FYS4150 - Computational Physics - Project 4

Eimund Smestad
Email: eimundsm@fys.uio.no
10th November, 2014

Abstract

In this report I will look at diffusion of neurotransmitter across synaptic cleft by solving the heat equation. I will compare the numerical result from the heat equation with an analytical solution of the diffusion problem. I have implemented the θ rule to solve the partial differential equation of the heat equation, and for the implicit schemes I will use a tridiagonal matrix solver of $4N$ FLOPS.

1 Diffusion of neurotransmitters

I will study diffusion as a transport process for neurotransmitters across synaptic cleft separating the cell membrane of two neurons, for more detail see [1]. The diffusion equation is the partial differential equation

$$\frac{\partial u(\mathbf{x},t)}{\partial t} = \nabla \cdot \left(D(\mathbf{x},t)\, \boldsymbol{\nabla} u(\mathbf{x},t) \right),$$

where u is the concentration of particular neurotransmitters at location $\mathbf{x}$ and time t with the diffusion coefficient D. In this study I consider the diffusion coefficient as constant, which simplify the diffusion equation to the heat equation

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

It is further assumed that the neurotransmitter concentration u is only dependent on the distance x in direction between the presynaptic to the postsynaptic across the synaptic cleft. Hence we have the differential equation

$$\frac{\partial u(x,t)}{\partial t} = D\frac{\partial^2 u(x,t)}{\partial x^2}\,. \tag{1}$$

The boundary and initial condition that I'm going to study is

$$\forall t \in \mathbb{R}_0 : u(0,t) = u_0\,, \quad \forall t \in \mathbb{R} : u(d,t) = 0 \quad \text{and} \quad \forall x \in \mathbb{R}_{0+}^{d-} \forall t \in \mathbb{R}^0 : u(x,t) = 0\,, \tag{2}$$

where u_0 are kept constant at the presynaptic, d is the distance between the presynaptic and the postsynaptic. Note that the notation $\forall x \in \mathbb{R}_{a+}^{b-} \Leftrightarrow a < x < b$, where as $\forall x \in \mathbb{R}_a^b \Leftrightarrow a \leq x \leq b$. Note also that these boundary conditions implies that the neurotransmitters are immediately absorbed at the postsynaptic, and for $t < 0$ there are no neurotransmitters between the pre- and postsynaptic.

To solve the differential equation (1) with the boundary condition (2) we start by separating the concentration $u(x,t)$ into two functions $u_1(x)$ and $u_2(x,t)$ in the time and space of the signal transmission of the neurotransmitters

$$\forall x \in \mathbb{R}_0^d \forall t \in \mathbb{R}_0 : u(x,t) = u_1(x) + u_2(x,t) \,, \tag{3}$$

such that u_2 satisfies the Dirichlet boundary condition $u_2(0,t) = u_2(d,t) = 0$, which forces $u_2(x,t)$ to be separated into two functions $u_3(x)$ and $u_4(t)$ as follows

$$u_2(0,t) = u_2(d,t) = 0 \quad \Rightarrow \quad u_2(x,t) = u_3(x)\,u_4(t) \qquad \text{when } d \neq 0 \text{ and } u_3(0) = u_3(d) = 0. \tag{4}$$

Now putting (2) and (3) into (1) yields

$$u_3(x)\frac{\partial u_4(t)}{\partial t} = D\left(u_4(t)\frac{\partial^2 u_3(x)}{\partial x^2} + \frac{\partial^2 u_1(x)}{\partial x^2}\right).$$

We wish that $\frac{\partial^2 u_1(x)}{\partial x^2} = 0$ because then we can separate this partial differential equation by variables, which puts the requirement $u_1(x) = a_0 + a_1 x$. Since the separation of $u(x,t)$ into $u_1(x)$ and $u_2(x,t)$ in (3) can be done arbitrarily without changing the solution of $u(x,t)$, means that the requirement $u_1(x) = a_0 + a_1 x$ is allowed. However we also need to investigate that $u_1(x) = a_0 + a_1 x$ satisfies the boundary condition in (2);

$$\begin{aligned} u(0,t) &= u_1(0) + u_2(0,t) = u_1(0) = u_0 \\ u(d,t) &= u_1(d) + u_2(d,t) = u_1(d) = 0\,, \end{aligned}$$

where the Dirichlet boundary condition in (4) are used. And we see that the boundary condition can satisfy the requirement $u_1(x) = a_0 + a_1 x$ when

$$u_1(x) = u_0\left(1 - \frac{x}{d}\right). \tag{5}$$

The differential equation in (1) can now be written as

$$\frac{1}{Du_4(t)}\frac{\partial u_4(t)}{\partial t} = \frac{1}{u_3(x)}\frac{\partial^2 u_3(x)}{\partial x^2} = -\lambda^2\,,$$

where λ is a constant, because t and x can vary independently. These two equations have the following solution

$$\begin{aligned} u_3(x) &= A\sin\left(\lambda x + \varphi\right) \qquad \text{and} \\ u_4(t) &= C\mathrm{e}^{-D\lambda^2 t}\,. \end{aligned}$$

Applying the boundary conditions for u_3 in (4) that $u_3(0) = u_4(d) = 0$ gives

$$\lambda = \frac{n\pi}{d} \qquad \text{for } n \in \mathbb{N}/\{0\}\,,$$

where I let $\mathbb{N} = \mathbb{N}_{-\infty}^{\infty}$ represent all positive and negative integers. Hence

$$u_2(x,t) = u_3(x)\,u_4(t) = \sum_{n=1} A_n \sin(n\pi x) \exp\left(-D\left(\frac{n\pi}{d}\right)^2 t\right),$$

where the negative values of n is absorbed into the coefficient A_n. Applying the initial condition from (2)

$$u(x,0) = u_0\left(1 - \frac{x}{d}\right) + \sum_{n=1} A_n \sin\left(n\pi\frac{x}{d}\right) = 0 \qquad \text{for } x \in \mathbb{R}_0^d. \tag{6}$$

We need to determine the coefficients A_n, and the trick is to do something with the equation above such that we isolate the A_n coefficients. To achieve this we use the fact that $\sin\left(n\pi\frac{x}{d}\right)$ is orthogonal with $\sin\left(m\pi\frac{x}{d}\right)$ under integration

$$\begin{aligned}\int \sin\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right)\mathrm{d}x &= -\frac{d}{m\pi}\cos\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right) + \frac{n}{m}\int \cos\left(m\pi\frac{x}{d}\right)\cos\left(n\pi\frac{x}{d}\right)\mathrm{d}x \\ &= -\frac{d}{m\pi}\cos\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right) + \frac{n}{m}\left(\frac{d}{\pi m}\sin\left(m\pi\frac{x}{d}\right)\cos\left(n\pi\frac{x}{d}\right) + \frac{n}{m}\int \sin\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right)\mathrm{d}x\right),\end{aligned}$$

where I have used integration by parts $\int u\dot{v} = uv - \int \dot{u}v$. Solving this equation with regard to the integral we get

$$\int \sin\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right)\mathrm{d}x = \frac{d}{\pi\left(n^2 - m^2\right)}\left(n\sin\left(m\pi\frac{x}{d}\right)\cos\left(n\pi\frac{x}{d}\right) - m\cos\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right)\right).$$

