Project 5 - Partial Differencial Equation

Solveig Andrea Devold Fjeld and Sarah Rastad

December 29, 2016

Abstract

This reseach looks at the diffusion equation in one and two dimensions and attempts to solve them using the explicit schemes Forward Euler and Jacobi and the implicit Backwards Euler and Crank Nicolson schemes. An error analysis showed that the truncation errors were respectively $\epsilon = 0.0025$ for the different Euler schemes and $\epsilon = 6.25 * 10^{-6}$ for Crank Nicolson. For carefully chosen initial conditions and time- and spacial steps the Forward Euler still provided the most accurate results. We found, however, that for $\frac{\Delta t}{\Delta x^2} > 0.5$ the explicit schemes fell far short.

1 Introduction

The diffusion equation can be solved analytically only for carefully chosen initial conditions. As with so many partial differential equations a far better approach is to adapt numerical methods. In this project we have done just that and look at the stability and accuracy of explicit and implicit schemes.

Our motivation for choosing this project was our curiosity to learn different kind of algorithms to solve different PDE equations, since PDE equations are so widespread in the natural sciences and especially in Physics. We wanted to get a better grasp on how to solve these equaitions and at the same time get a better handle on how the error in the solutions behave. Since many PDEs are impossible to solve analytical we thought that this project was the one we could benefit the most from.

2 Theory

2.1 Equations

We are solving the partial differencial equation:

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

We have the initial condition:

$$u(x,0) = 0 \tag{2}$$

And boundary conditions when $t > 0$:

$$u(0,t) = 0, u(L,t) = 1 \tag{3}$$

Here $L = 1$.

The steady state solution can be found easily by solving the laplace equation which is the harmonic solution. So we have to write our problem on a form that is easier to solve. We introduce:

$$w(x,t) = u(x,t) - v(x) \tag{4}$$

Where $v(x)$ is the steady state solution.

This gives us:

$$w(0,t) = 0 - 0 = 0 \tag{5}$$

$$w(L,t) = u(L,t) - v(L) = 1 - 1 = 0 \tag{6}$$

$$w(x,0) = u(x,0) - v(x) = 0 - x = -x \tag{7}$$

We can then solve the equation:

$$w_{xx} = w_t \tag{8}$$

This partial differencial equation can be seen as the temperature gradient in a rod of lenght L. It is also dimensionless in our case since there are no constant multiplied to the equation and x goes from zero to one.

To solve this equation we are looking for a solution by separating the variables:

$$w(x,t) = X(x)T(t) \tag{9}$$

If we take the partial derivatives of this expression we get:

$$w_{xx} = X''(x)T(t), and w_t = X(x)T'(t) \tag{10}$$

So if we set put this in the equation (8) we get:

$$\frac{T'(t)}{T(t)} = \frac{X''(x)}{X(x)} = constant = -\lambda \tag{11}$$

We see that this must be equal to a constant and that it therefore is an eigenvalue problem. We put a minus sign infront of the eigenvalue for convention.

This gives us the equations:

$$w(0,t) = X(0)T(t) = 0$$

$$w(1,t) = X(1)T(t) = 0$$

If we let $T(t) = 0$ we get the trivial solution, which we are not interested in.

The two-dimensional diffusion equation is given by eq(12).

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} = \frac{\partial u(x,y,t)}{\partial t} \tag{12}$$

with $t > 0$ and x=y=[0,1].

We choose to only look for the steady state solution with the boundary conditions:

$$u(0,0,t) = X(0)Y(0)T(t) = 0$$
$$u(1,0,t) = X(1)Y(0)T(t) = 0$$
$$u(0,1,t) = X(0)Y(0)T(t) = 0$$
$$u(1,1,t) = X(1)Y(1)T(t) = 0$$

The initial conditions are chosen for simplicity.

2.2 Algorithms

For solving PDE's we have to different kind of algorithms: explicit and implicit. An example of an explicit algorithm is the Forward Euler. What defines an explicit algorithm is that it takes basis in the forward timestep when differentiating and is therefore straightforward to program. In the implicit scheme, however, we calculate the differential by using the previous timestep and it becomes a series of matrix equations which can be solved using for example Gaussian elimination or LU-decomposition. Examples of implicit schemes are backward Euler and Crank-Nicolson.

2.2.1 Forward Euler

In forward euler we are approximating the time derivative by:

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t} \tag{13}$$

We are also using a centered difference in space with the approximation as you can see in equation (14).

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}. \tag{14}$$

Setting these two equations equal to eachother gives:

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{15}$$

$$\Rightarrow u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j} \tag{16}$$

And this is the form we choose for solving this. By looking at this equation we also see that stability requires (eq. (17))

$$\alpha = \frac{\Delta t}{\Delta x^2} < 0.5 \tag{17}$$

Else the second term vanishes, and our solution for the new time step is wrong.

