Project 5, FYS4150

Candidate 55

December 9, 2012

Contents

1 About the problem

In this project we will look at the transportation of neurotransmitter molecules across the so called synaptic cleft separating a brain cell and a target cell. This is a common way of communication between cells in the brain and it is known as chemical synapse. After an "action-potential" is recieved (in the axon terminal) vesicles inside the axon terminal merge with the presynaptic membrane, releasing the neurotransmitter molecules into the synaptic cleft. A vesicle is a kind of "bubble" inside a cell. Here we are talking about vesicles containing neurotransmitter moleules located in the axon (or nerve fibers) of a brain cell. The molecules then diffuse across the synaptic cleft, and are picked up by receptors on the target cell (or the postsynaptic side of the synaptic cleft if you want). We want to modell this particular event using a continuum modell, and the mathematical expression is the diffusion equation.

$$\frac{\partial u}{\partial t} = D\nabla^2 u$$

Where D is a diffusion coefficient. An illustrative figure of the problem can be found in figure (1).

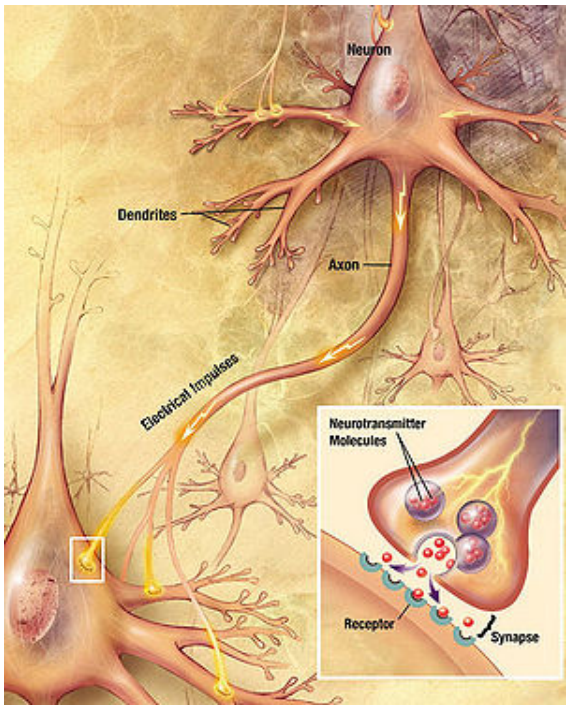

Figure 1: Illustration of communication between two brain cells and a close-up of the synaptic cleft separating the axon (presynaptic side) and the target cell (postsynaptic side). (From wikipedia search Chemical synapse)

We will assume that the synaptic cleft is of roughly equal width, and that the area of the synaptic cleft is large compared to its width which typically is around 20nm. The concentration of neurotransmitters therefore only vary in one direction, from presynaptic to postsynaptic side. The diffusion equation then reduces to

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$$

which we can rescale to be dimensionless by introducing $x = \alpha\tilde{x}$

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\alpha^2 \partial \tilde{x}^2}$$

and define $\alpha^2 = D$; $\tilde{x} = x$ giving us the diffusion equation in its simplest form.

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

We will say that at some time $t = 0$ the "action potential" is recieved, and the vesicles merge with the presynaptic membrane meaning that the initial distribution of neurotransmitters is $u(x, 0) = \delta(x)$, that is 1 at $x = 0$ and 0 everywhere else. Furthermore we say that the neurotransmitters which reach the postsynaptic membrane (or the receptors) are removed from the synaptic cleft (and our system). This means that at the far side ($x = d$) $u(d, t) = 0$. We are now ready to solve the equation, see sections 2 and 3.

We will also extend the diffusion equation to two spatial dimensions, also using dimensionless variables. The boundary and initial conditions of the two dimensional equation are quite different from the one dimensional case. We use $x, y \in [0, 1]$,

$$u(x, y, 0) = (1 - y)e^x$$

and

$$\begin{aligned} u(0, y, t) &= (1 - y)e^t \\ u(1, y, t) &= (1 - y)e^{1+t} \\ u(x, 0, t) &= e^{x+t} \\ u(x, 1, t) &= 0 \end{aligned}$$

1.1 Notation

For finite differences I will use the the notation

$$u(t_n, x_i, y_j) = u^n_{i,j}$$

where $t_n = t_0 + n \cdot \Delta t$, $x_i = x_0 + i \cdot \Delta x$ and $y_j = y_0 + j \cdot \Delta y$. Thus it is important not to confuse u^n (u to the power of n) with $u^n_{i,j}$ (u evaluated at timestep n and position i,j). I will at some points use the matrix B to simplify writing. B is defined to be

$$\mathbf{B} = \begin{pmatrix} 2 & -1 & 0 & \dots & & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & & 0 \\ 0 & \ddots & \ddots & \ddots & & & 0 \\ 0 & \dots & & 0 & -1 & 2 & -1 \\ 0 & \dots & & & 0 & -1 & 2 \end{pmatrix}$$

2 The algorithm

We will solve the 1+1 dimensional difusion equation by three different finite differnce schemes in this project. Using the standard approximation of the second derivative in space, we use successively more elaborate approximations to the time derivative.

2.1 Forward Euler

Starting off with the Forward Euler (FE) approximation we get the following scheme

$$\frac{u^{n+1}_i - u^n_i}{\Delta t} = \frac{u^n_{i+1} - 2u^n_i + u^n_{i-1}}{\Delta x^2}$$

$$u^{n+1}_i = \frac{\Delta t}{\Delta x^2}\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right) + u^n_i \tag{1}$$

So to solve the equation all we have to do is loop over the two variables and we are done.

```
for n = 0,1, ... , N
    for i = 0,1, ... , N_x
        u_new[i] = dtdx2*(u_prev[i+1]-2*u_prev[i] + u_prev[i-1]) + u_prev[i];
    u_prev = u_new;
end;
```

A discussion of stability and the error in this scheme can be found in the "Stability and precision" section.

2.2 Backward Euler

The Backward Euler (BE) approximation gives us a slightly more elaborate scheme seeing at it is an implicit one. The discretization gives

$$\frac{u^n_i - u^{n-1}_i}{\Delta t} = \frac{u^n_{i+1} - 2u^n_i + u^n_{i-1}}{\Delta x^2}$$

$$u^n_i - \frac{\Delta t}{\Delta x^2}\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right) = u^{n-1}_i$$

$$u^n_i\left(1 + 2\frac{\Delta t}{\Delta x^2}\right) - u^n_{i-1}\frac{\Delta t}{\Delta x^2} - u^n_{n+1}\frac{\Delta t}{\Delta x^2} = u^{n-1}_i$$

If we insert for a few steps we see that this takes the form of

$$\begin{pmatrix} \left(1+2\frac{\Delta t}{\Delta x^2}\right) & -\frac{\Delta t}{\Delta x^2} & 0 & \dots & & 0 & 0 \\ -\frac{\Delta t}{\Delta x^2} & \left(1+2\frac{\Delta t}{\Delta x^2}\right) & -\frac{\Delta t}{\Delta x^2} & 0 & \dots & & 0 \\ 0 & \ddots & \ddots & \ddots & 0 & \dots & 0 \\ 0 & \dots & & 0 & -\frac{\Delta t}{\Delta x^2} & \left(1+2\frac{\Delta t}{\Delta x^2}\right) & -\frac{\Delta t}{\Delta x^2} \\ 0 & \dots & & & 0 & -\frac{\Delta t}{\Delta x^2} & \left(1+2\frac{\Delta t}{\Delta x^2}\right) \end{pmatrix} \mathbf{u}^n = \mathbf{u}^{n-1} \quad (2)$$

$$\mathbf{A}\mathbf{u}^n = \mathbf{u}^{\mathbf{n-1}}$$

So for each timestep we need to multiply the inverse of $\mathbf{A}$ with a vector $\mathbf{u}^{n-1}$ containig the solution at the previous timestep. Looking a bit closer at equation (2) we realize two things. First of all, the matrix $\mathbf{A}$ is constant throughout the computation so we can get away with invertig it once. Second, and perhaps more important, we allready have a very efficient solver for this kind of problem from project 1 where we solved a system of a tridiagonal matrix times an unknown vector equal to a known vector. The only difference is that we must solve this system for each timestep. the algorithm for doing this is to do a specialized gaussian elimination utilizing the sparsity of the matirx A. Note that the tirdiagonal solver does not like boundary conditions so theese need to be excluded from the

A discussion of stability and the error in this scheme can be found in the "Stability and precision" section.

2.3 Crank Nicolson

Finally, we can use the Crank Nicolson approximation of the time derivative which is a special case of the so called theta -rule[1] with $\theta = 0.5$.

$$\begin{aligned} \frac{u_i^n - u_i^{n-1}}{\Delta t} &= \frac{1}{2\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \frac{1}{2\Delta x^2}\left(u_{i+1}^{n-1} - 2u_i^{n-1} + u_{i-1}^{n-1}\right) \\ u_i^n - \frac{\Delta t}{2\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) &= u_i^{n-1} + \frac{\Delta t}{2\Delta x^2}\left(u_{i+1}^{n-1} - 2u_i^{n-1} + u_{i-1}^{n-1}\right) \\ u_i^n(2+2C) - Cu_{i+1}^n - Cu_{i-1}^n &= u_i^{n-1}(2-2C) + Cu_{i+1}^{n-1} + Cu_{i-1}^{n-1} \end{aligned}$$

where $C = \frac{\Delta t}{\Delta x^2}$. This can also be expressed as a linear algebra problem

$$(2\mathbf{I} + C\mathbf{B})\mathbf{u}^n = (2\mathbf{I} - C\mathbf{B})\mathbf{u}^{n-1} \quad (3)$$

Again we observe that we will need to invert a matrix, but this time we will also need to do a matrix-vector multiplication.

$$\begin{aligned} (2\mathbf{I} + C\mathbf{B})\mathbf{u}^n = (2\mathbf{I} - C\mathbf{B})\mathbf{u}^{n-1} &= \tilde{\mathbf{u}}^{n-1} \\ \mathbf{u}^n &= (2\mathbf{I} + C\mathbf{B})^{-1}\tilde{\mathbf{u}}^{n-1} \end{aligned}$$

This is the same problem as we had in the BE case, only with a modified right-hand side

2.4 2+1 dimensional diffusion equation

The Jacobi algorithm for solving linear systems is an iterative scheme very well suited for solving the Poisson and Laplace equations in 2 or 3 dimensions. If we breefly look at the discretization of the Laplace equation in 2D (4) we see that there is no obvious way to solve for a new point since we do not know 3 out of 5 points needed to calculate a new one. The solution is then (since there is no initial condition either) to start out with an educated guess, and use the scheeme (5) point by point on the entire grid several times until the solution (hopefully) converges.

$$\begin{aligned} \nabla^2 u(x,y) &= 0 \\ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= 0 \end{aligned}$$