We want to make the result of this integral zero for $n \neq m$, which is the result if $x = \frac{kd}{2}$ and $x = \frac{\ell d}{2}$ where $k, \ell \in \mathbb{N}$ (for zero,negative and positive integers), because then sin cos parts above becomes zero. This means that we need to integrate from $x = \frac{kd}{2}$ and $x = \frac{\ell d}{2}$. However the result above is not defined for $n = m$ because we get $\frac{0}{0}$. So we redo the integration for $n = m$;

$$\begin{aligned}\int_{\frac{kd}{2}}^{\frac{\ell d}{2}} \sin^2\left(n\pi\frac{x}{d}\right)\mathrm{d}x &= -\left[\frac{d}{\pi n}\cos\left(n\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right)\right]_{\frac{kd}{2}}^{\frac{\ell d}{2}} + \int_{\frac{kd}{2}}^{\frac{\ell d}{2}} \cos^2\left(n\pi\frac{x}{d}\right)\mathrm{d}x = \int_{\frac{kd}{2}}^{\frac{\ell d}{2}} \cos^2\left(n\pi\frac{x}{d}\right)\mathrm{d}x \\ &= \int_{\frac{kd}{2}}^{\frac{\ell d}{2}} \left(1 - \sin^2\left(n\pi\frac{x}{d}\right)\right)\mathrm{d}x = [x]_{\frac{kd}{2}}^{\frac{\ell d}{2}} - \int_{\frac{kd}{2}}^{\frac{\ell d}{2}} \sin^2\left(n\pi\frac{x}{d}\right)\mathrm{d}x = \frac{d}{2}(\ell - k) - \int_{\frac{kd}{2}}^{\frac{\ell d}{2}} \sin^2\left(n\pi\frac{x}{d}\right)\mathrm{d}x = \frac{d}{4}(\ell - k),\end{aligned}$$

where I solve the equation with regard to in $\int \frac{d}{2}(\ell - k)\sin^2\left(n\pi\frac{x}{d}\right)\mathrm{d}x$ in the last step. I have also used integration by parts $\int u\dot{v} = uv - \int \dot{u}v$ and the Pythagoras trigonometric relation $\sin^2 x + \cos^2 x = 1$. So the solution of the following integral is

$$\int_{\frac{kd}{2}}^{\frac{\ell d}{2}} \sin\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right)\mathrm{d}x = \frac{d}{4}(\ell - k)\,\delta_{mn},$$

where δ_{mn} is the Kronecker delta, and hence I have showed the orthogonality of the above integral. Applying this to (6);

$$\sum_{n=1}\int_{\frac{kd}{2}}^{\frac{\ell d}{2}} A_n \sin\left(m\pi\frac{x}{d}\right)\sin\left(n\pi\frac{x}{d}\right)\mathrm{d}x = \int_{\frac{kd}{2}}^{\frac{\ell d}{2}} u_0 \sin\left(m\pi\frac{x}{d}\right)\left(\frac{x}{d}-1\right)\mathrm{d}x\,,$$

we can isolate A_n at $n = m$ because of the Kronecker delta δ_{nm};

$$\begin{aligned}
A_n &= \frac{4}{d\,(\ell-k)}\int_{\frac{kd}{2}}^{\frac{\ell d}{2}} u_0 \sin\left(n\pi\frac{x}{d}\right)\left(\frac{x}{d}-1\right)\mathrm{d}x = \frac{4u_0}{n\,(\ell-k)\,\pi}\left(-\left[\left(\frac{x}{d}-1\right)\cos\left(n\pi\frac{x}{d}\right)\right]_{\frac{kd}{2}}^{\frac{\ell d}{2}} + \frac{1}{d}\int_{\frac{kd}{2}}^{\frac{\ell d}{2}}\cos\left(n\pi\frac{x}{d}\right)\mathrm{d}x\right)\\
&= \frac{4u_0}{n\,(\ell-k)\,\pi}\left(-\left[\left(\frac{x}{d}-1\right)\cos\left(n\pi\frac{x}{d}\right)\right]_{\frac{kd}{2}}^{\frac{\ell d}{2}} + \frac{1}{n\pi}\left[\sin\left(n\pi\frac{x}{d}\right)\right]_{\frac{kd}{2}}^{\frac{\ell d}{2}}\right)\\
&= \frac{4u_0}{n\,(\ell-k)\,\pi}\left(\left(\frac{k}{2}-1\right)\cos\left(\frac{kn\pi}{2}\right) - \left(\frac{\ell}{2}-1\right)\cos\left(\frac{\ell n\pi}{2}\right) + \frac{1}{n\pi}\left(\sin\left(\frac{\ell n\pi}{2}\right) - \sin\left(\frac{kn\pi}{2}\right)\right)\right)\\
&= \frac{4u_0}{n\,(\ell-k)\,\pi}\begin{cases}
\frac{k-\ell}{2} & \text{when } kn \text{ and } \ell n \text{ is even,}\\
\frac{k}{2}-1+\frac{1}{n\pi} & \text{when } kn \text{ is even and } \ell n \text{ is odd,}\\
1-\frac{\ell}{2}+\frac{1}{n\pi} & \text{when } kn \text{ is odd and } \ell n \text{ is even,}\\
0 & \text{when } kn \text{ and } \ell n \text{ is odd.}
\end{cases}
\end{aligned}$$

Since $x \in \mathbb{R}_0^d$ in (6) leads to $k, \ell \in \{0, 2\}$ (because of $\frac{kd}{2}$ and $\frac{\ell d}{2}$ in the integration interval) and $k \neq \ell$ for A_n to apply to the whole interval of x, which means that kn and ℓn is even; hence there are only one possibility of the solution above

$$A_n = -\frac{2u_0}{n\pi} \qquad \text{for } x \in \mathbb{R}_0^d\,. \tag{7}$$

Therefore the analytical solution for the concentration of neurotransmitters in (3) is given by

$$\forall x \in \mathbb{R}_0^d \forall t \in \mathbb{R}_0 : u\,(x,t) = u_0\left(1 - \frac{x}{d} - \sum_{n=1}\frac{2}{n\pi}\sin\left(n\pi x\right)\exp\left(-D\left(\frac{n\pi}{d}\right)^2 t\right)\right). \tag{8}$$