We can implement this just by looping over the timesteps, followed by looping over the x values where $x \in [0,1]$.

2.2.2 Backward Euler

This is an implicit scheme where we approximating the time derivative by:

$$u_t \approx \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x_i,t_j) - u(x_i,t_j-\Delta t)}{\Delta t} \tag{18}$$

By setting $u_t = u_x x$ we get the equation:

$$u_{i,j-1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} - \alpha u_{j+1,i} \tag{19}$$

We then introduce the matrix:

$$\begin{bmatrix} 1+2\alpha & -\alpha & 0 & 0 & \dots & 0 \\ -\alpha & 1+2\alpha & -\alpha & 0 & \dots & 0 \\ 0 & -\alpha & 1+2\alpha & -\alpha & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & 0 & \dots & & 1+2\alpha \end{bmatrix}$$

We see that we can formulate this as a matrix multiplication problem:

$$\hat{A}V_j = V_{j-i} \tag{20}$$

which means we can rewrite our differential equation problem to:

$$V_j = \hat{A}^{-1}V_{j_1} = \hat{A}^{-1}(\hat{A}^{-1}V_{j_2}) = ... = \hat{A}^{-j}V_0 \tag{21}$$

To solve this matrix equation we utilize the Gaussian elimination for tridiagonal matrixes which we solved in project 1.

2.2.3 Crank Nicolson

In Cranc-Nicolson we use a time centered scheme where

$$u(x_i,t_{j+1}) \approx \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t} \tag{22}$$

This gives the equation :

$$\frac{u_{i,j+1}-u_{i,j}}{\Delta t} = \frac{1}{2}\left(\frac{u_{i+1,j+1}-2u_{i,j+1}+2u_{i-1,j+1}}{(\Delta x)^2} + \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\Delta x)^2}\right) \tag{23}$$

We can write this as:

$$-\alpha u_{i+1,j+1} + (1+2\alpha)u_{i,j+1} - \alpha u_{i-1,j-1} = \alpha u_{i+1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i-1,j} \tag{24}$$

Giving the matrix equation:

$$\hat{A}V_{j+1} = \hat{B}V_i \tag{25}$$

or

$$\hat{A}V_{j+1} = b_j \tag{26}$$

where we find V_{j+1} by using a forward euler and solving the matrix equation as in backward euler by using Gaussian elimination.

2.3 Jacobi

For an explicit solution to the 2-dimensional problem, $U_{x,t}$ and $U_{y,t}$ are given by eq(16). Combining these results in the explicit Jakobi-algorithm (27).

$u(i,j,t+dt) = u(i,j,t) + \alpha(u(i+1,j,t) + u(i-1,j,t) - 4(i,j,t) + u(i,j+1,t) +$

u(i,j-1,t)) (27)

3 Execution

3.0.1 Forward Euler

For the forward Euler algorithm we start by solving U(x,0), hereby referenced as U0, and define α as given by eq(16) with dx=0.1 and dt=dx*dx*0.25, as dictated by the restrictions for the stability for the explicit scheme (eq(17). We then call the forward step method (see below) for a given number of timesteps, each run increasing the total time T by dt.

```
vec forward_step(double n, double alpha, vec u, vec unew) {
    for (int i=1; i<n; i++) {
        unew(i) = alpha*u(i-1) + (1-2*alpha) * u(i) + alpha*u(i+1);
    }
    return unew;
}
```

3.0.2 Backward Euler

In the implicit Backward Euler scheme we use Gaussian elimination to advance in space and time, implemented in code below. Here, as eq (19) shows, b-value is defined as $1+2\alpha$, and a=c=-α, v being the solution given at a previous timestep, with the same initial condition as for the forward Euler scheme. We run the Gaussian elimination for each timestep dt until T(i) = final T.

```
Forward Substitution
    double m;
    for (int k=2; k<=n; k++) {
        m = a/b(k-1);
        b(k) = b_value - m*c;
        v(k) -= m*v(k-1);
    }

Backward Substitution
    u(n)= v(n)/b(n);
    for (int k= n-1; k>0; k--) {
        u(k) = (1.0/b(k))*(v(k) - c*u(k+1));
    }

    u(0) = 0;
    u(n) = 0;
```

