Project 4, FYS 3150 / 4150, fall 2013

Odd Petter Sand and Nathalie Bonatout

November 11, 2013

All our source code can be found at our GitHub repository for this project: https://github.com/NathalieB/Project4/

1 Introduction

How can biological processes such as the transport of signals between neurons be modelized on a computer? And what are the important criteria to look at, when writing such an algorithm ? These are the main questions that we used to lead our 4th project.

So the point here was to modelized the diffusion of neurotransmitters across the synaptic cleft. To do so, we implemented three methods, based on Taylor's expansion.:

- the Explicit Scheme, also called forward Euler Scheme
- the Implicit Scheme, or backward Euler Scheme
- the Crank Nicolson Scheme.

We chose to look at this problem as a one dimension problem, assuming that the concentration of neurotransmitters only varies in the direction across the synaptic cleft. Thus, the diffusion equation will be for the whole project a one dimension one. In the first part of this report, we will see how to apply these algorithms to our system. Then, we will compare the three of them, and try t determine which one is the most appropriate given this system.

2 Theory and Technicalities

2.1 Closed form solution

We subtitute

$$v(x,t) = u(x,t) - u_s(x) = u(x,t) + x - 1$$

where $u_s(x) = 1 - x$ is the steady state solution that satisfies our boundary and initial conditions. We then set up the diffusion equation for $v(x,t)$:

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

with known boundary conditions

$$v(0,t) = v(1,t) = 0 \qquad t \geqslant 0.$$

The initial condition $u(x,0) = 0$ then becomes

$$v(x,0) = u(x,0) - u_s(x) = 0 - (1-x) = x - 1 \qquad 0 < x < 1.$$

The solution of this equation is known from pp. 313-314 in the lecture notes, using $L = 1$:

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

We now find the Fourier series coefficients by partwise integration

$$A_n = 2\int_0^1 v(x,0)\sin(n\pi x)dx = 2\int_0^1 (x-1)\sin(n\pi x)dx$$

$$= 2\left(\left[-(x-1)\frac{1}{n\pi}\cos(n\pi x)\right]_0^1 - \int_0^1 -\frac{1}{n\pi}\cos(n\pi x)dx\right)$$

$$= \frac{2}{n\pi}\left([(1-x)\cos(n\pi x)]_0^1 + \left[\frac{1}{n\pi}\sin(n\pi x)\right]_0^1\right)$$

$$= \frac{2}{n\pi}(1+0) = \frac{2}{n\pi}$$

and finally, by substitution, our closed form solution is

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

The derivatives are then

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

2.2 Algorithms

2.2.1 Explicit scheme

```
input: nSteps (# of interior points), time, u_s(x)

deltaX = 1.0 / (nSteps + 1)
alpha = 0.5
deltaT = alpha * deltaX ^ 2
tSteps = 1.0 / deltaT

define v, vNext

for( i = 0; i < nSteps)
        x = (i + 1) * deltaX
        v[i] = -u_s(x)

for( t = 1; t <= tSteps)
        for( i = 1 --> nSteps )
                vNext[i] = (1 - 2 * alpha) * v[i]
                if( i > 0 ) : vNext[i] += alpha * v[i-1] // else += 0
                if( i < nSteps ) : vNext[i] += alpha * v[i+1] // else += 0
        v = vNext

for( i = 0; i < nSteps)
        x = (i + 1) * deltaX
        u[i] = v[i] + u_s(x)

output: u
```

2.2.2 Implicit scheme

```
input: nSteps (# of interior points), time, tSteps, u_s(x)
```

```
deltaX = 1.0 / (nSteps + 1)
deltaT = 1.0 / tSteps
alpha = deltaT / (deltaX ^ 2)

define v, vNext

for( i = 0; i < nSteps)
        x = (i + 1) * deltaX
        v[i] = -u_s(x)

a = -alpha          // diagonal element
b = 1 + 2*alpha     // off-diagonal element

for( t = 1; t <= tSteps)
        tridiagonalSolver(a, b, v, vNext)
        v = vNext

for( i = 0; i < nSteps)
        x = (i + 1) * deltaX
        u[i] = v[i] + u_s(x)

output: u
```

