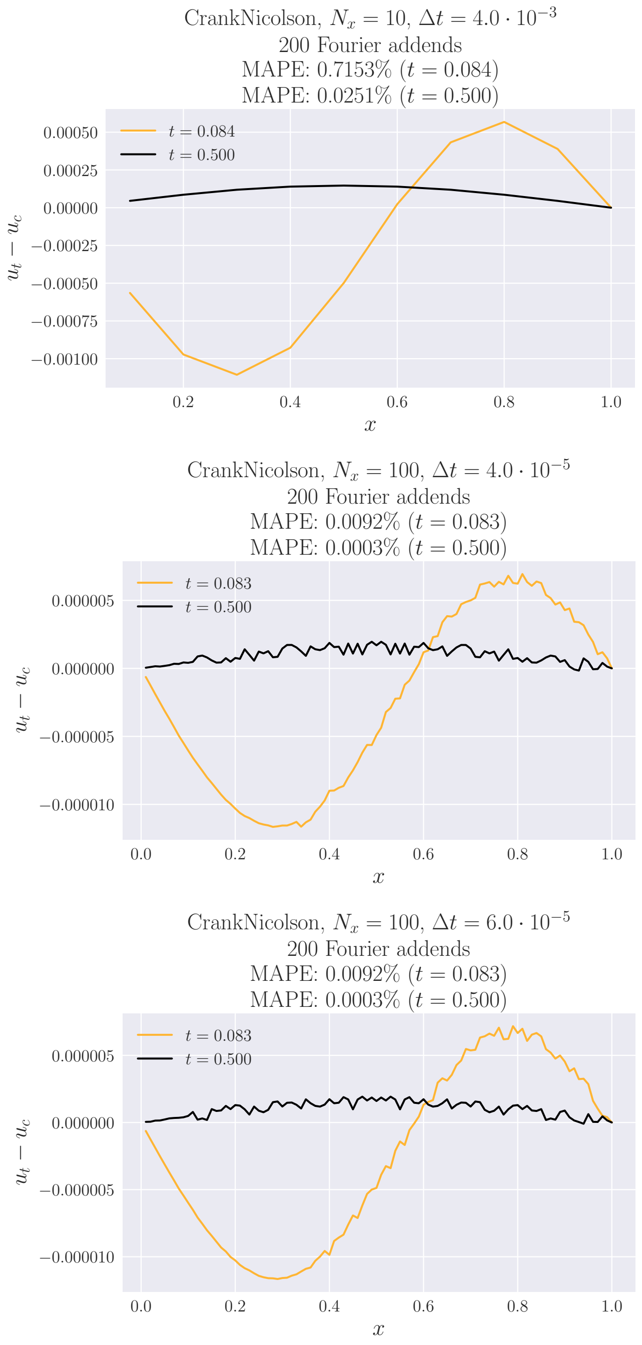

Figure 5. The difference between the computed values $u_c(x,t)$ and the theoretical values $u_t(x,t)$ as a function of x calculated using the Crank-Nicolson algorithm. The first plot is made with $N_x = 10$ and thus $\Delta x = 1/10$. For the second and third plot, we have $N_x = 100$ and $\Delta x = 1/100$. For the first two plots we have used a value of Δt such that $\Delta t/\Delta x^2 = 0.4$. In the third plot, we used a value of Δt such that $\Delta t/\Delta x^2 = 0.6$. Different colors correspond to different times. The mean absolute percentage error (MAPE) can be read from the title for the different times.

Crank-Nicolson schemes were not affected by the value of $\Delta t/\Delta x^2$ in the same way as the Forward Euler algorithm. From figures 4 and 5 we see that all the values for the error is decreased by a factor of approximately

80-100 when increasing the number of integration points from 10 to 100 while keeping the value $\Delta t/\Delta x^2 = 0.4$ constant.