3.0.3 Crank-Nicolson

Crank-Nicolson, being a combination of the explicit and implicit schemes, first runs forward step and then uses this updated solution v in the gaussian elimination for each timestep T(i).

```
void crank_Nicolson(int n, int t_steps, double alpha,
                                double dx, double dt) {
    char* method = "crank_nicolson";
    initializePrint(method, n, dx, dt, t_steps);
    double a_value, c_value, b_value;
    vec b(n+1);
    a_value = c_value = -alpha;
    b_value = 1 + 2*alpha;

    vec u(n+1);
    vec v(n+1);

    //making the vectors
    for (int k=1; k<n; k++){
        b(k) = b_value;
        u(k) = func(dx,k);
    }

    b(0) = b(n) = b_value;
    u(0) = 0.0;
    u(n) = 1.0;

    //The algorithm for solving the system
    for (int t=1;t<=t_steps;t++) {
        //One step with forward euler
        v = forward_step(n,alpha,u,v);
        v(0) = 0.0;
        v(n) = 1.0;
        //Then gaussian elimination
        u = GaussElim(a_value,b, b_value,c_value,n,u,v);
        ... Print to file method
    }
}
```

3.1 Jacobi

Implementing the jacobi-algorithm is quite straight forward using eq.(27). U(x,y,0)= A old = solution for t=0 found by separation of variables.

```
void jakobi_solver (double dx, double dt) {
    double n = 1.0/dx;
    double t_steps = 1000;
    mat A_old = zeros<mat>(n+1,n+1);
```

```
    for (int i=1;i<n;i++) {
        for (int j=1;j<n;j++) {
            A_old(i,j) = func(dx,i,j);
        }
    }

    double alpha = dt/(dx*dx);
    mat A_new = zeros<mat>(n+1,n+1);
    for (int t=1;t<=t_steps;t++) {
        for (int i=1;i<n;i++) {
            for (int j=1;j<n;j++) {
                A_new(i,j) = A_old(i,j) + alpha*(A_old(i+1,j)
                    + A_old(i-1,j) - 4*A_old(i,j) +
                     A_old(i,j+1) + A_old(i,j-1));
            }
        }
        A_old = A_new;
        }
    }
}
```

4 Results

4.1 Closed form solutions

4.1.1 Solution to the 1D heat equation

To solve the equation (1) we need to look for seperable solutions on the form:

$$w(x,t) = X(x)T(t) \tag{28}$$

If we set this in in the equation (1) we get:

$$\frac{\partial}{\partial t}(X(x)T(t)) = \frac{\partial^2}{\partial x^2}(X(x)T(t)) \tag{29}$$

To simplify the notation we write:

$$T'(t)X(x) = T(t)X''(x) \tag{30}$$

Which we can write:

$$\frac{T'(t)}{T(t)} = \frac{X''(x)}{X(x)} \tag{31}$$

We see that each side depends on a different variable R.H.S depends on x and L.H.S depends on t, so therefore this mus be equal to a constant. This is because if we change one and keep the other fixed the value must be the same. This constant we set to $-\lambda$ by convention so the equations to solve becomes:

$$X''(x) + \lambda X(x) = 0 \tag{32}$$

$$T'(t) + \lambda T(t) = 0 \tag{33}$$

With the boundary conditions:

$$w(0,t) = X(0)T(t) = 0 \tag{34}$$

$$w(1,t) = X(1)T(t) = 0 \tag{35}$$

From these boundary conditions we see that $X(0) = X(1) = 0$ because if $T(t) = 0$ we would only get the trivial solutions which we are not interested in.

So we solve the $X(x)$ equation first.

This is an equation which we have solved nmany times before. First we have the case $\lambda < 0$ which gives the solution:

$$X(x) = Ae^{\sqrt{k}x} + Be^{-\sqrt{k}x}, \lambda = -k \tag{36}$$

if we set in the boundary conditions we get that $X(0) = A + B$ and then $X(1) = Ae^{\sqrt{k}} - Ae^{\sqrt{k}} = A(e^{2*\sqrt{k}})$ and since k must be positive this gives that $A = B = 0$ which is the trivial solution which we are not interested in.

When $\lambda = 0$ this gives $A = B = 0$ which are also trivial solutions.

The last possibility is the harmonic equation which is:

$$X(x) = Acos(\sqrt{\lambda x}) + Bsin(\sqrt{\lambda x}) \tag{37}$$

And with our boundary conditions it gives $X(0) = A = 0$ and $X(1) = Bsin(\sqrt{\lambda}) = 0$. This means that $sin\lambda = 0$ This gives us the eigenvalue $\lambda = (n\pi)^2$ for any positive integer. This gives the solution:

$$X(x) = b_n sin(n\pi x) \tag{38}$$

The solution for $T(t)$ is then given by:

$$T'(t) = -n^2 * \pi^2 T(t) \tag{39}$$

Which is well known as

$$T(t) = c_n e^{-(n*pi)^2 t} \tag{40}$$

So the the solution becomes:

$$w(x,t) \sum_{n=1}^{\infty} B_n * sin(n * \pi x) e^{-(n^2\pi^2 t)} \tag{41}$$

With the initial condition

$$w(x,0) = \sum_{n=1}^{\infty} B_n * sin(n * \pi x) \tag{42}$$

multiply with $sin(m\pi x)$ on both sides and use the orthogonality of $sin(m\pi x)$ and $sin(n\pi x)$

