Project 5

FYS4150 Computational Physics

Even S. Håland

Abstract

1 Introduction

The purpose of this project is to solve the partial differential equation (PDE) known as the diffusion equation (or heat flow equation), both in one and two dimensions[1]. This equation has several applications in physics, and can for instance be used to describe the Brownian motion of particles and the heat flow in a material.

The one-dimensional equation is given as

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

where we have $t > 0$ and $x \in [0, L]$, where we will choose $L = 1$. In order to solve such a problem we also need an initial condition, which is given by

$$u(x,0) = 0 \quad \text{for} \quad 0 < x < L,$$

and boundary conditions, given as

$$u(0,t) = 0 \text{ and } u(L,t) = 1 \quad \text{for} \quad t \geq 0.$$

The two-dimensional equation reads

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

where we also have $t > 0$ and $x, y \in [0, L]$ with $L = 1$. The initial condition is quite similar to the previous case:

$$u(x,y,0) = 0 \quad \text{for} \quad 0 < x, y < L.$$

However, the boundary conditions becomes slightly more complicated. It might be tempting to extend the boundaries from the one dimensional case such that $u(x,L,t) = u(L,y,t) = 0$ and $u(L,y,t) = u(x,L,t) = 1$. However, since the boundary is now a square in the xy-plane, this leaves us with some unpleasant discontinuities in two of the corners. This problem can be avoided by instead choosing boundary conditions

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

where we always have $t \geq 0$, which gives a nice and continuous boundary.

For the one dimensional case we will implement and study three different schemes for solving the equation numerically. These are the forward Euler, the backward Euler and the Crank-Nicolson schemes. In particular we will study the stability of these algorithms when varying the step size in space and time. For the two

[1]By dimensions we here mean spatial dimensions. The diffusion equation is also always time dependent.

dimensional equation we will stick to one algorithm, which is an explicit scheme. In this case we will also study the stability of the algorithm.

An important part of the project is also to solve equations 1 and 2 analytically, with the given initial and boundary conditions. The analytical solutions will be compared with the numerical ones in order to study the performance of our algorithms.

We will start with the analytical solutions, before moving on to discussing the algorithms and how they are implemented. Finally we will look at (and discuss) the results.

2 Analytical solutions

When solving the equations analytically we will, both in one and two dimensions, make use of the technique for solving PDE's known as separation of variables. That is we assume that the solution can be written as a product of functions, where each function only depends on one of the independent variables (i.e x, y or t). In addition to this product we will add a term to our solution corresponding to the steady state solution. We will see that this last trick makes our lives quite a bit easier.

Note also that both in this section and later we will use a short-hand notation for derivatives, reading

$$\frac{\partial f}{\partial x} = f_x \quad \text{and} \quad \frac{\partial^2 f}{\partial x^2} = f_{xx}.$$

2.1 One dimension

For the one dimensional equation (eq. 1) we assume that the solution can be written as

$$u(x,t) = F(x)T(t) + u_s(x),$$

where $u_s(x)$ is the steady state solution, which naturally is independent of time. We know that steady state solution will be a straight line, i.e. $u_s(x) = x$, with the given boundaries.

Inserting our proposed $u(x,t)$ into the diffusion equation leaves us with

$$F_{xx}T = FT_t \quad \Rightarrow \quad \frac{F_{xx}}{F} = \frac{T_t}{T},$$

where each side of the latter equation must be a constant, which we (for later convenience) will call $-\lambda^2$. This means that we have two (ordinary) differential equations to solve:

$$F_{xx} + \lambda^2 F = 0 \quad \text{and} \quad T_t = -\lambda^2 T$$

The general solutions to these equations are

$$F(x) = A\cos(\lambda x) + B\sin(\lambda x) \quad \text{and} \quad T(t) = Ce^{-\lambda^2 t},$$

where A, B, C are constants. The first boundary conditions, $u(0,t) = 0$, implies that $B = 0$, which leaves us with the preliminary solution

$$u(x,t) = A' \sin(\lambda x) e^{-\lambda^2 t} + x,$$

where $A' = AC$. The second boundary condition, $u(L = 1, t) = 1$, gives

$$A' \sin(\lambda) = 0.$$

(Remember that the exponential is never zero.) Now we have two options. We can choose $A' = 0$, but this gives a trivial and non-interesting solution. A more interesting alternative is to choose $\lambda = n\pi$, where $n = 1, 2, 3, \ldots$. This means that the general solution is written as a Fourier series,