2.2.3 Crank-Nicolson scheme

```
input: nSteps (# of interior points), time, tSteps, u_s(x)

deltaX = 1.0 / (nSteps + 1)
deltaT = 1.0 / tSteps
alpha = deltaT / (deltaX ^ 2)

define v, w

for( i = 0; i < nSteps)
        x = (i + 1) * deltaX
        v[i] = -u_s(x)

a = 2 * (1 + alpha)     // diagonal element
b = -alpha              // off-diagonal element

for( t = 1; t <= tSteps)
        for( i = 0; i < nSteps )
                w[i] = 2 * (1 - alpha) * v[i]
                if( i > 0 ) : w[i] += alpha * v[i - 1] // else += 0
                if( i < nSteps - 1 ) : w[i] += alpha * v[i + 1] // else += 0
        tridiagonalSolver(a, b, w, v)
```

Scheme	Truncation error	Stability
Explicit	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t)$	stable for $\alpha \leq 0.5$, with $\alpha = \frac{\triangle t}{\triangle x^2}$
Implicit	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t)$	always stable
Crank-Nicolson	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t^2)$	always stable

Table 1: Error scaling and stability of our schemes.

```
for( i = 0; i < nSteps)
        x = (i + 1) * deltaX
        u[i] = v[i] + u_s(x)

output: u
```

2.3 Tridiagonal form of the implicit schemes

The methods to reformulate the problem into a tridiagonal matrix equation is described in great detail in pp. 308-312 in the lecture notes, so we will not reproduce them here.

In the case of the Crank-Nicolson scheme, it is easily seen (by the definition of matrix addition and multiplication of a matrix with a scalar) that the sum of a diagonal matrix and a tridiagonal matrix $(2\mathbf{I} + \alpha\mathbf{B})$ is also tridiagonal. We can then note that the known vector $\mathbf{w}_{j-1} \equiv (2\mathbf{I} - \alpha\mathbf{B})\mathbf{v}_{j-1}$is easily calculated in every step by

$$w_i = 2(1-\alpha)v_i + \alpha(v_{i-1} + v_{i+1})$$

Hence there is no reason to calculate the inverse matrix $(2\mathbf{I} - \alpha\mathbf{B})^{-1}$ or demand that it be tridiagonal. The operation above is $\mathcal{O}(n)$, just as the tridiagonal solver itself, so this does not affect the scaling of the algorithm.

2.4 Truncation errors and stability

For explanations of these results, we again refer to the lecture notes (p. 307-309, 312). We will investigate our actual error scaling and test the stability criteria.

3 Results and analysis

3.1 Explicit scheme

As discussed previously, the stability criteria for the explicit scheme is kind of a problem for the explicit scheme. Even though the algorithm is not so hard

to write, we really have to pay attention to the quantity $\frac{\triangle t}{\triangle x^2}$, since this is not stable for all x and t. As soon as we try to compute things with $\frac{\triangle t}{\triangle x^2} > 0.5$, things get messy, and at ssome point, irrelevant.