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = 0 \quad (4)$$

[1] See appendix

$$u_{i,j} = \frac{1}{4}\Big(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\Big) \tag{5}$$

Where we have assumed that $\Delta x = \Delta y$. Seeing as we have a time dependence in the diffusion equation, we also have an initial condition on our system. Therefore we allready know all values at previous timesteps (since the sceme is recursive), and we do not have to start with a quess of what the solution might look like. In stead, but still with the mentality of the Jacobi algorithm of using the previous value to get the new one, we can simply derive an explicit scheme by inserting our approximate derivatives. For example if we approximate the time derivative using the FE method we get

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = \frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}$$

$$u_{i,j}^{n+1} = \frac{\Delta t}{\Delta x^2}\big(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}\big) + \frac{\Delta t}{\Delta y^2}\big(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{ij-1}^{n}\big) + u_{i,j}^{n}$$

This is the same scheme as in 1 only with one more dimension. Notice that $u_{i,j}^{n+1}$ only appears once, and so all values of $u_{i,j}^{n}$ are known from the previous timestep.
As we will se in the "stability and precision" section the centered difference or "Leap Frog" scheme has a better truncation error with respect to time than the FE scheme, and this will therefore (most likely) yield better results. The Leap Frog scheme is

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n-1}}{2\Delta t} = \frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}$$

$$u_{i,j}^{n+1} = \frac{\Delta t}{2\Delta x^2}\big(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}\big) + \frac{\Delta t}{2\Delta y^2}\big(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{ij-1}^{n}\big) + u_{i,j}^{n-1}$$

We could also make an explicit scheme by the mentality in the Gauss-Seidel algorithm. That is to use some of the new values to get a new value. In one dimension this is known as the Euler-Chromer method and is known to be quite good for its simplicity. If we look closer at the FE or Leap Frog scheme we notice that we allready know the values $u_{i-1,j}^{n+1}$ and $u_{i,j-1}^{n+1}$. Using these values might give us better accuracy (at least compared to the FE case). This is stictly a spontanious guess, I do not have anything more than I have explained to back up this. I shall comment on the results from the comparison with the FE scheme in section 7. The explicit scheme for the two dimensional Euler Chromer scheme is

$$u_{i,j}^{n+1} = \frac{\Delta t}{\Delta x^2}\big(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n+1}\big) + \frac{\Delta t}{\Delta y^2}\big(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{ij-1}^{n+1}\big) + u_{i,j}^{n}$$

3 Analytic solution

As a comparison to the nummeric solution of our problem we can find the analytic solution to this problem, using the dimensionless variables from section 1 as follows.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad u(x,0) = u(d,t) = 0 \tag{6}$$

$$x \in [0,d], \quad d = u(0,t) = 1 \tag{7}$$

We see right away that the boundary $x = 0$ could give us some problems, so we start off with a small trick

$$u(x,t) = v(x,t) + u_s(x)$$

where the $u_s = 1 - x$ term is the steady-state solution to equation 6. This trick leaves us with new boundary conditions on $v(x,t)$

$$u(0,t) = v(0,t) + u_s(0,t) = 1, \;\; u_s(0,t) = 1 \implies v(0,t) = 0$$

which makes the whole procedure much simpler. We now assume that $u(x,t)$ can be separated into factors

$$v(x,t) = F(x)G(t) \implies \frac{\partial v}{\partial t} = \frac{\partial^2 v}{\partial x^2} \to F(x)\frac{\partial G}{\partial t} = G(t)\frac{\partial^2 F}{\partial x^2}$$

$$\frac{1}{G(t)}\frac{\partial G}{\partial t} = -k^2 = \frac{1}{F(x)}\frac{\partial^2 F}{\partial x^2}$$

We start with the simplest of the equations which is the time dependence

$$\frac{1}{G(t)}\frac{\partial G}{\partial t} = -k^2$$

$$G(t) = Ce^{-k^2 t}$$

and leave it like this for now. The x-dependent equation is somewhat more complicated

$$\begin{aligned}
\frac{1}{F(x)}\frac{\partial^2 F}{\partial x^2} &= -k^2 \\
F(x) &= A\sin(kx) + B\cos(kx) \\
F(0) &= A\sin(0) + B\cos(0) = 0 \implies B = 0 \\
F(d) &= A\sin(kd) = 0 \implies k = \frac{m\pi}{d} = \pi m, \;\; A = A_m
\end{aligned}$$

we now combine all the equations to determine $u(x,t)$

$$\begin{aligned}
u(x,t) &= 1 - x + \sum_{m=1}^{\infty} B_m e^{-(m\pi)^2 t}\sin(m\pi x) \\
u(x,0) &= 1 - x + \sum_{m=1}^{\infty} B_m \sin(m\pi x) \\
\implies &\int_0^1 \sin(m\pi x)\sin(n\pi x)dx = \delta_{mn} = \int_0^1 (x-1)\sin(m\pi x)dx = -\frac{2}{m\pi} = B_m
\end{aligned}$$

This gives us the full analytic solution

$$u(x,t) = 1 - x - \frac{2}{\pi}\sum_{m=1}^{\infty}\frac{1}{m}e^{-(m\pi)^2 t}\sin(m\pi x) \tag{8}$$

which satisfies all initial and boundary conditions.

4 Results/Verification

From section 3 we have the analytic solution to our problem, and we can use this to verify our numerical schemes. Now we know from section 5 that both the FE and the BE schemes have errors going like $\mathcal{O}(\Delta t)$, and that the FE scheme has a stability criterion of $\Delta t \leq \frac{\Delta x^2}{2}$. From experience I know that using exactly this criterion can give some oscillations in the solution, so we will use half of this and we will use the same for all the schemes so that we can easily compare results. In the analytic solution we have an infinite sum, which we of course have to cut at some point. Looking at our solution (8) we see that $t = 0$ will have the slowest convergence because there will be no additional dampening in the exponential factor. From figure 2 we see that there is very little change in the solution after 25 terms. It will then be very safe to terminate the sum after some 100-200 terms to make sure no "invisible" roundoffs are made. We also notice that the analytic solution cannot reproduce the perfect delta function that is the initial condition. This is expected from Fourier series since this is a rather nasty discontinuity.

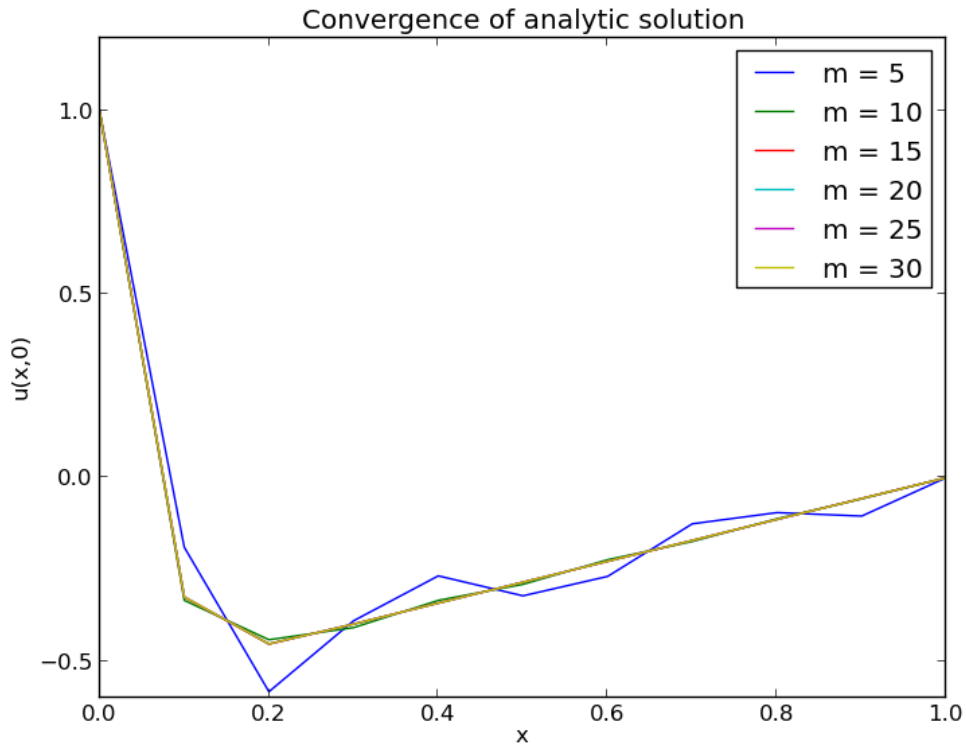

Figure 2: Plot of the analytic solution at $t = 0$ while adding more terms.

The errors we get from the different schemes have been visualized in figures (3) and (4). As mentioned the error should go like $\mathcal{O}(\Delta t)$.

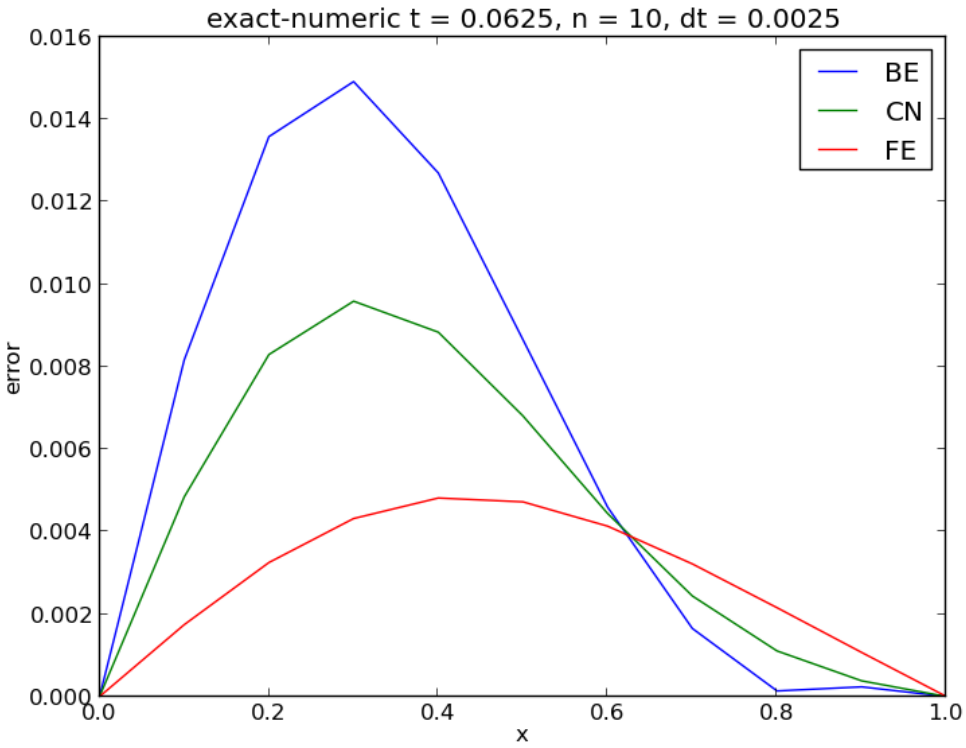

Figure 3: The absolute error for $\Delta x = {}^{1}/_{10}$ and Δt restricted by the stability criterion on the FE scheme.