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x) e^{-n^2\pi^2 t} + x.$$

In order to determine the Fourier coefficients, A_n, we can use the initial condition, $u(x,0) = 0$, which gets rid of the time part, and we find that

$$\sum_{n=1}^{\infty} A_n \sin(n\pi x) = -x.$$

In e.g. ref. [1] on can find the Fourier expansion

$$x = 2 \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n} \sin nx,$$

from which we easily can find the coefficients we need by letting $x \to \pi x$. Our general solution then becomes

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

2.2 Two dimensions

The method for solving the two-dimensional equation (eq. 2) goes much in the same pattern, but there are some complications because of the extra dimension. Let us start by assuming that

$$u(x,y,t) = F(x,y)T(t) + u_s(x,y).$$

The first complication we meet is that it is a bit more tricky to realise what the steady state solution must be. The only possibility I have been able to come up with in order to satisfy the boundary conditions is $u_s(x,y) = xy$. (We will see later that this also matches the numerical solution quite good.)

We insert our $u(x,y,t)$ into the diffusion equation, and gets

$$\frac{F_{xx}}{F} + \frac{F_{yy}}{F} = \frac{T_t}{T} = -\lambda^2,$$

where λ again is a constant, and we see that the time dependent part gets exactly the same form as in one dimension. However, the spatial part becomes

$$F_{xx} + F_{yy} + \lambda^2 F = 0,$$

which is still a PDE, and we need to separate it further. Let us assume that $F(x, y) = G(x)H(y)$, giving

$$G_{xx}H + GH_{yy} + \lambda^2 GH = 0 \quad \Rightarrow \quad \frac{G_{xx}}{G} = -\frac{H_{yy}}{H} - \lambda^2 = -\kappa^2,$$

where κ is a constant, hence

$$G_{xx} + \kappa^2 G = 0 \quad \text{and} \quad H_{yy} + \nu^2 H = 0,$$

where $\nu^2 = \lambda^2 - \kappa^2$. These have the same form as the spatial part of the 1D problem, and we can also eliminate the cosine part of the solution using the boundary conditions $u(0, y, t) = u(x, 0, t) = 0$. This gives a preliminary solution of the form

$$u(x, y, t) = A\sin(\kappa x)B\sin(\nu y)e^{-\lambda^2 t} + xy,$$

where A, B are constants. Using the other boundary conditions we get (neglecting the time part)

$$A\sin(\kappa)B\sin(\nu y) + y = y \quad \text{and} \quad A\sin(\kappa x)B\sin(\nu) + x = x,$$

giving $\kappa = n\pi$ and $\nu = m\pi$, where $m, n = 1, 2, 3, \ldots$, (hence $\lambda^2 = (n^2 + m^2)\pi^2$), and the full solution as a double Fourier series

$$u(x, y, t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x) \sum_{m=1}^{\infty} B_m \sin(m\pi y)e^{-(n^2+m^2)\pi^2 t} + xy.$$

As for the one-dimensional case we can use the initial condition to determine the Fourier coefficients, and the general solution becomes

$$u(x, y, t) = xy - \frac{4}{\pi^2}\sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \frac{(-1)^{n+m-2}}{nm} \sin(n\pi x)\sin(m\pi y)e^{-(n^2+m^2)\pi^2 t}.$$

3 Algorithms and code

Having solved the equations analytically we will now move to the numerical part of the project, and discuss the algorithm we have implemented. All code written for the project is found in the following git repository:

https://github.com/evensha/FYS4150/tree/master/Project5/Programs

This repository contains four important programs/scripts:

- `Diffusion_1d.cpp`

- `Diffusion_2d.cpp`
- `Diffusion_1d.py`
- `Diffusion_2d.py`

The first two are the C++ programs used to solve the one and two dimensional diffusion equation numerically, while the last two are python scripts used to plot and analyse the output from the C++ programs, as well as calculate the analytical solution(s).

