1 PDE methods

In implementing the following three methods we used the substitution $v(x,t) = u(x,t) - u_s(x,t)$ outlined in the theory section. This simplifies the initial condition to equation REF 2 and the boundary conditions to REF 3.

1.1 Forward

This algorithm has a simple implementation as v_i^{j+1} is explicitly given by equation REF 10. We compute the new distribution at every time step j with the code

```
for (int i=1; i<Nx; i++) {
    v(i) = alpha*v_prev(i-1) + (1-2*alpha)*v_prev(i) +
        alpha*v_prev(i+1);
}
```

1.2 Backward

As described in the theory section we find the solution by solving the matrix equation REF (ligningen over 14) at every time step j. As seen the solution is given implicitly, but rather than inverting the matrix we solve it on the form $(\mathbb{1} + \alpha\hat{B})V_j = V_{j-1}$ with Gaussian elimination using row operations. The simple form of the three diagonal matrix means that the row operations are fairly easily implemented. The lower non-diagonals $a = -\alpha$ turn out zero while the upper non-diagonals $c = -\alpha$ are unchanged. The diagonals $b = 1 + 2\alpha$ and the right hand side v_i^{j-1} are changed accordingly:

$$b_{i_{new}} = b - \frac{a \cdot c}{b_{i-1_{new}}} \tag{1}$$

$$v_{i_{new}}^{j-1} = v_i^{j-1} - \frac{a \cdot v_{i-1_{new}}^{j-1}}{b_{i-1_{new}}} \tag{2}$$

where *new* signifies the value after a row operation. This constitutes the forward substitution. These lines are implemented in the code as

```
// Forward substitution
v_prev_rowoperated(1) = v_prev(1);
b_rowoperated(1) = b;
for (int i=2; i<Nx; i++) {
    b_rowoperated(i) = b - ac/b_rowoperated(i-1);
    v_prev_rowoperated(i) = v_prev(i) -
        a_c*v_prev_rowoperated(i-1)/b_rowoperated(i-1);
}
```

The backward substitution to finally obtain v_i^j is completed by computing

$$v_i^j = \frac{v_{i_{new}}^{j-1} - c \cdot v_{i+1}^j}{b_{i_{new}}} \tag{3}$$

Implemented in the code as

```
// Backward substitution
v(Nx-1) = v_prev_rowoperated(Nx-1)/b_rowoperated(Nx-1);
for (int i=Nx-2; i>0; i--) {
    v(i) = (v_prev_rowoperated(i)-a_c*v(i+1))/b_rowoperated(i);
}
```

These computations are done in a loop iterating over every time step j to obtain the final vector $V_{N_t} = v_i^{N_t} = v(x, t = T)$.

1.3 Crank-Nicolson

From equation REF 18 we have that

$$(2\mathbb{1} + \alpha\hat{B})V_j = (2\mathbb{1} - \alpha\hat{B})V_{j-1} \tag{4}$$

Thus V_j can be obtained by first computing the vector on the right hand side and solving the resulting linear equations problem. This is explicitly given and so the method from the Forward Euler procedure can be applied here. After this we are left with an implicit problem similar to the Backward Euler method and we solve it by the same procedure of Gaussian elimination. At each time step j we then have a two step procedure combining the code from Forward and Backward Euler.

2 Monte Carlo Theory

The second approach to our problem is a Monte Carlo scheme with random walks. This is a method closely related to diffusion for the following reasons. The framework of random walks allows for a microscopic description of Brownian motion. Brownian motion is the behaviour exhibited by small fractions of any system when exposed to random fluctuations of the medium - generally it is the random movement of particles. Diffusion can be the behaviour of a large number of particles subjected to Brownian motion. While the behaviour may be truly random at the microscopic level of each particle, the behaviour at the macroscopic level of the system becomes seemingly deterministic. This is because macroscopic system states which are only obtained from certain combinations of particle movements

become infinitesimally unlikely compared to system states that are possible results of varous different motions of the particles. This is equivalent to the fact that entropy increases. Diffusion results from the overwhelming probability of movement towards lower concentration.

In our case the diffusing particles are the neurotransmitters navigating the synaptic cleft. This leads to the idea of achieving a simulation of the diffusion process by modelling the neurotransmitters with random walkers. A Markov process is a simple random walk in that two conditions for the microscopic movements are met - ergodicity and detailed balance, assuming there is an equal probability for every microscopic motion and that the probabiltity is also equal to that of the reverse motion.

Underlining the close link between Markov chains and diffusion is the fact that the discretized diffution equation can be derived from a Markov process. Or stated differently, a Markov process yields in the limit of infinitely many steps the diffution equation. REF KOMP.

It can be shown that for the diffusion process the root mean square displacement after a time Δt is

$$\sqrt{\langle x^2 \rangle - \langle x \rangle^2} = \sqrt{2D\Delta t} \tag{5}$$

We will implement this as our step length l_0 when developing the algorithm.

3 Monte Carlo

3.1 Discrete

In the first implementation of the Monte Carlo scheme the set of possible particle positions is discretized. The particles move with a constant step length $l_0 = \sqrt{2D\Delta t}$ with equal probability for moving left and right.

The algorithm is implemented as two for loops. The outer loop iterates over the given number of Monte Carlo cycles, each cycle representing a new particle. The inner loop iterates over the given number of time steps. Thus it simulates a random walk for each particle, moving either left or right at every time step. If the particle moves outside the synaptic cleft the loop breaks. This accounts for the boundary conditions. While the result is an array representing the concentration of neurotransmitters at a final time T, we count the transmitter's position at every time step. This is meant as a computationally efficient way of accounting for the constant stream of new transmitters released into the cleft. The transmitter's movement at a time earlier than T might well represent the movement of a transmitter that entered the cleft at a later time. The code for the random walk loop is as follows

```
for (double time = dt; time <= T; time+=dt) {
    if (ran0(&idum) <= move_prob) { // move to the right
        pos_index += 1;
        // if it has reached the end, stop counting it
        if (pos_index >= Nx) break;
    } else {                    // move to the left
        pos_index -= 1;
        // if it has returned to start, stop counting it
        if (pos_index <=0) break;
    }
    // count the particle's position
    pos_count(pos_index) += 1;
}
```

3.2 Continuous

This method implements the Monte Carlo scheme with a continuous set of positions. This is done by changing the step length to $l_0 = \xi\sqrt{2D\Delta t}$ where ξ is a random number chosen from a Gaussian distribution with mean value 0 and standard deviation 1. See the code for the implementation. The initial positions of the particles are given by a randomly distributed position within the first interval, as opposed to them initially being set to 0. There exists some alternative approaches here, such as initiating the particles at an interval left of zero, or in a small interval centered at zero. The initial condition is given as the Dirac delta function and it is suggested that the problem is how to represent this well at continuous values.