Like we observed in the two-dimensional case, satisfying the stability condition is paramount to having good results. Looking at table I we see that the MAPE is around $10^{34-115}\%$, which means we are far away from the actual solution. When satisfying the condition however, we notice (also from table I) good correlation between the actual values (MAPE of around $10^{-3}\%$). We tried to increase the number of Fourier addends, however saw no change in error, meaning the actual solution should be accurate. Another thing which we observed in the one dimensional problem was oscillations in the computed values. It seems to be a product of not satisfying the stability conditions. If you don't know the stability conditions, this could be something to be vary of.

Each algorithm have their strength. Backward Euler and Crank-Nicolson are both unconditionally stable, giving it a clear edge over forward Euler. If stability conditions are not satisfied, the unconditionally algorithms still gave decent results, forward Euler however gave results with no resemblance of the actual result. On the other hand, when stability conditions were satisfied, forward performed similar to backward Euler in our tests. For $N_x = 10$, $N_x = 100$ and $\Delta t = 4 \cdot 10^{-3}$ forward Euler had around the same values of MAPE as backward. When $N_x = 10$ forward had MAPE= 0.0155% ($t = 0.83$) and MAPE=0.0005% ($t = 0.50$), backward had MAPE= 1.4169% ($t = 0.83$) and MAPE=0.1048% ($t = 0.50$). Looking at the plots 3 and 4, they also seem similar. They both stray away from zero equally far (around $6 \cdot 10^{-3}$). We notice the same trend for $N_x = 100$, implying that forward is equal to backward Euler when stability conditions are satisfied. Crank-Nicolson outperformed forward and backward in all cases. The MAPE-percentages were lower for all cases, also looking at the error-plots, they do not stray as far away from zero as forward and backward. This is expected, as we mentioned in the theory section backward and forward has errors $O(\Delta t)$ and $O(\Delta x^2)$ and Crank-Nicolson errors $O(\Delta t^2)$ and $O(\Delta x^2)$. When considering ease of implementation, forward and backwards Euler comes on top. Forward Euler is notoriously easy to implement, and as we have seen in this report, it gives good solutions when stability conditions are satisfied.

Code optimization

We did see a significant speed-up from the parallelization, the code was approximately twice as fast with eight cores as with one core. Given more time, there are several avenues we could have explored to improve on our solution, both to reduce memory consumption and computing time.

One possibility is to use only one-dimensional arrays for storing our results. We have done this to some extent in our code by representing the three-dimensional array of results (timesteps $\times\, x \times y$) as a two-dimensional array. Here, one one-dimensional sub-array represents the matrix in the x, y-plane at one time step. We could have gone further and represented the entire result array with a one-dimensional array, which is generally thought to reduce computing times. We have made a working attempt at this, which can be viewed on a developmental branch in our GitHub repository[5]. We were surprised to find that this code was actually slower than our first implementation. Simple timing indicated that the computing time was around 50% greater. It is possible that our way of indexing this one-dimensional array was sub-optimal and it is definitely something that would be interesting to investigate further.

A possible improvement, mainly with regard to memory usage, is the way we keep track of which diagonals have been calculated by the different cores. This is now implemented as a boolean array with one value for each diagonal of each time step that is simultaneously being calculated. This is possibly ineff SKAL DETTE MED?

Another alteration, which most certainly would increase the efficiency of our program, is to allow each core to calculate values for more than one time step between each time the threads are gathered and the result matrix reset. In our implementation, we let every core calculate the values for the entire grid in one time step each. After all the cores have completed this task, the threads are gathered, the result array is reset and the process is repeated. To split and gather the threads introduces overhead and takes time. By allowing each core to calculate more than one time step each before the reset, we could easily reduce this overhead cost manyfold over the course of the total calculation. If we for instance use two cores for calculation, then allowing the first core to calculate the first and third next time steps, and the second core to calculate the second and fourth next time steps before the reset would half the number of times the threads are split and gathered. This would require more memory as the result array would have to be larger. Representing the results for all time steps in one large array, while probably the fastest approach, would in most cases require more memory than is available. Our solution would have to land somewhere in-between, using enough memory to allow each core to calculate several time steps each, but not exhaust the memory resources of the computer. This tradeoff between memory consumption and computational efficiency is a recurring theme in computer science.