3.1 One dimension

For solving the one dimensional equation we have implemented three different algorithms, which will be briefly described in the following. Where we use equal step lengths, with the spatial step, Δx, always given by

$$\Delta x = \frac{1}{n+1},$$

where n is the number of steps we consider. Also for the time step, Δt, we will only consider equal step sizes, but when choosing the actual value of Δt we should take into account the stability limit of the algorithm in question.

Position and time are discretized as

$$\begin{aligned} t_j &= j\Delta t \quad j \geq 0, \\ x_i &= i\Delta x \quad 0 \leq i \leq n+1, \end{aligned}$$

giving

$$u(x,t) = u(x_i, t_j) = u_{i,j}.$$

3.1.1 Forward Euler

With the forward Euler scheme we approximate the derivatives as

$$u_t \approx \frac{u(x, t+\Delta t) - u(x,t)}{\Delta t} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t},$$

and

$$u_{xx} \approx \frac{u(x+\Delta x) - 2u(x,t) + u(x-\Delta x, t)}{\Delta x^2} = \frac{u_{i+1,j} + u_{i,j} - u_{i-1,j}}{\Delta x^2}.$$

The approximation of the time derivative is the standard forward Euler formula with truncation error $O(\Delta t)$, while the double derivative in x is the three point formula with truncation error $O(\Delta x^2)$.

Now we have approximations for both sides of the 1D diffusion equation, and a simple manipulation of the equation ($u_{xx} = u_t$) leads us to the expression

$$u_{i,j+1} = \alpha u_{i-j,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j}, \tag{3}$$

where

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

This is a very simple equation to implement, since the solution for a given time step only depends on the solution for the previous time step. This means that, with given initial and boundary conditions, the solutions can be calculated directly without any "fancy" mathematics, hence this is referred to as an explicit scheme. This does however come with a price, as the algorithm is stable only if

$$\Delta t \leq \frac{1}{2}\Delta x^2,$$

meaning that we need small (and hence many) time steps. The algorithm is implemented in the function `Forward_Euler` in the one-dimensional program.

3.1.2 Backward Euler

With the backward Euler scheme the time derivative is approximated by the backward Euler formula

$$u_t \approx \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u_{i,j} - u_{i,j-1}}{\Delta t},$$

while u_{xx} is approximated as previously. This leads to the expression

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

with α defined as previously. We call this an implicit scheme, since we determine u at time t_{j-1} based on u at time t_j. This means that we can't implement this equation directly (like we did for the forward Euler method), but we rather need to implement a method for solving the equation (or set of equations). This can be done by realizing that we can write this as a linear algebra problem. We do this by introducing a vector V_j containing $u_{i,j}$ at a given time t_j, i.e.

$$V_j = \begin{pmatrix} u_{1,j} \\ u_{2,j} \\ \vdots \\ u_{n-1,j} \end{pmatrix}.$$

We don't include the boundary points, as these are known. It is then fairly easy to realize that equation 4 can be rewritten as

$$V_{j-1} = \hat{A}V_j,$$

where $\hat{A}$ is a tridiagonal matrix given by

$$\hat{A} = \begin{pmatrix} 1+2\alpha & -\alpha & 0 & \cdots & \cdots & 0 \\ -\alpha & 1+2\alpha & -\alpha & 0 & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & 0 & -\alpha & 1+2\alpha & -\alpha \\ 0 & \cdots & \cdots & 0 & -\alpha & 1+2\alpha \end{pmatrix},$$

thus we find the values of u at time t_j by finding the inverse of $\hat{A}$, i.e.

$$V_j = \hat{A}^{-1} V_{j-1}.$$

This problem can be solved by the Thomas algorithm, which first performs a Gauss elimination and then a backward substitution, in order to find each element of V_j. This method is implemented in the `Backward_Euler` function in the code.

As we in this scheme use similar approximations to the derivatives as in the forward Euler scheme, the truncation errors are the same, i.e. $O(\Delta t)$ and $O(\Delta x^2)$. This algorithm is however stable for *all* choices of Δt and Δx.