And this gives:

$$\sum_{n=1}^{\infty} b_n \int_0^1 sin(n\pi x) sin(m\pi x) dx = \frac{1}{2} b_n = -\int_0^1 sin(m\pi x) x dx \tag{43}$$

This gives the integral

$$b_n = -2\int_0^1 xsin(m\pi x) \tag{44}$$

solving this by using partial integration and simplifying gives:

$$b_n = (-1)^m(-\frac{1}{m\pi}), m = 1, 2, ... \tag{45}$$

So setting this into the whole solution from equation (4) we get:

$$u(x,t) = v(x) + w(x,t) = x + \sum_{n=1}^{\infty}(-1)^n(\frac{2}{n\pi})sin(n\pi * x)e^{-(n\pi)^2 t} \tag{46}$$

4.2 3D- Heat equation

4.2.1 Analytical Solution

The equation (12), as the 2D equation, is solved by assuming separation of solutions:

$$u(x,y,t) = X(x)Y(y)T(t) \tag{47}$$

With the boundary conditions $u(0,y,t) = u(1,y,t) = 0$ and $u(x,0,t) = u(x,1,t) = 0$ Using this, the equation takes the form:

$$\frac{X''(x)}{X(x)} + \frac{Y''(y)}{Y(y)} = \frac{T'(t)}{T(t)} \tag{48}$$

By the same logic as in the 2D-case this becomes:

$$\frac{X''(x)}{X(x)} + \frac{Y''(y)}{Y(y)} = -\lambda \tag{49}$$

If we first keep y constant and varies x we get the equation:

$$X''(x) + (\lambda + \frac{Y''(y)}{Y(y)})X(x) = 0 \rightarrow X''(x) + (\lambda + \mu)X(x) = 0 \tag{50}$$

As in the 2D-case when we get the equations:

$$Y''(y) + (\lambda + \mu)Y(y) = 0 \tag{51}$$

in both dimensions, giving

$$X(x) = b_n sin(n\pi x) \tag{52}$$

$$Y(y) = c_m sin(m\pi y) \tag{53}$$

And the time equation then becomes:

$$T(t) = d_{n,m}e^{-(m^2\pi^2 + n^2\pi^2)t} \tag{54}$$

So the equation becomes with $m = n = 1$ and $b_n c_n d_{n,m} = 1$

$$u(x,y,t) = sin(\pi x)sin(\pi y)e^{-2pi^2 t} \tag{55}$$

4.2.2 Error Analysis

To calculate the error we use taylor expansion which are defined:

$$u_n = \frac{f^{(n)}(b)}{n!} \tag{56}$$

So to calculate the error in the forward difference for u'(t) we make a Taylor expansion around t_n:

$$u(t_{n+1}) = u(t_n) + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{57}$$

This gives the error:

$$R = \frac{1}{2}u''(t_n)\Delta t + \mathcal{O}\Delta t^2 \tag{58}$$

This means that the forward euler has an error in time in the first order. For backwards euler we taylor expand $u(t_{n-1}$ and get:

$$R = \frac{1}{2}u''(t_n)\Delta t + \mathcal{O}\Delta t^2 \tag{59}$$

So the same as in the Forward Euler scheme

In Crank-Nicolson we use a time centered scheme so we have to taylor expand $u_{n+1/2}$ and $u_{n-1/2}$ and combine these.

$$u(t_{n+1/2}) = u(t_n) + \frac{1}{2}u'(t_n)\Delta t + \frac{1}{4}u''(t_n)\Delta t^2 +$$

$$\frac{1}{12}u'''(t_n)\Delta t^3 + \frac{1}{48}u^{(4)}(t_n)\Delta t^4 +$$

$$\frac{1}{240}u^{(5)}(t_n)\Delta t^5 + \mathcal{O}(\Delta t^6)$$

and

$$u(t_{n-1/2}) = u(t_n) - \frac{1}{2}u'(t_n)\Delta t + \frac{1}{4}u''(t_n)\Delta t^2 -$$

$$\frac{1}{12}u'''(t_n)\Delta t^3 + \frac{1}{48}u^{(4)}(t_n)\Delta t^4 -$$

$$\frac{1}{240}u^{(5)}(t_n)\Delta t^5 + \mathcal{O}(\Delta t^6)$$

If we subtract the last from the first we get the error:

$$R = \frac{1}{24}u'''(t_{n+1/2})\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{60}$$