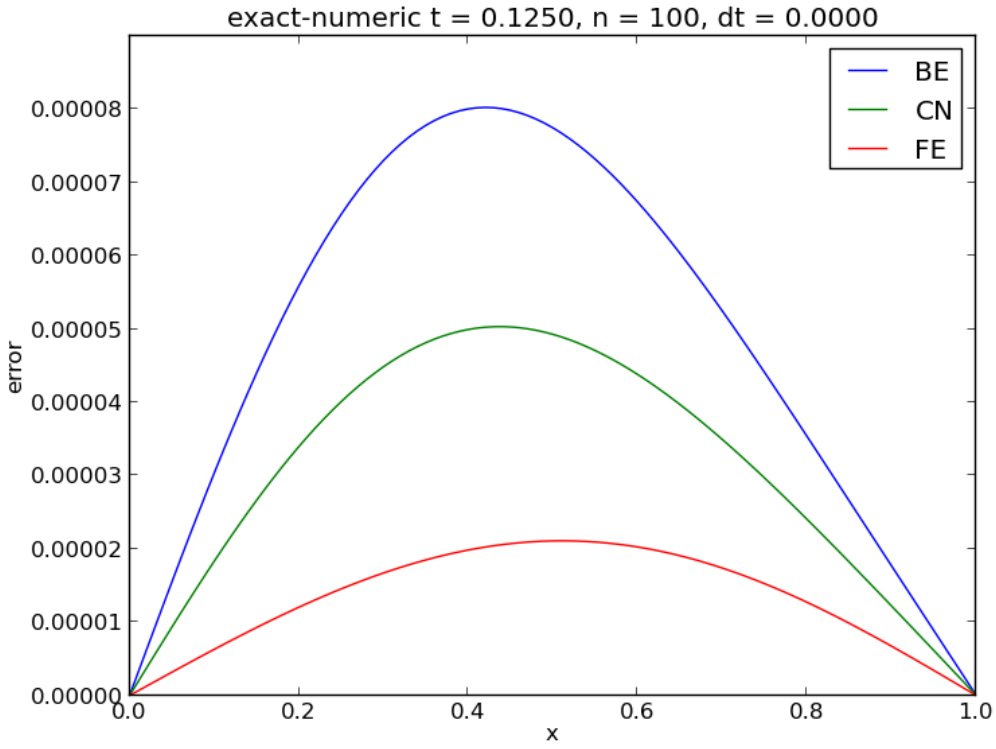

Figure 4: The absolute error for $\Delta x = {}^{1}/_{100}$ and Δt restricted by the stability criterion on the FE scheme..

We notice that contrary to the results of the truncation error the FE scheme gives the best results. As another verification we could let the simulation run untill the steady state is reached. Since the steadu state is a first order polynomial, this should be represented to machine precision, which means that the error should go to 0 as the number of time-steps gets large. However, as is also expected in a diffusion equation, the convergence is very slow after the simulation has run for a while. we can see the results of this experiment in figures (5) and (6).

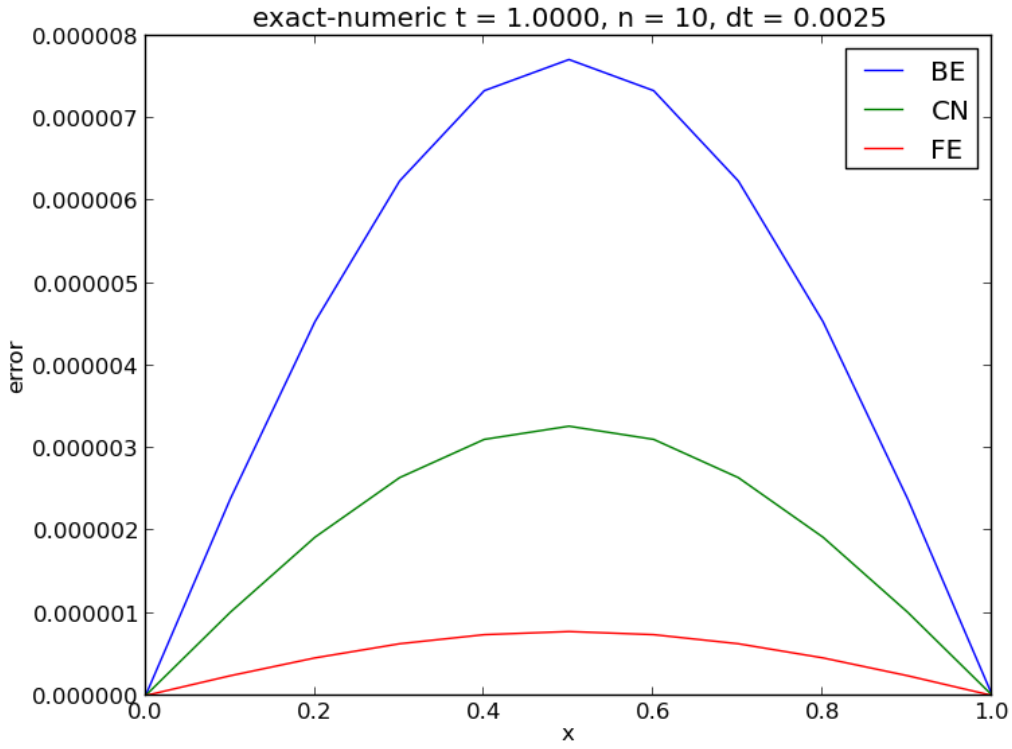

Figure 5: The absolute error for $\Delta x = {}^{1}/_{100}$ and Δt restricted by the stability criterion on the FE scheme.

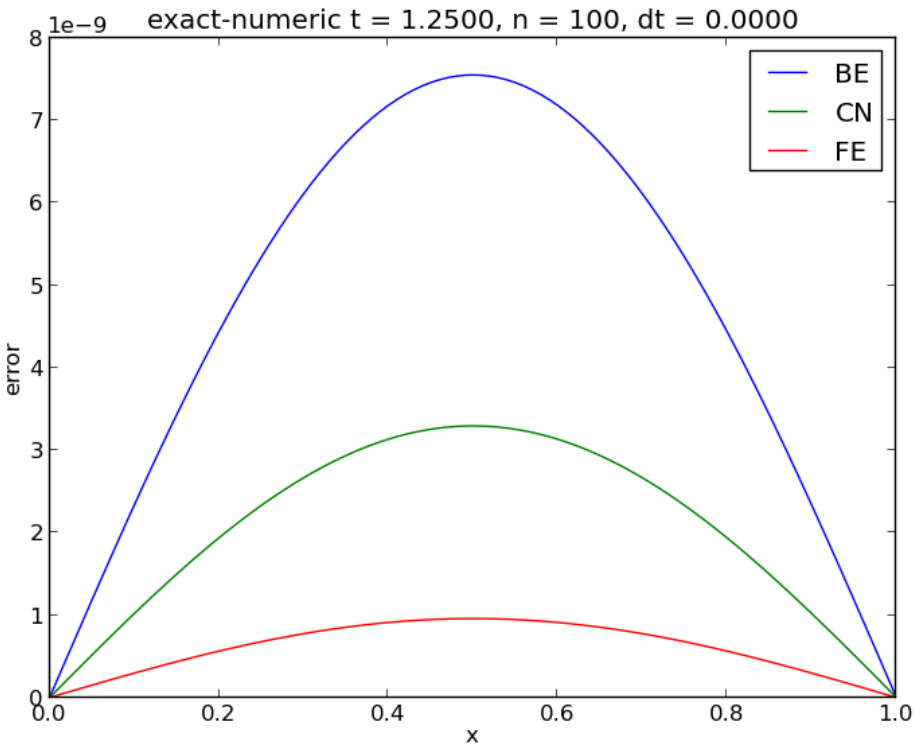

Figure 6: The absolute error for $\Delta x = {}^{1}/_{100}$ and Δt restricted by the stability criterion on the FE scheme.

For the two-dimensional case we also have an analytic solution which we can compare our results with. We will use the absolute error in this verification as well, and make a plot over the entire area as before. As

we discussed in section 2 there are many different ways to simulate the diffusion equation in more than one dimesion as well. The simplest of which is of course the Forward Euler scheme, which is one of the schemes we have implemented. The results of the same verification as we did in 1D is illustrated in figures (7) and (8)

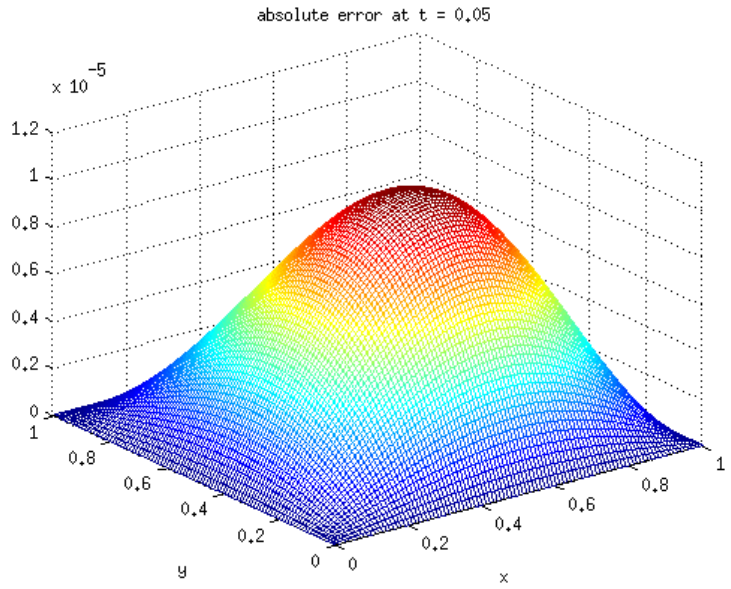

Figure 7: The absolute error for $\Delta x = {}^{1}/_{100}$ and Δt restricted by the stability criterion on the FE scheme in two spatial dimensons

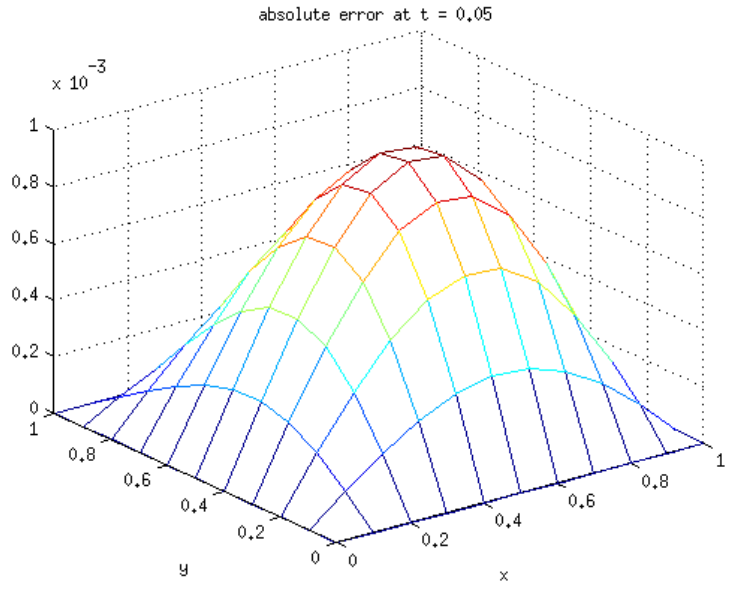

Figure 8: The absolute error for $\Delta x = {}^{1}/_{10}$ and Δt restricted by the stability criterion on the FE scheme in two spatial dimensons

The "modified" Euler Chromer scheme has also been implemented, and run through the same test as the FE scheme. The error has been measured for the same time as for the FE scheme for a thorogh comparison.