3.1.3 Crank-Nicolson

Finally we will, for the one-dimensional case, also consider the Crank-Nicolson scheme. The expressions used for approximations of the derivatives in this scheme are based on Taylor series expansion around $t' = t + \Delta/2$ (instead of $t + \Delta t$ as usual), hence this is referred to as a time centred scheme. The expression for the time derivative now becomes

$$u_t \approx \frac{u(x,t) - u(x, t-\Delta t)}{\Delta t} = \frac{u_{i,j} - u_{i,j-1}}{\Delta t},$$

which is the same expression as for the backward Euler scheme, but the difference being that the truncation error is now reduced to $O(\Delta t^2)$, because of the time centring. This also leads to a slightly more complicated expression for u_{xx}, given by

$$u_{xx} \approx \frac{1}{2}\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{\Delta x^2}\right),$$

which has truncation error $O(\Delta x^2)$. Putting these approximations together leads to the expression

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

with α defined as previously. This can be written on matrix form as

$$(2\hat{I} + \alpha\hat{B})V_j = (2\hat{I} - \alpha\hat{B})V_{j-1},$$

where $\hat{I}$ is the identity matrix, V_j is like defined in the previous section, and $\hat{B}$ is tridiagonal matrix given as

$$\hat{B} = \begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix}.$$

We can then calculate V_j as

$$V_j = (2\hat{I} + \alpha\hat{B})^{-1}(2\hat{I} - \alpha\hat{B})V_{j-1},$$

which can again be solved by the Thomas algorithm, with only a few modifications compared to the backward Euler scheme. This is implemented in the `Crank_Nicolson` function in the code, and this scheme is also stable for all choices of Δx and Δt.

3.2 Two dimensions

For the two dimensional diffusion equation we will only consider an explicit scheme, very similar to the forward Euler scheme described above, only with an extra space dimension. We now discretize x, y, t by

$$x_i = x_0 + ih, \quad y_j = y_0 + jh \quad \text{and} \quad t_l = t_0 + l\Delta t,$$

where $0 \leq i, j \leq n+1$ and $l \geq 0$, with

$$h = \frac{1}{1+n},$$

meaning that we use the same step size in x- and y-direction. Using the same approximations for the derivatives as for the 1D forward Euler scheme we arrive to the explicit expression

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

This is implemented in `Forward_Euler` function in the 2D program. (Notice that in this program I have also tried to implement the iterative Jacobi method for solving the implicit 2D scheme, without great success. I have however not spent a lot of time on figuring out what I have done wrong here, but rather decided to just stick to the explicit solver.)

4 Results

All results are presented in the following section, including comparisons of the different schemes and comparisons with the analytical solution.

4.1 One dimension

In one dimension we will study the solution two different time points, t_1 and t_2, where we have chosen these points so that the system is in "motion" at t_1, while at t_2 it has reached the steady state. We will also look at two different choices of Δx, and choose Δt as dictated by the stability limit of the forward Euler method.

The numerical solutions using each of the solvers are plotted together with the analytical solution in Figure 1 for t_1 and t_2, using $\Delta x = 1/10$ in the upper plots and $\Delta t = 0.01$ in the lower plots. The information we are able to extract from these plots is however quite limited, as all solvers seem to give good agreement with the analytical solution.

It might be more interesting to study Figure 2 which shows the absolute value of the difference between the analytical solution and each of the numerical ones. Here we see that the solver that (in general) seems to be performing best is backward Euler. This is a bit surprising, as the truncation error in the Crank-Nicolson method should be smaller, suggesting that there might be something slightly wrong with the implementation of the Crank-Nicolson algorithm. We also notice that the forward Euler solver shows some weird behaviour, but we have to keep in mind that the solution is obtained at the stability limit of this solver (i.e. $\Delta t = 1/2\Delta x^2$). In Figure 3 we can see the catastrophic consequences of increasing the time step by a factor of two, to $\Delta t = \Delta x^2$.