2 Numerical methods

2.1 The θ-rule

The Taylor expansion is given by

$$u\,(x) = \sum_{n=0}\frac{\overset{(n)}{u}(x_0)}{n!}\,(x-x_0)^n \tag{9}$$

where $\overset{(n)}{u} = \frac{\mathrm{d}^n u}{\mathrm{d}t^n}$ and x_0 is a initial value where we step from to x . If we now use the first order approximation

$$u\,(x) \approx u\,(x_0) + \overset{(1)}{u}(x_0)\,(x-x_0)\,.$$

The first order differential equation $\overset{(1)}{u}(x) = f(x)$ is determined when we have the initial condition $u(x_0)$, however $\overset{(1)}{u}(x_0)$ is not an initial condition, and it depends on how we calculate it numerically from the initial condition. Now note that $\overset{(1)}{u}(x_0)$ is the same for different values of x in the approximation above and lets say that we calculate it as given from the approximation above;

$$\overset{(1)}{u}(x_0) \approx \frac{u(x) - u(x_0)}{x - x_0} . \tag{10}$$

So now use this in another point $x_\theta = \theta x + (1-\theta) x_0$ which we also approximate to the first order, and if we use the expression above for $\overset{(1)}{u}(x_0)$ we get

$$\begin{aligned} u(x_\theta) &\approx u(x_0) + \overset{(1)}{u}(x_0)(x_\theta - x_0) = u(x_0) + \theta \overset{(1)}{u}(x_0)(x - x_0) \\ &\approx u(x_0) + \frac{u(x) - u(x_0)}{x - x_0}\theta(x - x_0) = \theta u(x) + (1-\theta) u(x_0) , \end{aligned} \tag{11}$$

this is known as the θ-rule. The θ-rule can be used to approximate the solution of the following first order differential equation

$$\overset{(1)}{u}(x) = f(u(x)) , \tag{12}$$

where we use (10) to approximate the expression $\overset{(1)}{u}(x)$ and given an even better or worse approximation to the solution $u(x)$ by approximating $f(x) \approx f(x_\theta)$;

$$\frac{u(x) - u(x_0)}{x - x_0} \approx f\Big(u(x_\theta)\Big) = f\Big(\theta u(x) + (1-\theta) u(x_0)\Big) ,$$

which discretize to

$$\frac{u_{i+1} - u_i}{x_{i+1} - x_i} = f\Big(\theta u_{i+1} + (1-\theta) u_i\Big) \qquad \text{where } i \in \mathbb{N}_0 \text{ and } u_0 \text{ is an initial contidion.} \tag{13}$$

We can find the the next step in the numerical solution to (12) by solving this difference equation with regard to u_{i+1}. Note the above discretization is known as Forward Euler scheme (Explicit) when $\theta = 0$, Backward Euler scheme (Implicit) when $\theta = 1$ and Crank-Nicolson scheme when $\theta = \frac{1}{2}$.

The truncation error of the Forward and Backward Euler scheme can be found by an alternative derivation, where expand the Taylor series in (9) around the point $x_0 \pm \Delta x$ accordingly;

$$u(x_0 \pm \Delta x) = \sum_{n=0} \frac{\overset{(n)}{u}(x_0)}{n!} (\pm \Delta x)^n , \tag{14}$$

and solve it with regard to $\overset{(1)}{u}(x_0)$

$$\overset{(1)}{u}(x_0) = \frac{u(x_0 \pm \Delta x) - u(x_0)}{\Delta x} + O(\Delta x) ,$$

which means that we have a local truncation error of $O(\Delta x)$ with the Forward and Backward Euler scheme. However the Crank-Nicolson scheme can be found by subtraction the Taylor expansion above for the two points

$$u(x_0 + \Delta x) - u(x_0 - \Delta x) = 2 \sum_{n=1} \frac{\overset{(2n-1)}{u}(x_0)}{(2n-1)!} \Delta x^{2n-1} ,$$

and solve it with regard to $\overset{(1)}{u}(x_0)$

$$\overset{(1)}{u}(x_0) = \frac{u(x_0 + \Delta x) - u(x_0 - \Delta x)}{2\Delta x} + O\left(\Delta x^2\right) ,$$

which means that we have a local truncation error of $O\left(\Delta x^2\right)$. With the θ-rule we can get even better or worse truncation error, because we can change the θ value to change the approximation.

2.2 Second order derivative

We approximated the first order derivative in (10), but we need to approximate the second order derivative to be able to solve the diffusion in (1).
Now adding the two expansions in (14)

$$u(x_0 + \Delta x) + u(x_0 - \Delta x) = 2 \sum_{n=0} \frac{\overset{(2n)}{u}(x_0)}{(2!)} \Delta x^{2n} = 2u(x_0) + \overset{(2)}{u}(x_0) \Delta x^2 + 2 \sum_{n=2} \frac{\overset{(2n)}{u}(x_0)}{(2!)} \Delta x^{2n} ,$$

and solve it with

$$\begin{aligned} \overset{(2)}{u}(x_0) &= \frac{u(x_0 + \Delta x) - 2u(x_0) + u(x_0 - \Delta x)}{\Delta x^2} - 2 \sum_{n=2} \frac{\overset{(2n)}{u}(x_0)}{(2!)} \Delta x^{2(n-1)} \\ &= \frac{u(x_0 + \Delta x) - 2u(x_0) + u(x_0 - \Delta x)}{\Delta x^2} + O\left(\Delta x^2\right) , \end{aligned}$$

So the second order derivative can be approximated with

$$\overset{(2)}{u}(x_0) \approx \frac{u(x_0 + \Delta x) - 2u(x_0) + u(x_0 - \Delta x)}{\Delta x^2} \tag{15}$$

with the local truncation error $O\left(\Delta x^2\right)$.

2.3 The heat equation

We want to discretize the dimensionless heat equation from (1), where we use $D = 1$, $u_0 = 1$ and $d = 1$,

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2} ,$$

to numerically solve diffusion of neurotransmitters. First we do the θ-rule discretization in (13)

$$\frac{u_{i(j+1)} - u_{ij}}{\Delta t} = \frac{\partial^2 u_{i(j+\theta)}}{\partial x^2} = \frac{\partial^2 \left(\theta u_{i(j+1)} + (1-\theta) u_{ij}\right)}{\partial x^2} = \theta \frac{\partial u_{i(j+1)}}{\partial x^2} + (1-\theta) \frac{\partial u_{ij}}{\partial x^2}$$

and then implement discretization of the second order in (15)

$$\frac{u_{i(j+1)} - u_{ij}}{\Delta t} = \frac{\theta}{\Delta x^2}\left(u_{(i+1)(j+1)} - 2u_{i(j+1)} + u_{(i-1)(j+1)}\right) + \frac{1-\theta}{\Delta x^2}\left(u_{(i+1)j} - 2u_{ij} + u_{(i-1)j}\right), \tag{16}$$

where index i is stepping of x and j is stepping of t. And from the discussion before we have the local truncation error $O(\Delta t)$ and $O\left(\Delta x^2\right)$ for the Forward and Backward Euler scheme, and $O\left(\Delta t^2\right)$ and $O\left(\Delta x^2\right)$ for Crank-Nicoloson scheme.