There are also possible improvements to be made with the way we reset the values in the result matrix. We store the results in a matrix where each row corresponds to a specific time step. During reset, the values of the first row are updated to the values of the last row. The

[5] https://github.com/sigurdru/FYS3150/tree/vegard_test/Project5/code

values of the other rows are set to zero. Firstly, while setting to zero the values that should later be updated is practical for troubleshooting, it is highly unnecessary and inefficient during larger calculations. The values will later be updated and setting them all to zero is simply a costly operation that in the end serves no purpose. Secondly, we probably do not need to move the elements from the first row to the last row one by one. We could change the pointer to the first row to instead point to the last row and vice versa. This would save a lot of indexing and memory handling during the reset, and could potentially reduce the computing time by a considerable amount.

VI. CONCLUSION

In one dimension, forward and backward Euler performed similarly when stability conditions were complied with. Looking both at the error-plots and MAPE, they oscillated between which was best, however close to each other in all cases. This was expected. Crank-Nicolson, like we predicted theoretically, outperformed them both across the board. However always within an order of magnitude, both when looking at MAPE and the error-plots. When we used Δt above the stability conditions, we saw slight improvements in backward Euler and no changes in Crank-Nicolson. Forward Euler on the other hand became unstable and produced values in the order of magnitude 10^8 which makes no sense.

For our two dimensional problem we used forward Euler as our solver. Here we also tested right above and below the stability conditions. As we saw in the one dimensional case, when below stability conditions we got good results (MAPE of around $10^{-3}\%$). When above the condition however, the solver produced results in the order of 10^{35-116}, and MAPE in the same order.

When it comes to which algorithm to use, there are some things to consider. Crank-Nicolson has better results, however is harder to implement. Backward Euler is easy to implement and unconditionally stable. Forward Euler is conditionally stable, and if these conditions are not met, the results does not correspond with the closed solution.

We have had a lot of focus code-implementation in this report. Extensive unit-tests have made us confident in our results. Our parallelization has made the code around twice as fast when running with eight cores. We suspect this is not fully optimized however, and we recommend reading the discussion section on parallelization if one plans on implementing this themselves. There we have discussed many ways of optimizing the code to make it run quicker.

[1] Morten Hjorth-Jensen, Computational Physics, Lecture Notes Fall 2015, August 2015, https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf.
[2] Ryan C. Daileda, Trinity University, Partial Differential Equations, March 6, 2012, http://ramanujan.math.trinity.edu/rdaileda/teach/s12/m3357/lectures/lecture_3_6_short.pdf.
[3] massachusetts institute of technology, Numerical Methods for Partial Differential Equations, March 31. 2009, https://ocw.mit.edu/courses/mathematics/18-336-numerical-methods-for-partial-differential-equations-spring-2009/lecture-notes/MIT18_336S09_lec14.pdf.
[4] Tveito, Aslak and Winther, Ragnar, Introduction to Partial Differential Equations A Computational Approach, 2005.

VII. APPENDIX

A. Derivation of equation for the Crank-Nicolson scheme

Again we define

$$\alpha = \frac{\Delta t}{(\Delta x)^2}$$

Inserting the approximations of u_t and u_{xx} stated in the Theory section into our differential equation, we get