5 Stability and precision

To get a feeling of how good the numerical schemes are we will analyze their errors and stability criteria individually.

5.1 Forward Euler

Let us first look at the truncation error of this scheme. If we do a Taylor expansion of $u(x+\Delta x,t)$, $u(x-\Delta x,t)$ and $u(x,t+\Delta t)$, assuming $\Delta x, \Delta t \to 0$ we get the following

$$u(x+\Delta x,t) = u(x,t) + \frac{\partial u(x,t)}{\partial x}\Delta x + \frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2 + \mathcal{O}(\Delta x^3)$$
$$u(x-\Delta x,t) = u(x,t) - \frac{\partial u(x,t)}{\partial x}\Delta x + \frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2 + \mathcal{O}(\Delta x^3)$$
$$u(x,t+\Delta t) = u(x,t) + \frac{\partial u(x,t)}{\partial t}\Delta t + \mathcal{O}(\Delta t^2)$$

Theese are the local errors, meaning that our complete scheme gives the error of

$$\frac{u^{n+1}-u^n}{\Delta t} \approx \frac{\partial u(x,t)}{\partial t} + \mathcal{O}(\Delta t)$$
$$\frac{u_{i+1}-2u_i+u_{i-1}}{\Delta x^2} \approx \frac{\partial^2 u(x,t)}{\partial x^2} + \mathcal{O}(\Delta x^2)$$

Let us also look at the stability of the scheme, but by in slightly different way than what has been done in the rest of the course called Von Neumann analysis. Let us assume that the slution of the diffusion equation takes the form

$$u(x,t) = A^n e^{ikq\Delta x} \tag{9}$$

Where A^n is a damping factor which is time dependent. If we insert this expression 9 into the scheme 1 we get

$$A^n e^{ikq\Delta x}(A-1) = CA^n e^{ikq\Delta x}(e^{i(q+1)k\Delta x} - 2e^{iqk\Delta x} + e^{i(q-1)k\Delta x})$$
$$A - 1 = C(e^{ik\Delta x} - 2 + e^{-ik\Delta x})$$
$$A = 1 - 2C(cos(k\Delta x) - 1) = 1 - 4C\sin^2(k\Delta x/2)$$

Note that x_i has been replaced by x_q. This means that the final solution is on the form

$$u_q^n = \left(1 - 4C\sin^2(k\Delta x/2)\right)^n e^{ikq\Delta x}$$

It is obvious that we need $-1 \leq A \leq 1$ to avoid divergence. $A \leq 1 \implies C \geq 0$ is not of to much interest, however if we insert for

$$-1 \leq A \implies 1 - 4C\sin^2(k\Delta x/2) \geq -1$$
$$C \leq \frac{1}{2\sin^2(k\Delta x/2)}$$

and since $max(\sin^2(k\Delta x/2)) = 1$ we are left with

$$C = \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2} \tag{10}$$

5.2 Backward Euler

The truncation error of the BE scheme is very similar to the FE case. In fact it is obvious that the spacial part is identical, so we only need to redo the time dependence.

$$u(x,t-\Delta t) = u(x,t) - \frac{\partial u(x,t)}{\partial t}\Delta t + \mathcal{O}(\Delta t^2)$$

And so the complete scheme gives an error of

$$\frac{u^n-u^{n-1}}{\Delta t} \approx \frac{\partial u(x,t)}{\partial t} + \mathcal{O}(\Delta t)$$
$$\frac{u_{i+1}-2u_i+u_{i-1}}{\Delta x^2} \approx \frac{\partial^2 u(x,t)}{\partial x^2} + \mathcal{O}(\Delta x^2)$$

We can investigate the stability of the BE scheme in the same way as we did for the FE scheme. Assume again that we have a general solution on the form 9. Insert it in 2 as follows

$$A^n e^{ikq\Delta x}(1 - A^{-1}) = CA^n\left(e^{ik(q+1)\Delta x} - 2e^{ikq\Delta x} + e^{ik(q-1)\Delta x}\right)$$
$$A^{-1} = 1 - 4C\sin^2(k\Delta x/2)$$
$$A = \frac{1}{1 - 4C\sin^2(k\Delta x/2)}$$

giving us the general term

$$u_q^n = \left(\frac{1}{1 - 4C\sin^2(k\Delta x/2)}\right)^n e^{ikq\Delta x}$$

but now we have that $0 \leq A \leq 1$.

$$\frac{1}{1 - 4C\sin^2(k\Delta x/2)} \leq 1$$
$$1 \leq 1 - 4C\sin^2(k\Delta x/2) \implies C \geq 0$$

The other limitation ($A \geq 0$) gives us nothing and thus the scheme is stable for all values of Δt and Δx.

5.3 Crank Nicolson

The truncation error of the CN scheme is slightly more intricate to calculate because we need to Taylor expand around $t' = t + \Delta t/2$

$$\begin{aligned}
u(x+\Delta x, t+\Delta t) &= u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial x}\Delta t/2 + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2/4 \\
&\quad + \frac{\partial^2 u(x,t')}{\partial x \partial t}\Delta x \Delta t/2 + \mathcal{O}(\Delta x^3) \\
u(x-\Delta x, t+\Delta t) &= u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial x}\Delta t/2 + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2/4 \\
&\quad - \frac{\partial^2 u(x,t')}{\partial x \partial t}\Delta x \Delta t/2 + \mathcal{O}(\Delta x^3) \\
u(x+\Delta x, t) &= u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial x}\Delta t/2 + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2/4 \\
&\quad - \frac{\partial^2 u(x,t')}{\partial x \partial t}\Delta x \Delta t/2 + \mathcal{O}(\Delta x^3) \\
u(x-\Delta x, t) &= u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial x}\Delta t/2 + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2/4 \\
&\quad + \frac{\partial^2 u(x,t')}{\partial x \partial t}\Delta x \Delta t/2 + \mathcal{O}(\Delta x^3) \\
u(x, t+\Delta t) &= u(x,t') + \frac{\partial u(x,t')}{\partial t}\Delta t/2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2/4 + \mathcal{O}(\Delta t^3) \\
u(x,t) &= u(x,t') - \frac{\partial u(x,t')}{\partial t}\Delta t/2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2/4 + \mathcal{O}(\Delta t^3)
\end{aligned}$$

Which means that the error of the whole scheme scales like

$$\frac{\partial u(x,t')}{\partial t} \approx \frac{\partial u(x,t')}{\partial t} + \mathcal{O}(\Delta t^2)$$
$$\frac{\partial^2 u(x,t')}{\partial x^2} \approx \frac{\partial^2 u(x,t')}{\partial x^2} + \mathcal{O}(\Delta x^2)$$

We can check the stability of the CN scheme in the same way that we checked it for FE and BE schemes by assuming the same solution 9 as we did for the FE and BE case, and insert it in the CN scheme.

$$\begin{aligned}
A^n e^{ikq\Delta x}(1 - A^{-1}) &= A^n \frac{C}{2}\left(e^{i(q+1)k\Delta x} - 2e^{iqk\Delta x} + e^{i(q-1)k\Delta x}\right) + A^{n-1}\frac{C}{2}\left(e^{i(q+1)k\Delta x} - 2e^{iqk\Delta x} + e^{i(q-1)k\Delta x}\right) \\
(1 - A^{-1}) &= \frac{C}{2}\left(-4\sin^2(k\Delta x/2) - 4A^{-1}\sin^2(k\Delta x/2)\right) \\
1 + 2C\sin^2(k\Delta x/2) &= A^{-1}\left(1 - 2C\sin^2(k\Delta x/2)\right) \\
A &= \frac{1 - 2C\sin^2(k\Delta x/2)}{1 + 2C\sin^2(k\Delta x/2)}
\end{aligned}$$

And the limitations are $-1 \leq A \leq 1$

$$\frac{1 - 2C\sin^2(k\Delta x/2)}{1 + 2C\sin^2(k\Delta x/2)} \leq 1$$
$$1 - 2C\sin^2(k\Delta x/2) \leq 1 + 2C\sin^2(k\Delta x/2)$$
$$\implies C \geq 0$$

The other limitation ($A \geq -1$) gives us nothing and thus the CN scheme is stable for all choises of Δt and Δx.

5.4 Schemes for diffusion in multiple dimensions

We can try the same approach to investigate the stability of the schemes in multiple dimesions. We start out with a general solution on the form

$$A^n e^{ik(p\Delta x+q\Delta y)}$$

and insert it in the respective schemes.

$$\begin{aligned}
&A^n e^{ik(p\Delta x+q\Delta y)}\big(A-1\big) = CA^n e^{ikq\Delta y}\big(e^{ik(q+1)\Delta x} - 2e^{ikq\Delta x} + e^{ik(q-1)\Delta x}\big) + \\
&CA^n e^{ikp\Delta x}\big(e^{ik(p+1)\Delta y} - 2e^{ikp\Delta y} + e^{ik(p-1)\Delta y}\big) \\
&(A-1) = -4C\sin^2(k\Delta x/2) - 4C\sin^2(k\Delta y/2)
\end{aligned}$$

Plugging in for A we see that the limitations on A are $-1 \leq A \leq 1$. We will also insert the maximum value for the sine terms.

$$\begin{aligned}
-8C + 1 &\geq -1 \\
8C \leq 2 \implies C &\leq \frac{1}{4}
\end{aligned}$$

We see that the other limit only gives us $C \geq 0$ which really is not that exciting.
For the leap frog method we get

$$\begin{aligned}
&A^n e^{ik(p\Delta x+q\Delta y)}\big(A-A^{-1}\big) = 2CA^n e^{ikq\Delta y}\big(e^{ik(q+1)\Delta x} - 2e^{ikq\Delta x} + e^{ik(q-1)\Delta x}\big) + \\
&2CA^n e^{ikp\Delta x}\big(e^{ik(p+1)\Delta y} - 2e^{ikp\Delta y} + e^{ik(p-1)\Delta y}\big) \\
&(A-A^{-1}) = -8C(\sin^2(k\Delta x/2) + \sin^2(k\Delta y/2))
\end{aligned}$$

To avoid som unnececarily nasty expressions we will from here on look at the "worst case" where $\sin^2(k\Delta x/2) = \sin^2(k\Delta y/2) = 1$

$$\begin{aligned}
A^2 - 16CA - 1 &= 0 \\
A = \frac{1}{2}\big(-16C \pm \sqrt{16^2C^2+4}\big) &= -8C \pm \sqrt{64C^2+1} \qquad (11)
\end{aligned}$$

That is A will be a linear combination of the two roots $A_1 = -8C - \sqrt{64C^2+1}$ and $A_2 = -8C + \sqrt{64C^2+1}$. We notice that A_1 is a negative root which will cause the solution to oscillate. From this we get that the scheme is (at least) stable for $\Delta t \leq \frac{1}{8}$, but for this scheme a very small Δt is preferred (see appendix).