The dimensionless initial condition from (4) gives us

$$u_{i0} = \begin{cases} 1 & \text{when } i = 0, \\ 0 & \text{elsewhere.} \end{cases}$$

The Forward Euler scheme is when $\theta = 0$ which gives explicit solution of (16)

$$u_{i(j+1)} = u_{ij} + \alpha\left(u_{(i+1)j} - 2u_{ij} + u_{(i-1)j}\right), \tag{17}$$

where

$$\alpha = \frac{\Delta t}{\Delta x^2} \qquad \text{and} \qquad n = \frac{1}{\Delta x}.$$

When we step forward in time with j Implicitly $\theta > 0$, we use (16) to find the next time step $j+1$ which is unknown and collect the unknowns on one side of the equation;

$$\begin{array}{lcll} u_{0(j+1)} & & 1 & \text{boundary condition,} \\ -u_{(i-1)(j+1)} + \left(2 + \frac{1}{\alpha\theta}\right) u_{i(j+1)} - u_{(i+1)(j+1)} & = & \left(\frac{1}{\theta} - 1\right)\left(u_{(i+1)j} - 2u_{ij} + u_{(i-1)j}\right) + \frac{u_{ij}}{\alpha\theta} & \text{when } i \in \mathbb{N}_1^{n-1}, \\ u_{n(j+1)} & & 0 & \text{boundary condition.} \end{array}$$

We can rewrite further to omit the boundary condition

$$\begin{array}{lcll} \left(2 + \frac{1}{\alpha\theta}\right) u_{1(j+1)} - u_{2(j+1)} & & \left(\frac{1}{\theta} - 1\right)\left(u_{2j} - 2u_{1j} + u_{0j}\right) + \frac{u_{1j}}{\alpha\theta} + u_{0j} & \text{when } i = 1, \\ -u_{(i-1)(j+1)} + \left(2 + \frac{1}{\alpha\theta}\right) u_{i(j+1)} - u_{(i+1)(j+1)} & = & \left(\frac{1}{\theta} - 1\right)\left(u_{(i+1)j} - 2u_{ij} + u_{(i-1)j}\right) + \frac{u_{ij}}{\alpha\theta} & \text{when } i \in \mathbb{N}_2^{n-2}, \\ -u_{(n-2)(j+1)} + \left(2 + \frac{1}{\alpha\theta}\right) u_{(n-1)(j+1)} & & \left(\frac{1}{\theta} - 1\right)\left(u_{nj} - 2u_{(n-1)j} + u_{(n-2)j}\right) + \frac{u_{(n-1)j}}{\alpha\theta} + u_{nj} & \text{when } i = n-1, \end{array} \tag{18}$$

where the left side constructs a tridiagonal matrix with the elements $\left(-1, 2 + \frac{1}{\alpha\theta}, -1\right)$ when we extract the unknowns to a vector. Note that the boundary conditions are added in $i = 1$ and $i = n-1$ in addition to the expression for $i \in \mathbb{N}_2^{n-2}$, which we get by taking the Gaussian elimination on row $i = 1$ and $i = n-1$ with regard to the row $i = 0$ and $i = n$ accordingly.

The Explicit solver in (17) requires $5N$ FLOPS to be solve for each timestep. The implicit solver requires a tridiagonal solver in addition (18), and the tridiagonal solver uses $4N$ FLOPS. For the Backward Euler scheme $\theta = 1$ so $\frac{1}{\theta} - 1 = 0$ and we need only one FLOP to calculate (18) since $\frac{1}{\alpha\theta}$ can be precalculated, and with the tridiagonal solver we then have $5N$ FLOPS. For a general *theta*-rule implicit scheme (18) we need $6N$ FLOPS to calculate (18) since $\frac{1}{\theta} - 1$ and $\frac{1}{\alpha\theta}$ can be precalculated, and with the tridiagonal solver we then have $10N$ FLOPS.

2.3.1 Tridiagonal matrix

We found that heat equation in (1) can be solved by solving a matrix problem on the form

$$\mathbf{Au} = \mathbf{v}$$

where $\mathbf{u}$ is the unknowns that we want to find, $\mathbf{v}$ are given values and $\mathbf{A}$ is a tridiagonal matrix on the form $(-1, a, -1)$ which we represent with the elements

$$a_{ij} = \begin{cases} a & \text{when } i = j. \\ -1 & \text{when } j \in \{i-1, i+1\} \text{ and } i, j \in \mathbb{N}_1^{n-1}, \\ 0 & \text{otherwise.} \end{cases}$$

We then use Gaussian elimination to reduce the tridiagonal matrix $\mathbf{A}$ to a upper triangular matrix $\check{\mathbf{A}}$ in the following way

$$\check{\mathbf{A}}\mathbf{u} = \check{\mathbf{v}}\,.$$

To find the upper tridiagonal matrix we need to eliminate the elements $a_{i(i-1)}$ so $\check{a}_{i(i-1)} = 0$, which means that we need to multiply the values (without changing them) in row $i-1$ with

$$\frac{a_{i(i-1)}}{\check{a}_{(i-1)(i-1)}} = -\frac{1}{\check{a}_{(i-1)(i-1)}}$$

and subtract them with the values in row i, which leads obviously to

$$\check{a}_{i(i-1)} = a_{i(i-1)} - \frac{a_{i(i-1)}}{\check{a}_{(i-1)(i-1)}}\check{a}_{(i-1)(i-1)} = 0\,.$$

We can also show that off-tridiagonal elements in the upper tridiagonal remains zero under this Gaussian elimination

$$\check{a}_{ij} = a_{ij} - \frac{a_{i(i-1)}}{\check{a}_{(i-1)(i-1)}}\check{a}_{(i-1)j} = a_{ij} = 0 \quad \text{when } i \in \mathbb{N}_2^{n-3} \text{ and } j \in \mathbb{N}_{i+2}^{n-1}, \text{ because } \check{a}_{1j} = a_{1j} = 0.$$

and similarly that the off-tridiagonal elements in the lower tridiagonal also remains zero

$$\check{a}_{ij} = a_{ij} - \frac{a_{i(i-1)}}{\check{a}_{(i-1)(i-1)}}\check{a}_{(i-1)j} = a_{ij} = 0 \quad \text{when } i \in \mathbb{N}_3^{n-1} \text{ and } j \in \mathbb{N}_1^{i-2}, \text{ because } \check{a}_{i(i-1)} = a_{i(i-1)} = 0.$$

We can also show that elements $a_{i(i+1)} = -1$ also remains unchanged under this Gaussian elimination

$$\breve{a}_{i(i+1)} = a_{i(i+1)} - \frac{a_{i(i-1)}}{\breve{a}_{(i-1)(i-1)}}\breve{a}_{(i-1)(i+1)} = a_{i(i+1)} = -1\,.$$

This results in that the diagonal elements are then given by

$$\breve{a}_{ii} = a_{ii} - \frac{a_{i(i-1)}}{\breve{a}_{(i-1)(i-1)}}\breve{a}_{(i-1)i} = a_{ii} - \frac{1}{\breve{a}_{(i-1)(i-1)}} \quad \text{when } \breve{a}_{11} = a_{11} = a\,. \tag{19}$$

We have now calculated all the elements in the upper triangular matrix $\breve{\mathbf{A}}$, and we can summarize the the elements as

$$\breve{a}_{ij} = \begin{cases} a_{11} & \text{when } i = j = 1 \\ a_{ii} - \frac{1}{\breve{a}_{(i-1)(i-1)}} & \text{when } i = j \text{ and } i \in \mathbb{N}_2^{n-1} \\ -1 & \text{when } j = i+1 \text{ and } i, j \in \mathbb{N}_1^{n-1} \\ 0 & \text{otherwise}\,. \end{cases}$$

We also need to do the Gaussian elimination on the vector $\mathbf{v}$ as well, which leads to

$$\breve{v}_i = \begin{cases} v_1 & \text{when } i = 1 \\ v_i + \frac{\breve{v}_{i-1}}{\breve{a}_{(i-1)(i-1)}} & \text{when } i \in \mathbb{N}_2^{n-1}\,. \end{cases} \tag{20}$$