$$\begin{aligned}
u_t &= u_{xx} \\
\frac{u_{i,j+1}-u_{i,j}}{\Delta t} &= \frac{1}{2\Delta x^2}\Big(u_{i+1,j}-2u_{i,j}+u_{i-1,j}+u_{i+1,j+1}-2u_{i,j+1}+u_{i-1,j+1}\Big) \\
u_{i,j+1}-u_{i,j} &= \frac{\alpha}{2}\Big(u_{i+1,j}-2u_{i,j}+u_{i-1,j}+u_{i+1,j+1}-2u_{i,j+1}+u_{i-1,j+1}\Big) \\
u_{i,j+1}-\frac{\alpha}{2}\Big(u_{i+1,j+1}-2u_{i,j+1}+u_{i-1,j+1}\Big) &= u_{i,j}+\frac{\alpha}{2}\Big(u_{i+1,j}-2u_{i,j}+u_{i-1,j}\Big) \\
2u_{i,j+1}-\alpha u_{i+1,j+1}+2\alpha u_{i,j+1}-\alpha u_{i-1,j+1} &= 2u_{i,j}+\alpha u_{i+1,j}-2\alpha u_{i,j}+\alpha u_{i-1,j} \\
-\alpha u_{i-1,j+1}+2(1+\alpha)u_{i,j+1}-\alpha u_{i+1,j+1} &= \alpha u_{i-1,j}+2(1-\alpha)u_{i,j}+\alpha u_{i+1,j}
\end{aligned} \tag{27}$$

Written out explicitly for all i, the equation reads

$$\begin{aligned}
-\alpha u_{0,j+1}+2(1+\alpha)u_{1,j+1}-\alpha u_{2,j+1} &= \alpha u_{0,j}+2(1-\alpha)u_{1,j}+\alpha u_{2,j} \\
-\alpha u_{1,j+1}+2(1+\alpha)u_{2,j+1}-\alpha u_{3,j+1} &= \alpha u_{1,j}+2(1-\alpha)u_{2,j}+\alpha u_{3,j} \\
&\vdots \\
-\alpha u_{n-3,j+1}+2(1+\alpha)u_{n-2,j+1}-\alpha u_{n-1,j+1} &= \alpha u_{n-3,j}+2(1-\alpha)u_{n-2,j}+\alpha u_{n-1,j} \\
-\alpha u_{n-2,j+1}+2(1+\alpha)u_{n-1,j+1}-\alpha u_{n,j+1} &= \alpha u_{n-2,j}+2(1-\alpha)u_{n-1,j}+\alpha u_{n,j}
\end{aligned}$$

Similar to the case of Backward Euler, this set of equations can be rearranged as follows

$$\begin{aligned}
2(1+\alpha)u_{1,j+1}-\alpha u_{2,j+1} &= \alpha u_{0,j}+2(1-\alpha)u_{1,j}+\alpha u_{2,j}+\alpha u_{0,j+1} \\
-\alpha u_{1,j+1}+2(1+\alpha)u_{2,j+1}-\alpha u_{3,j+1} &= \alpha u_{1,j}+2(1-\alpha)u_{2,j}+\alpha u_{3,j} \\
&\vdots \\
-\alpha u_{n-3,j+1}+2(1+\alpha)u_{n-2,j+1}-\alpha u_{n-1,j+1} &= \alpha u_{n-3,j}+2(1-\alpha)u_{n-2,j}+\alpha u_{n-1,j} \\
-\alpha u_{n-2,j+1}+2(1+\alpha)u_{n-1,j+1} &= \alpha u_{n-2,j}+2(1-\alpha)u_{n-1,j}+\alpha u_{n,j}+\alpha u_{n,j+1}
\end{aligned}$$

We can define the vector $\mathbf{b}_j$ to hold the values on the right side of each of these equations:

$$\mathbf{b}_j = \begin{bmatrix} \alpha u_{0,j}+2(1-\alpha)u_{1,j}+\alpha u_{2,j}+\alpha u_{0,j+1} \\ \alpha u_{1,j}+2(1-\alpha)u_{2,j}+\alpha u_{3,j} \\ \vdots \\ \alpha u_{n-3,j}+2(1-\alpha)u_{n-2,j}+\alpha u_{n-1,j} \\ \alpha u_{n-2,j}+2(1-\alpha)u_{n-1,j}+\alpha u_{n,j}+\alpha u_{n,j+1} \end{bmatrix} \tag{28}$$

Using this and the definition of the vector $\mathbf{v}_j$ from (11), we can write the set of equations as

$$\mathbf{A}\mathbf{v}_j = \mathbf{b}_j$$

where the matrix $\mathbf{A}$ is a tridiagonal matrix with $2(1+\alpha)$ on the diagonal and $-\alpha$ directly above and below the diagonal.