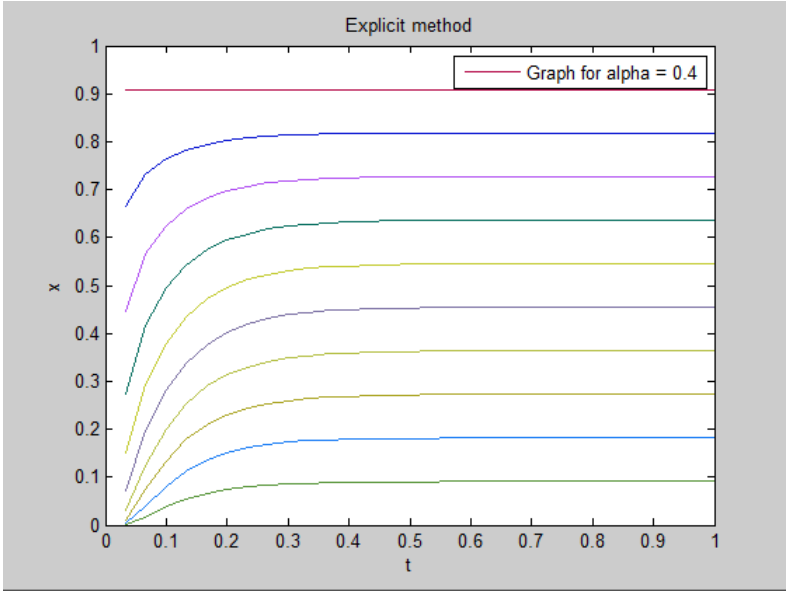

Figure 1: Plot for $\frac{\triangle t}{\triangle x^2} = 0.4$

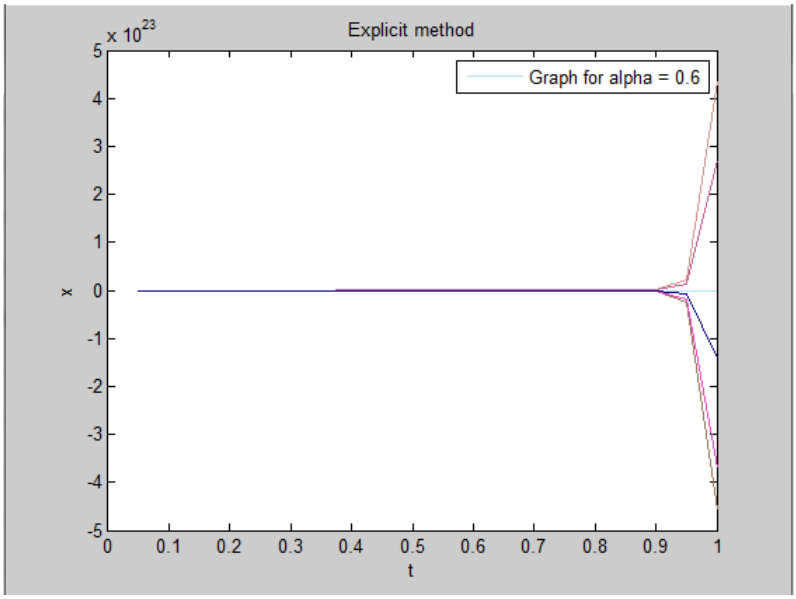

Figure 2: Plot for $\frac{\triangle t}{\triangle x^2} = 0.6$

3.2 Implicit scheme

To implement this method, and the Crank Nicolson one, we chose to re-use the tridiagonal solver we implemented during the first lab. Event though the first step was to make some corrections, since we decided to write it for our specific case last time, it was still quite interesting to see that re-using an "old" piece of code was possible.

3.3 Crank-Nicolson scheme

With the Crank Nicolson scheme, the conditions on the α used for the stability of the explicit scheme is no longer relevant: this algorithm is stable for every x and t.