We now want to eliminate the upper off-diagonal elements $\breve{\mathbf{A}}$ with Gaussian elimination so we get at diagonal matrix $\hat{\mathbf{A}}$ in the following way

$$\hat{\mathbf{A}}\mathbf{u} = \hat{\mathbf{v}}.$$

To find the diagonal matrix we need to eliminate the elements $\breve{a}_{i(i+1)}$ so $\hat{a}_{i(i+1)} = 0$, which means that we need to multiply the values (without changing them) in row $i+1$ with

$$\frac{\breve{a}_{i(i+1)}}{\hat{a}_{(i+1)(i+1)}} = -\frac{1}{\hat{a}_{(i+1)(i+1)}}$$

and subtract them with the values in row i, which leads obviously to

$$\hat{a}_{i(i+1)} = \breve{a}_{i(i+1)} - \frac{\breve{a}_{i(i+1)}}{\hat{a}_{(i+1)(i+1)}}\hat{a}_{(i+1)(i+1)} = 0\,.$$

We can also show that off-tridiagonal elements in the upper tridiagonal remains zero under this Gaussian elimination

$$\hat{a}_{ij} = \breve{a}_{ij} - \frac{\breve{a}_{i(i+1)}}{\hat{a}_{(i+1)(i+1)}}\breve{a}_{(i+1)j} = \breve{a}_{ij} = 0 \quad \text{when } i \in \mathbb{N}_1^{n-3} \text{ and } j \in \mathbb{N}_{i+2}^{n-1}, \text{ because } \hat{a}_{i(i+1)} = \breve{a}_{i(i+1)} = 0,$$

and similarly that the off-diagonal elements in the lower tridiagonal also remains zero

$$\hat{a}_{ij} = \check{a}_{ij} - \frac{\check{a}_{i(i+1)}}{\hat{a}_{(i+1)(i+1)}}\hat{a}_{(i+1)j} = a_{ij} = 0 \quad \text{when } i \in \mathbb{N}_2^{n-2} \text{ and } j \in \mathbb{N}_1^{i-1}, \text{ because } \check{a}_{i(i+1)} = a_{i(i+1)} = 0.$$

This results unchanged diagonal elements

$$\hat{a}_{ii} = \check{a}_{ii} - \frac{\check{a}_{i(i+1)}}{\hat{a}_{(i+1)(i+1)}}\check{a}_{(i+1)i} = \check{a}_{ii} = a_{ii} - \frac{1}{\check{a}_{(i-1)(i-1)}} \qquad \text{where } \check{a}_{11} = a_{11}\,.$$

We have now calculated all the elements in the diagonal matrix $\hat{\mathbf{A}}$, and we can summarize the elements as

$$\hat{a}_{ij} = \begin{cases} \check{a}_{ii} & \text{when } i = j \text{ and } i \in \mathbb{N}_1^{n-1} \\ 0 & \text{otherwise.} \end{cases}$$

We also need to do the Gaussian elimination on the vector $\mathbf{v}$ as well, which leads to

$$\hat{v}_i = \begin{cases} \check{v}_{n-1} & \text{when } i = n-1 \\ \check{v}_i + \frac{\hat{v}_{i+1}}{\check{a}_{(i+1)(i+1)}} & \text{when } i \in \mathbb{N}_1^{n-2}\,. \end{cases}$$

To find the unknowns in $\mathbf{u}$ we need to make the diagonal matrix $\hat{\mathbf{A}}$ to an identity matrix $\mathbf{I}$ such that $\bar{\mathbf{v}}$

$$\mathbf{Iu} = \bar{\mathbf{v}}\,,$$

which we achieve by dividing each row with the diagonal element a_{ii}. And the solution is given by

$$u_i = \bar{v}_i = \frac{\hat{v}_i}{\check{a}_{ii}}\,. \tag{21}$$

But we now see that when we calculate $\check{v}_i$ actually uses the previous solution u_{i+1}, and we can rewrite $\check{v}_i$ to

$$\hat{v}_i = \begin{cases} \check{v}_{n-1} & \text{when } i = n-1 \\ \check{v}_i + u_{i+1} & \text{when } i \in \mathbb{N}_1^{n-2}\,. \end{cases} \tag{22}$$

So what we need to calculate the solution in (21) is first to calculate $\check{a}_{ii}$ in (19), then $\check{v}_i$ in (20) followed by $\hat{v}_i$ (22). The calculation of $\check{a}_{ii}$ coefficient in (19) is actually independent of the input values v_i, and can therefore be precalculated. Which means that we need 2N FLOPS to calculate $\check{v}_i$ in (20), 1N FLOP to calculate u_i in (21) and 1N FLOP to calculate $\hat{v}_i$ in (22). Hence we need 4N FLOPS to find the solution u_i.

3 Implementation

3.1 Property class

I made myself a `Property` class to manage public variables of a class in a better way. The purpose of the `Property` class is to protect public variables of a class, such that the address of the variable can only be changed by the owner of the variable, but still read and write to variable is allowed outside the class. This means that the `Property` class can't be overwritten outside the class either. For the `Property` class to know how to do this it needs a owner type as a template argument in it's declaration, which is done by the template argument `C` in `template<PropertyType A, typename C, typename T, typename... L> class Property`, and the class `C` is a friend of the `Property` class, which enables the owner to get access to private and protected instances inside the `Property` class that no one else gets. Type of the variable is determined by the template argument `T` and the owner allowed typecast of the variable is determined by template argument `L` (which is a varadic template argument which can be of an arbitrary length). The access to the variable is controlled by the template argument `A` which is of the following type

enum class PropertyType

```
enum class PropertyType {
  ReadWrite,
  ReadOnly,
  WriteOnly
};
```

A variable that is protected by a `Property` class is accessed by read and write as if it's a regular variable, which is achieved by overloading the operators for typecasting and assignment. This is made possible by making `Property` inherent from `template<typename T, typename C, typename... L> class TypeCast<T,C,L...>`, which is again inherited by itself until all the typecasts are represented. However the compiler does not know what to do when a variable protected by a `Property` class is sent to a template function, unless you specify the template arguments explicitly when calling the function. This is due to that the it ambiguous which type is should cast to when it's not specified. Another way to resolve this to use the overloaded operator for &, which unwraps the variable from the property class by making a copy of it self.

The owner of a property variable may want to add user defined set and get function, to enable synchronization of data inside the owner class when a property variable is read or written to. This is made possible by using the classes `PropertyGet`, `PropertySet` or `PropertySetGet` as type to a `Property` class. This classes requires the use of a `Delegate` class which contains a function pointer an it's owner, which makes it easier to pass function pointers of class members, because the recipient does not need to know who owns the function pointer, it can just call it as a regular function.

In addition I made `ArrayLength` class that has the length of an array as a variable, and automatically reallocate memory and reinitialize it by the owners wishes. This class does not contain the array it self but rather a pointer to the array. Which enables the array by be declared in regular way in it's owner class, and one don't have to deal with extra overhead code to access it. But it makes life much easier when dealing with arrays that changes sizes, because one don't have to think about reallocation of memory, one just sets the size of the array.