When it comes to the truncation error for schemes in multiple dimensions, they are the same for the Forward Euler scheme (only adding a term of $\mathcal{O}(\Delta y^2)$). For the Leapfrog scheme however the error in time is a bit different

$$\begin{aligned}
u(t+\Delta t, x, y) &= u(t,x,y) + \frac{\partial u(t,x,y)}{\partial t}\Delta t + \frac{\partial^2 u(t,x,y)}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3) \\
u(t-\Delta t, x, y) &= u(t,x,y) - \frac{\partial u(t,x,y)}{\partial t}\Delta t + \frac{\partial^2 u(t,x,y)}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3)
\end{aligned}$$

which we recognize as the same error we got from the second order spatial derivatives. Thus the truncation error in time of the Leapfrog scheme is of second order. There is of course a problem, and that is the oscillating solutions. The oscillations are (in general) small in the beginning of the simulation, but blows up towards the end, causing the solution to diverge completely. There are methods to stabilize the leapfrog scheme, and the scheme is therefore widely used in meteorology, but I do not have the time to implement theese methods as of now.

6 Solving by Finite Element Methods

The normal approach to solving partial differential equations (especially in more than one dimension) in industry is to discretize using a finite difference scheme in time, and a finite element method in space. The finite element method is quite a lot more complex both in its discretization and in its implementation, which is why we will use the FEniCS software pack to solve the equation using finite elements. The reason one would typically use finite elements for PDEs is that it easily generalizes to complex geometries with dynamic choise of resolution, and that one can choose freely how accurate the approximation should be.
Consider this a very short and superficial explaniation of the finite element method. We will stick to 1 dimension. The basic idea is to map our approximation of the solution on a function space of some piecewise continous

functions. That is, we divide the unit square, on which our equation is defined, into some number of elements. Notice that an element typically takes the form of a triangle, and that the elements can vary in size (for simplicity we will use elements of equal size). A triangle in one dimension is simply an interval, defined by two points. Theese points are called nodes. We now use the nodes to define linear functions on each element, and demand that each linear function is exactly 1 on one node and 0 on the other. We can of course use any function we want which fulfills this cirterion, but the standard functions to use are some orthogonal polynomial, specifically the Lagrange polynomials (of some degree depending on the number of nodes per element). So far we have our equation and we have a finctionspace $V = span\{\phi_i\}$ where ϕ_i is the Lagrange polynomial at element number i. We discretize the equation in time (u_{xx} denotes the duble derivative of u wrt x).

$$\begin{aligned} u^{n+1} - u^n - \Delta t u^n_{xx} &= 0 \\ R &= 0 \end{aligned}$$

we now want to approximate R on V, minimizing the error. There are a few ways to do this, and again we will use a quite general one, the Galerkin method. To minimize the error we take the inner product $(R, V) = 0 \forall \phi_i \in V$ which leads to the following integrals

$$\begin{aligned} \int_\Omega u^{n+1}\phi_i d\Omega - \int_\Omega u^n \phi_i d\Omega - \Delta t \int_\Omega u^n_{xx}\phi_i d\Omega = 0 \\ \int_\Omega \sum_{j=1}^N (\phi_j \phi_i) u^{n+1} d\Omega - \int_\Omega \sum_{j=1}^N (\phi_j \phi_i) u^n d\Omega - \Delta t \int_\Omega \sum_{j=1}^N (\phi''_j \phi_i) u^n d\Omega = 0 \end{aligned}$$

where we have used the approximation of u on V $u \simeq \sum_{j=1}^N \phi_j u$. We notice that taking a double derivative of ϕ_j leaves us with 0, so we try integrating it by parts

$$-\Delta t \int_\Omega \sum_{j=1}^N (\phi''_j \phi_i) u^n d\Omega = \Delta t \big([\phi'_j \phi_i u]_{\partial\Omega} - \int_\Omega \sum \phi'_j \phi'_i u^n d\Omega \big)$$

which is a boundary term and a new integral. This is actually sufficient for FEniCS to work with. The rest of the computation is simply assembly of a linear system from the integrals, and solving the linear system.
We can easily implement the BE scheme in time using the FEniCS software, and solve by iterating in the same way we have been doing throughout the project. In figures 9 and 10 we see the results of the same error analysis we did for the FE scheme in 2D.

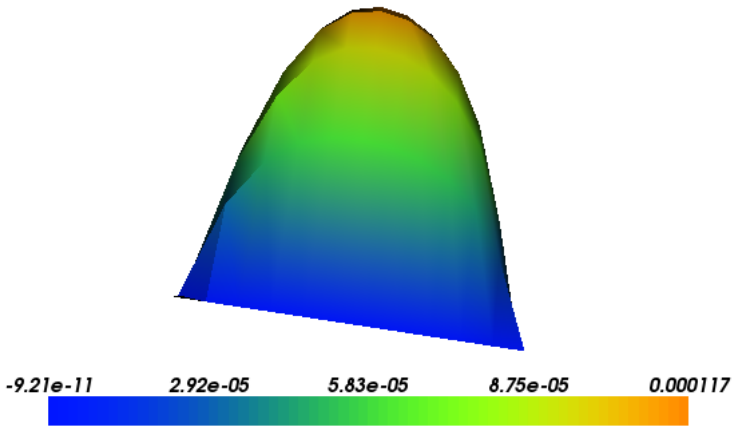

Figure 9: The absolute error for the Backward Euler scheme solved by the Finite element method using first order Lagrange ploynomials as basis functions. $\Delta x = 0.1$ and $\Delta t = 0.2\Delta x^2$

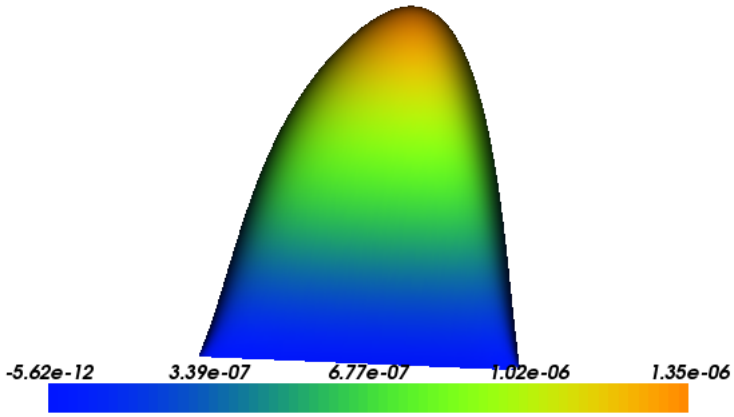

Figure 10: The absolute error for the Backward Euler scheme solved by the Finite element method using first order Lagrange ploynomials as basis functions. $\Delta x = 0.01$ and $\Delta t = 0.2\Delta x^2$

Notice that the error is basically the same as for the FE scheme underlining the fact that finite elements are best suited for the more complex problems.

7 Final comments

The results from section 5 tells us that the Crank Nicolson scheme should give an error going as $\mathcal{O}(\Delta t^2)$, which is better than both the Forward and Backward Euler schemes. Notice however that the Crank Nicolson is known to be somewhat difficult to implement in such a way that the error actually has a second order convergence. We will therefore assume that this scheme also has a first order convergence. Looking at the results form section 4 however, we see a different story. Surprisingly the Forward Euler scheme seems to give the best approximation to the exact solution. Notice also that all off the schemes have errors of roughly $\mathcal{O}(\Delta t)$, suggesting that at least all the larger bugs in my program have been eradicated. I find theese results strange, but not impossible. It could be that for some reason the Forward Euler scheme is simply better on this particular problem than the two others. Concidering simplicity of implementation (and in this case the error) the Forward Euler scheme would be my choise, but it comes with the drawback of a strict stability criterion. For a more fool-proof (and brute force) approach which is allways stable and can therefore jump over the boring part where the solution has a slow convergence, I would say that the Backward Euler scheme is definately the best one. I say this despite the fact that it shows the worst error of the schemes because it generally has a better error and I suspect that there is still a small bug somewhere. The Backward Euler scheme also has a somewhat slower convergence towards the steady state solution because it uses values from the new timestep to calculate the new timestep. A newer timestep has come slightly further in the convergence, and thus has slightly smaller derivatives giving a slower convergence.
When it comes to the two dimesional solver we see in the verification of the error (figure 7 and 8) that the error is of the expected order which is $\mathcal{O}(\Delta t)$ for both gridsizes chosen. The shape of the error is the same as in the one dimesional case (a parabola). This is also expected seeing as we force the numerical solution to be exact at the boundarys, and so the error will approach zero towards the boundarys. As mentioned in section 2 there is no need for an iterative scheeme when we know the values at perious times. We could however use such a scheme combined with an implicit solver such as the BE scheme. This would be set up in such a way that for each timestep we iterate over the grid to approximate the solution at the new timestep. This gives us an explicit scheme from an implicit one such as the BE scheme. Or we could solve a linear system where the solution is a matrix and we would have to invert a block matrix at each timestep (This is the approach of the FEniCS software pack). I know that I promised an implenmentations of the Leap frog scheme in two spatial dimensions, and I have tried to make this work, but to no use. The scheme seems to be unstable even for rather small Δt and simulation times. The code is enclosed in appendix A, should it be of interest.
It also seems like my idea of an Euler Chromer solver did not quite work out as planned. I did at some point get the same error as with the FE scheme, but I suspect that I was a bit sleep deprived at the moment and that I had actually plotted the results of the FE solver twice. Anyways, theese results are gone, and I am unable to replicate them, so they do not hold. The final word on this scheme is that we improve the error by at least two orders of magnitude when using the FE scheme.