But to get the full error we have to take in consideration the standard arithmetic mean wich is:

$$\frac{1}{2}(u(t_{n+1/2}) + u(t_{n-1/2}) \tag{61}$$

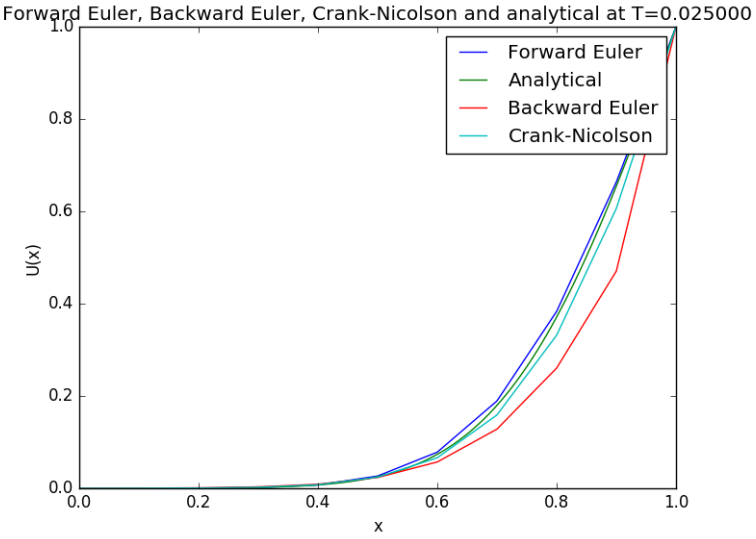

Figure 1: The three schemes with the analytical solutiion for when $T = 0.025$ and $dt = 0.0025$

This has the error term:

$$R = \frac{1}{8}u''(t_{n+1/2})\Delta + \frac{1}{384}u''''(t_n)\Delta t^4 + \mathcal{O}(\Delta t^6) \tag{62}$$

If we add these two we get the total error in time as:

$$R = (\frac{1}{24}u'''(t_{n+1/2}) + \frac{1}{8}u''(t_n))\Delta t^2 + \mathcal{O}\Delta t^4 \tag{63}$$

So the error in the Crank-Nicolson scheme is Δt^2

And the error in spatial differencial for all three is equal to by usin taylor expansion:

$$R_x = \frac{1}{12}u''''(x_n)\Delta x^2 + \mathcal{O}(\Delta x^4) \tag{64}$$

So the error in x is of the second order.

In figure (1) and (2) we see the numerical calculations against the analytical for two times $t_1 = 0.025$ and $t_2 = 0.65$.

4.3 3D PDE

Explicit numeric solutions for different values of dx and dt are figured in plots ((3), (5), (5), (7), (9), (11), (13) and (15)). The corresponding analytical solutions are shown in figures ((4), (6), (8), (10), (10), (12), (14) and (16).

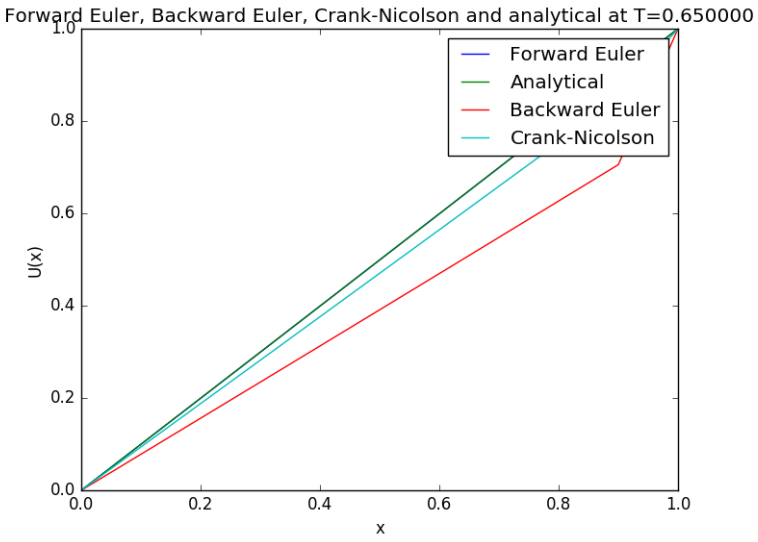

Figure 2: The three schemes with the analytical solutiion for when $T = 0.025$ and $dt = 0.65$