This classes that I have described here are made to protected vital variables in a class that are made public, and ensure correct of the class use from outsiders by synchronize the variables. The implementation by the class of this protected variables is bit cumbersome, however when you have tested your class you can be a lot sure that outsiders can't case bugs inside your class. Which again makes it easier to locate bugs when you program, because you can exclude where the bugs may occur.

3.2 Heat equation solver

I made a my own class that solves the heat equation as described in section 2.3 with `template<typename T, unsigned int D> class HeatEquation`. The template argument `T` is the variable type that the calculation is done with, and the template argument `D` is a preparation to support arbitrary dimension of the heat equation. However this implementation only support `D=1` for the time being. `HeatEquation` inherits from `template<typename T, unsigned int D> class Boundary` which handles the boundary conditions. The solver of the `HeatEquation` supports arbitrarily θ values in the θ-rule, which is set by `HeatEquation::Theta` prior to calling the solver.

`HeatEquation::Solve()`

```
void Solve() {
  T* _u[D], *_v[D]; // Preparations
  T* u, *v;
  T alpha;
  int n[D+1], _n;
  n[0] = this->n[0];
  for(int i = 1; i <= D; i++)
  n[i] = this->n[i]-2;
  for(int i = 0; i < D; i++) {
    _v[i] = new T[n[i+1]+2];
    _v[i][0] = this->u[i][0];
    _v[i][n[i+1]+1] = this->u[i][n[i+1]+1];
    _u[i] = this->u[i];
  }

  if(theta == 1.0) { // Backward Euler Implicit
    for(int i = 1, j, k; i < n[0]; i++) {
      for(j = 0; j < D; j++) {

        // Enable faster accessing of arrays
        u = this->u[j];
        v = _v[j];
        alpha = this->alpha[j];
        _n = n[j+1];

        // Calculate eq_18 with theta=1
        v[1] = alpha*u[1] + u[0];
        for(k = 2; k < _n; k++)
          v[k] = alpha*u[k];
        v[_n] = alpha*u[_n] + u[_n+1];

        matrix[j]->Solve(&v[1], _n);  // Tridiagonal solver

        // Prepare for next timestep
        _v[j] = this->u[j];
        this->u[j] = v;
      }
    }
  } else if(theta) { // Theta Implicit
    for(int i = 1, j, k; i < n[0]; i++) {
      for(j = 0; j < D; j++) {

        // Enable faster accessing of arrays
        u = this->u[j];
```

```
        v = _v[j];
        alpha = this->alpha[j];
        _n = n[j+1];

        // Calculate eq_18
        v[1] = _1_theta*(u[2] - 2.0*u[1] + u[0]) + alpha*u[1] + u[0];
        for(k = 2; k < _n; k++)
          v[k] = _1_theta*(u[k+1] - 2.0*u[k] + u[k-1]) + alpha*u[k];
        v[_n] = _1_theta*(u[_n+1] - 2.0*u[_n] + u[_n-1]) + alpha*u[_n] + u[
            _n+1];

        matrix[j]->Solve(&v[1], _n);  // Tridiagonal solver

        // Prepare for next timestep
        _v[j] = this->u[j];
        this->u[j] = v;
      }
    }
  } else { // Forward Explicit
    for(int i = 1, j, k; i < n[0]; i++) {
      for(j = 0; j < D; j++) {

        // Enable faster accessing of arrays
        u = this->u[j];
        v = _v[j];
        alpha = this->alpha[j];
        _n = n[j+1];

        // Calculate eq_17
        for(k = 1; k <= _n; k++)
          v[k] = u[k] + alpha*(u[k+1] - 2*u[k] + u[k-1]);

        // Prepare for next timestep
        _v[j] = this->u[j];
        this->u[j] = v;
      }
    }
  }

  for(int i = 0; i < D; i++) // Clean up
    delete [] _v[i];
}
```

Number of timesteps are set by `HeatEquation::n[0]`, and number of grid points in x direction is set by `HeatEquation::n[1]`. The start and end time are set by `HeatEquation::lower[0]` and `HeatEquation::upper[0]` accordingly, and the lower and upper boundary of x are set by `HeatEquation::lower[1]` and `HeatEquation::upper[1]` accordingly. You are allowed full access to the array of $u(x,t)$ through `HeatEquation::u`, so you can set up the initial conditions prior to calling the solver, and you can read the result afterwards. The elements of `HeatEquation::u` are set to zero by calling `HeatEquation::Initialize`.

3.3 Tridiagonal solver

The tridiagonal solver used by the heat equation is implemented in `template<class T> class`

`Matrix<MatrixType::Tridiagonal_m1_C_m1, T>`, where the template argument `T` is the type of variable that the calculation is done by. This matrix store only a constant for the main diagonal of the matrix, since this elements are constant, and the matrix looks like $(-1, C, -1)$ for each row where C is located on the main diagonal and we have leading and succeeding zeros on the row. Which means that this matrix does not need any memory to represent it expect a constant C. However to enable the $4N$-solver we need to store an array of length N (the size of the tridiagonal matrix) for the precalculated values (19), which is named `Matrix<MatrixType::Tridiagonal_m1_C_m1, T>::factor`.

`Matrix<MatrixType::Tridiagonal_m1_C_m1, T>::Solve`

```
bool Solve(T* f, int n) {
  if(this->n < n)    // Checks if the matrix size has changed
    this->n = n;     // and if so new precalculated values are calculated

  n -= 1;
  int i = 0;
  while(i < n)
    f[i] += f[i]*factor[i++]; // eq_20
  while(i) {
    f[i] *= factor[i];        // eq_21
    f[i] += f[i--];           // eq_22
  }
  f[0] /= this->b;

  return true;
}
```

4 Result

Order of relative error of v to u is calculated by

$$\epsilon = \log_{10} \left| \frac{v-u}{u} \right| .$$

$n_x \backslash n_t$	201	20001	2000001
10	9e-06	0.000832	0.066476
100	6.8e-05	0.007114	0.700974
1000	0.000655	0.066341	6.76555

Table 4.1: *Calculation time for the Forward Euler scheme in seconds.*

$n_x \backslash n_t$	201	20001	2000001
10	2.4e-05	0.002679	0.180817
100	0.000224	0.023228	2.28261
1000	0.002245	0.228861	23.0031

Table 4.2: *Calculation time for the Backward Euler scheme in seconds.*

$n_x \backslash n_t$	201	20001	2000001
10	2.8e-05	0.003475	0.211697
100	0.000242	0.025092	2.46297
1000	0.002615	0.250891	24.9284

Table 4.3: *Calculation time for the Crank-Nicoloson scheme in seconds.*

$n_x \backslash n_t$	201	20001	2000001
10	4.09865	4.14892	4.14942
100	-1.22908	-0.36132	-0.356169
1000	inf	-1.64976	-1.57965

Table 4.4: *Order of relative error of Forward Euler scheme to the exact solution in* (8)*, for* $t = 0.01$*,* $u_0 = 1$*,* $d = 1$ *and* $D = 1$*. Relative error of calculated and exact value less than* 10^{-6} *are excluded.*