The section on finite element methods is ment only as a short glance onward from this project into more complex problems which are still on the same form as this one. Examples are nonlinear diffusion equations, diffusion equations in three spatial dimensions etc.
Source code can be found in the appendix.

8 Appendices

A Source code

A.1 C++ files

```
/*
 * File:   diffusion.cpp
 * Author: candidate 55
 *
 * Created on 28. november 2012, 09:58
 */

#include "diffusion.h"


int main(int argc, char** argv){
    //--------Common variables----------------
    //int N = atoi(argv[1]);
    ofstream outfile;
    int tofile = atoi(argv[1]);
    int spacing = atoi(argv[2]);
    int FE1D = atoi(argv[3]);
    int BE1D = atoi(argv[4]);
    int CN1D = atoi(argv[5]);
    int FE2D = atoi(argv[6]);
    int LeapFrog = atoi(argv[7]);
    int nx = atoi(argv[8]);
    int n_t = atoi(argv[9]);

    vec u_new = zeros<vec>(nx+1);
    vec u_prev = u_new;
    double dx = 1.0/(nx);
    double dt = dx*dx/4.0;       //Stability criterion dt <= dx*dx/2
    double dtdx2 = dt/(dx*dx);
    mat U = zeros<mat>(nx+1,nx+1);
    mat U_p = U;
    mat U_pp = U;

    if(FE1D){
    //################################################
    //--------Forward Euler scheme----------##
    //################################################
        if(dtdx2>0.5){dtdx2 = 0.5;} //make sure the stability criterion is fulfilled
        u_prev = linspace<vec>(-1.0,0.0,nx+1);
        cout<<"alpha = "<<dtdx2<<endl;
        for(int n = 0;n <= n_t; n++){
            for(int i = 1; i <nx; i++){
                u_new(i) = dtdx2*(u_prev(i+1)-2*u_prev(i) + u_prev(i-1)) + u_prev(i);
            }
            u_new(0) = 0; u_new(nx) = 0;
            u_prev = u_new;
            /*write to file for plotting*/
            if(tofile && (n%spacing)==0){output(&outfile,u_prev,n,0,nx);cout<<n<<endl;}
```

```
    }
    cout<<"Explicit scheme finished. "<<endl;
    }
if(BE1D){
//################################################
//-------Backward Euler scheme----------##
//################################################
    //dtdx2 = 1/(dx*dx*200);
    u_prev = linspace<vec>(-1.0,0.0,nx+1);
    double a = -dtdx2;
    double c = a;
    double b = 1+2*dtdx2;
    u_new.zeros();
    for(int n = 1;n<=n_t;n++){
        tridiag(a,b,c, u_new, u_prev,nx);
        u_new(0)=0; u_new(nx) = 0;
        //u_prev = u_new;
        for(int k=0;k<=nx;k++){
            u_prev(k) = u_new(k);
        }
        /*Write to file for plotting*/
        if(tofile && (n%spacing)==0){output(&outfile,u_prev,n,1,nx);}

    }
    cout<<"Backward Euler scheme finished"<<endl;
}
if(CN1D){
//################################################
//------Crank Nicolson scheme----------##
//################################################

    u_prev = linspace<vec>(-1.0,0.0,nx+1);
    double a1 = -dtdx2;
    double c1 = a1;
    double b1 = 2+2*dtdx2;
    double a2 = dtdx2;
    double c2 = a2;
    double b2 = 2-2*dtdx2;
    u_new = u_prev;
    for(int n=1; n<=n_t; n++){
        make_uprev(u_prev,u_new,a2,c2,b2,nx);
        u_prev(0)=0;u_prev(nx)=0;
        tridiag(a1,b1,c1, u_new, u_prev,nx);
        //u_prev = u_new;
        for(int k=0;k<=nx;k++){
            u_prev(k) = u_new(k);
        }
        u_prev(0)=0;u_prev(nx)=0;
        /*Write to file for plotting*/
        if(tofile && (n%spacing)==0){output(&outfile,u_prev,n,2,nx);}
    }
    cout<<"Crank Nicolson scheme finished"<<endl;
}
//################################################
//---------------2D solvers----------##
//---------------Forward Euler----------##
//################################################
if(0){
    double C = dtdx2;
    if(C >= 0.25)
    {//Insert for stability criterion!
```

```
            C = 0.2;
        }
        dx = 1.0/nx;
        dt = C*dx*dx;
        cout<<"dt = "<<dt<<" C = "<<C<<endl;
        initial_condition(U_p,dx,nx);
        for(int t=0; t<n_t; t++){
            for(int i=1; i<nx; i++){
                for(int j=1; j<nx; j++){

                    U(i,j) = U_p(i,j) + C*(U_p(i+1,j)-2*U_p(i,j)+U_p(i-1,j)) \
                    + C*(U_p(i,j+1)-2*U_p(i,j)+U_p(i,j-1));
                }
            }
            update_boundarys(U,t*dt,dx,nx);
            for(int k=0;k<=nx;k++){
                for(int l=0;l<=nx;l++){
                    U_p(k,l) = U(k,l);
                }
            }
             //Write to file for plotting
            if(tofile && (t%spacing)==0){output2D(&outfile,U_p,t,3,nx);}
        }
        cout<<"Forward Euler done!"<<endl;
    }
    if(LeapFrog){
        double C = dtdx2;
        if (C>(1.0/8.0))
        {
            C = 1.0/10.0;
        }
        dt = C*dx*dx;
        initial_condition(U_p,dx,nx);
        for(int i=1; i<nx; i++){
            for(int j=1; j<nx; j++){
                U(i,j) = U_p(i,j) + C*(U_p(i+1,j)-2*U_p(i,j)+U_p(i-1,j)) \
                + C*(U_p(i,j+1)-2*U_p(i,j)+U_p(i,j-1));
            }
        }
        update_boundarys(U,dt,dx,nx);
        U_pp = U_p;
        U_p = U;
        for(int t=1; t<n_t; t++){
            for(int i=1; i<nx; i++){
                for(int j=1; j<nx; j++){
                    U(i,j) = C*(U_p(i+1,j)-2*U_p(i,j)+U_p(i-1,j)) \
                    + C*(U_p(i,j+1)-2*U_p(i,j)+U_p(i,j-1))+0.5*U_pp(i,j);
                }
            }
            update_boundarys(U,t*dt,dx,nx);
            for(int k=0;k<=nx;k++){
                for(int l=0;l<=nx;l++){
                    U_pp(k,l) = U_p(k,l);
                }
            }
            for(int k=0;k<=nx;k++){
                for(int l=0;l<=nx;l++){
                    U_p(k,l) = U(k,l);
                }
            }
            //Write to file for plotting
```

```
            if(tofile && (t%spacing)==0){output2D(&outfile,U_p,t,4,nx);}
    }
    cout<<"Leap Frog scheme finished"<<endl;
    }
    if(FE2D){
    //#########################################################
    //----------Euler Chromer scheme----------------##
    //#########################################################
        double C = dtdx2;
        if(C >= 0.25)
        {//Insert for stability criterion!
            C = 0.2;
        }
        dx = 1.0/nx;
        dt = C*dx*dx;

        cout<<"dt = "<<dt<<" C = "<<C<<endl;
        initial_condition(U_p,dx,nx);
        for(int t=1; t<=n_t; t++){
            for(int i=1; i<nx; i++){
                for(int j=1; j<nx; j++){
                    U(i,j) = U_p(i,j) + C*(U_p(i+1,j)-2*U_p(i,j)+U(i-1,j)) \
                    + C*(U_p(i,j+1)-2*U_p(i,j)+U(i,j-1));
                }
            }
            update_boundarys(U,t*dt,dx,nx);
            for(int k=0;k<=nx;k++){
                for(int l=0;l<=nx;l++){
                    U_p(k,l) = U(k,l);
                }
            }
             //Write to file for plotting
            if(tofile && (t%spacing)==0){output2D(&outfile,U_p,t,3,nx);}
        }
        cout<<"Euler Chromer done!"<<endl;
    }
    return 0;
}

/*
Headerfile for diffusion porject
*/

#include <cstdlib>
#include <omp.h>
#include <armadillo>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <fstream>

/*from lib.h*/
#include <new>
#include <cstdio>
#include <cstdlib>
#include <cstring>


using namespace std;
using namespace arma;
```

```
#ifndef DIFFUSION_H
#define DIFFUSION_H

double timediff(double time1, double time2);
char *make_filename(int n, int scheme);
void tridiag(double a, double b, double c, vec &v, vec &f, int n);
void make_uprev(vec &uprev,vec &u, double a, double c, double b1, int n);
void output(ofstream* outfile, vec &u, int n, int scheme, int N);
void output2D(ofstream* outfile, mat &u, int n, int scheme, int N);
void initial_condition(mat &u, double dx, int n);
void update_boundarys(mat &u, double t, double dx, int n);
#endif  /* DIFFUSION_H */


double timediff(double time1, double time2){
        // This function returns the elapsed time in milliseconds
        return ((time2 - time1)*1000)/CLOCKS_PER_SEC;
}
char *make_filename(int n, int scheme){
        //Returns a filename saying something about the particular run.
        char* buffer = new char[60];
                if(scheme == 0){
                sprintf(buffer,"results_FE_n%d.txt",n);
        }
        else if(scheme ==1){
                sprintf(buffer,"results_BE_n%d.txt",n);
        }
        else if(scheme ==2){
                sprintf(buffer,"results_CN_n%d.txt",n);
        }
        else if(scheme ==3){
            sprintf(buffer,"results_FE2D_n%d.txt",n);
        }
        else{
            sprintf(buffer,"results_LF2D_n%d.txt",n);
        }
        return buffer;
}

void tridiag(double a, double b, double c, vec &v, vec &f, int n){
    vec bv = zeros<vec>(n+1);
    double temp = 0;
    bv(1)=b;
    v(1) = f(1)/b;
    //v(0) = 1;

    for(int i=2;i<=n-1;i++){
        //forward substitution withoutvectors
        temp = a/bv(i-1);
        bv(i) = b -c*temp;
        f(i) -= f(i-1)*temp;
    }
    v(n-1)= f(n-1)/bv(n-2);
    //v(n)=0;

    for(int i=n-2;i>=1;i--){
        //Backward substitution
        v[i] = (f(i)-c*v(i+1))/bv(i);
    }
```

```
}

void make_uprev(vec &uprev, vec &u, double a, double c, double b, int n){
    uprev(1) =b*u(1) +c*u(2);
    for(int i =2;i< n-1; i++){
        uprev(i) = a*u(i-1) +c*u(i+1) + b*u(i);
    }
    uprev(n-1) = a*u(n-2)+b*u(n-1);
}

void output(ofstream* outfile, vec &u, int n, int scheme, int N){
    /*outfile is an ofstram-object letting us open a file
    **u is an armadillo-object containing the solution at time n
    **n is the timestep number
    **scheme is an integer telling what scheme is used to obtain the solution
    **N is the size of the array*/
        outfile->open(make_filename(n,scheme));
        for(int i=0;i<=N;i++){
                *outfile <<u(i)<<setprecision(12)<<endl;
        }
        outfile->close();
}

void output2D(ofstream* outfile, mat &u, int n, int scheme, int N){
    /*outfile is an ofstram-object letting us open a file
    **u is an armadillo-object containing the solution at time n
    **n is the timestep number
    **scheme is an integer telling what scheme is used to obtain the solution
    **N is the size of the array (in one direction)*/
    outfile->open(make_filename(n,scheme));
    for(int i=0;i<=N;i++){
        for(int j=0;j<=N;j++){
            *outfile <<u(i,j)<<setprecision(12)<<"  ";
            }
        if(i<N){*outfile <<endl;}
    }
    outfile->close();
}

void initial_condition(mat &u, double dx, int n){
    for(int i=0;i<=n;i++){
        for(int j=0;j<=n;j++){
            u(j,i) = (1- j*dx)*exp(i*dx);
        }
    }
}

void update_boundarys(mat &u, double t, double dx, int n){
    for(int i=0; i<=n;i++){
        for(int j=0; j<=n;j++){
            u(j,0) = (1- j*dx)*exp(t);
            u(j,n) = (1- j*dx)*exp(1+t);
            u(0,i) = exp(i*dx+t);
            u(n,i) = 0;
        }
    }
}
```