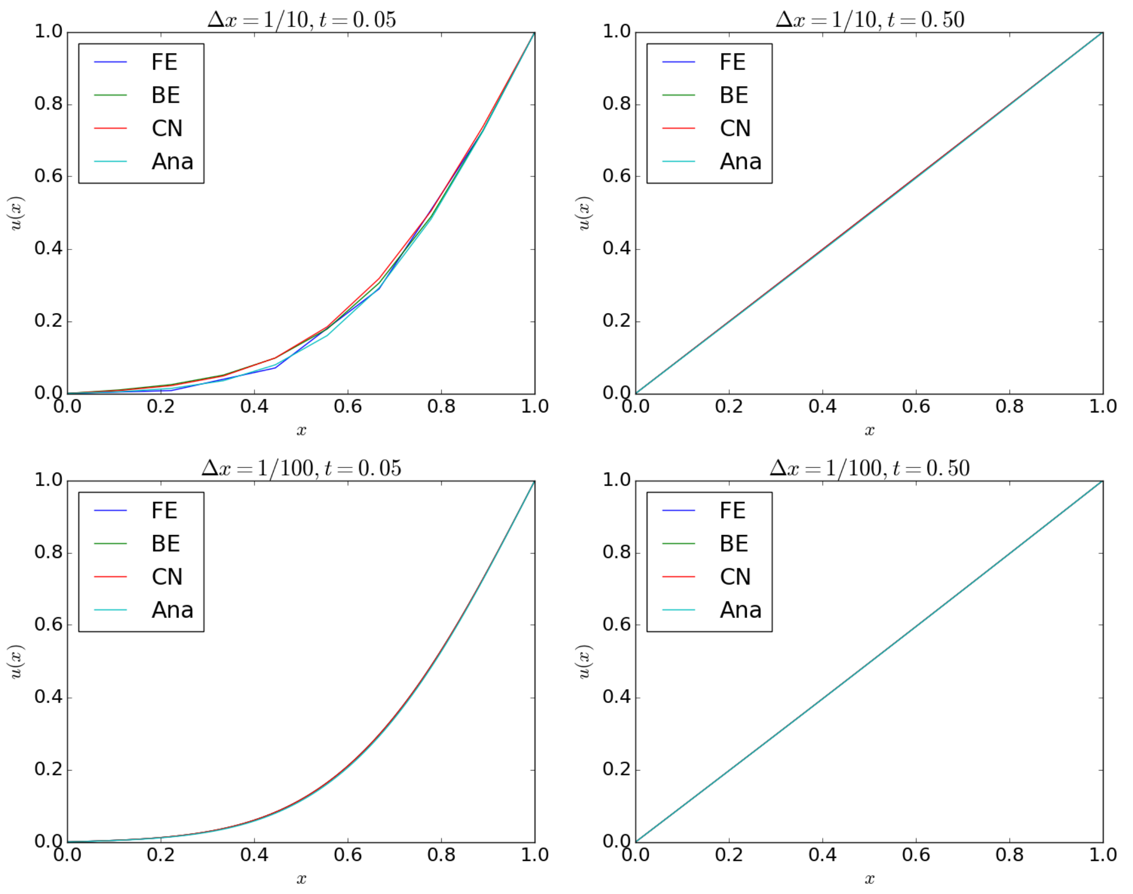

Figure 1: Numerical and analytical solution for the 1D diffusion equation, using $\Delta x = 0.1$ in the upper plots and $\Delta x = 0.01$ in the lower plots.

4.2 Two dimensions

5 Summary and conclusions

References

[1] K. Rottman (2003). *Matematisk formelsamling*, Spektrum forlag, Norway.