$n_x \backslash n_t$	201	20001	2000001
10	4.17117	4.14965	4.14943
100	-0.0643332	-0.352659	-0.356082
1000	-0.48892	-1.53688	-1.57856

Table 4.5: *Order of relative error of Backward Euler scheme to the exact solution in* (8)*, for* $t = 0.01$*,* $u_0 = 1$*,* $d = 1$ *and* $D = 1$*. Relative error of calculated and exact value less than* 10^{-6} *are excluded.*

$n_x \backslash n_t$	201	20001	2000001
10	4.13565	4.14929	4.14942
100	-0.444706	-0.356972	-0.356125
1000	-1.47976	-1.5896	-1.5791

Table 4.6: *Order of relative error of Crank-Nicoloson scheme to the exact solution in* (8)*, for* $t = 0.01$*,* $u_0 = 1$*,* $d = 1$ *and* $D = 1$*. Relative error of calculated and exact value less than* 10^{-6} *are excluded.*

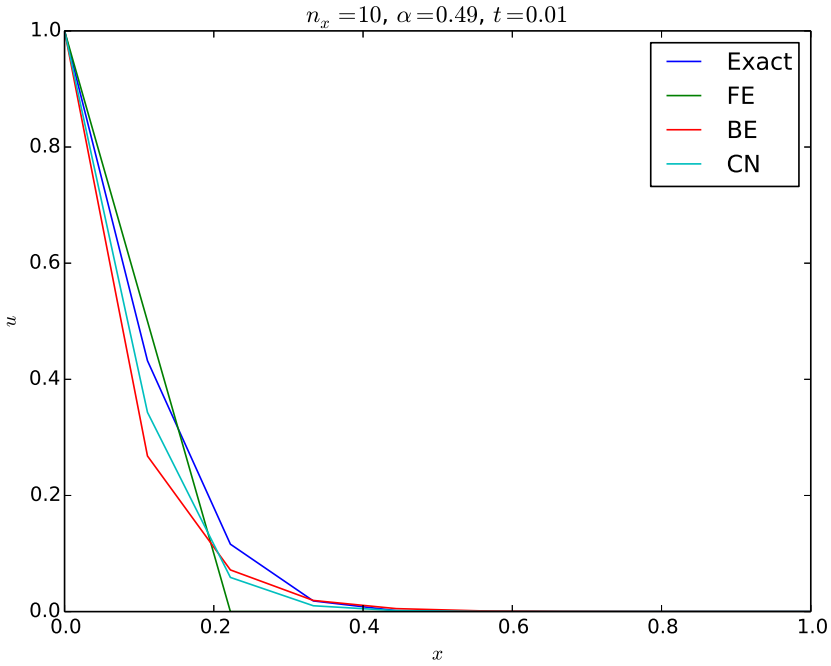

Figure 4.1: *Exact solution calculated* (8)*, FE is Forward Euler, BE is Back Euler and CN is Crank Nicholoson scheme. The following values is set as* $u_0 = 1$*,* $d = 1$ *and* $D = 1$*.*

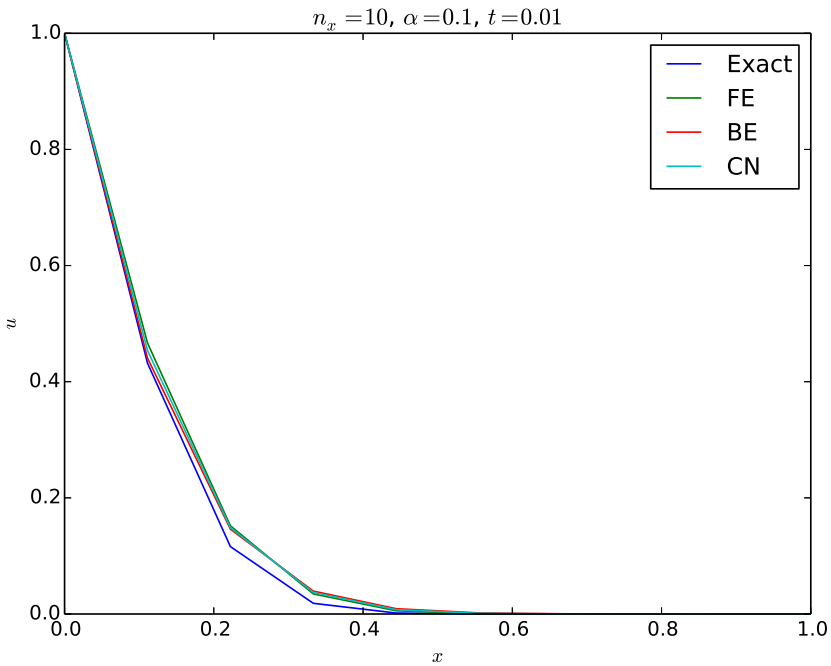

Figure 4.2: *Exact solution calculated* (8)*, FE is Forward Euler, BE is Back Euler and CN is Crank Nicholoson scheme. The following values is set as* $u_0 = 1$*,* $d = 1$ *and* $D = 1$*.*

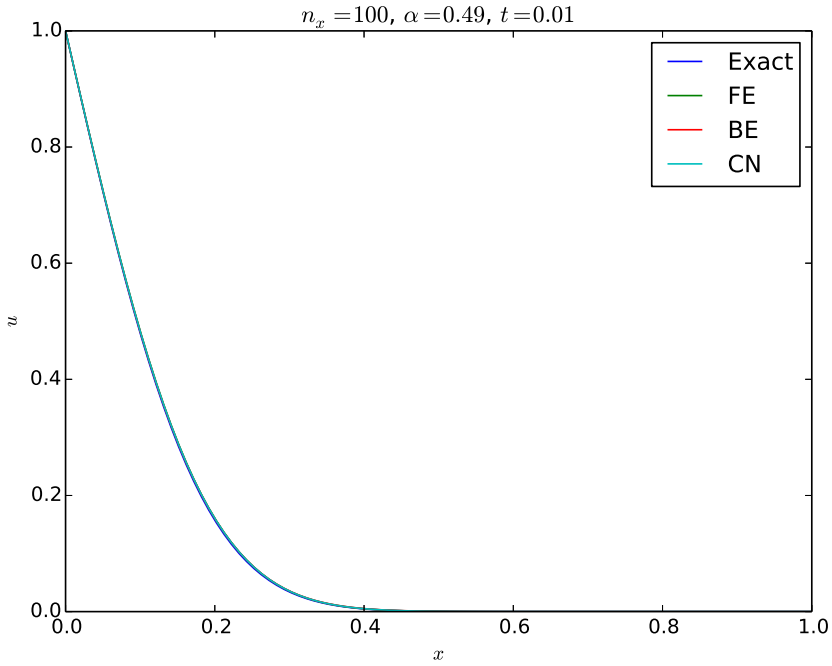

Figure 4.3: *Exact solution calculated* (8)*, FE is Forward Euler, BE is Back Euler and CN is Crank Nicholoson scheme. The following values is set as* $u_0 = 1$*,* $d = 1$ *and* $D = 1$*.*