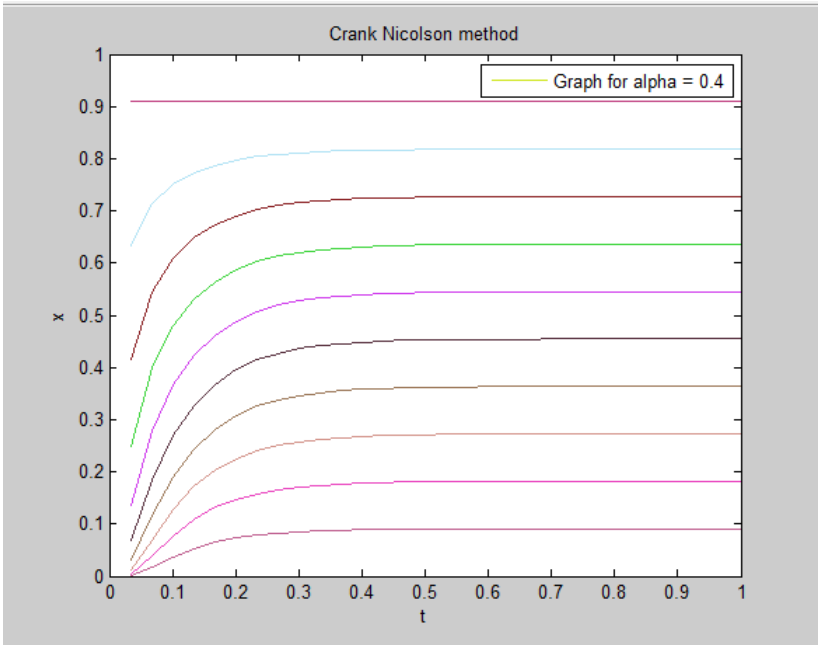

Figure 3: Plot for $\frac{\triangle t}{\triangle x^2} = 0.4$

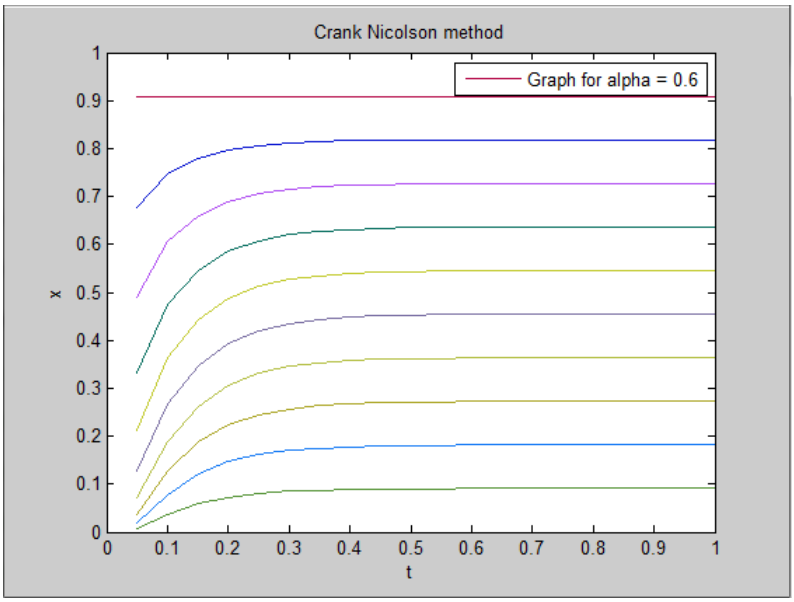

Figure 4: Plot for $\frac{\triangle t}{\triangle x^2} = 0.6$

Finally, we can see that even though the Explicit scheme is easy to implement, the restriction given by its stability domain cannot make it the best scheme for our diffusion equation. The Crank Nicolson scheme is really interesting because we do not have this restriction anymore. But the number of steps to be computed increases, so we have to find a good balance between a fast, and a general algorithm. But it is important to underline the fact that our system can be seen as a tridiagonal one. Thus, solving this problem, with our old tridiagonal algorithm, is rather efficient, since our code is written for tridiagonal matrices.

4 Conclusion

What we learned:

- Fourier series solutions of PDEs is a handy thing to remember, so we can compare to a closed form solution.
- In the implicit and Crank-Nicolson schemes, we do not need to show that the inverse matrices are tridiagonal. It just so happens that they are (though it seems non-trivial to prove), but our algorithm can work around that.

4.1 Critique

- It would be helpful if the exercise text used $u(x,t)$ throughout and not the confusing $u(x)$ formulation that hides the time dependency. This is fine for $u_s(x)$ which is steady-state, but not for u and v.
- It seems that exercise b) and c) merely asks you to reproduce what is in the lecture notes already. It is fine that we have to understand it, but just copying it seems unnecessary, so we skipped it in the report.