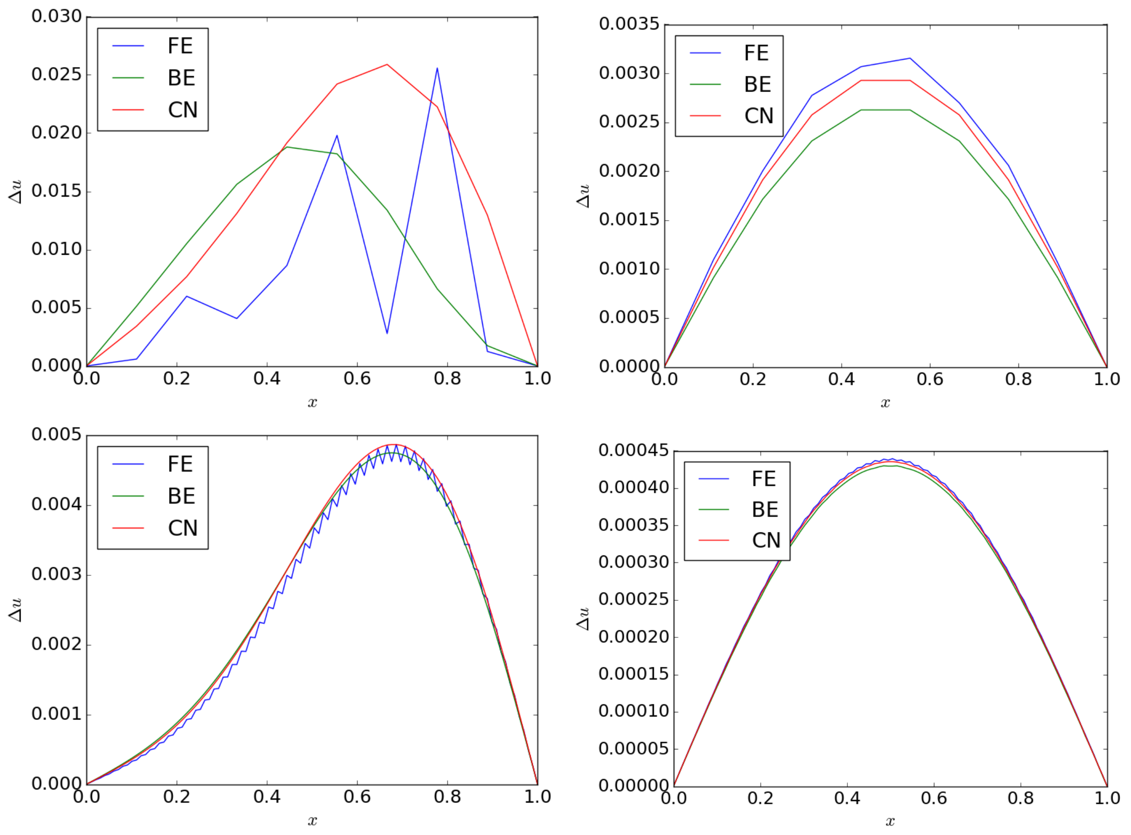

Figure 2: Absolute differences (Δu) between numerical and analytical solution, using $\Delta x = 0.1$ in the upper plots and $\Delta x = 0.01$ in the lower plots.

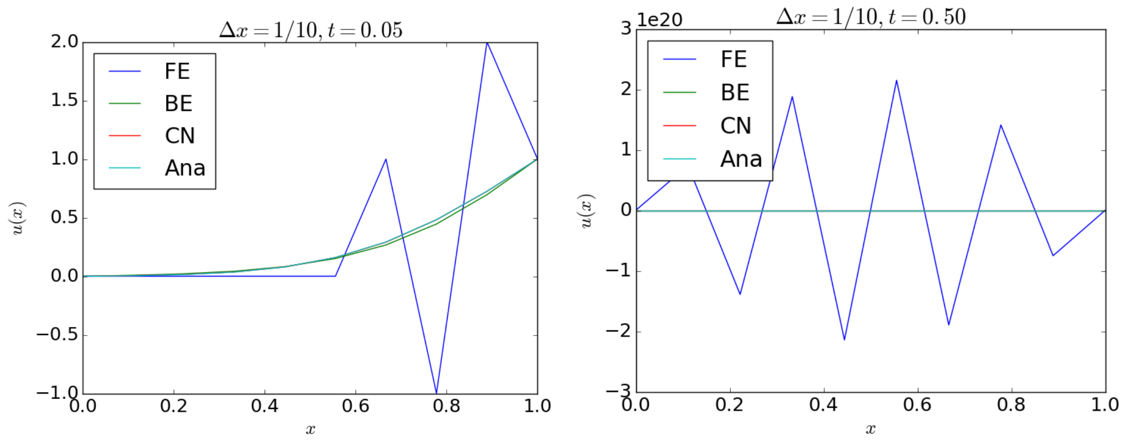

Figure 3: Diffusion equation solved using $\Delta t = \Delta x^2$, meaning that the forward Euler solver is (obviously) unstable.