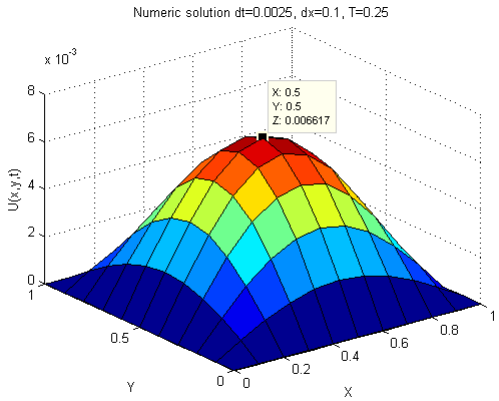

Figure 3: Explicit solution for dx=0.1, $\alpha = 0.25$, T=0.25

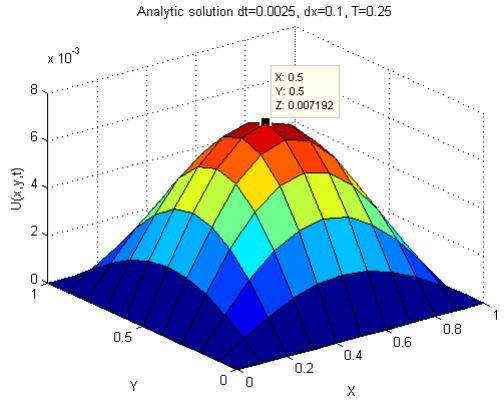

Figure 4: Analytical solution for dx=0.1, $\alpha = 0.25$, T=0.25

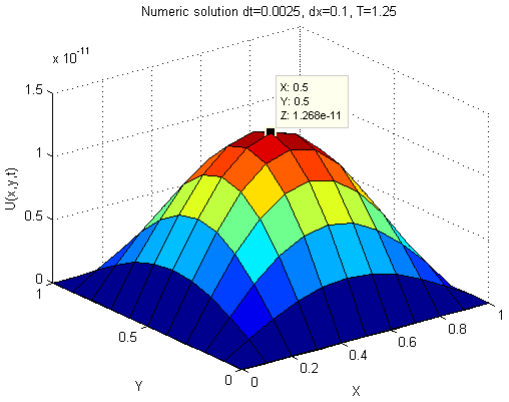

Figure 5: Explicit solution for dx=0.1, $\alpha = 0.25$, T=1.25

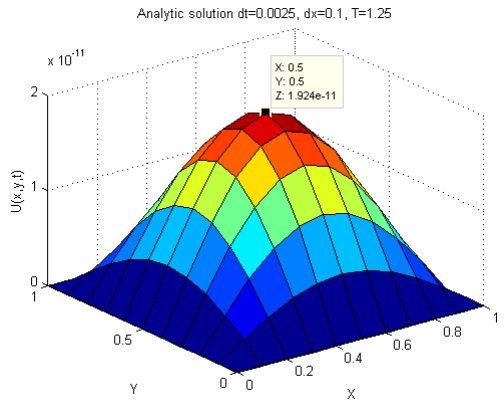

Figure 6: Analytical solution for dx=0.1, $\alpha = 0.25$, T=1.25

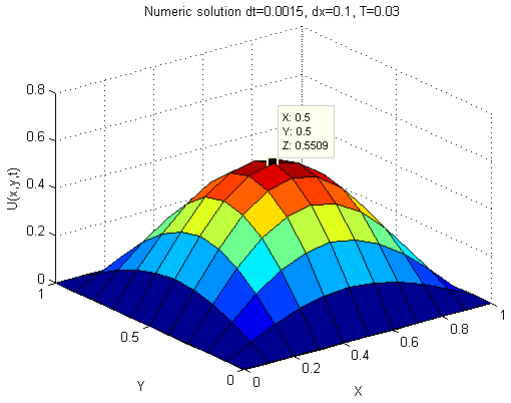

Figure 7: Explicit solution for dx=0.1, $\alpha = 0.15$, T=0.03

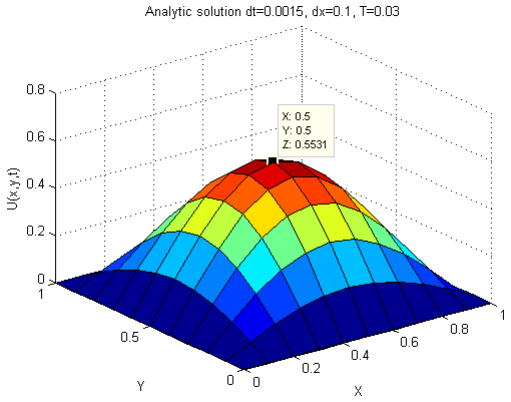

Figure 8: Analytical solution for dx=0.1, $\alpha = 0.15$, T=0.03

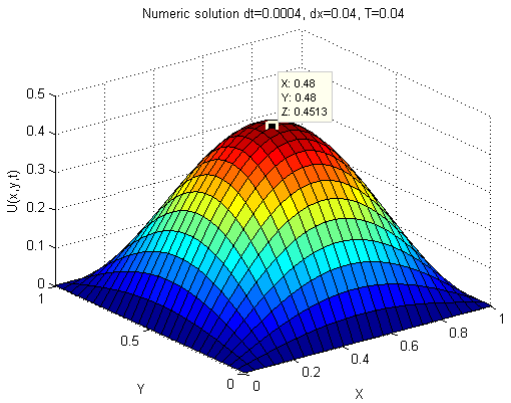

Figure 9: Explicit solution for dx=0.04, $\alpha = 0.25$, T=0.04

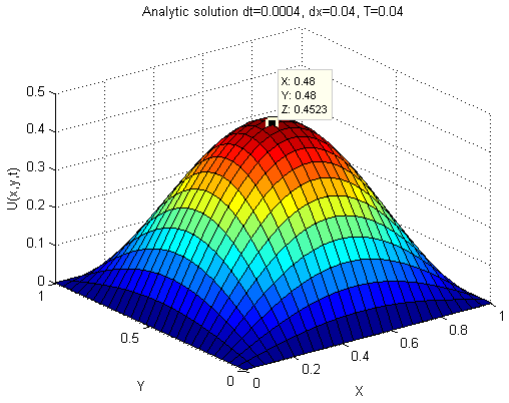

Figure 10: Analytic solution for dx=0.04, $\alpha = 0.25$, T=0.04

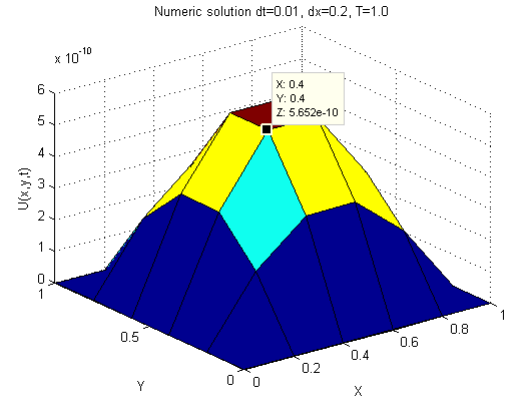

Figure 11: Numeric solution for dx=0.2, $\alpha = 0.25$, T=1.0

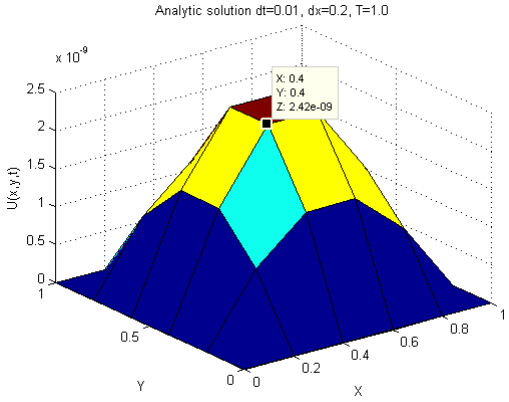

Figure 12: Analytic solution for dx=0.2, $\alpha = 0.25$, T=1.0

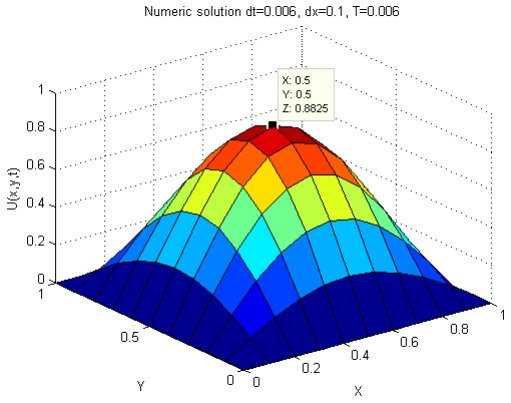

Figure 13: Explicit solution for dx=0.1, $\alpha = 0.6$, T=0.006

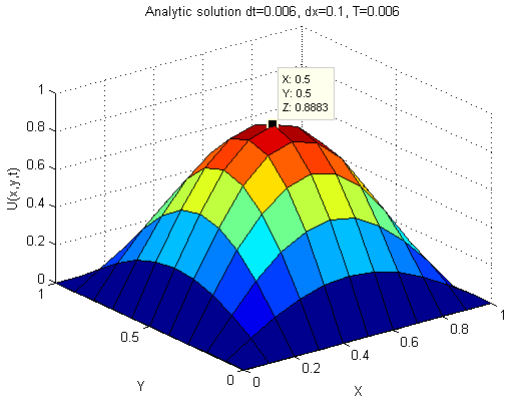

Figure 14: Analytical solution for dx=0.1, $\alpha = 0.6$, T=0.006

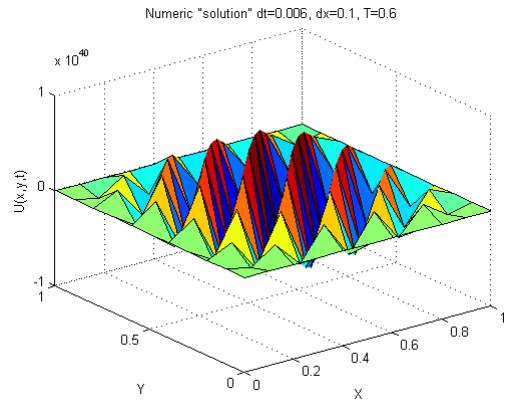

Figure 15: Explicit solution for dx=0.1, $\alpha = 0.6$, T=0.6

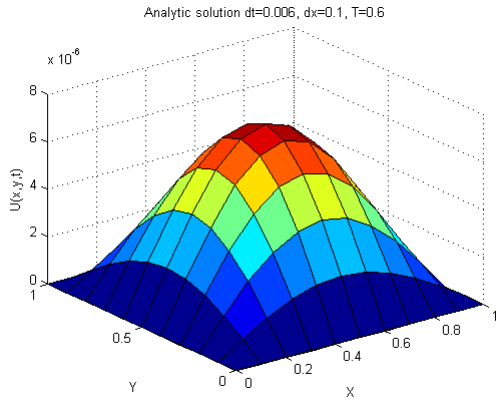

Figure 16: Analytical solution for dx=0.1, $\alpha = 0.25$, T=0.6

5 Discussion

5.1 2D

We see in figure (1) that all the solutions are very correct which is not strange concidering it has only calculated approximately 10 timesteps. But in figure (2) we see that the one that most closely resembles the analytical solution is Forward Euler, followed by Crank-Nicolson and lastly Backward Euler. For these plots we have used as stated before the timestep: $\Delta t = 0.0025$.