A.2 Python scripts for plotting

```
import os,argparse,glob,numpy as np
```

```
import matplotlib.pyplot as mpl
#from scitools.std import *
from mayavi import mlab
from mayavi.api import OffScreenEngine

parser = argparse.ArgumentParser()
parser.add_argument("-run",action="store_true", help="run the diffusion.cpp excecutable fi
parser.add_argument("-compile",action="store_true", help="complie the diffusion project")
parser.add_argument("-tofile", action="store_true", help="write results to file (for plotti
parser.add_argument("-spacing", type=int, action="store",dest="spacing",default=10, help="
parser.add_argument("-removefiles",action="store_true", help="remove the resultfiles")
parser.add_argument("-FE1D", action="store_true",help="Run the Forward Euler discretizatior
parser.add_argument("-BE1D", action="store_true",help="Run the Backward Euler discretizatio
parser.add_argument("-CN1D", action="store_true",help="Run the Crank Nicolson discretizatio
parser.add_argument("-FE2D", action="store_true",help="Run the Forward Euler discretizatior
parser.add_argument("-LF2D", action="store_true",help="Run the Leap Frog discretization in
parser.add_argument("-nx", type=int, action="store",dest="nx",default=10, help="number of
parser.add_argument("-nt", type=int, action="store",dest="nt",default=100, help="number of
args = parser.parse_args()

tofile = 1 if args.tofile else 0

FE1D = 1 if args.FE1D else 0
BE1D = 1 if args.BE1D else 0
CN1D = 1 if args.CN1D else 0
FE2D = 1 if args.FE2D else 0
LF2D = 1 if args.LF2D else 0

if args.compile:
        os.system('g++ -o willy -O3 diffusion.cpp -larmadillo')

if args.run:
                os.system('./willy %d %d %d %d %d %d %d %d %d' %(tofile,args.spacing,FE1D,l


'''
solvers = [FE1D,BE1D,CN1D,FE2D,LF2D]
names = ['FE*.txt','BE*.txt','CN*.txt']
i=0
for method in names:
        filenames = 'results_'+names[i]
        i+=1
        for files in sorted(glob.glob(filenames)):
                picname = files.split('.')
                picname[0] += '.png'
                print files
                u = np.loadtxt(files)
                u += np.linspace(1,0,len(u))
                #ax.cla()
                #ax.imshow(u)
                mpl.plot(u)

                #fig.savefig(picname[1])
                mpl.show()
                if args.removefiles:
                        os.remove(files)
'''
names2d =['FE2D*.txt','LF2D*.txt']
i=0
X,Y = np.meshgrid(np.linspace(0,1,args.nx+1),np.linspace(0,1,args.nx+1))
for method in names2d:
```

```
        filenames = 'results_'+names2d[i]
        #fig = mpl.figure(figsize=(5,5))
        #ax = fig.add_subplot(111)
        i+=1
        for files in sorted(glob.glob(filenames)):
                picname = files.split('.')
                picname[0] += '.png'
                u = np.loadtxt(files)
                #mlab.options.offscreen = True
                mlab.mesh(X,Y,u)
                mlab.savefig(picname[0])
                mlab.clf()

                #ax.cla()
                #ax.imshow(u)
                #mpl.plot(u)

                #fig.savefig(picname[1])
                #mpl.show()
                if args.removefiles:
                        os.remove(files)

'''
os.system("mencoder 'mf://_tmp*.png' -mf type=png:fps=10 \
        -ovc lavc -lavcopts vcodec=wmv2 -oac copy -o animation.mpg")
'''

import os,numpy as np, matplotlib.pyplot as mpl, glob,sys

try:
        timestep = int(sys.argv[1])
except IndexError:
        print "Bad usage: provide timestep on commandline"
        sys.exit(1)
tofile = 1
FE1D = 0
BE1D = 0
CN1D = 0
FE2D = 0
LF2D = 1
nx = 100
nt = 100
spacing = 5
cmpile = True
run = True

if cmpile:
        os.system('g++ -o willy -O3 diffusion.cpp -larmadillo')

if run:
                os.system('./willy %d %d %d %d %d %d %d %d %d' %(tofile,spacing,FE1D,BE1D,


x = np.linspace(1,0,nx+1)
axis = np.linspace(0,1,nx+1)
dx = 1./(nx)
dt = dx**2/4.0

def solution(t):
        x = np.linspace(0,1,nx+1)
        exact = np.linspace(1,0,nx+1)
```

```
        fourier = np.zeros(nx+1)
        for m in xrange(1,250):
                fourier[:] += (1.0/m)*np.exp(-(m*np.pi)*(m*np.pi)*(t+dt))*np.sin(m*np.pi*x
                '''
                if m%5 ==0:
                        mpl.plot(np.linspace(0,1,nx+1),fourier)
                '''
                #print m,fourier
        fourier *= (2.0/np.pi)
        '''
        #mpl.legend(['m = 5', 'm = 10','m = 15','m = 20','m = 25','m = 30'])
        mpl.title('Convergence of analytic solution')
        mpl.xlabel('x')
        mpl.ylabel('u(x,0)')
        mpl.show()
        '''
        return exact-fourier

def plot_error(timestep):
        solver = ['FE','BE','CN']
        n = 0
        for results in sorted(glob.glob('results*n%d.txt'%timestep)):
                infile = np.loadtxt(results)
                approx = x+infile
                mpl.plot(axis,abs(solution(timestep*dt)-approx))
                print results
                n +=1
        mpl.title('exact-numeric t = %.4f, n = %d, dt = %.4f'%(timestep*dt,nx,dt))
        mpl.xlabel('x'); mpl.ylabel('error')
        mpl.legend(['BE','CN','FE'])
        mpl.show()

def plot_normal(timestep):
        solver = ['FE','BE','CN']
        n = 0
        for results in sorted(glob.glob('results*n%d.txt'%timestep)):
                infile = np.loadtxt(results)
                approx = x+infile
                mpl.plot(axis,approx)
                n +=1
        mpl.plot(axis,solution(timestep*dt))
        mpl.xlabel('x'); mpl.ylabel('u(x,t = %g)'%(timestep*dt))
        mpl.legend(['BE','CN','FE','exact'])
        mpl.show()

def solution_2d(timestep):
        X,Y = np.meshgrid(np.linspace(0,1,nx+1),np.linspace(0,1,nx+1))
        U = np.zeros((nx+1,nx+1))
        U[:,:] = (1-Y[:,:])*np.exp(X[:,:]+timestep)
        return U

def plot_error_2d(timestep,filename=None):
        from mayavi import mlab
        infile = np.loadtxt(filename) if filename is not None else np.zeros((nx+1,nx+1))
        X,Y = np.meshgrid(np.linspace(0,1,nx+1),np.linspace(0,1,nx+1))
        U = solution_2d(timestep)
        print np.shape(infile)
        mlab.mesh(X,Y,abs(U-infile))
        mlab.colorbar()
        mlab.show()
        #mlab.savefig('test.png')
```

```
filename = 'results_LF2D_n%d.txt'%timestep
#plot_normal(timestep)
#plot_error(timestep)
#f = solution(0*dt)
print timestep*dt

plot_error_2d(timestep*dt,filename)
```

A.3 FEniCS program/wrapper in Python

```
from dolfin import *
import numpy as np

nx = 100
dt = 0.2*(1./nx)*(1./nx)
mesh = UnitSquare(nx+1,nx+1)
V = FunctionSpace(mesh,'Lagrange',1)


u0 = Expression('(1-x[1])*exp(x[0]+t)', t = 0)
u_1 = interpolate(u0,V)

def u0_boundary(x,on_boundary):
        return on_boundary

bc = DirichletBC(V,u0,u0_boundary)
u = TrialFunction(V)
v = TestFunction(V)
a = u*v*dx + dt*inner(nabla_grad(u),nabla_grad(v))*dx
L = u_1*v*dx

A = assemble(a)

u = Function(V)
T = 0.5
t = dt
b = None
tol = 1e-12

while t<=T:
        b = assemble(L, tensor=b)
        u0.t = t
        bc.apply(A,b)
        solve(A,u.vector(),b)
        t+=dt
        u_1.assign(u)
        u_e = interpolate(u0, V)
        if(abs(t-0.05)<tol):
                plot(abs(u_e-u))
                interactive()
```