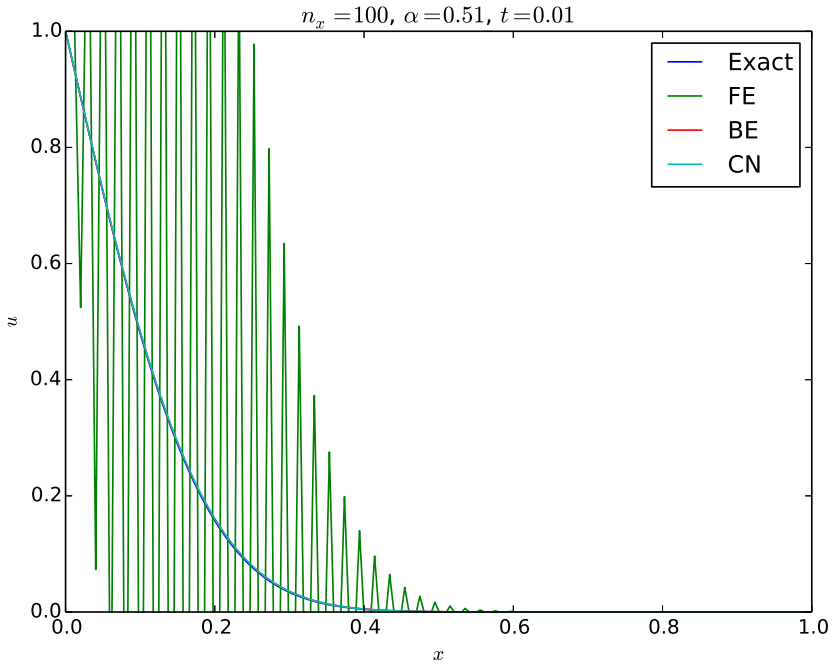

Figure 4.4: *Exact solution calculated* (8)*, FE is Forward Euler, BE is Back Euler and CN is Crank Nicholoson scheme. The following values is set as* $u_0 = 1$*,* $d = 1$ *and* $D = 1$*.*

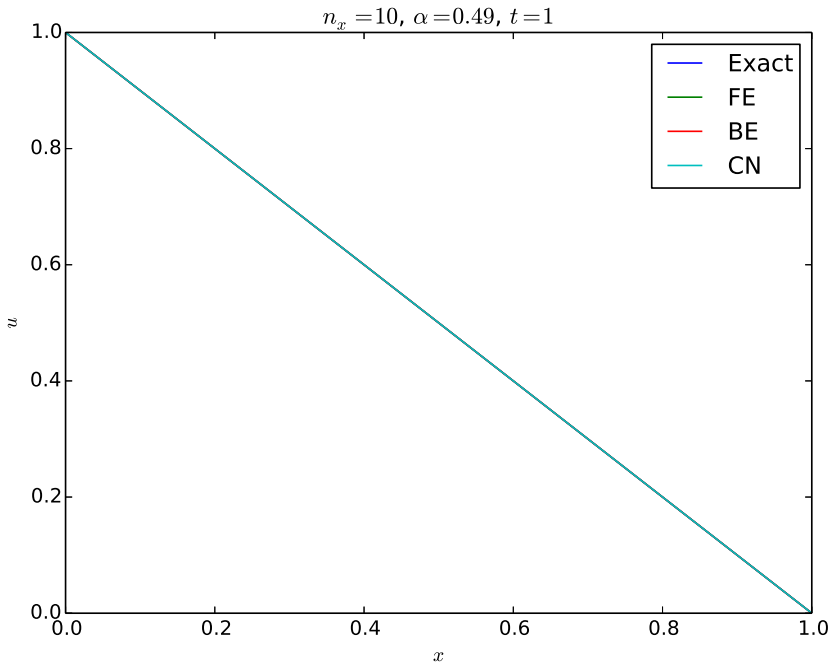

Figure 4.5: *Exact solution calculated* (8)*, FE is Forward Euler, BE is Back Euler and CN is Crank Nicholoson scheme. The following values is set as* $u_0 = 1$*,* $d = 1$ *and* $D = 1$.

5 Conclusion

We see from the results that the different solvers that I made for the heat equation gives good approximation to the analytical solution in (8). However if we look at the tables Table 4.4, Table 4.5 and Table 4.6 the Crank-Nicoloson scheme is the most stable, where the solution converges the fastest. This is due that Crank-Nicoloson scheme has second order local truncation error in both time and space, whereas the the Euler schemes have only first order in time and second order in space. The Backward Euler scheme is also stable for all α values, as for Crank-Nicoloson. But the Forward Euler scheme is only stable for $\alpha < \frac{1}{2}$, which I have demonstrated in Figure 4.3 and Figure 4.4. The different solvers converges faster to the stationary solution i Figure 4.5 than a more dynamic solution in Figure 4.1.

The Forward Euler scheme is about 3.5 times faster the implicit scheme, seen in Table 4.1, Table 4.2 and Table 4.3. This is not in agreement with number of FLOPS needed to solve with these schemes. The Forward and Backward scheme we found to both use $5N$ FLOPS for each timestep, but still the Backward Euler scheme is 3.5 times slower than the Forward Euler scheme. This is due to the tridiagonal matrix solver. Even though the tridiagonal matrix solver is taking into account in the number of FLOPS, it seems to run slower because it accesses two different arrays at the same instances. This also explains why the Crank-Nicoloson scheme is almost as fast as the Backward Euler scheme, even though the Crank-Nicoloson scheme use twice as many FLOPS.

We can also see from Table 4.4, Table 4.5 and Table 4.6 that if we want to have good approximation for small values for the concentration $u(x,t)$, we need to increase the number of grid points, where the number timestep does not affect this significantly. The error in these tables are dominated by the values close to zero of $u(x,t)$.

6 Attachments

The files produced in working with this project can be found at
https://github.com/Eimund/UiO/tree/master/FYS4150/Project%204

The source files developed are

1. Array.h
2. Boundary.h
3. Delegate.h
4. HeatEquation.h
5. Matrix.h
6. Property.h
7. Type.h
8. project4.cpp

7 Resources

1. QT Creator 5.3.1 with C11
2. Eclipse Standard/SDK - Version: Luna Release (4.4.0) with PyDev for Python
3. Ubuntu 14.04.1 LTS
4. ThinkPad W540 P/N: 20BG0042MN with 32 GB RAM

References

[1] Morten Hjorth-Jensen, *FYS4150 - Project 4* - *Diffusion of neurotransmitters in the synaptic cleft*, University of Oslo, 2014

[2] Morten Hjorth-Jensen, *Computational Physics - Lecture Notes Fall 2014*, University of Oslo, 2014

[3] http://en.wikipedia.org/wiki/Diffusion_equation

[4] http://en.wikipedia.org/wiki/Heat_equation

[5] http://en.wikipedia.org/wiki/Dirichlet_boundary_condition

[6] http://en.wikipedia.org/wiki/Integration_by_parts

[7] http://en.wikipedia.org/wiki/Taylor_series

[8] http://en.wikipedia.org/wiki/Euler_method

[9] http://en.wikipedia.org/wiki/Backward_Euler_method

[10] http://en.wikipedia.org/wiki/Crank%E2%80%93Nicolson_method

[11] http://en.wikipedia.org/wiki/Gaussian_elimination