So the truncation error in Forward and Backward euler is both $\epsilon = 0.0025$ and in Crank Nicolson it is $\epsilon = 6.25 * 10^{-6}$. Since the truncation error is the error committed by one step of the method, and it assumes the earlier step is the exact solution it will be an accumulated error. But in our plot (figure (2) we see that the Forward Euler method is incredible correct. This is because of the initial values are tailored for the forward Euler method. But if we had choosen other timesteps then the forward Euler method would have failed miserable, while the two other method will be stable for all choosen timesteps and positionsteps.

The number of FLOPS in the forward Euler scheme is $\propto n$ for every timestep, and in backward euler which uses gaussian elimination on a tridiagonal matrix also have number of FLOPS $\propto n$. The Crank-Nicolson scheme is a combination of these two so number of FLOPS will be $\propto 2n$. These round off errors is for every timestep, so the errors will accumulate with time but as we see all the round off errors are proportional with n for every timestep so the error coming from the round off will be approximately the same for the three methods.

5.2 3D

We see from the 3D-plots that the explicit method gives results very close to the analytical solution for closely defined α's. In anology to the one(two)-dimensional case, the algorithm provides an accurate solution because of the initial functions correspondence to the analytical solution for T=0.

Because of the stability limitations care must be taken when choosing optimal resolution in the plots. A reduction in dx in order to increase accuracy

means a dx^2 increase in the number of timesteps needed to analyse the time evolution of the solution up to a point T, meaning we are using a lot more computer power than we would have needed using an implicit scheme where dt is independent of dx. If we are to analyse the temperature evolution during a long timeframe we would therefore need to sacrifice spacial accuracy. And looking at figures (11) and (12) we see that the numeric solutions diverges from the analytical solution when the spacial resolution reduces.