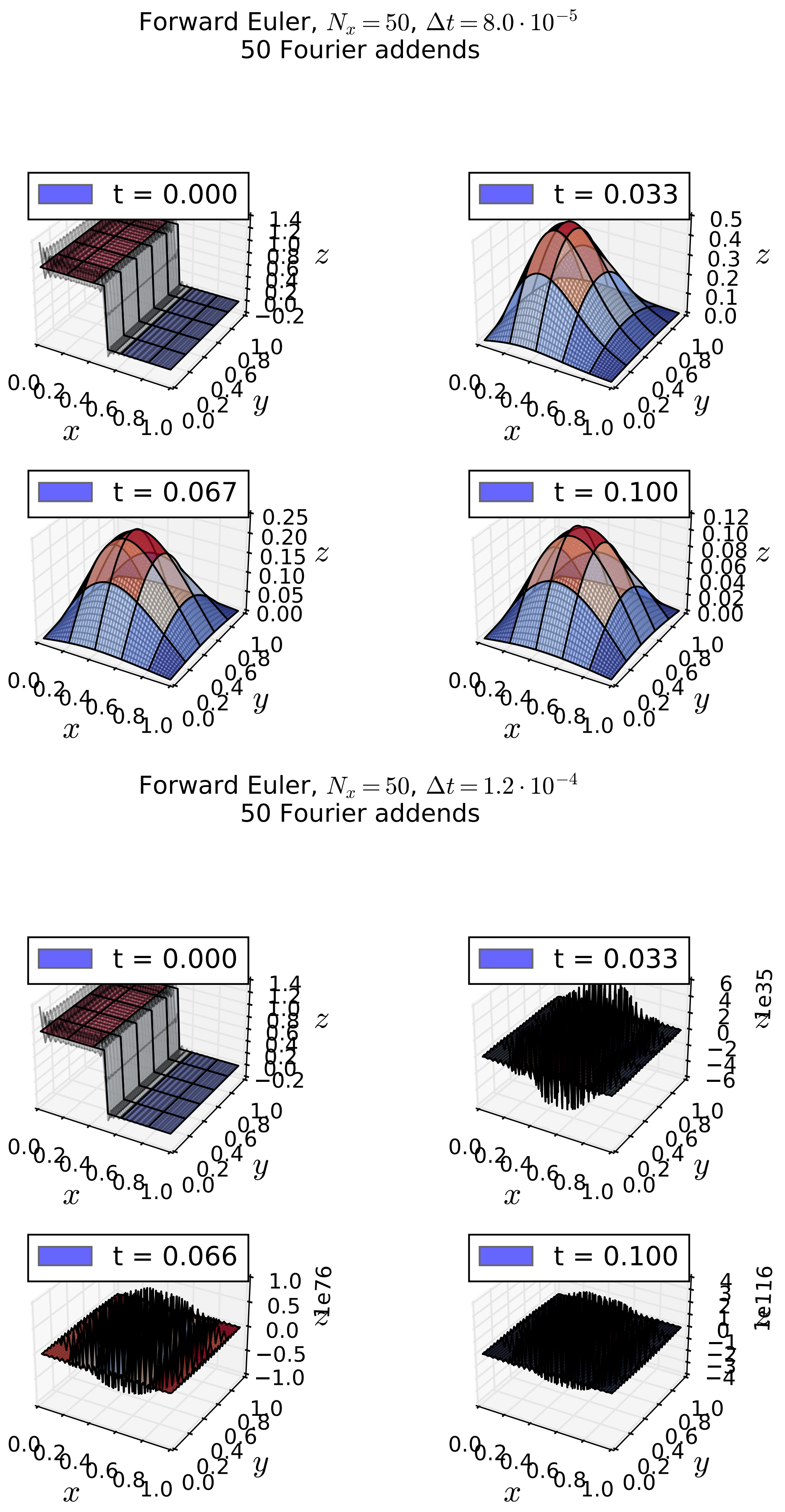

Figure 6. In this figure you see the analytical and computed result for the two dimensional problem. The colored plot is the computed, and the black mesh plot is the analytical plot. The top four plots are the stable solutions, and the bottom four are the unstable ones. They are all plotted at the same time steps, $n = m50$ Fourier addends and number of integration points in $x-$ and $y-$direction.

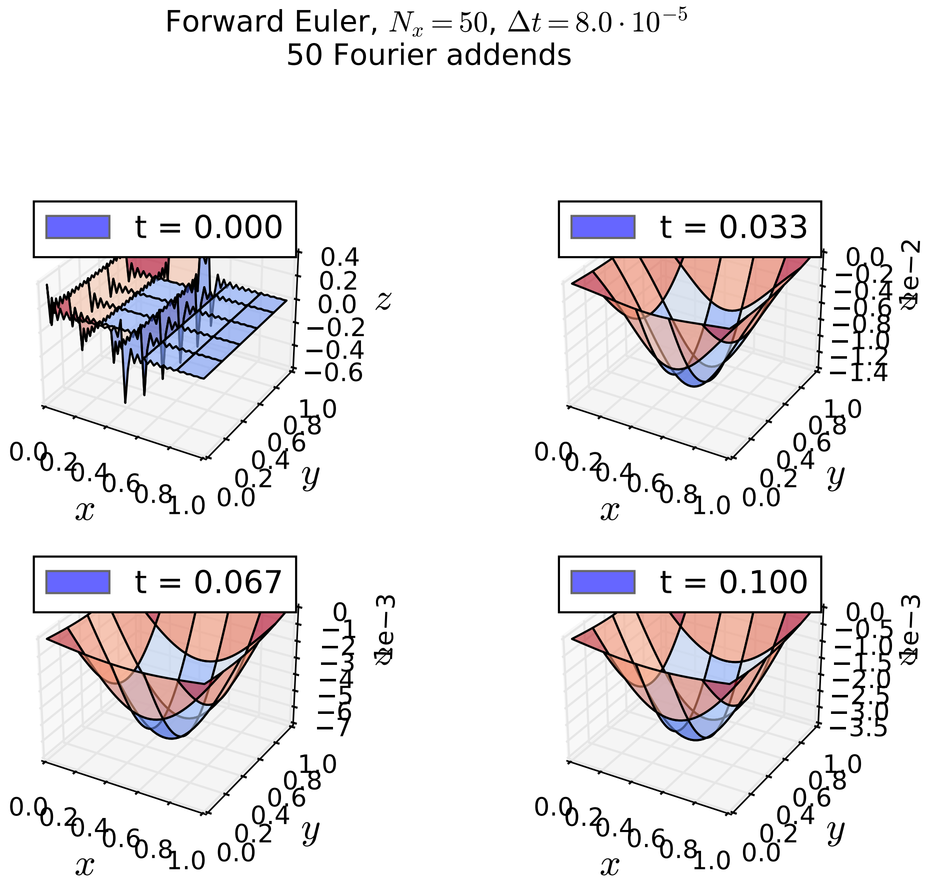

Figure 7. In this figure we have plotted the difference between the theoretical and stable computed values. We used $n = m = 50$ Fourier addends, and plotted the difference for three different time steps, the same as in figure 6.