A.4 Leapfrog test program

```
from numpy import *
from matplotlib.pyplot import *

def solver(I, a, b, T, dt):
    """
Solve u'(t) = -a(t)u(t) + b(t), u(0) = I,
for T in (0,T] with steps dt.
a and b are python functions of t.
```

```
"""
    dt = float(dt)
    N = int(round(T/dt))
    T = N*dt
    u = zeros(N+1)
    t = linspace(0, T, N+1)

    u[0] = I
    u[1] = u[0] + dt*(b(t[0]) - a(t[0])*u[0])

    for n in range(1, N):
        u[n+1] = u[n-1] + 2*dt*(b(t[n]) - a(t[n])*u[n])
    return u, t


def exact_solution(t, I, a, b):
    u_e = zeros(len(t))
    for i in range(len(t)):
        u_e[i] = b(t[i])/a(t[i]) + (I- b(t[i])/a(t[i]))*exp(-a(t[i])*t[i])
    return u_e


def plot_solution(u, t, u_e, t_e, dt, T):
    figure()
    plot(t, u, 'r--')
    plot(t_e, u_e, 'b-')
    legend(['numerical', 'exact'], 'best')
    xlabel('t')
    ylabel('u(t)')
    title('Leapfrog scheme for T = %g, dt = %g' % (T, dt))
    savefig('LF_T%g_dt%g.png' % (T, dt))
    show()


def dt_experiments(I, a, b, T, dt_values):

    M = 1000
    t_e = linspace(0, T, M+1)
    u_e = exact_solution(t_e, I, a, b)

    for dt in dt_values:
        N = int(round(T/dt))
        u, t = solver(I, a, b, T, dt)

        plot_solution(u, t, u_e, t_e, dt, T)

def T_experiments(I, a, b, T_values, dt):
    M = 1000


    for T in T_values:
        N = int(round(T/dt))
        u, t = solver(I, a, b, T, dt)
        t_e = linspace(0, T, M+1)
        u_e = exact_solution(t_e, I, a, b)
        plot_solution(u, t, u_e, t_e, dt, T)


def main():
    I = 0.1
    T = 4
```

```
    dt = 0.001
    dt_values = [0.5, 0.1, 0.05, 0.01, 0.005, 0.001]
    T_values = [2, 4, 6, 8, 10]

    def a(t):
        return 1

    def b(t):
        return 1

    #dt_experiments(I, a, b, T, dt_values)
    T_experiments(I, a, b, T_values, dt)

main()
```

B Other resources

B.1 The θ-rule

As mentioned in section 2 the Crank Nicoloson scheme is a special case of the θ-rule. The theta rule is to calculate the new solution at some time between t_n and t_{n+1}. We call this time $\theta \in [0, 1]$, and get

$$t_{n+\theta} = \theta t_{n+1} + (1 - \theta)t_n$$

inserting this in our function gives us the same thing

$$u(\tilde{t}) \simeq \theta u^{n+1} + (1 - \theta)u^n$$

We now force the equation to hold at $\tilde{t} \in [t_n, t_{n+1}]$ and approximate the right hand side by an average leaving

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = \theta u^{n+1} + (1 - \theta)u^n$$

which is the general θ-rule. Inserting $\theta = 0$ gives the Forward Euler scheme, $\theta = 1$ gives the Backward Euler scheme, and $\theta = \frac{1}{2}$ is the Crank Nicolson scheme.

B.2 Closer analysis of the Leapfrog scheme in 1 dimension

Picking up at (11) where we left off we can write A as (from now on we are in one dimenson, but all calculations done in section 5 are pricipally the same.

$$A^n = \alpha_1 A_1^n + \alpha_2 A_2^n$$

we can determine the coeficcients α_1 and α_2 from the initial condition $u(x, 0) = I$ setting $8C = x$

$$A^0 = \alpha_1 + \alpha_2 = I \implies \alpha_1 = I - \alpha_2$$
$$A^1 = \alpha_1(-x - \sqrt{x^2 + 1}) + \alpha_2(-x + \sqrt{x^2 + 1}) = I(1 - x)$$

And from this we can run some tests on just how these roots affect the solution.
To illustrate the problem with this scheme I have done some tests. Figures 11 and 12 show the nummeric and the analytic solution to the equation $u'(t) = -au(t)$ in the same plot.

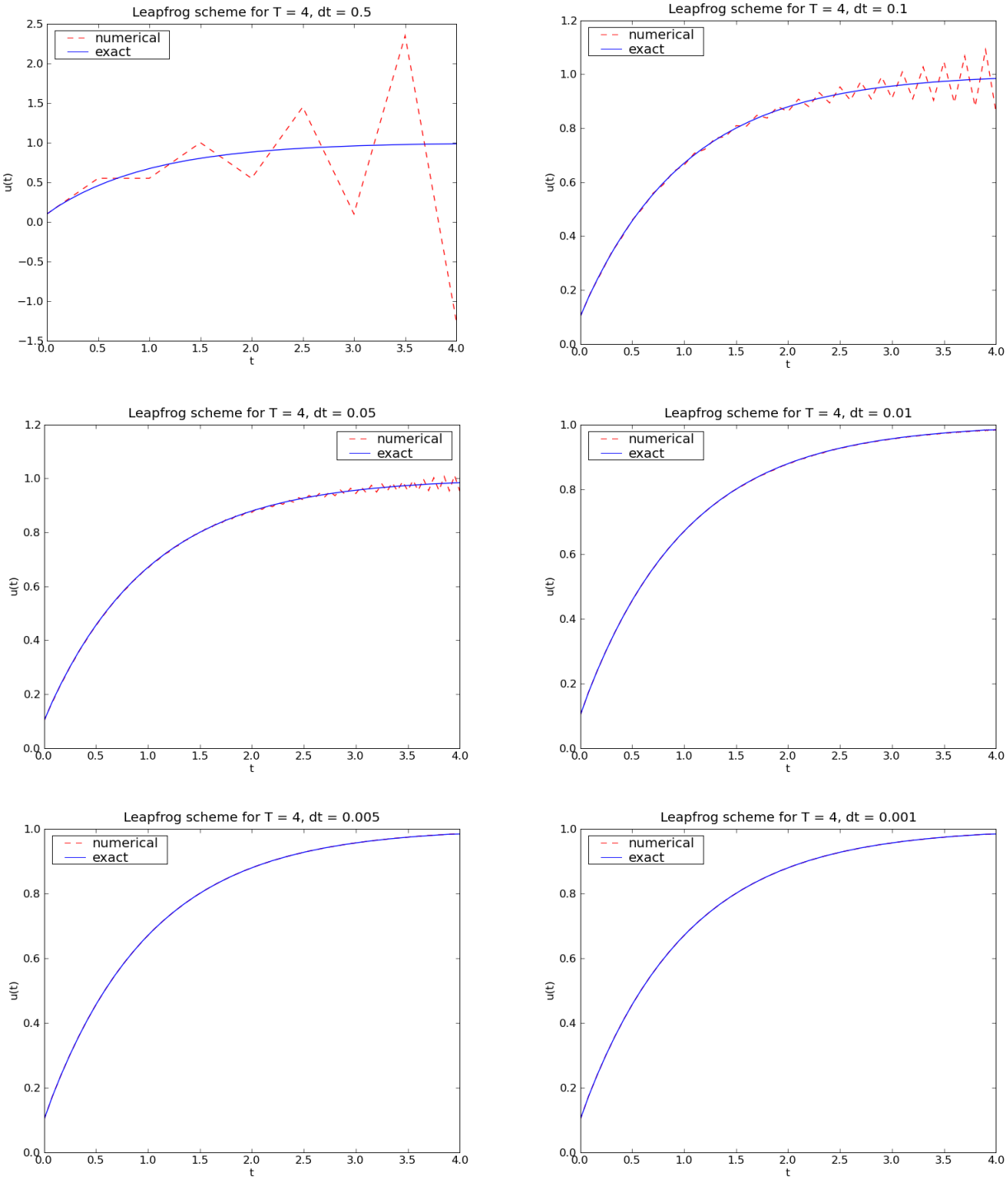

Figure 11: Leapfrog scheme for increasingly better dt

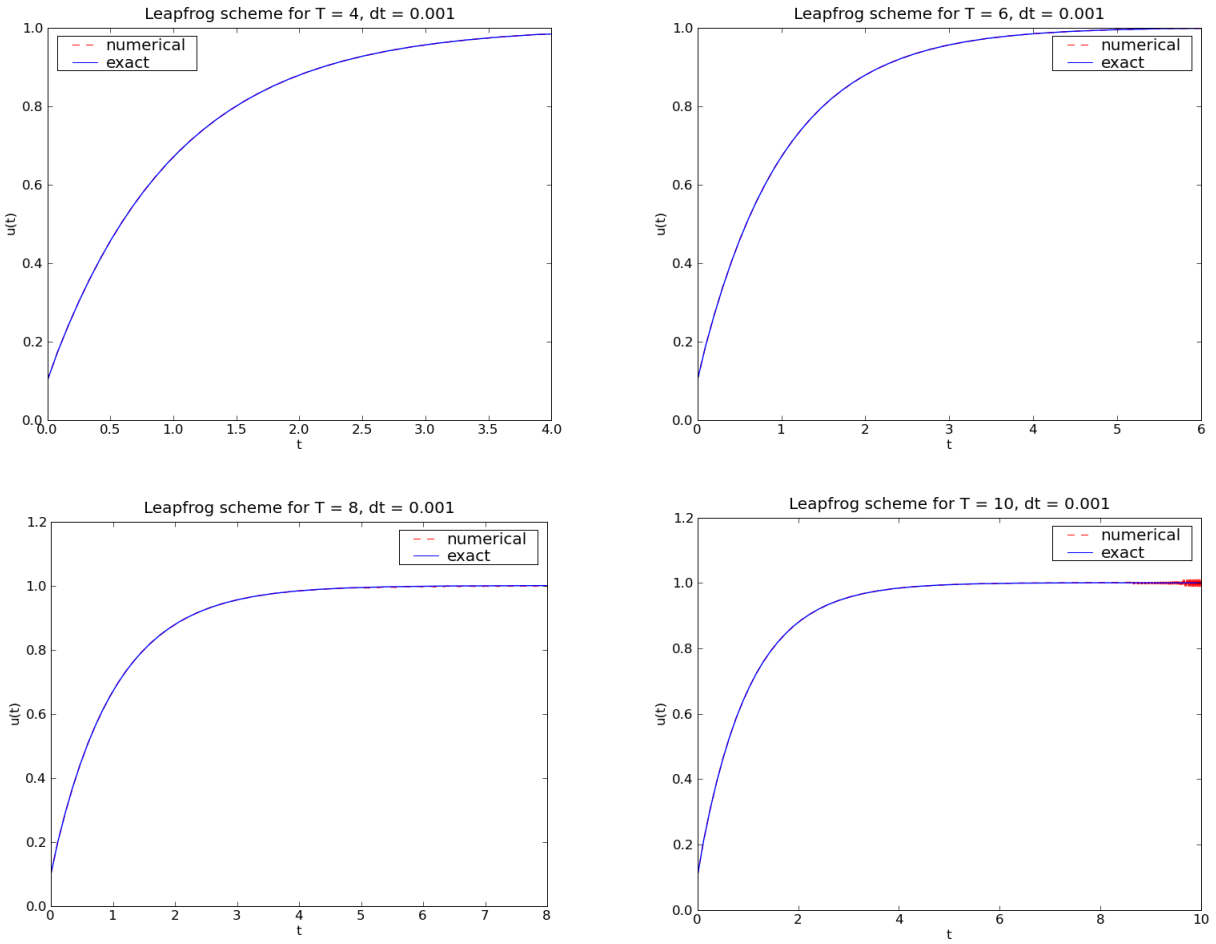

Figure 12: Leapfrog scheme with best dt for longer simulation times

We notice that the scheme seems to be doing very well for a long time when Δt is small, but eventually breaks down, just as expected.