Figure (15) shows the results of not obeying the stability requirements of the explicit scheme after only 100 timesteps. We would not even have needed this many steps, seeing as the results of even a few time-loops gives the wrong solution. α is here given as 0.6, not far from the upper limit of 0.5, meaning the stability limit is very strict.

The number of FLOPS required for the jacobi-algorithm is $\propto 2n^2$ for each timestep, meaning there's an even larger round-off error in the 3D-case.

6 Conclusion

We saw that the truncation error of the Crank-Nicolson scheme was in the order of $10^{(-3)}$ less than that of both the forward and backwards Euler schemes. In spite of this, we saw that for $\alpha = 0.25$ and with an accurate initial condition which the explicit scheme builds on, the forward Euler solutions most closely resembled the analytical solutions for the timesteps analyzed.

In the threedimensional case we saw the limits of the explicit schemes when we analyzed time- and positionsteps not as precicely defined. The stability requirements are strict and gives wrong results when not abided by. For long time frames a reduction in accuracy may also be necessary in order to reduce the number of computations needed ($2n^2\times$ timesteps).

7 Codes and data

The following are all from the github-domain

`https://github.com/sarahbra/Project5:`

/PDE2dim (main)
/PDESolver (main)
/Results
/tex

8 Sources

The sources we have used are: Computational Physics, Lecture notes Fall 2015 (Morten Hjort-Jensen)

Knut Mørken: Numerical Algorithms and Digital Representation, Kompendium 2012 (Knut Mørken)

`http://hplgit.github.io/num-methods-for-PDEs/doc/pub/trunc/sphinx/._main_trunc001.html#example-the-backward-difference-for`

(Hans Petter Langtangen)